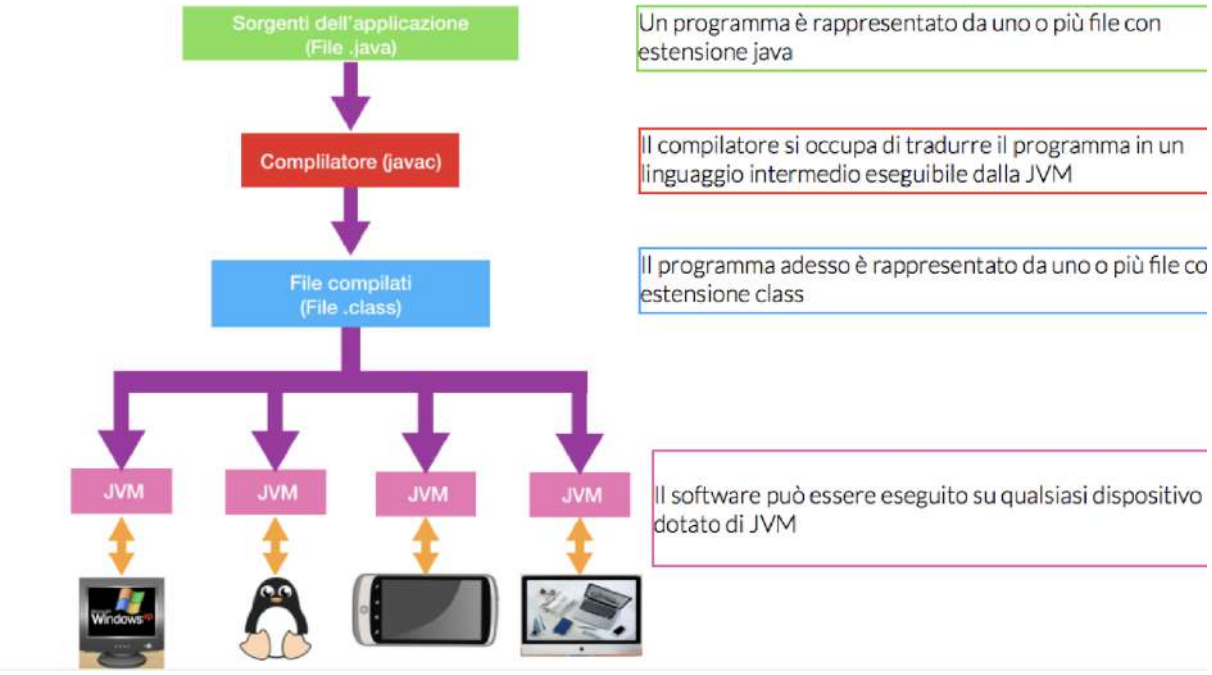
Appunti Java

Introduzione a Java

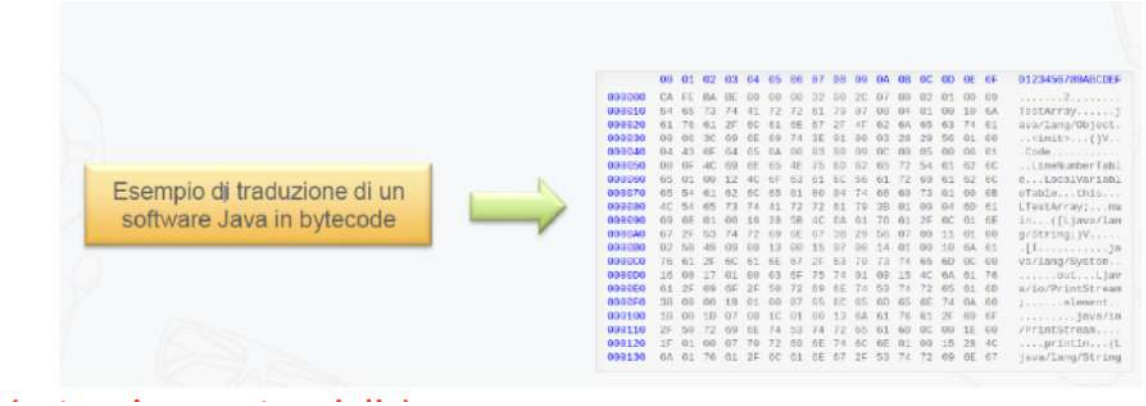


Java è un linguaggio di programmazione con le seguenti caratteristiche:

1. Object Oriented (orientato agli oggetti
2. Portabile, ovvero l'indipendenza dal sistema operativo. Gli elementi che rendono il linguaggio Java portabile sono:
    1. Java Virtual Machine (La macchina virtuale o JVM)
    2. Java Platform

La macchina virtuale

Java Virtual Machine o JVM è un software che si occupa di eseguire i programmi tradotti in bytecode dal compilatore. La JVM è una CPU virtuale: traduce i bytecodes nelle istruzioni macchina della CPU del dispositivo reale sul quale si vuole eseguire il programma



Vantaggi e svantaggi di Java

Vantaggi:

- indipendenza del linguaggio bytecode: consente di eseguire lo stesso programma su più dispositivi dotati di JVM
- velocità di sviluppo
- grande disponibilità di librerie
- alta integrazione con il web

Svantaggi:

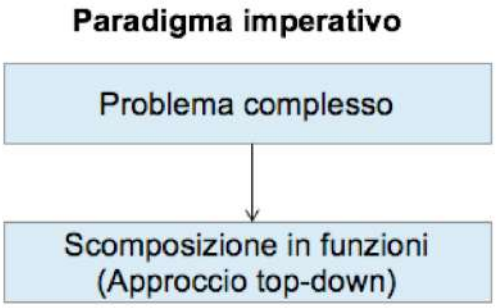
- Velocità di esecuzione: il programma viene eseguito ed elaborato dalla JVM che a sua volta traduce le istruzioni in linguaggio macchina. Pertanto il tempo di esecuzione è leggermente più lento rispetto ad un programma scritto in C++.
- Attraverso la de compilazione è possibile risalire al codice sorgente (a meno di usare opportuni strumenti che si chiamano java obfuscator)

I paradigmi della programmazione

Un paradigma è l'approccio che si segue per risolvere un problema (insieme d'idee) I paradigmi più comuni:

–Imperativo (importante è l'algoritmo e la programmazione strutturata) ==> dati e funzioni sono separati

–A oggetti (importante è l'oggetto, descritto dai dati e dalle funzioni, dette metodi, che operano sui dati) ==> dati e funzioni non sono più separati



Vantaggi della OOP:

–Riutilizzo massiccio del codice: si usa un oggetto già creato (e testato) da altri programmatori. Un oggetto può essere usato anche per crearne un altro grazie all'ereditarietà.

Cosa bisogna sapere di un oggetto?

–Come si usa, non come è fatto!

–Esempio: per assemblare un PC, si usano oggetti (schede madri, CPU, schede video etc) già creati da altri.

E' necessario sapere la CPU come è fatta?

Parole riservate del JDK: Keywords

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

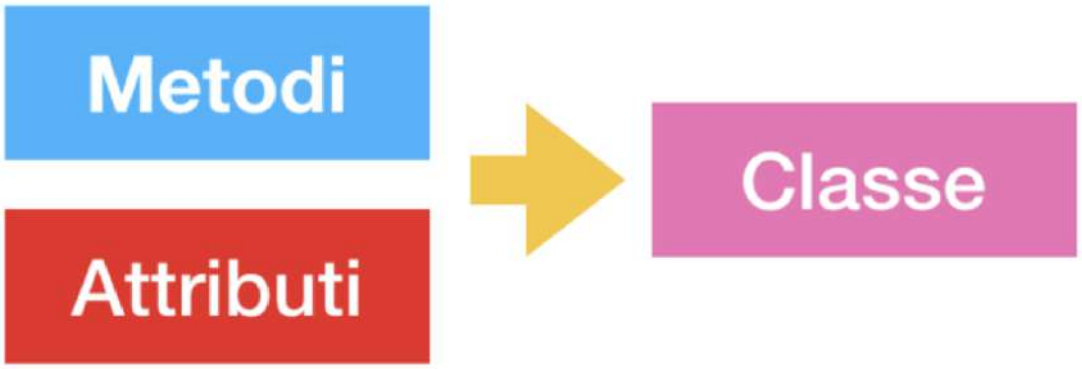
Struttura di un programma

I programmi in Java sono costituiti da classi. Potrebbero esserci decine di migliaia di classi. Un programma minimo è costituito da almeno una classe. Per ogni classe, viene creato un file separato. Il nome del file corrisponde al nome della classe.

Gli elementi che compongono una classe sono:

1. **Metodi**
2. **Attributi**

I metodi e gli attributi in un programma Java devono essere contenuti in una classe:



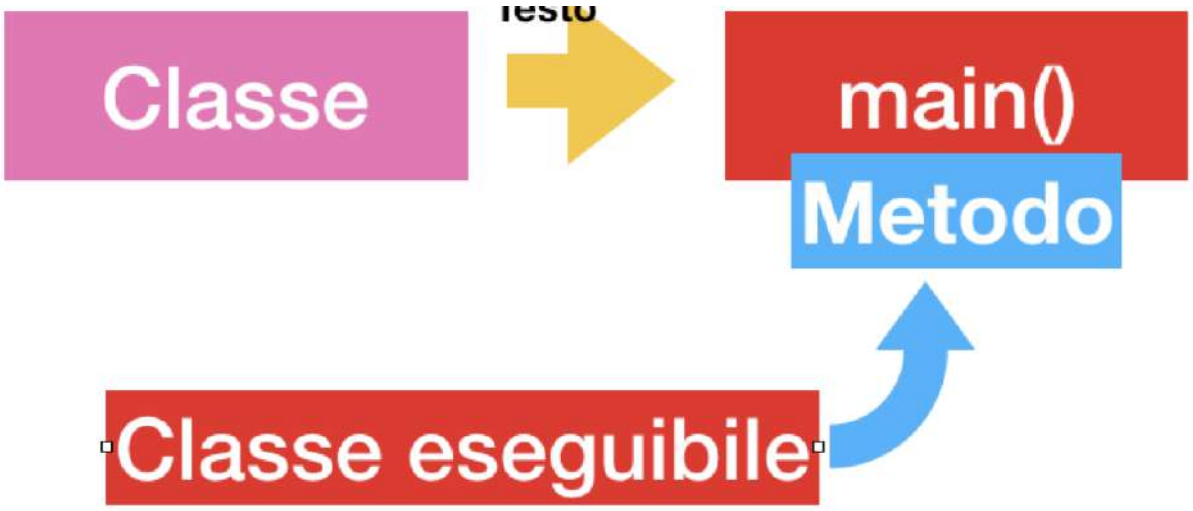
Il codice di una classe è costituito dal nome della classe e dal corpo della classe racchiuso tra parentesi graffe.

```
public class Main{

    CLASS BODY

}
```

Gli attributi descrivono le proprietà della classe mentre i metodi sono le azioni che si possono compiere.

Classe eseguibile (metodo main)

Per poter essere eseguito un programma Java si deve rendere eseguibile una sua classe. Per far questo bisogna definire nella classe che si vuole rendere eseguibile un metodo chiamato **main();**




```
public static void main(String[] args)
```

1. La parola chiave **public** indica che il metodo può essere invocato da qualsiasi luogo
2. La parola chiave **static** indica che il metodo può essere invocato senza creare un'istanza della classe
3. La parola chiave **void** indica che il metodo non restituisce alcun valore
4. La variabile d**ell'array args** contiene argomenti inseriti nella riga di comando se non ci sono argomenti, l'array è vuoto

Questi concetti saranno visti nei prossimi argomenti. Per ora bisogna sapere come scrivere ed eseguire un semplice programma Java con il metodo principale.

```java
class Main {
  public static void main(String[] args) {
    System.out.println("Hello world!");//out in java
  }
}
```

L'output su schermo si ottiene tramite il comando :

System.out.println()

Tra parentesi si scrive l'argomento da stampare.

- System.out.println(1); Visualizza il numero 1 sullo schermo
- System.out.println("Java"); Visualizza "Java" sullo schermo
- System.out.println("Java e C++"); Visualizza "Java e C++" sullo schermo

Questo metodo ha due versioni:

1. System.out.println() stampa e manda a capo il cursore, se si usa più volte ogni volta l'argomento passato viene visualizzato su una riga separata.
2. System.out.print(), il testo viene visualizzato sulla stessa riga.

La classe che avvia il programma può avere qualsiasi nome, ma il metodo principale deve sempre avere lo stesso aspetto:

```java
public class Home
{
  // parte invariante
  public static void main(String[] args)
  {


    //CODICE DEL METODO


  }
}
```

Se il metodo principale ha una dichiarazione non valida, sono possibili due casi:

1. Il programma non può essere compilato
2. Il programma è stato compilato correttamente ma non può essere avviato

Il programma non può essere compilato

È il caso in cui la dichiarazione del metodo principale viola la sintassi di Java.

Esempio:

public static main(String[] args)

nessun valore restituito

Il programma può essere compilato ma non può essere eseguito:

È il caso in cui il metodo, è scritto in modo corretto come metodo normale, ma non soddisfa il requisito del metodo principale

Esempio:

public static void main(String args)

dovrebbe essere String [] args

public void main(String[] args)

manca la parola static

Quindi, il metodo principale è il punto d'ingresso di qualsiasi programma Java. Ha una sintassi molto specifica che bisogna ricordare.

Per realizzare un programma e renderlo eseguibile si devono compiere i seguenti passi:

1. Scrivere il codice sorgente;

2. compilare il codice sorgente;

3. eseguire il codice compilato.

Per editare il codice Java si può usare qualsiasi editor di testi e salvarlo in un file con estensione .java. Il nome da dare al file corrisponde al nome della classe. Se il programma è composto da più classi, si deve memorizzare ogni classe in un file diverso.

Per compilare il programma occorre usare il Prompt dei comandi. Il compilatore Java viene richiamato usando il seguente comando:

```
javac nomefile
```

La compilazione genera un file compilato che ha estensione .class e rappresenta il bytecode.

Per eseguire il programma bisogna attivare l'interprete Java. L'interprete prende il codice compilato (bytecode), lo traduce in codice macchina e lo esegue. L'interprete Java viene richiamato usando il comando:

```
java nomefile
```

Il nome del file passato all'interprete corrisponde alla classe che contiene il metodo main.

Esistono molti IDE che permettono di eseguire tutte le precedenti operazioni esempi

eclipse, netbeans ecc…

Definizione di una classe

Un programma Java è costituito da classi: ogni classe è memorizzata in un singolo file, il cui nome coincide con il nome della classe. L'estensione del file è java.

Un programma consiste in una serie di file con l'estensione 'java', e ogni file contiene il codice per una sola classe

Se un file si chiama Moto.java, contiene la classe Moto.

Quando si hanno molti file si raggruppano in cartelle e sottocartelle, inoltre le classi sono raggruppate in pacchetti e sotto-pacchetti.

I package

I package rappresentano un meccanismo per organizzare classi Java in sottogruppi ordinati. Si tratta di uno strumento utile per organizzare le classi in modo logico e ordinato sotto un unico nome. Le classi principali della libreria di base di Java sono raccolti nel package java.lang. I package consentono anche di creare classi pubbliche con nomi uguali. È sufficiente collocarle in package diversi. Il nome di un package deve essere univoco. Per usare una classe di un package si usa la parola chiave package seguita dal percorso della classe. La strutturazione dei tipi in package consente di raggiungere i seguenti scopi.

- Evitare conflitti di nome tra tipi, poiché il nome della classe è parte del nome del package.
- Riusare tipi già scritti da altri programmatori importando i relativi package.
- Raggruppare i tipi secondo criteri funzionali e di correlazione.

I nomi dei pacchetti e dei sotto pacchetti vanno indicati nel codice della classe, e devono avere lo stesso nome delle cartelle.

Quindi, da un lato, ci sono i file archiviati nelle cartelle e dall'altro le classi memorizzate nei pacchetti. Un nome di classe deve anche coincidere con il nome del file. Il nome del pacchetto coincide con il nome della cartella in cui è archiviata la classe.

Variabili e Data Type

Per introdurre i primi concetti risolviamo un problema di matematica da prima elementare e vediamo i passi da fare per risolverlo.

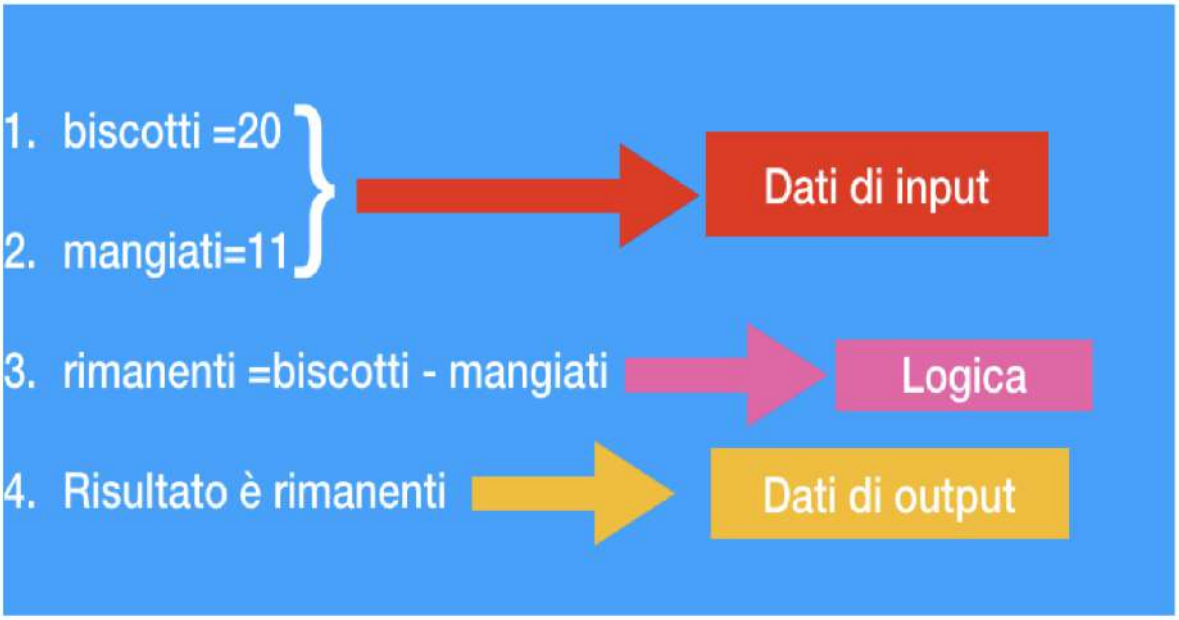
Roberta ha 20 biscotti e ne mangia 11. Quanti biscotti rimangono a Roberta?

Scomponiamo il problema nei passi elementari:

20 biscotti

11 mangiati

Quanti biscotti rimangono?



Le prime due righe rappresentano i dati d'input, la terza riga la logica di risoluzione e la quarta riga l'output.

Convertiamo questo pseudo codice in java:

Bisogna inserire il codice dentro il metodo main in una classe:

```
public class Main{
public static void main(String[] args)
```

```
  {
    int biscotti;
    biscotti =20;
    int mangiati=11;
  int biscottiRimasti=biscotti-mangiati;
  System.out.println("biscotti rimasti "+biscottiRimasti);
  }
}
```

In queste semplici righe di codice abbiamo introdotto alcuni concetti fondamentali:

Variabili

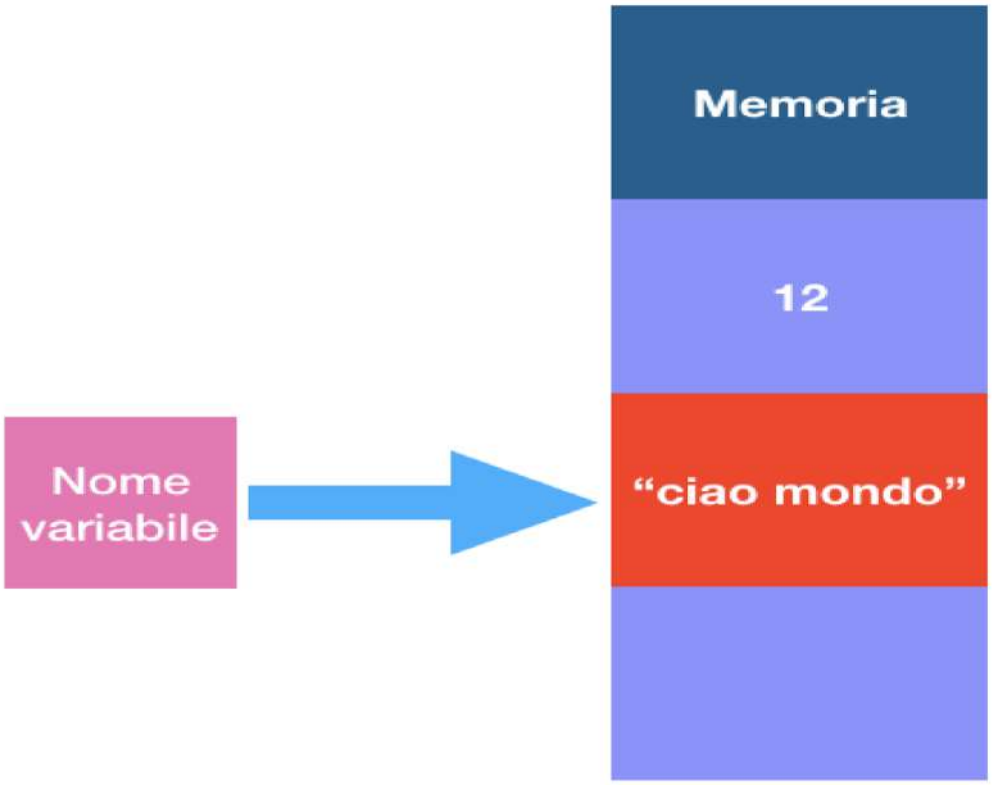
Tramite la keyword **int, si dichiara** che **biscotti** è una **variabile (di tipo intero)**, ossia un contenitore che permette di salvare, aggiornare e recuperare i dati.

Una variabile in Java ha tre proprietà importanti:

tipo, nome e valore.

Il nome, ci permette di distinguere una variabile da un'altra. È come un'etichetta su una scatola.

Rappresenta un nome che si assegna a uno spazio di memoria alterabile. Il nome della variabile, rappresenta l'indirizzo fisico in memoria. Serve per identificare la locazione della variabile in memoria al fine di accedere o modificarne il valore durante l'esecuzione.



Una variabile può memorizzare solo dati il cu**i tipo è uguale al suo**.

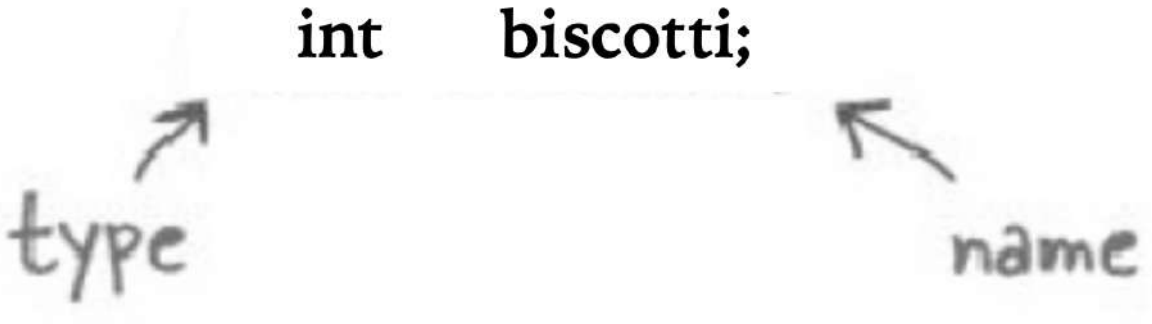
Il valore assegnato alla variabile può essere modificato durante l'esecuzione del programma, ma il nome e il tipo rimangono inalterati.

Nell'utilizzo delle variabili distinguiamo due fasi:

1. **Dichiarazione:** si effettua utilizzando la parola chiave che identifica il **tipo** seguita da un identificatore.

tipo biscotti;

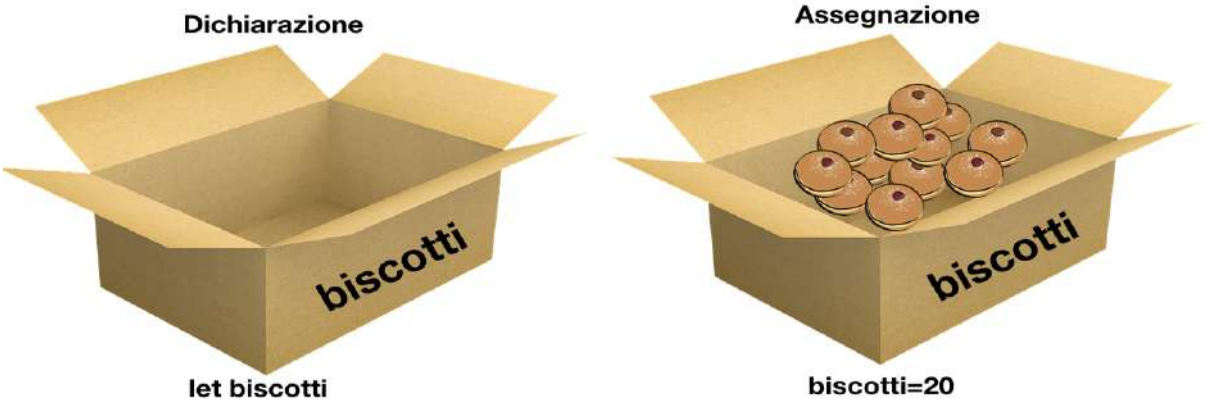
Questa istruzione crea una nuova variabile (contenitore) che si chiama biscotti, tale variabile ancora non è definita ossia non contiene nessun valore.



2. **Assegnazione**: si effettua utilizzando l'operatore di assegnamento, rappresentato dal simbolo **uguale (=**) Seguito dal valore da attribuire, ricordando che tale valore deve essere dello stesso tipo della variabile in questione.

biscotti=20;

Viene valorizzata la variabile con il valore 20, (inseriamo un contenuto nella scatola).



Le due fasi si possono raggruppare in un unica istruzione:

int mangiati=11;

Ogni dichiarazione/inizializzazione può essere effettuata su più variabili in una sola riga utilizzando il simbolo di virgola (,).

Codice

Il nome delle variabili delle costanti, dei metodi e degli oggetti viene detto identificatore e deve rispettare alcune regole:

- non deve coincidere con una delle parole chiave del linguaggio;
- non può iniziare con un numero;
- non può contenere caratteri speciali come ad esempio:
  - lo spazio,
  - il trattino (-),
  - il punto interrogativo (?),
  - il punto (.),
- Sono però ammessi:
  - l'underscore (_)
  - il simbolo del dollaro ($).

Java è case sensitive, quindi i nomi miaVariabile e MiaVariabile indicano due variabili diverse. Per convenzione il nome di una variabile viene scritto in minuscolo e se è composta da più parole si utilizza la rappresentazione a gobba di cammello ossia si scrivono in maiuscolo le iniziali delle parole che seguono la prima:

int biscottiRimasti

```
int number_1; // CORRETTO
int number 1; // ERRORE - ';' expected
int 1number; // ERRORE - not a statement
// a e A sono variabili DIVERSE!!!
int a;
int A;
```

```
// dichiarazione e inizializzazione di variabili primitive
int a = 44, b = 55;
// dichiarazione e inizializzazione di variabili riferimento
String my_str = new String("ciao mondo");
// dichiarazione
float c,d;
// inizializzazione
c = 33.33f;
d = 44.44f;
```

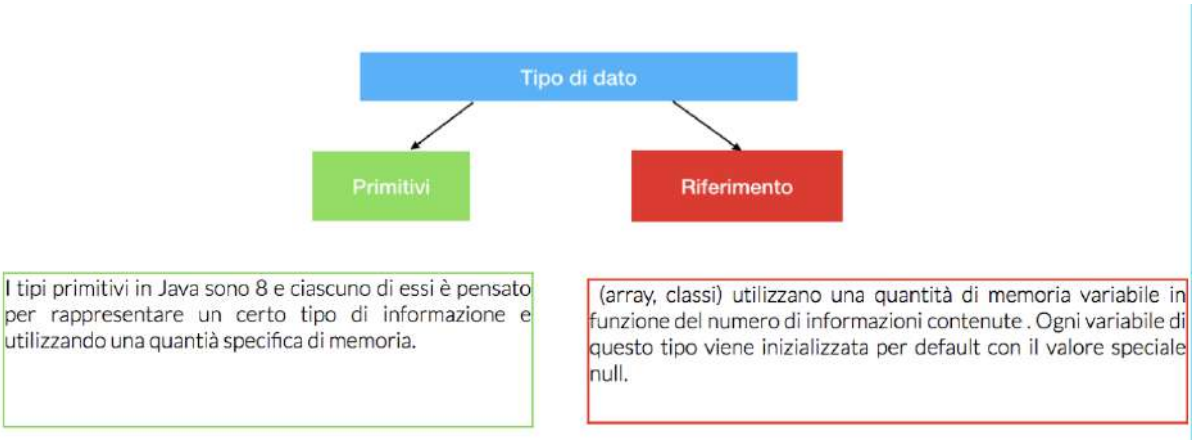
Java è un linguaggio fortemente tipizzato, di ogni variabile si deve specificare il Data Type di appartenenza il quale classifica in modo preciso:

- un insieme di valori
- le operazioni definite su tali valori.

Per esempio, il data type int, definisce un insieme di valori Interi, più un insieme di operazioni ammesse su tali valori (addizione, sottrazione, e così via).

In Java esistono due tipi di dato:

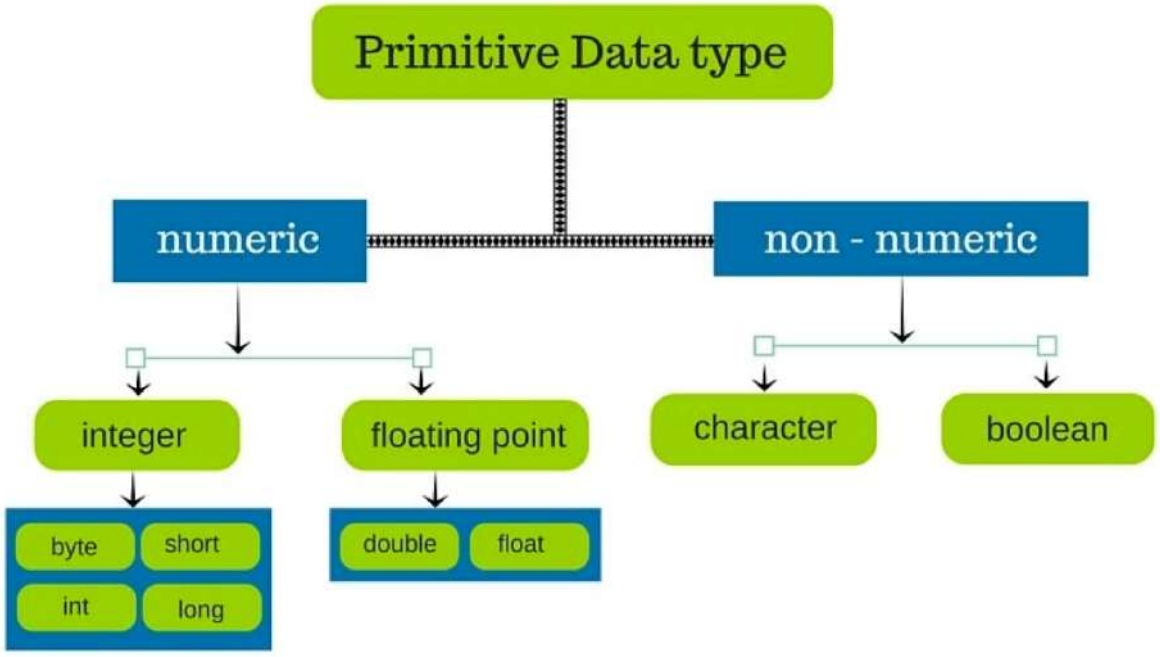
1. Primitivi
2. Riferimento



Tipi di dati primitivi

Java definisce otto tipi di dati primitivi:

- Interi: byte, short, int, long. (differiscono solo per il numero di byte occupati)
- floating point (o a virgola mobile): float e double
- testuale: char.
- logico-booleano: boolean.



Tipi di dati interi (Literal)

Literal è la forma letterale con cui si può rappresentare un tipo primitivo all'interno del codice sorgente, e in questa forma il compilatore è in grado di determinare il data Type di riferimento.

Un numero intero può essere rappresentato come un insieme di valori espressi in base Decimale.

Oltre alla notazione decimale si possono usare anche le notazioni:

Binaria: la notazione che utilizzano i processori dei computer, composta solo da 0 e 1. Bisogna anteporre al numero intero, uno 0 (zero) e una b (maiuscola o minuscola).

Ottale: si utilizzano solo i numeri da 0 a 7. Bisogna anteporre al numero intero uno 0 (zero).

Esadecimale: si utilizzano oltre ai numeri da 0 a 9 anche le lettere da A ad F. Bisogna anteporre al numero intero 0 (zero) e una x (indifferentemente maiuscola o minuscola).

Esempi:

byte b = 10; //notazione decimale: b vale 10

short s = 022; //notazione ottale: s vale 18

long l = 0x12acd; //notazione esadecimale: l vale 76493

int i = 1000000000; //notazione decimale: i vale 1000000000

int n = 0b10100001010001011010000101000101 //notazione binaria: //n vale -1589272251

Tipi di dati a virgola mobile Literal

I numeri floating point (a virgola mobile)sono quei numeri composti da due parti:

1. un parte intera
2. una parte decimale

La forma literal prevede che queste due parti siano separate da un punto.

Questa è la forma più comune di rappresentare i numeri reali, esiste anche la possibilità di rappresentare un floating point con la notazione detta scientifica(esponenziale).

In tale rappresentazione si:

Prende una sola cifra per la parte intera

Si aggiunge il suffisso e (esponenziale)

il numero che segue il suffisso si chiama esponente e rappresenta una potenza di dieci che moltiplicata per il numero ritorna il numero di partenza

Java utilizza per i valori floating point (a virgola mobile) lo standard di decodifica IEEE-754. I due tipi che possiamo utilizzare sono:

Tipo	Intervallo di rappresentazione
float	32 bit (da +/–1.40239846^{-45} a +/–3.40282347^{+38})
double	64 bit (da +/–4.94065645841246544^{-324} a +/–1.79769313486231570^{+328})

Per default un literal in virgola mobile viene considerato da java come un double.

Per assegnare un valore a virgola mobile a un float, non possiamo fare a meno di un cast. Per esempio, la seguente riga di codice provocherebbe un errore in compilazione:

float f = 3.14;

il cast con la sintassi breve:

```
float f = 3.14F;
```

La “effe” può essere sia maiuscola sia minuscola.

Esiste, per quanto ridondante, anche la forma contratta per i double:

double d = 10.12E24; è equivalente a double d = 10.12E24D;

Alcune operazioni matematiche potrebbero dare risultati che non sono compresi nell’insieme dei numeri reali (per esempio “infinito”). Nella libreria standard sono definite delle classi dette classi wrapper (in italiano classi involucro), che non sono altro che classi che rappresentano i tipi di dati primitivi. Gli oggetti istanziati da queste classi sono interscambiabili con i dati primitivi grazie alla caratteristica di Java nota come autoboxing-autounboxing. È per questo che le classi wrapper Double e Float forniscono le seguenti costanti statiche:

- Float.NaN Float.NEGATIVE_INFINITY Float.POSITIVE_INFINITY
- Double.NaN Double.NEGATIVE_INFINITY Double.POSITIVE_INFINITY

Dove NaN sta per “Not a Number” (“non un numero”).

Tipo di dato primitivo letterale(char)

Il tipo char permette di immagazzinare caratteri (uno per volta). Per caratteri si intendono:

Java utilizza l'insieme 16 bit per memorizzare un carattere e usa lo standard Unicode per la decodifica dei caratteri. Per rappresentare un carattere si utilizzano gli apici singoli

'A ' ' ' '3' ecc…

Si possono specificare delle sequenze particolari che iniziano con il simbolo di escape

\

Sequenza di escape	Descrizione
\t	Inserisce una Tabulazione
\b	Inserisce un backspace n
\n	Inserisce una nuova riga
\r	Inserisce un ritorno a capo
\'	Inserisce un singolo carattere di citazione
\"	Inserisce un carattere con doppia virgoletta
\\	Inserisce un carattere barra rovesciata
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)

Literal Boolean

Il data Type boolean è un data type che ammette solo due valori logici, true e false che non vengono convertiti in una rappresentazione numerica. Il true in Java non è uguale a 1, ne il valore false è uguale a 0. In Java, i valori literal boolean possono essere assegnati solo a variabili dichiarate come booleane o utilizzate in espressioni con operatori booleani.

Literal String

I valori literal delle stringhe in Java sono specificati racchiudendo una sequenza di caratteri tra una coppia di virgolette. Esempi

"Ciao mondo"

"due \ n"

Le sequenze di escape funzionano allo stesso modo all'interno di stringhe. Le stringhe in Java devono iniziare e finire sulla stessa riga.

Tipi di dati interi, casting e promotion

Type	Size Byte	Range	Default
byte	1	-128, +127	0
short	2	-32768, +32767	0
int	4	-2147483648, +2147483647	0
long	8	-9.223E18, +9.223E18	0
float	4	+3.4 E+38	0
double	8	+1.7 E+308	0
char	2	0, 65535	0
boolean	1	true, false	false

```
6   package tipi;
7   public class Tipi {
8   incompatible types: possible lossy conversion from int to byte {
    ----
    (Alt-Enter shows hints)
9
10          byte b = 128; //il massimo per byte è 127
12
13
            short s = 32768; //il massimo per short è 32767
15                  incompatible types: possible lossy conversion from int to short
                    ----
                    (Alt-Enter shows hints)
16
            int i = 2147483648; //il massimo per int è 2147483647
18                                  integer number too large: 2147483648
19                                  ----
                                    (Alt-Enter shows hints)
20      }
21  }
22
```

provocano errori in compilazione.

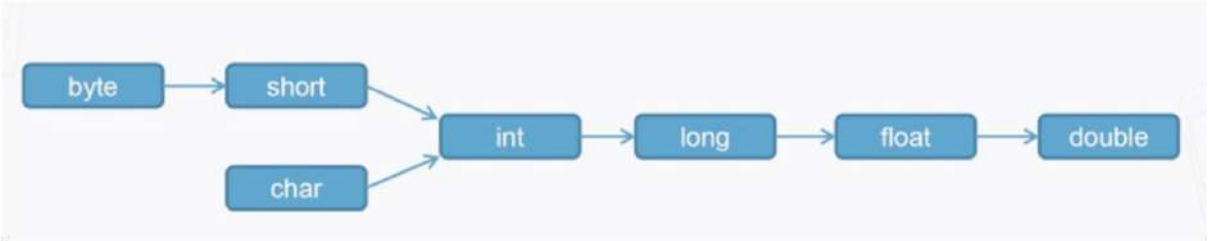
Quando si assegna una variabile a un altra variabile java effettuerà una conversione automatica (implicita) con ampliamento dei valori. Tale conversione è attuata, tuttavia, solo se si verificano due condizioni:

1. I tipi sono compatibili tra loro

2. il tipo finale è più grande del tipo iniziale.

Essa è sempre lecita fra tipi interi (int, byte, short e long), decimali (double float) e char, ma non con i tipi boolean.

```
int a=100;
double d;
d=a;//100.0
```



Casting Esplicito

Se i tipi sono incompatibili, si può provare comunque ad assegnare il valore effettuando un'operazione di conversione esplicita con riduzione del valore tramite un operatore definito di cast che ha la sintassi che segue:

```
(target-type)exp
```

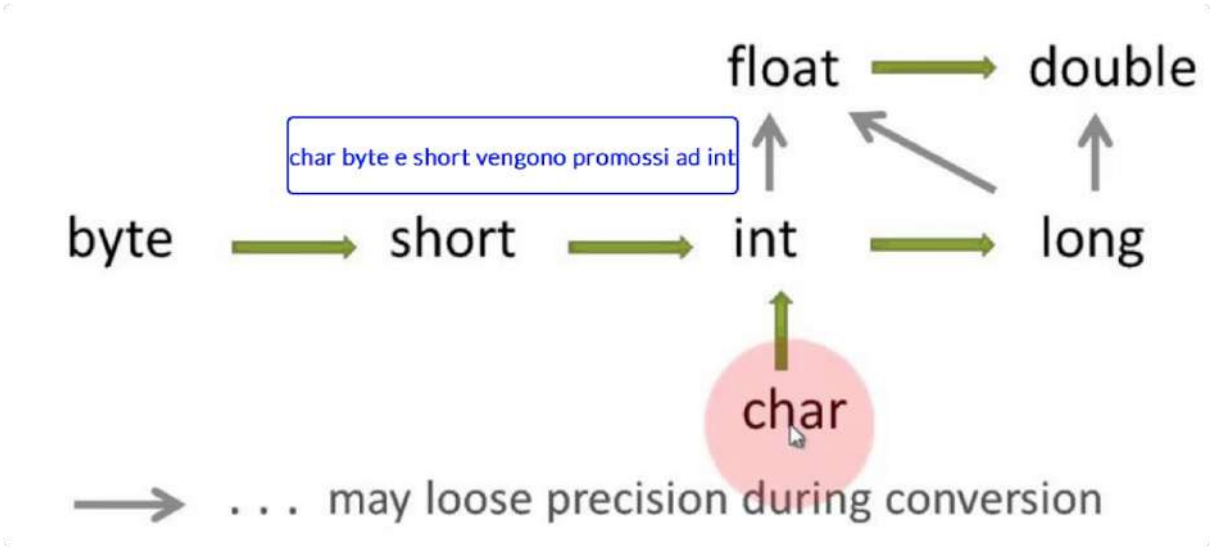
Dove target-type è il tipo di dato in cui si vuole convertire l'espressione. Nell'attuare la conversione possono verificarsi i seguenti casi:

- Se si forza l'assegnamento di una variabile contenente un valore decimale a una variabile di tipo intero, si avrà un troncamento della sua parte frazionaria;
- Se si assegna un valore di una variabile che è più grande del valore massimo contenibile nell'altra variabile, sarà assegnato un valore (detto modulo) che rappresenta il resto della divisione tra i due valori.

```
int a;

doubled=123,45;

a= (int)d; //123
```

Conversione di Tipo all'interno delle espressioni casting e promotion

Un espressione può contenere variabili costanti operatori e valori numerici (literal), se sono presenti tipi di dati differenti java esegue una promozione applicando delle regole.



- Se uno degli operandi è long, l'altro operando sarà convertito in long; altrimenti
- se uno degli operandi è un float, l'altro operando sarà convertito in float; altrimenti
- se uno degli operandi degli operandi è un double, l'altro operando sarà convertito in double;

La promozione automatica degli operandi avviene prima che sia eseguita una qualsiasi operazione binaria.

short a=100,b=130;

int c=a+b;//230

Promozione avvenuta con successo. Se invece consideriamo:

short a=100,b=130;

short c=a+b; //errore

Questo perché java nell'espressione promuove i due short in un intero che quindi non può essere assegnato a uno short, quello che si deve fare è eseguire un casting

short a=100,b=130;

short c=(short)(a+b); //230

Alto esempio:

byte b=50;//nessun problema

b=b+1;//errore perché in un espressione il byte viene promosso a int

sintassi corretta

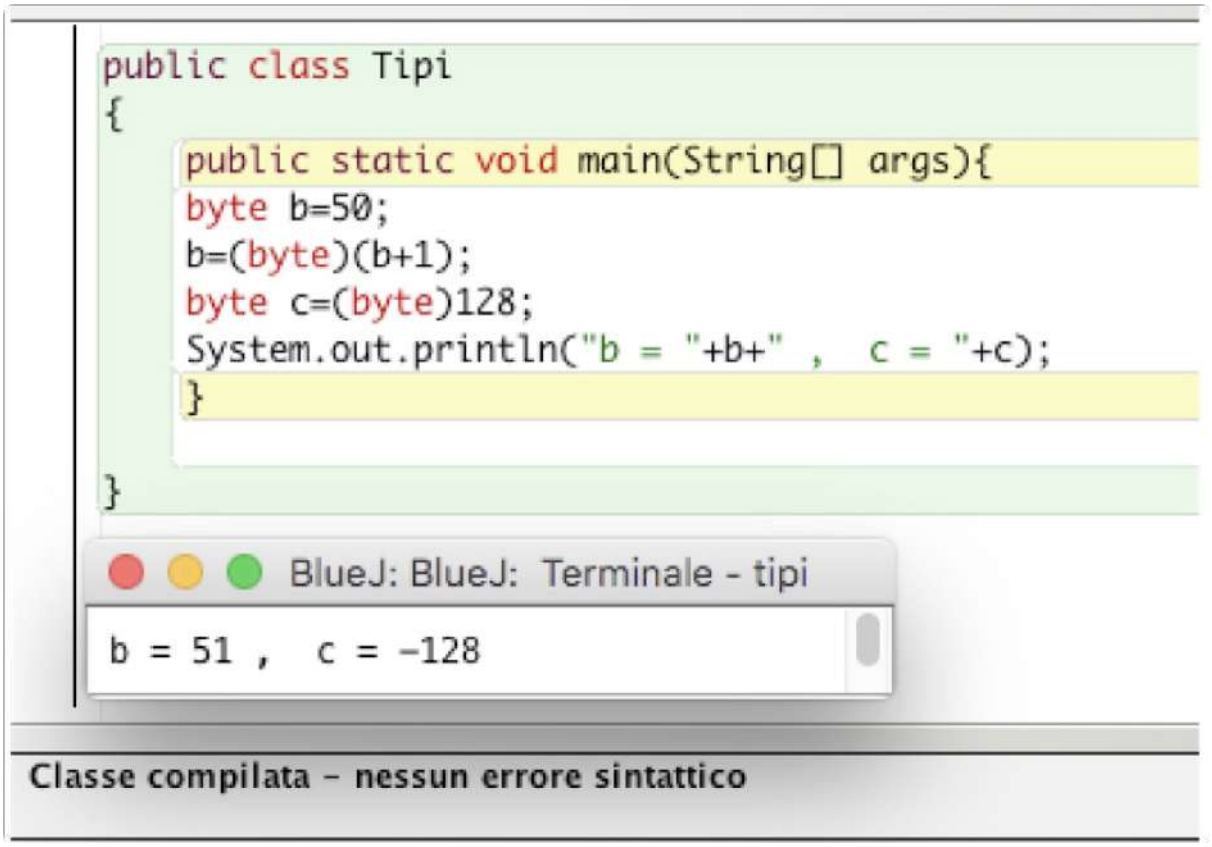
byte b=50;//nessun problema

b=(byte)(b+1); corretto conferiamo l'intero in byte

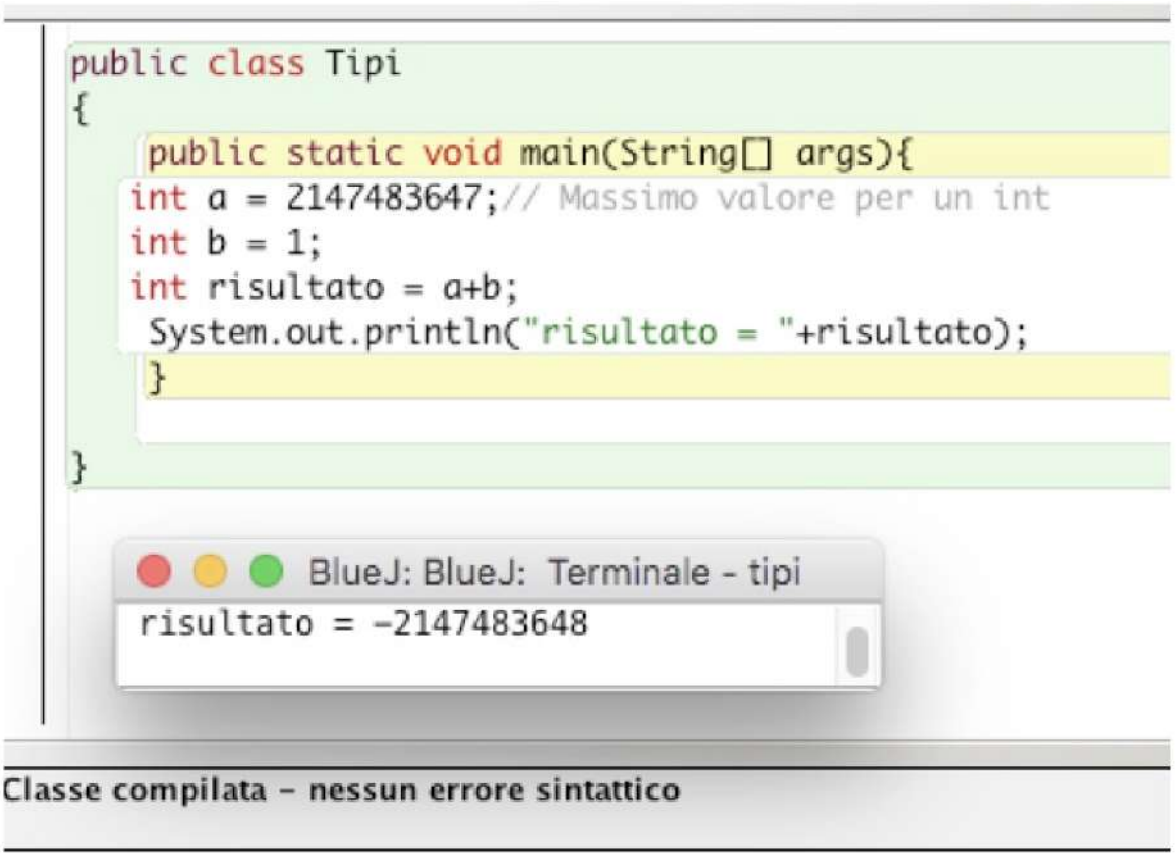
In questo modo il compilatore sarà avvertito di un'eventuale perdita di precisione. Bisogna essere però molto prudenti nell'utilizzare il casting in modo corretto. Infatti se scrivessimo:

b = (byte) 128;

il compilatore non segnalerebbe nessun tipo d'errore. Siccome il cast agisce troncando i bit in eccedenza (nel nostro caso, dato che un int utilizza 32 bit, mentre un byte solo 8, saranno troncati i primi 24 bit dell'int), la variabile b avrà il valore di -128 e non di 128.



Un altro tipico problema di cui preoccuparsi è la somma di due interi. Se la somma di due interi supera il range consentito, è comunque possibile immagazzinare il risultato in un intero senza avere errori in compilazione ma il risultato sarà diverso da quello previsto.



Anche la divisione tra due interi rappresenta un punto critico! Infatti il risultato finale, per quanto detto sinora, non potrà che essere immagazzinato in un intero, ignorando così eventuali cifre decimali. Inoltre, se utilizziamo una variabile long, a meno di cast espliciti, essa sarà sempre inizializzata con un intero. Quindi, se scriviamo:

```
long a = 2000;
```

Dobbiamo tener ben presente che 2000 è un int per default, ma il compilatore non ci segnalerà errori perché un int può essere immagazzinato in un long. Per la precisione dovremmo scrivere:

```
long a = 2000L;
```

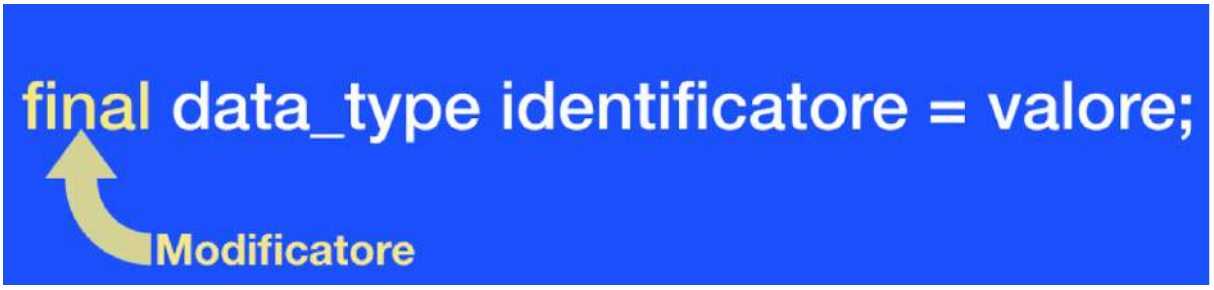
che esegue con una sintassi più compatta il casting da int a long.

Da: / A:	byte	short	char	int	long	float	double
byte	----	*Impl.*	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
short	(byte)	----	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
char	(byte)	(short)	----	*Impl.*	*Impl.*	*Impl.*	*Impl.*
int	(byte)	(short)	(char)	----	*Impl.*	*Impl.*	*Impl.*
long	(byte)	(short)	(char)	(int)	----	*Impl.*	*Impl.*
float	(byte)	(short)	(char)	(int)	(long)	----	*Impl.*
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

Codice

Costanti

Una costante rappresenta uno spazio di memoria in cui è memorizzato un valore che non può essere più alterato dopo che vi è stato assegnato. In Java una costante si dichiara utilizzando la **keyword final.**



final int A = 82;

A = 90; // ERROR – cannot assign a value to final variable a

La dichiarazione di una costante impone delle regole relative a quando e se essa debba essere inizializzata, e tali regole presentano delle differenze a seconda che la costante sia locale a un metodo o globale di classe. Vedremo tali differenze quando studieremo i metodi e le classi.

Operatori aritmetici

Nel codice è presente l'operatore aritmetico

per il calcolo dei biscotti rimasti. Altri operatori aritmetici che agiscono su operandi numerici sono:

Operatore	Descrizione
+	Addizione
-	Sottrazione o segno
*	Moltiplicazione
/	Divisione
%	Modulo

Nelle espressioni la precedenza degli operatori è la stessa dell'algebra:

Prima sono valutati gli operatori di segno delle singole variabili o costanti numeriche.

Successivamente sono valutate le operazioni di moltiplicazione, divisione e modulo.

Per ultime le operazioni di addizione e sottrazione.

Questi criteri di precedenza possono essere modificati con il ricorso ai simboli delle parentesi tonde.

L'unico operatore che può risultare poco familiare è l'operatore modulo. Il risultato dell'operazione modulo tra due numeri coincide con il resto della divisione fra essi. Per esempio:

5 % 3 = 2

10 % 2 = 0

100 % 50 = 0

Questi operatori si dicono binari in quanto si applicano a due operandi.

Esistono anche degli operatori (unari) di pre e post-incremento (e decremento)

Operatore	Descrizione
--	Decremento
++	Incremento

Operatore unario d'incremento "++"

Questo Operatore incrementa l'operando di uno, cioè l'espressione:

x ++

Sarà equivalente all'espressione:

x = x + 1

Tale operatore può essere usato in due modi:

In posizione post, con operatore dopo operando (ad esempio, x++), restituisce il valore prima d'incrementare:

```
int y= 5;
int x = y++;
System.out.println(x);
System.out.println(y);
```

Dà come risultato x = 5, y = 6, poiché l'operazione d'incremento avviene DOPO che alla variabile x viene assegnato il valore di y.

- **Come prefisso** quindi prima dell'operando (ad esempio, ++x), restituisce il valore dopo l'incremento.

y=5;

x = ++y

restituisce il risultato x = 6, y = 6, poiché prima il valore della variabile y aumenta di uno e poi tale valore viene assegnato alla variabile x

Operatore unario di decremento "--"

Questo Operatore decrementa il valore della variabile di uno:

```
x--
```

è equivalente a:

```
x = x - 1
```

Anche per questo operatore valgono le stesse regole viste per l'operatore incremento.

Per le operazioni di assegnazione oltre uguale (=) usato in precedenza, java mette a disposizione gli operatori:

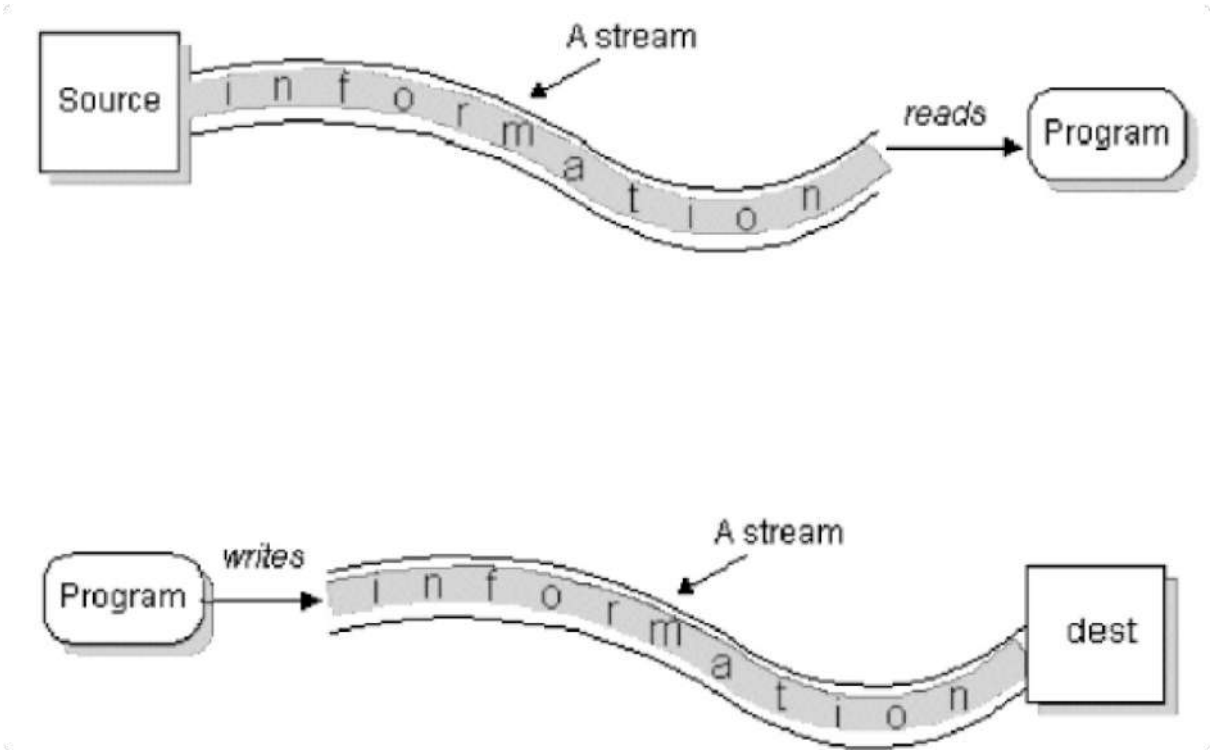
Operatore	Descrizione
+=	Somma e assegna x+=y equivale a x=x+y
−=	Sottrae e assegna x-=y equivale a x=x-y
*=	Moltiplica e assegna x*=y equivale a x=x*y
/=	Divide e assegna x/=y equivale a x=x/y
%=	Calcola il reste assegna x%=y equivale a x=x%y

Introduzione all'input

La classe Scanner

Nel programma precedente le variabili d'input sono state assegnate direttamente nel codice. Per realizzare un programma più efficiente si deve gestire l'input dell'utente. Partiamo dal concetto fondamentale che è alla base del discorso: lo stream (flusso). Per prelevare informazioni da una fonte esterna (la tastiera, un file, una rete etc.), un

programma deve aprire uno stream su essa e leggerne le informazioni in maniera sequenziale. Allo stesso modo un programma può inviare a una destinazione esterna aprendo uno stream su essa e scrivendo le informazioni sequenzialmente.



La classe

java.util.Scanner

permette di leggere i dati da un'origine specificata:

o una stringa

o un file

o la console (caso che trattiamo)

Successivamente, riconosce le informazioni e le elabora in modo appropriato.

La classe Scanner presenta diversi costruttori che permettono di ottenere un oggetto di tipo Scanner a partire da

oggetti di altri tipi, quali ad esempio:

1. System.in (se si deve leggere da tastiera).

2. String, utile in quei casi in cui sia necessario procurarsi degli input da una stringa.

3. File utile in quei casi in cui l'input proviene da un file di testo.

Per approfondimenti si rimanda alla documentazione ufficiale del linguaggio Java (link).

Per utilizzare la Classe Scanner per prima cosa importiamo la classe con l'istruzione:

import java.util.Scanner;

Posizionata in alto prima della definizione della classe:

```
import java.util.Scanner:
public class ProvaScanner{
public static void main(String[] args) {

}
```

Il costruttore che utilizziamo per la lettura da tastiera è:

Scanner in = new Scanner(System.in); //istanzia un oggetto lettore di tipo Scanner
Usando la libreria System.in abbiamo in ingresso un buffer con le informazioni sotto forma di byte da convertire.

Scanner si occupa di convertire il buffer in ingresso nel tipo di variabile che vogliamo (int, String, double….).

La classe Scanner suddivide lo stream dei caratteri in, spezzoni di stringhe(token) separate dai caratteri delimitatori.

I caratteri delimitatori di default sono:

o gli spazi,

o i caratteri di tabulazione

o i caratteri di newline.

Alcuni metodi della classe Scanner:

int nextInt(): legge un numero intero e lo restituisce al chiamante

double nextDouble(): legge un numero reale e lo restituisce al chiamante;

```
import java.util.Scanner;
public class Main {
     public static void main(String[] args) {
          int i;
          double j;
          Scanner scanner = new Scanner (System.in);
          System.out.print("Primo numero:");
          i = scanner.nextInt();
          System.out.print ("Secondo numero: ");
          j = scanner.nextDouble();
          System.out.print ("La somma di"+i+"+"+j+"è: ");
          System.out.println(i + j);
          scanner.close();
     }
}
```

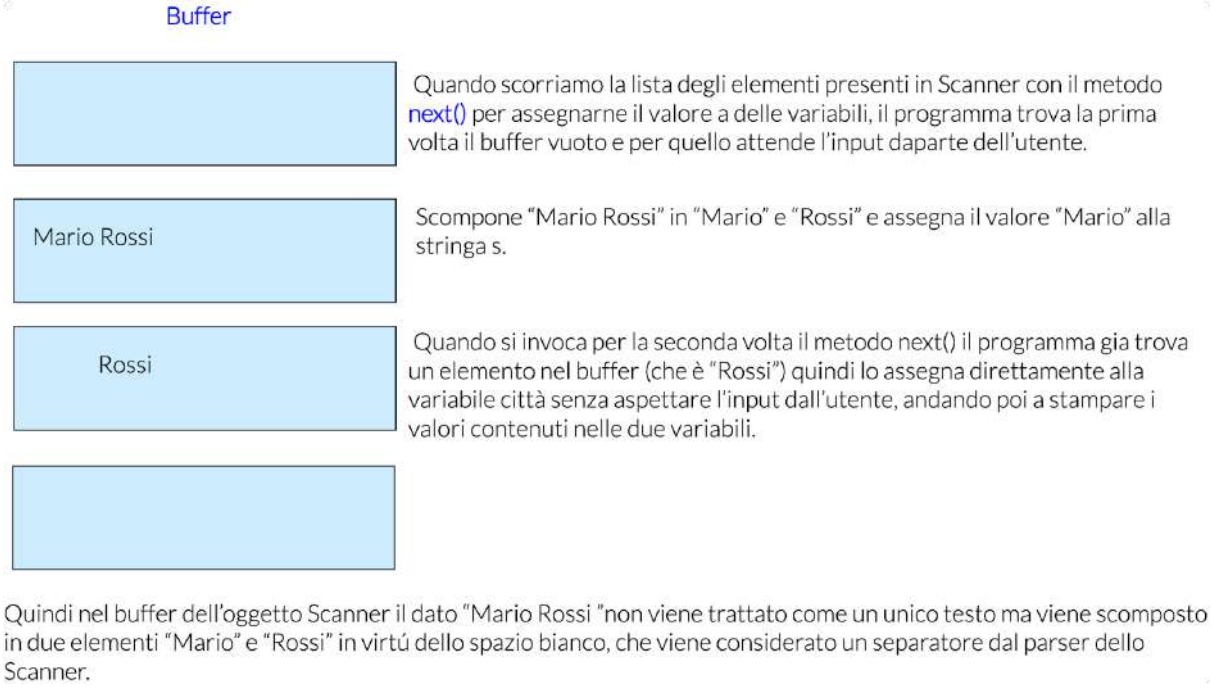
Codice

String next(), legge un blocco di testo (una sottostringa), ossia una sequenza di caratteri contigui senza delimitatori, e lo restituisce al chiamante: questo metodo considera come delimitatori di sottostringhe gli spazi, i caratteri di tabulazione e i caratteri di newline;

String nextLine(): legge una linea di testo e la restituisce al chiamante;

```java
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    String s;
     String città;
    Scanner scanner = new Scanner(System.in);
    System.out.print("Inserisci Nome e Cognome ");
    s = scanner.next();
    System.out.print("inserisci città ");
    città = scanner.next();
    System.out.println("Sig: "+s);
    System.out.println("Nato a: "+città);
    scanner.close();
  }
}
```

Il codice si compora in modo corretto?



Quindi nel buffer dell'oggetto Scanner il dato "Mario Rossi "non viene trattato come un unico testo ma viene scomposto in due elementi "Mario" e "Rossi" in virtú dello spazio bianco, che viene considerato un separatore dal parser dello Scanner.

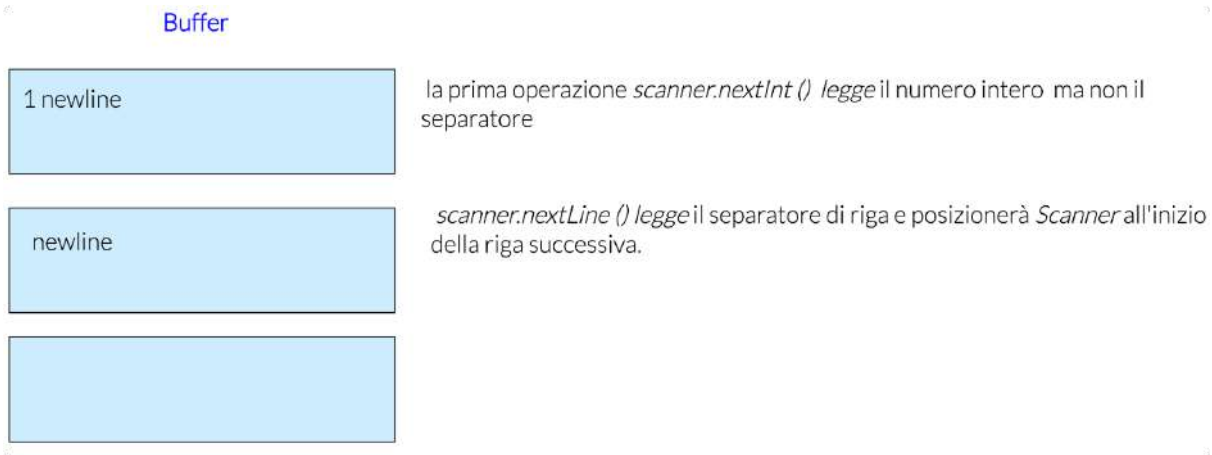
Codice

Soluzione: usare il metodo

nextLine() in grado di leggere un'intera riga e posizionare il cursore nella linea successiva.

Se si usa nextLine() bisogna fare attenzione alla sequenza di letture che si eseguono.

```
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    int i;
     String concorrente;
    Scanner scanner = new Scanner(System.in);
    System.out.print("Inserisci il numero del concorrente ");
    i = scanner.nextInt();
    System.out.print("inserisci Cognome e nome ");
    concorrente = scanner.nextLine();
    System.out.print("\nNumero partecipante:\t
"+"Concorrente\n"+i);
    System.out.println("\t"+concorrente);
    scanner.close();
  }
}
```



Quindi le due istruzioni vengono eseguite entrambe.

Per risolvere il problema si deve svuotare il buffer questo lo si fa inserendo dopo l'istruzione per la lettura di un numero un'istruzione nextLine() che va a eliminare il fine riga.

Codice

• **boolean hasNextInt():** restituisce vero se il prossimo blocco può essere interpretato come un numero intero, falso altrimenti.

boolean hasNextDouble(): restituisce vero se il prossimo blocco può essere interpretato come un numero reale,falso altrimenti.

• **boolean hasNextLine():** restituisce vero se in input è disponibile una ulteriore riga, falso altrimenti.

boolean hasNext(): restituisce vero se in input è disponibile un ulteriore blocco, falso altrimenti.

Questi metodi vengono utilizzati per controllare l'input.

Esercizi

1 Scrivere un programmaSommaApprossimata che chiede all'utente d'inserire due numeri con la virgola, li somma e poi stampa il risultato come numero intero.

2 Scrivere il programmaAreaTriangolo assumendo le seguenti dichiarazioni di variabili:

int base , altezza; double area;

3 Scrivere l'algoritmo che, ricevuto in input un orario attraverso le sue tre componenti (ore, minuti e secondi), ne calcoli il valore totale in secondi.

4 Scrivere un programma che dato un numero di secondi calcolare da quante ore minuti e secondi è composto.

Questo problema è l'inverso del precedente

5 File di posti al Cinema

Scrivi un programma che dato:

a) Un numero totale di Persone

b) Il numero di posti presenti in ogni fila

Restituisca in output:

- Il numero di file

- Nel caso l'ultima fila risulti incompleta indicare il numero di persone mancanti per completarla.

Strutture di controllo del flusso

Sequenza

Tutte le istruzioni che compongono il codice sorgente vengono eseguite a run-time in ordine sequenziale, dall'alto in basso e da sinistra a destra.

Le istruzioni di controllo del flusso, consentono di modificare il normale flusso di esecuzione del programma.

Esercizi:

1 Scrivere un algoritmo che calcoli l'area di un trapezio, note le misure delle basi e dell'altezza.

2 Scrivere l'algoritmo che, dati due numeri interi x e y, calcoli il risultato e il resto della divisione intera tra x e y.

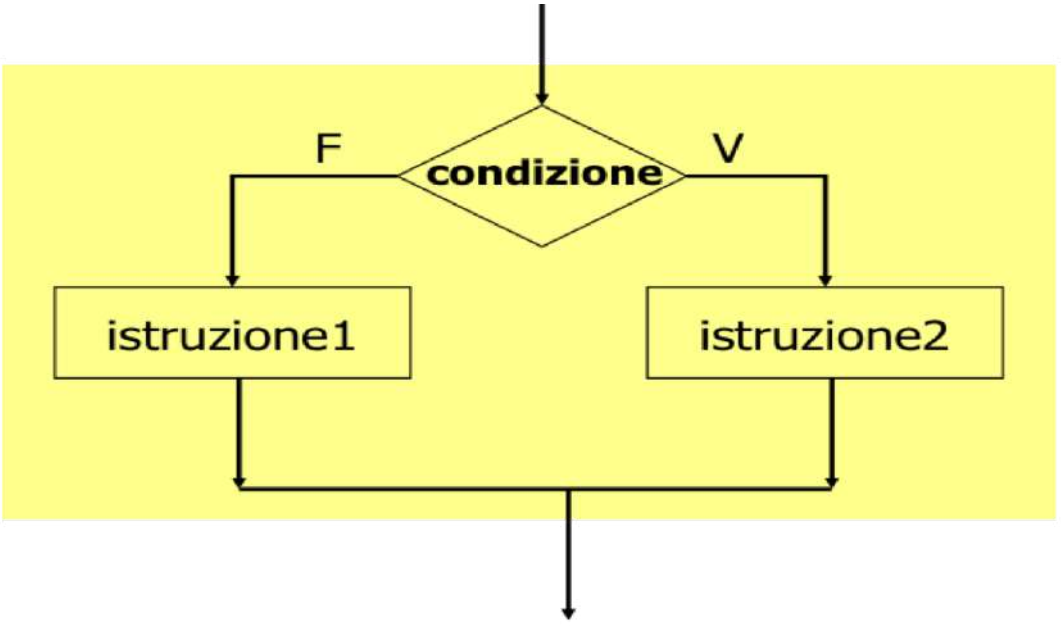
3 Scrivere un algoritmo che, date le età di tre persone, calcoli l'età media.

4 Scrivere un algoritmo che, dato il prezzo di un prodotto, calcoli il prezzo scontato del 20%.

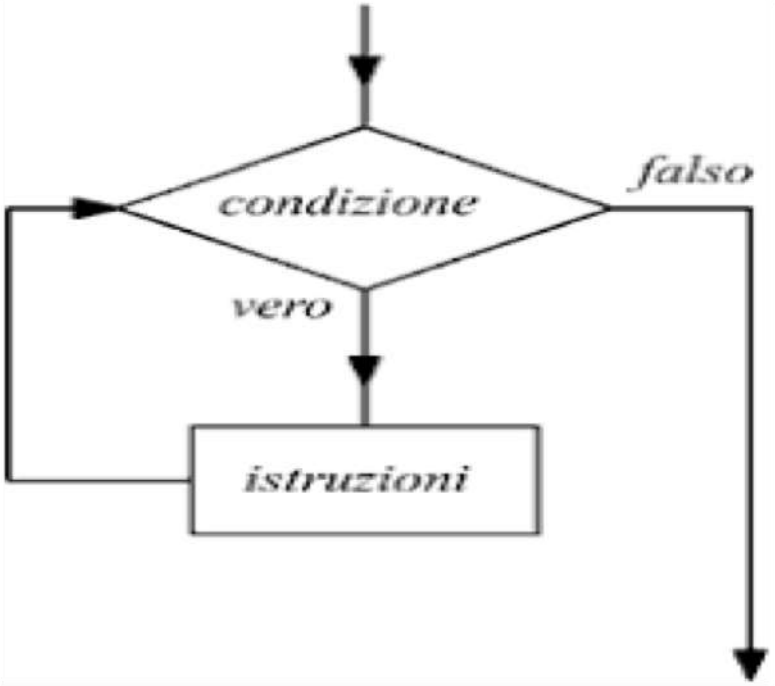
5 Scrivere un algoritmo che, lette le coordinate di due punti del piano, calcoli la distanza tra essi.

Il flusso di esecuzione può essere alterato attraverso tre categorie d'istruzioni:

istruzioni di salto condizionale: sono le istruzioni if-else, switch, case. La modifica del flusso avviene a seconda che si verifichi o meno una determinata condizione



istruzioni iterative: for, do, while. La modifica del flusso avviene ripetendo una sequenza di istruzioni finché è valida una determinata condizione. Supponendo di avere un blocco di 10 istruzioni che vengono ripetute finché si verifica una determinata condizione, il flusso eseguirà le istruzioni del blocco dalla 1 alla 10, dopodiché anziché eseguire l'istruzione 11, ritornerà ad eseguire l'istruzione 1 del blocco; questo finché la condizione sarà valida.



istruzioni di salto incondizionato: break, continue, return. Queste istruzioni determinano il salto del codice "forzato" ad una istruzione differente dalla successiva nell'ordine sequenziale

Inoltre esistono le istruzioni di gestione delle eccezioni: **try-catch-finally**. Queste istruzioni verranno approfondite più avanti nel corso. Per ora possiamo iniziare a dire che hanno un comportamento simile alle istruzioni condizionali. Qui però la condizione che determina il salto nel flusso di codice è determinata dal verificarsi di un errore. Ovvero, se si verifica un errore (in Java eccezione o E**xception**) il codice "salta" l'ordine sequenziale. Il significato di try-catch (prova e cattura) lo potremmo riassumere così: "prova (try) ad eseguire questo blocco di istruzioni, se si verificasse un'eccezione (errore), cattura (catch) il flusso ed esegui quest'altro blocco di istruzioni"

Struttura di selezione singola if

La prima e più semplice struttura di controllo è quella definita di selezione singola if, con cui si valuta se un'espressione è vera o falsa.

```
if (espressione) { istruzioni; }
```

dove:

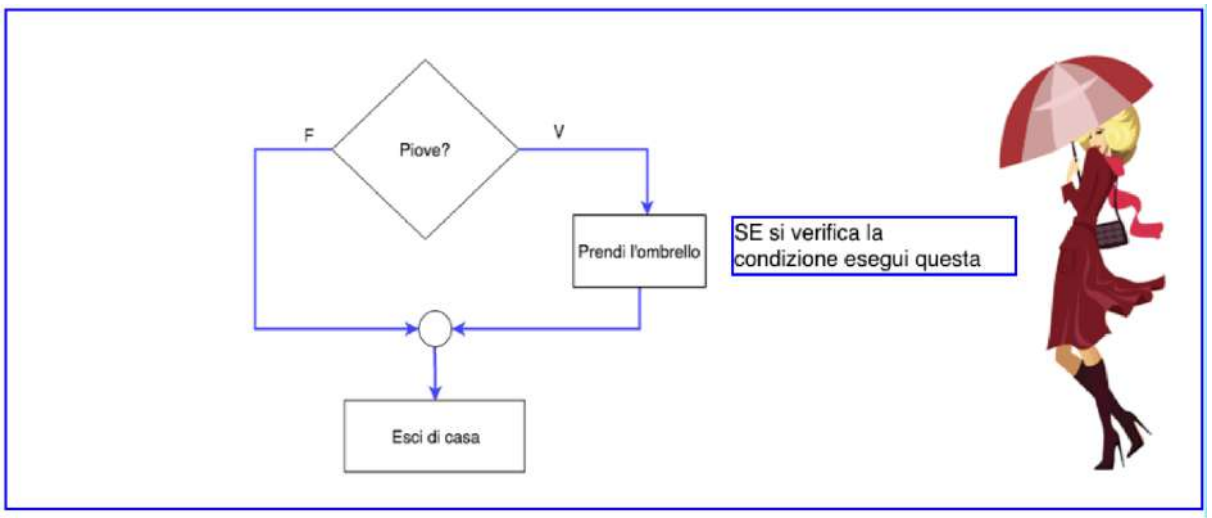
- All'interno delle parentesi tonde ( ) è presente un'espressione da valutare.

- Tra le parentesi graffe { } vanno poste le istruzioni che vanno eseguite se l'espressione risulterà vera.
- Se l'espressione risulterà falsa, le istruzioni non saranno eseguite e il flusso esecutivo del programma proseguirà alla prima istruzione posta subito dopo la parentesi graffa di chiusura.

Es:

Prima di uscire di casa si valutano le condizioni del tempo:

Se piove si prende l'ombrello



Si esce di casa

```
boolean piove=true;
if(piove){
System.out.println("Prendi l'ombrello");
}
System.out.println("Esci di casa");
```

```
Output - JavaApplication28 (run)
run:
Prendi l'ombrello
Esci di casa
BUILD SUCCESSFUL (total time: 0 seconds)
```

Codice

Esempio di Selezione

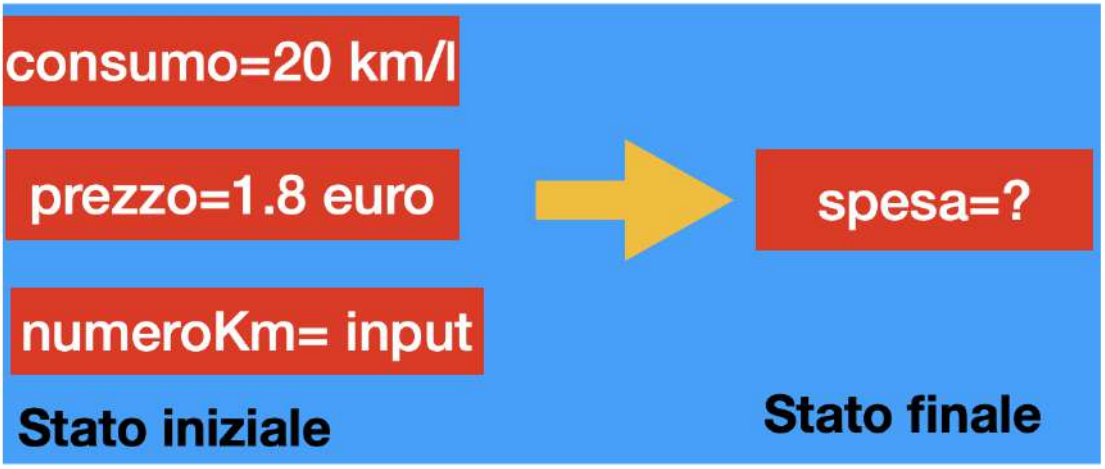
Un'automobile percorre 20 km con un litro di benzina. Calcolare la spesa necessaria a percorrere n km. Se la spesa è maggiore di €100, applicare uno sconto del 5%.

Dati:

Siccome al prezzo viene applicato uno sconto se la spesa è superiore ai 100 euro spesa>100 verifichiamo la condizione

se la risposta è vera calcoliamo il prezzo scontato

```
spesa=spesa - spesa*5/100
```




```
consumo =20 -> costante
prezzo=1.8 -> costante
numeroKm = n -> dato d'input
litriConsumati= numeroKm/consumo -> logica
spesa= litriConsumati*prezzo
```

```
import java.util.Scanner;
class Main {
```

```
  public static void main(String[] args) {
    final int consumo =20;
final double prezzo=1.8;
Scanner tastiera=new Scanner(System.in);
System.out.println("Inserisci il numero di Km");
int numeroKm = tastiera.nextInt();
double litriConsumati= (double)numeroKm/consumo;
double spesa= litriConsumati*prezzo;
if (spesa>100)
    spesa-=spesa*5/100;
System.out.println("La spesa da sostenere è: "+spesa);
  }
}
```

Codice

Nella condizione si è fatto uso dell'operatore relazionale:

>

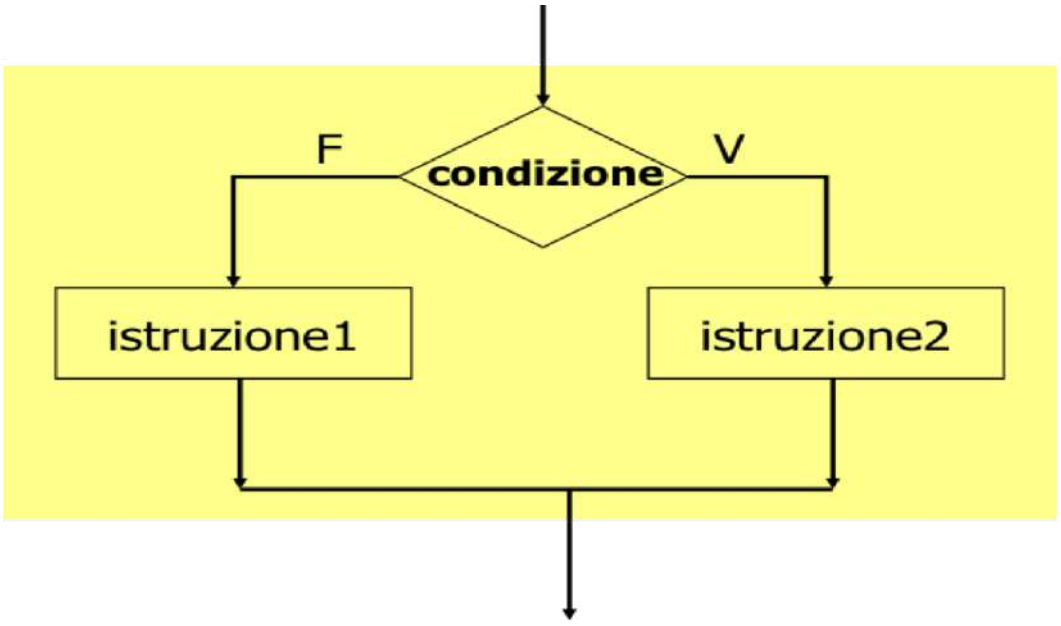
che permette di confrontare due espressioni e restituisce true se il primo operando è maggiore del secondo, oppure false se non lo è.

Oltre a questo java dispone di altri operatori per confrontare espressioni essi sono:

Operatore	Simbolo	Applicabilità
Uguale a	==	Tutti i tipi
Diverso da	!=	Tutti i tipi
Maggiore	>	Solo i tipi numerici
Minore	<	Solo i tipi numerici
Maggiore o uguale	>=	Solo i tipi numerici
Minore o uguale	<=	Solo i tipi numerici

Il risultato delle operazioni basate su operatori relazionali è sempre un valore boolean, ovvero true o false.

Struttura di selezione doppia if/else



La struttura di selezione doppia if-else consente di eseguire delle istruzioni se l'espressione è valutata vera oppure altre istruzioni se l'espressione è valutata falsa. Tali istruzioni sono mutuamente esclusive.

if (espressione) { istruzione1; }

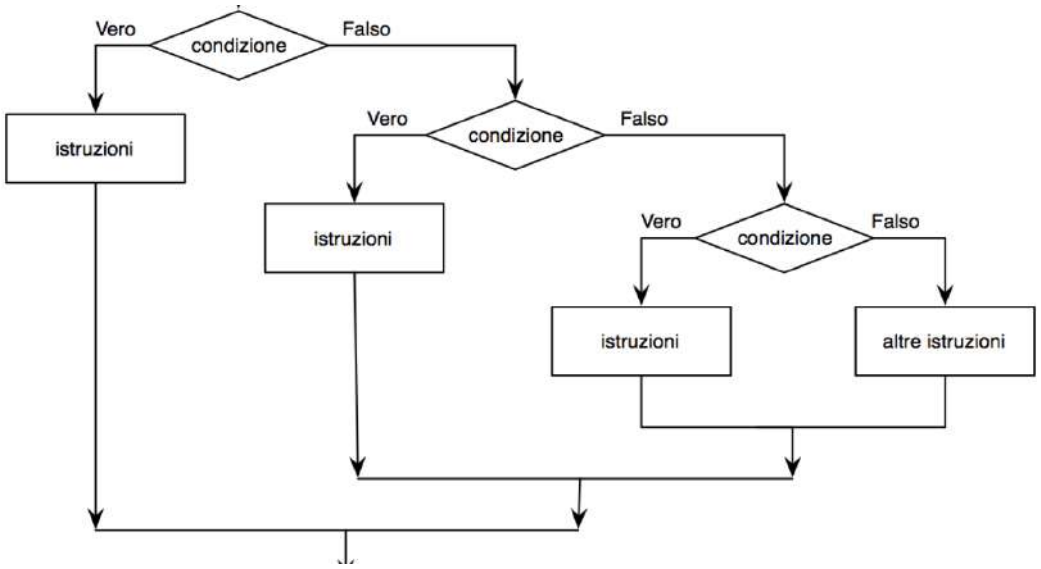
else { istruzione2; }

Oltre al consueto costrutto **if(se)**, è presente anche un blocco, definito costrutto **else(altrimenti),** che eseguirà le istruzioni poste al suo interno solamente se l'espressione sarà falsa.

```
*/
public class Condizione_If_else {

    public static void main(String[] args) {
        int eta = 15;
        if (eta < 18) {
            System.out.println("Funzione vietata: hai meno di 18 anni");
        } else {
            System.out.println("Accesso consentito: hai più di 18 anni");
        }
    }
}
```

Codice

La struttura if/else può essere costruita con più livelli di annidamento.

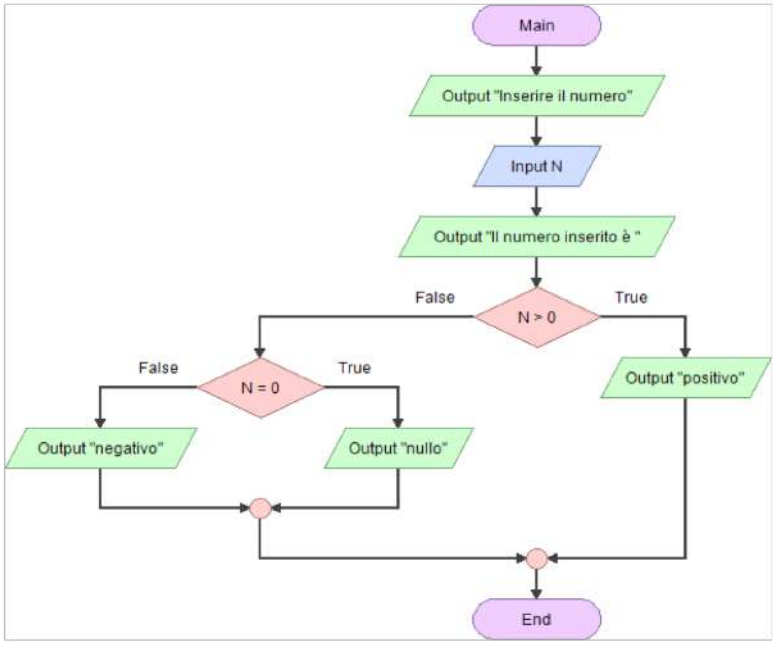


La logica è :

se è vera una delle condizioni, allora vengono eseguite le istruzioni corrispondenti e il programma salta al di fuori di tutti gli altri if/else;

se nessuna condizione è vera, allora il programma le salta tutte.

Algoritmo che permette di verificare se un numero inserito è positivo, negativo o nullo.

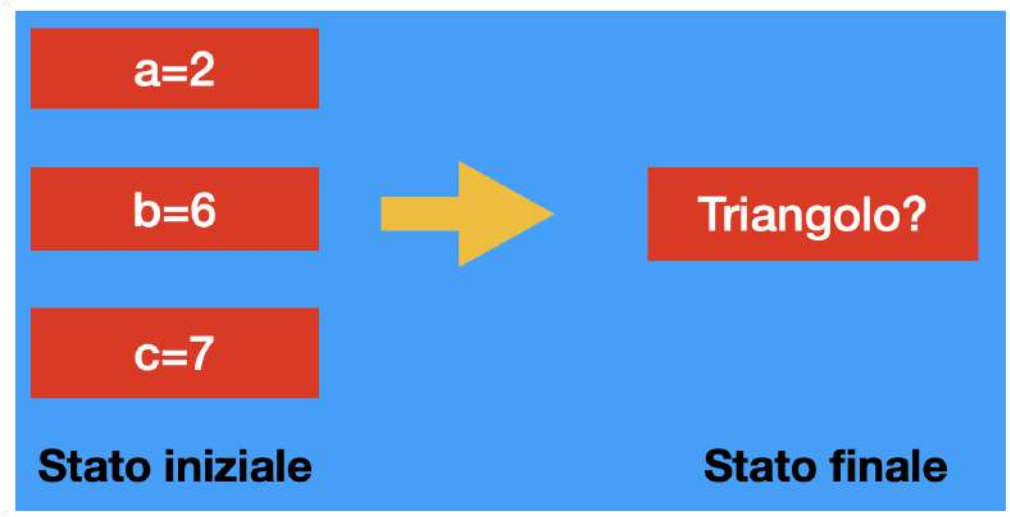


Codice

Lati di un triangolo

Stabilire se tre segmenti possono costituire i lati di un triangolo

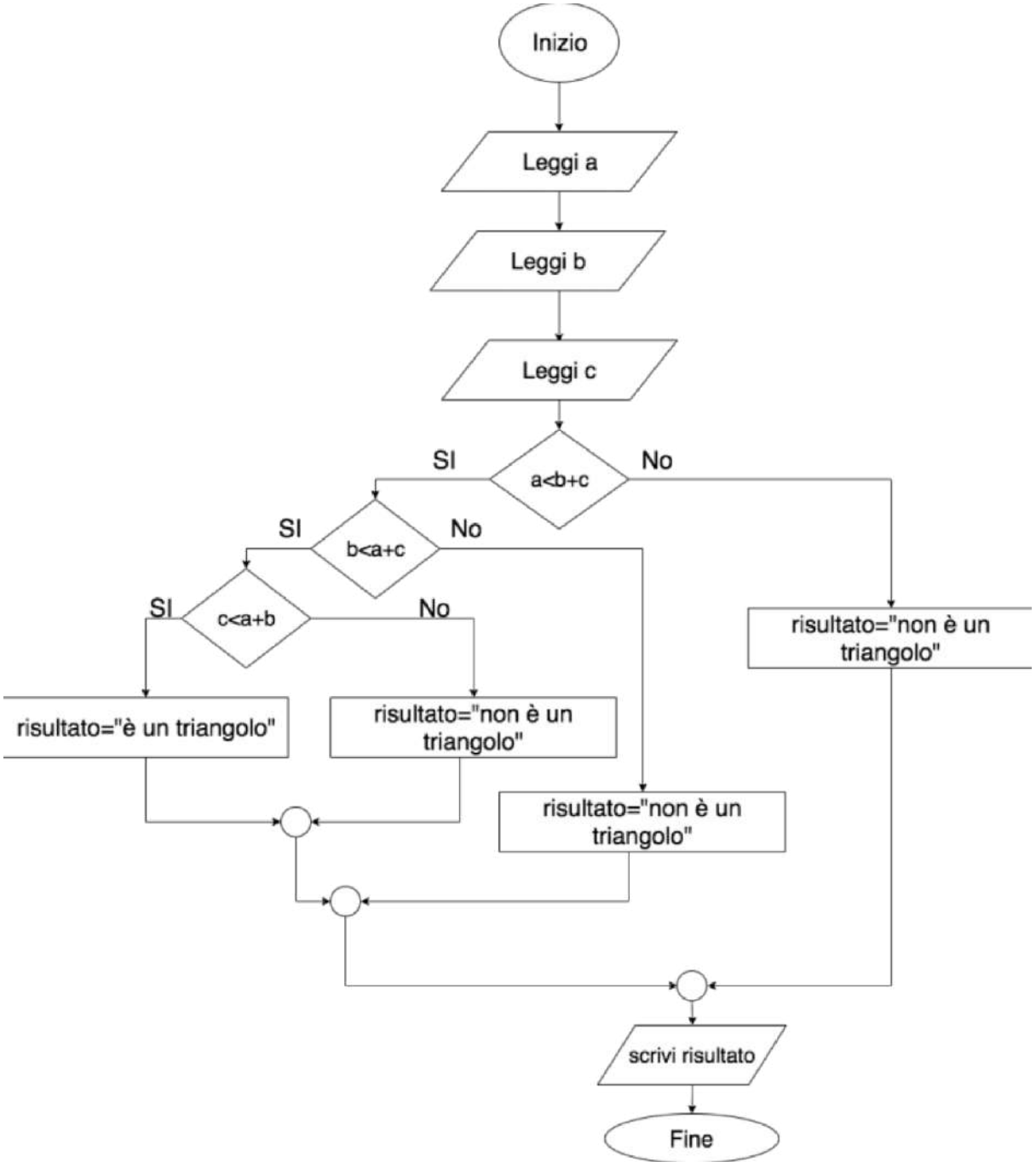
Dati



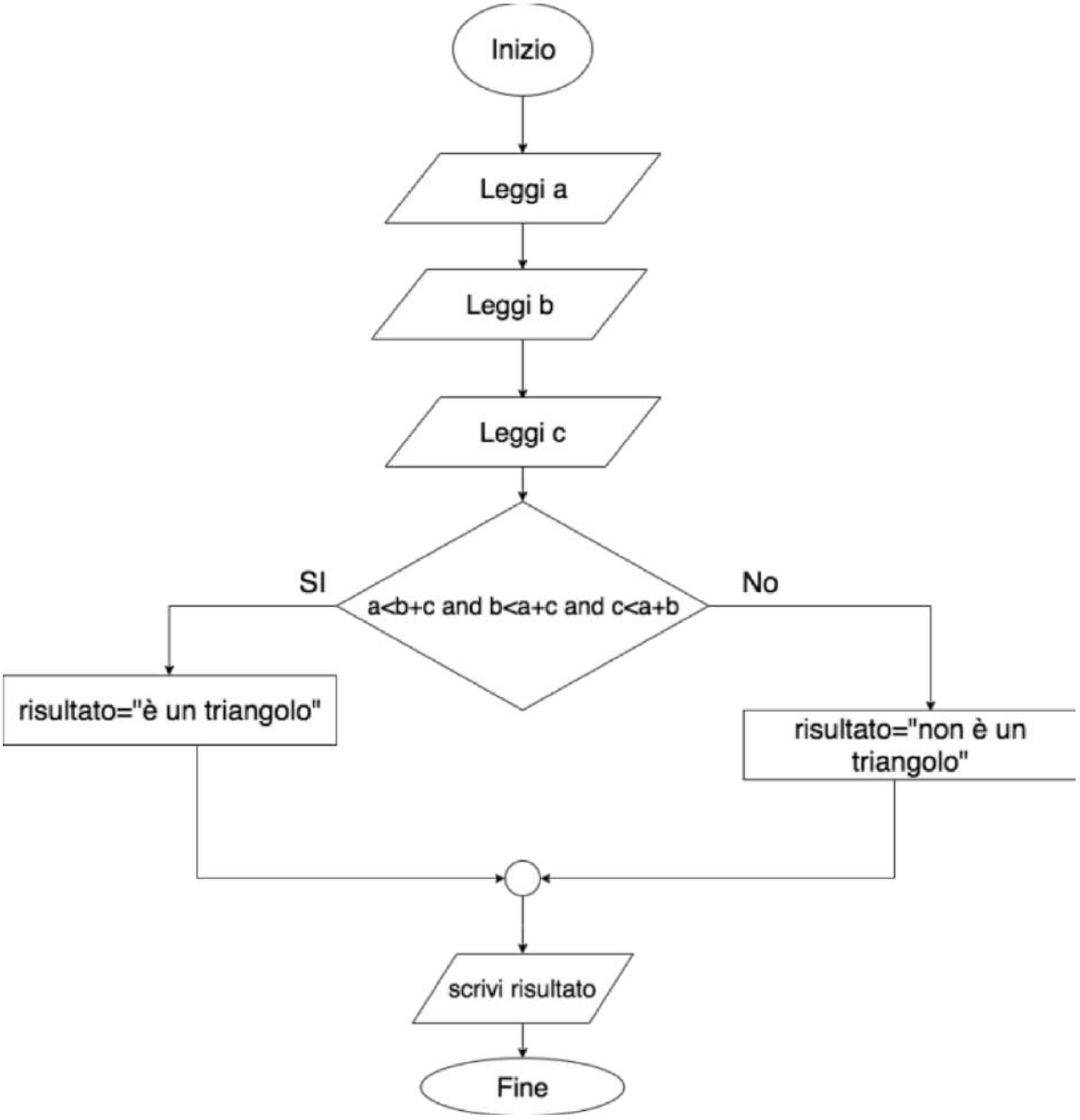
In un triangolo, ciascun lato deve essere minore della somma degli altri due.

$$A < B + C \text{ e } B < A + C \text{ e } C < A + B$$

Algoritmo



oppure in modo più semplice



usando gli:

Operatori booleani

Per verificare tre condizioni contemporaneamente abbiamo usato l'operatore **logico AND** che opera su due operandi di tipo boolean e restituisce true se i due operandi sono veri. Gli altri operatori logici disponibili sono:

Operatore	Descrizione
&&	"and logico" Restituisce true se sono vere le espressioni a destra e sinistra dell'uguale.

||	"or logico" Restituisce true se una una delle due espressione è vera
!	"not" nega: l'espressione – se è false La cambia in true – se è true la cambia in false

Quando si esegue un'operazione AND, se il primo operando è falso, il risultato è falso indipendentemente dal valore del secondo operando.

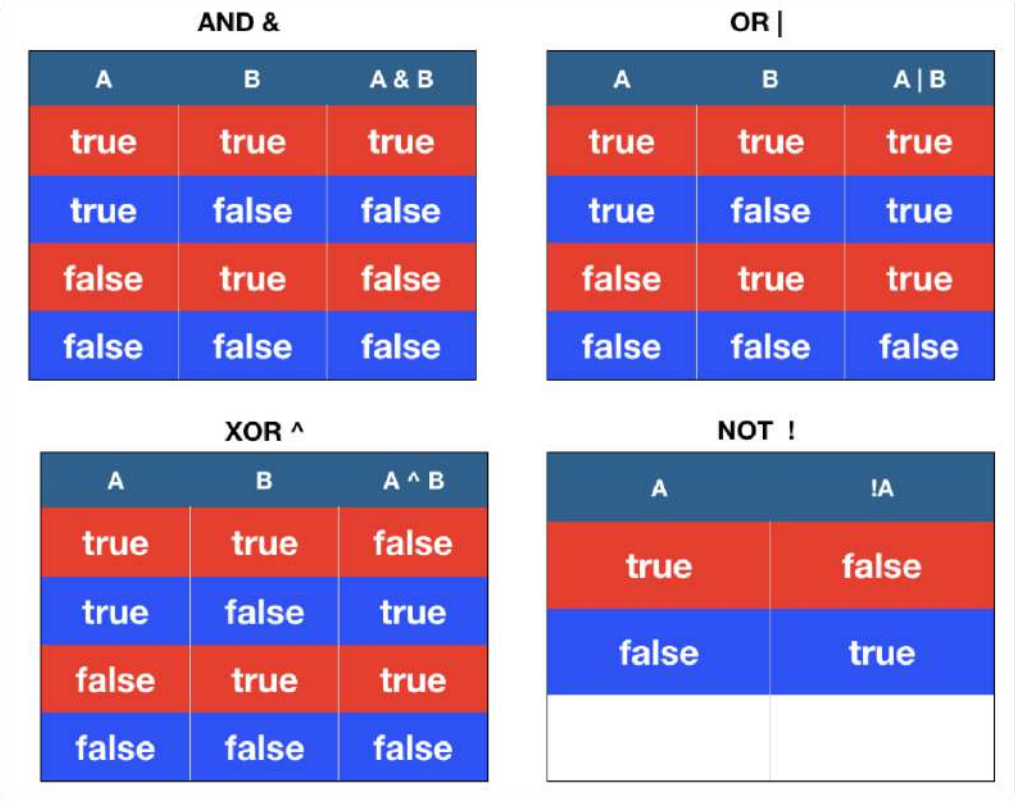
In un'operazione OR, se il primo operando è true, il risultato dell'operazione è true indipendentemente dal valore del secondo operando. Pertanto, in questi due casi non è necessario valutare il secondo operando. Non valutando il secondo operando, il tempo di esecuzione è minore e il codice risulta più efficiente. Oltre a questi operatori detti di cortocircuito javascript possiede degli operatori analoghi che operano sempre su ogni bit.

Operatori bitwise ("bit a bit")

Gli operatori sui bit od operatori bitwise trattano i valori numerici come sequenze di bit applicando le relative operazioni. Gli operatori previsti in Java Sono:

Operatore	Descrizione
&	And per la congiunzione logica bit a bit
|	OR per la disgiunzione logica bit a bit
^	Xor per l'Or esclusivo bit a bit (
~	per il complemento a uno dell'operatore, nega cioè bit a bit il dato
>>	Spostamento a destra con segno (shift) del primo operando di tanti bit quanti indicati dal secondo e riempi gli spazi vuoti con il bit di segno
>>>	Spostamento a destra senza segno (shift) del primo operando di tanti bit quanti indicati dal secondo e riempie gli spazi vuoti con zero
<<	per lo scorrimento a sinistra(shift)

Tavole della verità



Esercizi

1. Scrivere un programma che, letto in input un valore numerico, dica se è positivo o negativo.
2. Scrivere un programma che, dato un numero intero in input, visualizza il suo doppio se è pari, il triplo se è dispari.
3. Scrivere un programma che, dati due numeri, calcoli la somma se sono entrambi positivi, il prodotto altrimenti.
4. Scrivere un programma che, dato il prezzo di un prodotto, applichi uno sconto del 12% se il prezzo è inferiore a € 30,00, del 25% altrimenti.
5. Scrivere un programma che, dati base e altezza di un triangolo, calcoli l'area se sono entrambi positivi, oppure stampi il messaggio "Valori di input errati".
6. Scrivere un programma che, presi in input gli estremi a e b di un intervallo e un valore x, visualizzi il messaggio "Il valore è interno all'intervallo" se $a \leq x \leq b$, altrimenti "Il valore è esterno all'intervallo". Scrivere un algoritmo che, letti in input due numeri interi, verifichi se il primo è multiplo del secondo.
7. Scrivere un programma che, dato un numero intero in input, visualizza il suo doppio se è pari, il triplo se è dispari.
8. Scrivere un programma che, dati due numeri, calcoli la somma se sono entrambi positivi, altrimenti calcoli la differenza tra il maggiore e il minore.
9. Scrivere un programma che, dati i lati di un triangolo, calcoli il perimetro se questi tre valori possono essere i lati di un triangolo, altrimenti stampi il messaggio "Valori d'input errati".

10. Scrivere un programma che, preso in input un voto, dica se è corretto (compreso tra 1 e 10).
11. Scrivere un programma che, dato il consumo di acqua di un utente, espresso in m3, calcoli l'importo della bolletta, sapendo che ogni bolletta comprende una quota fissa di 20 euro e una quota variabile di 2,50 euro/m3 per i primi 100 metri cubi d'acqua, di euro 4,00/m3 per i metri cubi in eccesso.

12. Scrivere un programma che determini il prezzo d'ingresso al cinema considerando che clienti con un'età compresa tra 18 e 65 anni pagano il prezzo pieno, mentre per età diverse da queste il prezzo è la metà del prezzo pieno. Si effettuino i dovuti controlli sui valori immessi in input.

13. Scrivere un programma per controllare la correttezza di una data ricevuta in ingresso attraverso tre diversi input: giorno, mese e anno. Tutti gli input devono essere numerici. Si effettuino i dovuti controlli sui valori immessi in input.

14. Progettare un programma che dati in input due orari della stessa giornata visualizzi la differenza in ore, minuti e secondi.
    Esempio:

```
Ora 1: 10
Minuti 1: 14
Secondi 1: 18
Ora 2: 5
Minuti 2: 9
Secondi 2: 2
Differenza: 5:5:16
```

15. Realizzare un programma che fornite due date in input, visualizzi il numero di giorni che intercorrono tra loro. Ad esempio:

```
Data antecedente: 10/08/2018 Data successiva: 08/01/2022

Giorni che intercorrono fra le due date: 1243
```

    Per ogni data prevedere l'inserimento distinto di giorno, mese, anno. Non è previsto che la prima data inserita debba necessariamente essere l'antecedente.
    Non si considerino gli anni bisestili, i mesi si suppongono tutti di 30 giorni e gli anni compresi tra il 1970 e l'anno corrente.
    Si controlli la correttezza delle date inserite.

16. Scrivere un programma che determina se un anno inserito da tastiera è bisestile.

17. Scrivere un programma che, letto in input un valore numerico, dica se è positivo, negativo o nullo.

18. Scrivere un programma che, presi in input 3 numeri, visualizzi il valore maggiore.
19. Scrivere un programma per visualizzare in ordine crescente tre valori numerici ricevuti in input.
20. Scrivere un programma che legga da tastiera i valori delle lunghezze dei tre lati di un triangolo e determini se il triangolo è equilatero, isoscele, rettangolo o scaleno.
21. Si realizzi un programma che, dato il prezzo di un prodotto e la quantità acquistata, calcoli il prezzo totale, tenendo conto che il venditore applica uno sconto del 10% se si acquistano più di 5 pezzi, del 15% se si acquistano più di 10 pezzi o del 20% se si acquistano più di 20 pezzi.
22. Progettare un programma che chieda in input tre valori interi compresi nell'intervallo [10-1000]. L'algoritmo deve verificare se almeno due dei tre numeri condividono la cifra meno significativa, e in questo caso dovrà visualizzare la cifra condivisa. Nel caso non ci sia alcun numero che condivide la cifra meno significativa si dovrà visualizzare un messaggio appropriato.
    Esempi:
    Input: 41, 22, 71 → Output: 1
    Input: 23, 32, 42 → Output: 2
    Input: 57, 37, 17 → Output: 7
    Input: 14, 53, 98 → Output: "Nessun valore condivide l'ultima cifra"
    Input: 9, 99, 999 → Output: "Una delle cifre non si trova nell'intervallo [10-1000]"

23. Lo spazio espresso in metri di frenata di un'automobile è stimato mediante la seguente formula, supponendo che il tempo di reazione del guidatore sia pari ad 1 secondo: spazio = velocita2 / (250 * COEFFICIENTE)
    dove velocita è la velocità in km/h, e COEFFICIENTE è un coefficiente relativo alle condizioni stradali, come indicato dalla seguente tabella:

Condizioni stradali	COEFFICIENTE
Asfalto ruvido	0,6
Asfalto liscio	0,5
Asfalto bagnato	0,4
Asfalto ghiacciato	0,1

Progettare un programmache calcola lo spazio di frenata a partire dalla velocità data in input e dal coefficiente sulle condizioni stradali. Usare lo switch-case per risolvere l'algoritmo.

24. Gli abbonamenti della GTT di Torino possono riguardare zone diverse (urbano - 1, suburbano - 2, urbano+suburbano - 3) e possono essere settimanali (S), mensili (M) o annuali (A). I costi sono quelli indicati nella seguente tabella:

Zona\Durata	S	M	A
1	12,00 €	38,00 €	310,00 €
2	9,80 €	35,50 €	319,50 €
3	15,70 €	56,50 €	508,50€

Gli abbonamenti per gli studenti hanno una riduzione del 50%.
Progettare un programma che calcola il costo dell'abbonamento a partire dal tipo, dalla durata e se ad acquistarlo è uno studente o meno.

25. Progettare un programma che accetti tre valori numerici interi e li visualizzi in ordine crescente.

26. Progettare un programma che richieda l'inserimento di una temperatura in gradi Celsius e visualizzi un messaggio come indicato nella seguente tabella:

Temperatura (t)	Messaggio
t >= 30	"Molto caldo"
20 <= t < 30	"Caldo"
10 <= t < 20	"Ideale"
0 <= t < 10	"Freddo"
t < 0	"Molto freddo"

27. Progettare un programma che calcoli le radici nel campo reale di un'equazione di secondo grado, dati in input i coefficienti a e b e il termine noto c.

28. Progettare un programma che visualizzi il maggiore fra tre numeri dati in input.

Dati

Input	Output	Lavoro

a	max	
b		
c		

Relazione tra ingresso e uscita

se a>b trovare il max tra a e c

altrimenti trovare il max tra b e c

se a>b
   se a>c

      max = a

     altrimenti il max è C

altrimenti

se b>c

allora il max è b

altrimenti il max è c

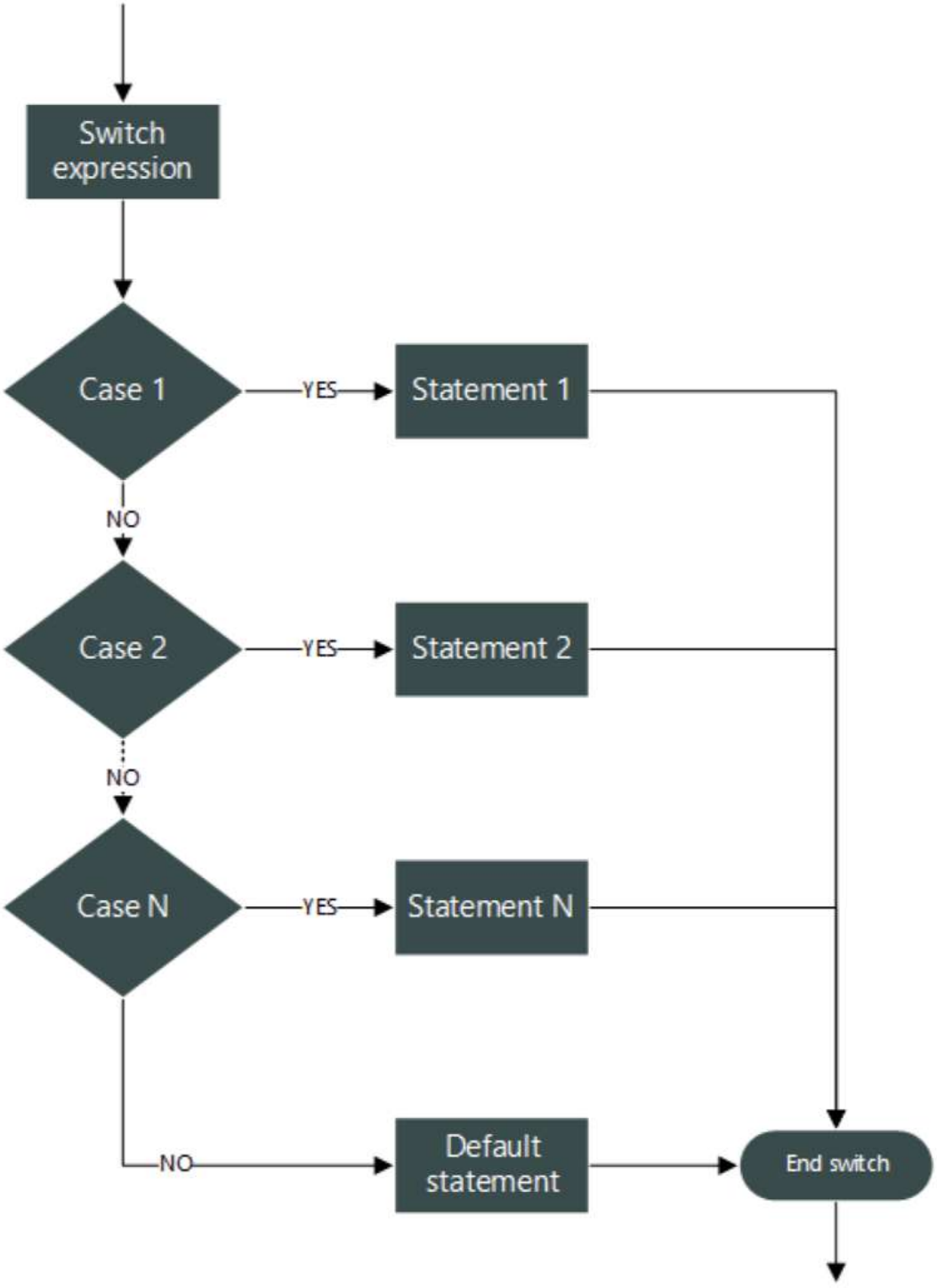
29.

Struttura di selezione multipla switch/case

La struttura di selezione multipla switch/case consente di eseguire le istruzioni di un blocco di codice, identificato da una particolare etichetta, se il valore costante che questa rappresenta è uguale al valore dell'espressione da valutare, che può essere di tipo byte, short, char, int, String ed enumerativo.

```
switch (expression)
{
```

```
case value1: statements;
break;
case value2:
statements;
break;
... [default]: statements;
}
```



La keyword switch è seguita dalle parentesi tonde ( ) che racchiudono l'espressione da valutare.

Tra le parentesi graffe { }, si ha un insieme di etichette (**keyword case**). Il valore dell'etichette deve essere una costante. Chiude il blocco switch una clausola non obbligatoria (keyword default), che esegue delle istruzioni se tutti i blocchi case non hanno un valore corrispondente al valore dell'espressione switch.

L'istruzione break interrompe il flusso esecutivo del codice facendolo uscire dalla struttura switch, altrimenti viene valutato anche il caso successivo.

```
class ProvaSwitch {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.println("inserisci il giorno da 1 a 7");
    int giornoDellaSettimana = in.nextInt();
    String nomeGiorno;
    switch (giornoDellaSettimana) {
    case 1:
      nomeGiorno = "Lunedì";
      break;
    case 2:
      nomeGiorno = "Martedì";
      break;
    case 3:
      nomeGiorno = "Mercoledì";
      break;
    case 4:
      nomeGiorno = "Giovedì";
      break;
    case 5:
      nomeGiorno = "Venerdì";
      break;
    case 6:
      nomeGiorno = "Sabato";
      break;
    case 7:
      nomeGiorno = "Domenica";
      break;
    default:
      nomeGiorno = "non valido";
    }
    System.out.println("il giorno della settimana è: " +
nomeGiorno);
  }
}
```

Codice

Multi-case

Se si omette il break, vengono eseguiti tutti i blocchi a partire da quello per cui l'etichetta è uguale all'espressione.

```
int valore = 0;
switch (valore) {
 case -1:
 System.out.println("-1 negativo");
 break;
 case 0: // valore è 0 quindi il criterio è verificato; questo
blocco verrà eseguito
 System.out.println("valore verificato" +valore);
 // NOTA: il break non è stato inserito
 case 1: // manca il comando break in 'case 0:' quindi anche
questo blocco sarà eseguito
System.out.println("case 1 eseguito per mancanza del break");
 break; //incontra il break e non proseguirà in 'case 2:'
 case 2:
 System.out.println(2);
 break;
 default:
System.out.println('default');
}
```

Si sfrutta questa proprietà del break per fare dei controlli multipli. Realizzare un programma che visualizzi il numero di giorni in base al mese e all'anno inseriti:

Siccome più mesi hanno trentuno giorni si raggruppano in una sequenza di case vuote, l'ultima case fissa il numero di giorni e è chiusa con il break. La stessa cosa per i mesi di trenta giorni.

```
class SwitchMultiCase {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
```

```
    System.out.println("inserisci mese");
    int mese = in.nextInt();
    System.out.println("inserisci anno");
    int mese = in.nextInt();
    int numeroGiorni = 0;
    switch (mese) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
      numeroGiorni = 31;
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      numeroGiorni = 30;
      break;
    case 2:
      if (((anno % 4 == 0) && !(anno % 100 == 0)) || (anno % 400
== 0))
        numeroGiorni = 29;
      else
        numeroGiorni = 28;
      break;
    default:
      numeroGiorni = -1;
      break;
    }
    System.out.println("Numero di giorni = " + numeroGiorni);
  }
}
```

Nel case due si verifica se l'anno è bisestile.

Esercizi:

1. Scrivere un programma che, preso in input un valore compreso tra 1 e 12, visualizzi il nome del mese corrispondente.
2. Scrivere un programma per convertire un numero intero N compreso tra 1 e 365, fornito in input, nel giorno e mese corrispondente. Si consideri un anno non bisestile.
3. Il biglietto d'ingresso a un teatro ha le seguenti tariffe. Per i bambini di età inferiore a 6 anni l'ingresso è gratuito, per gli studenti 8 euro, per i pensionati 10 euro, per tutti gli altri 15 euro. Creare un programma in cui l'utente inserisce un numero tra 1 e 4 e viene comunicato il prezzo relativo all'opzione scelta. Se il numero non è un'opzione valida viene mostrato un messaggio di errore.
4. Progettare un programma che prenda in input un numero intero, sia negativo, sia positivo, e visualizzi il numero come parola se è compreso nell'intervallo [-9, +9], mentre visualizzi la scritta "Other" diversamente. Si provi ad usare il costrutto switch-case.

Operatore ternario ?:

Oltre ai costrutti if ... else e switch è possibile modificare il flusso di un programma (a seconda che si verifichi una condizione) tramite l'operatore ternario. La cui sintassi è:

condizione ? istruzione1; : istruzione2;

Che si traduce come: se la condizione è vera esegui l'istruzione 1, in caso contrario esegui l'istruzione 2.

Questo operatore è comodo se si devono affrontare condizioni semplici del tipo if... else mentre è meno adatto a controlli complessi, anche perché si scrive su una linea singola e non si può andare a capo.

L'operatore ternario può essere usato per assegnare un valore a una variabile usando la forma:

<variabile> = <condizione>?<valore1>:<valore2>

Alla variabile viene assegnato direttamente il risultato del controllo ternario.

```
int numero=in.nextInt();
String risultato=numero%2==0 ? "numero pari" : "numero
```

```
dispari";
System.out.println(risultato);
```

è possibile combinare tra loro due o più operatori ternari per gestire situazioni più complesse.

Loop

Capita che per risolvere un problema una o più istruzioni devono essere ripetute più volte. Se si vogliono visualizzare i numeri da 1 a 10 (ma potrebbe essere da 1 a 1000) invece di scrivere le dieci istruzioni:

1. System.out.println(1);
2. …
3. …..
10. System.out.println(10);

Si può usare un LOOP cambiando semplicemente il numero con una variabile che aumenta di uno.

```
numero=1
ripeti per 10 volte
 scrivi(numero)
 numero=numero+1
```

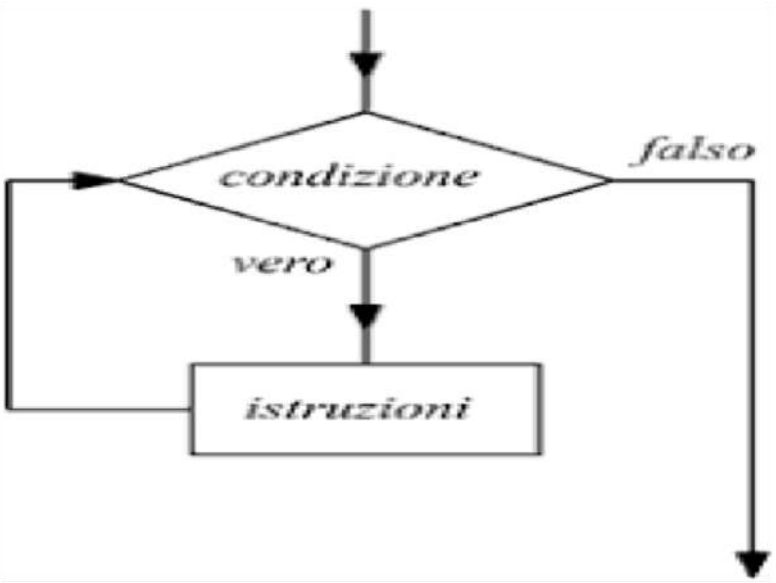
Per fare questo tutti i linguaggi possiedono dei costrutti di tipo loop. Java consente tre strutture iterative:

Struttura di iterazione while

Questo costrutto esegue lo stesso blocco d'istruzioni finché una condizione è vera. La sua sintassi è la seguente:

```
while (condizione) {
```

```
istruzioni;
}
```



Abbiamo:

- **la keyword while, contenente la condizione** da valutare
- un blocco di **codice scritto tra le parentesi graffe { }.**

l'esempio di prima diventa:

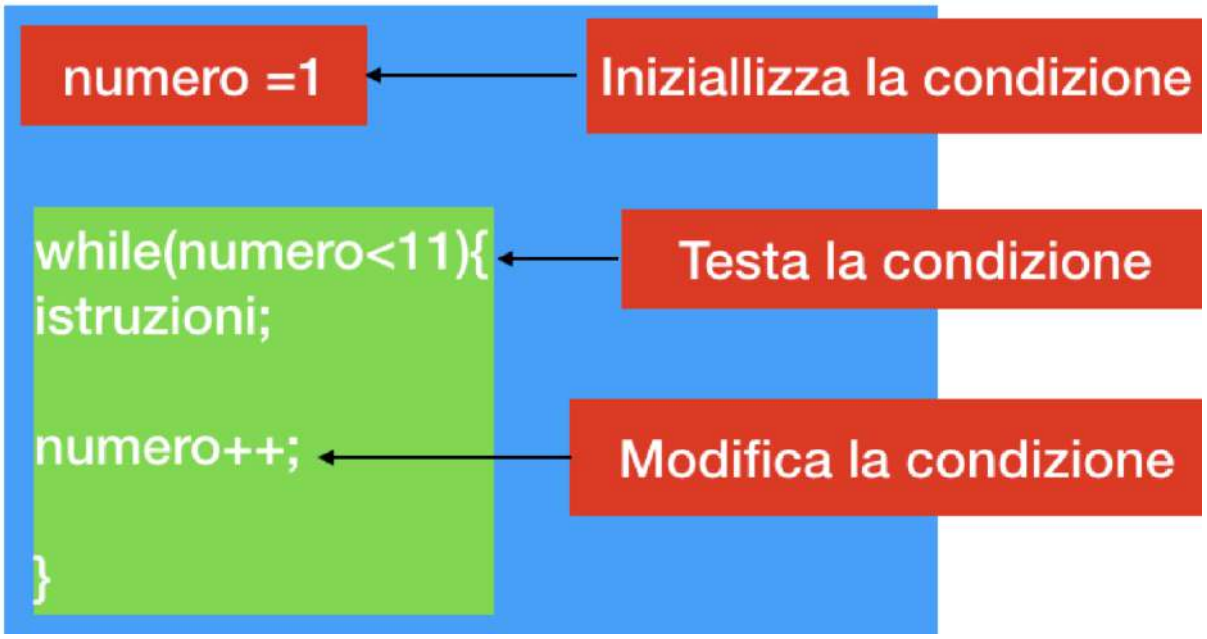
```
public class While {
  public static void main(String[] args) {
    int numero = 1;
    while (numero < 11) {
      system.out.println(numero);
      numero++;
    }
  }
}
```

Il ciclo while si può interpretare in questo modo:

Finché la variabile **numero è minore al valore di 11,** stampare il numero.

L'istruzione numero++ è fondamentale poiché permette di modificare il valore di numero. Se non ci fosse questa istruzione il ciclo while sarebbe infinito, perché la condizione sarebbe sempre vera, dato che la variabile numero è sempre minore di 11.

Lo schema principale è:



Nei cicli spesso si fa uso di due variabili particolari:

1. **contatore:** utilizzata per contare quante volte sono eseguite determinate istruzioni oppure l'intero ciclo. Viene inizializzata (spesso a zero) e incrementata ogni volta di uno e è usata anche come condizione per uscire dal ciclo.
2. **accumulatore:** cioè una variabile nella quale ogni nuovo valore non sostituisce il valore corrente ma si accumula a esso. Si pensi al display della cassa di un supermercato in cui ogni volta che si passa un prodotto il prezzo si somma al precedente.

Esempio di Codice

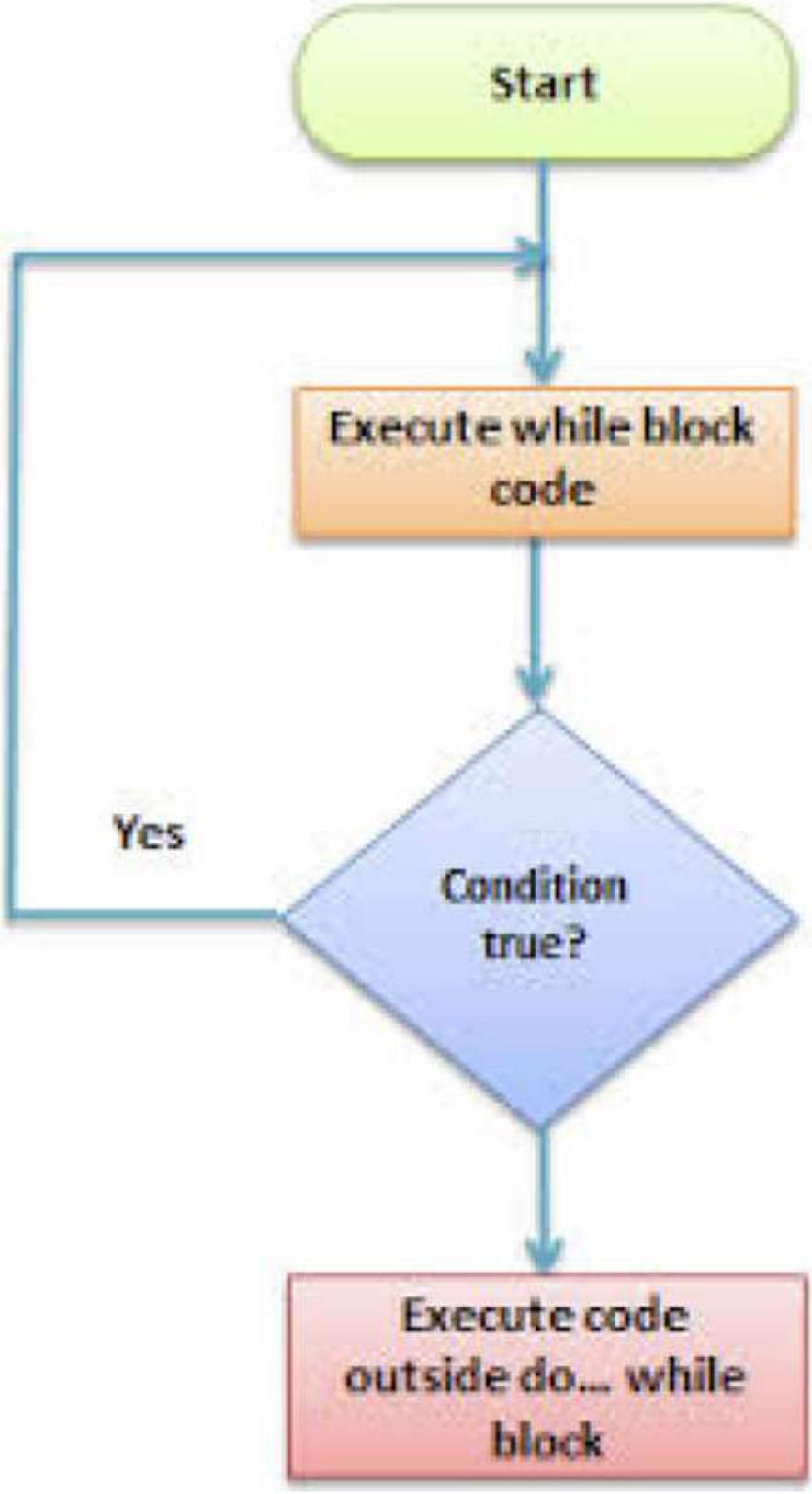
Struttura di iterazione do/while

La struttura do/while consente, come la struttura while, di ripetere un blocco d'istruzioni finché una condizione è vera. Nel do while però le istruzioni vengono eseguite almeno una volta, in quando la condizione viene testata in coda. Sintassi do/while:

```
do {
istruzione;
} while (condizione);
```

dove:

- la keyword do rappresenta l'inizio del ciclo
- Le istruzioni poste tra le parentesi graffe { }
- la keyword while seguita con la condizione da valutare.




```
public class DoWhile {
  public static void main(String[] args) {
    int a = 8;
    System.out.print("a = ");
    do {
```

```
      System.out.print(a-- + " ");
    } while (a >= 0); // finché a >= 0
  }
}
Output.
a = 8 7 6 5 4 3 2 1 0
```

Struttura di iterazione for

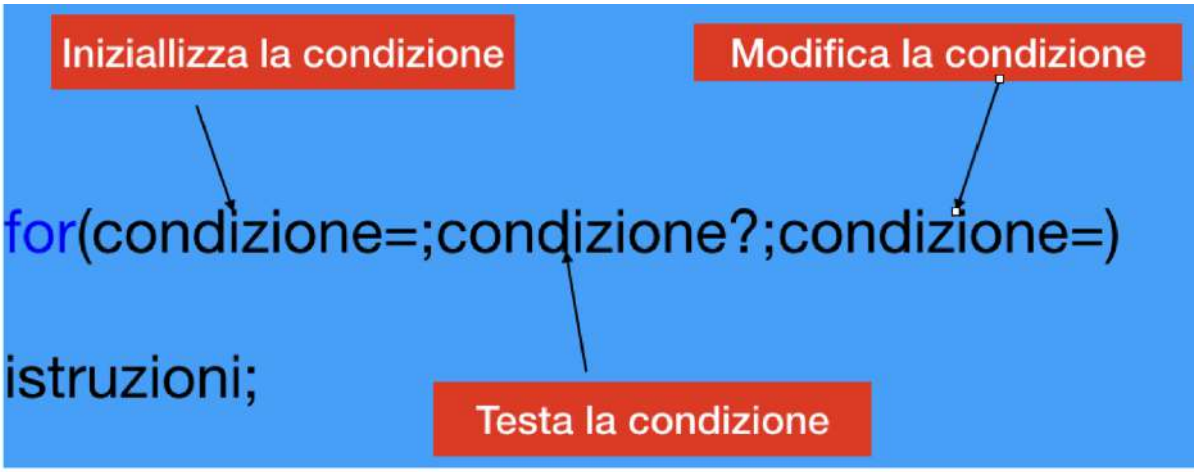
La struttura for consente di ripetere un blocco d'istruzioni finché una condizione è vera. A differenza di while e do/while, for consente di gestire all'interno del suo costrutto delle espressioni aggiuntive con cui:

- Si inizializzano
- Si testano
- Si modificano delle variabili di controllo.

Sintassi for:

for (expression1; expression2; expression3) {

istruzione; }



La keyword for è seguita dalle parentesi tonde ( ) che racchiudono tre espressioni:

1. expression1: inizializza le variabili,
2. expression2: controlla se la condizione è vera
3. expression3: che modifica le variabili di expression2.

Il costrutto termina con le parentesi graffe { } del blocco d'istruzioni che sarà eseguito ciclicamente finché la condizione è vera.

```
public class For {
  public static void main(String[] args) {

    System.out.print("a = ");
    for (int a = 8; a >= 0; a--) // finché a >= 0 esegue il ciclo
    {
      System.out.print(a-- + " ");
    }
  }
}
Output a = 8 7 6 5 4 3 2 1 0
```

Il ciclo for è semplice da usare quando si conosce il numero di volte che il ciclo deve essere ripetuto, infatti fa quasi sempre uso (non è un obbligo) di una variabile contatore:

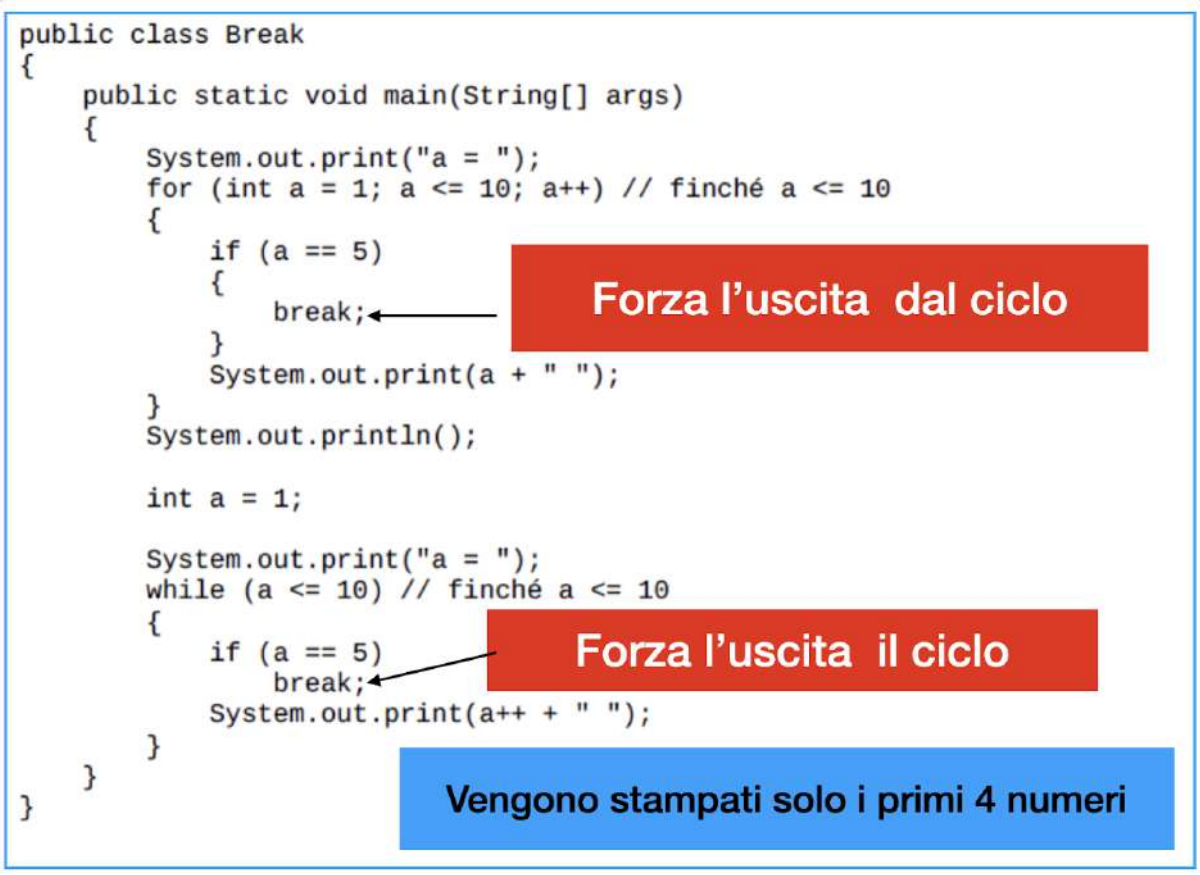
- inizializzata nella prima espressione
- controllata nella seconda
- modificata nella terza

Istruzioni break, continue

Le strutture iterative possono essere alterate durante il loro flusso esecutivo mediante le istruzioni break e continue:

L'istruzione **break** interrompe l'iterazione:

Nel programma che stampa i primi dieci numeri inseriamo un controllo sul numero da stampare se questo numero è 5 inseriamo l'istruzione break:



Output :

a = 1 2 3 4

a = 1 2 3 4

L'istruzione break interrompe l'iterazione sia del ciclo for sia del ciclo while; infatti quando la variabile a è uguale a 5 il programma esce dall'iterazione, pertanto saranno stampati solo i valori fino a 4.

L'istruzione **continue,** invece, salta le rimanenti istruzioni del corpo della struttura e procede con la successiva iterazione.

Nel programma precedente sostituiamo break con continue.

Output :

a = 1 2 3 4 6 7 8 9 10

a = 1 2 3 4 6 7 8 9 10

Esercizi

1. Scrivere un algoritmo che visualizza i numeri naturali dispari da 3 a 21.
2. Scrivere un algoritmo che visualizza in ordine decrescente i numeri pari positivi inferiori a 50.
3. Scrivere un algoritmo che visualizza tutti i numeri naturali inferiori al valore assoluto di un numero scelto dall'utente.
4. Scrivere un algoritmo che visualizza in ordine crescente tutti i numeri naturali compresi tra due numeri scelti dall'utente (estremi inclusi).
5. Scrivere un algoritmo che, presi in input 15 numeri interi, dica quanti valori pari sono stati inseriti.
6. Scrivere un algoritmo che, presi in input 20 numeri interi, dica quanti valori negativi sono stati inseriti.

7. Scrivere un algoritmo che, presi in input due numeri interi N e X (con N>0), visualizzi gli N numeri interi successivi a X.
8. Scrivere un algoritmo che, presi in input N valori interi (N > 0), calcoli la somma dei numeri positivi e la somma dei valori assoluti dei numeri negativi.
9. Calcolare il prodotto di due numeri naturali, mediante somme successive.
10. Calcolare quoziente e resto intero della divisione tra due numeri naturali, mediante differenze successive.
11. Scrivere un algoritmo che, dato un numero compreso nell'intervallo [1, 10], visualizzi i suoi primi 10 multipli.
12. Si sviluppi un programma che, come nel caso di una macchina distributrice di caffè, riceve in ingresso un numero intero positivo N (corrispondente a un importo da pagare in centesimi) e, successivamente, una sequenza di numeri interi corrispondenti alle monete inserite, che possono essere da 1, 5, 10, 20 e 50 centesimi. Il programma deve ripetere l'acquisizione di ciascun numero se non corrisponde a una moneta tra quelle indicate. Appena l'importo richiesto N viene raggiunto o superato, il programma interrompe l'acquisizione della sequenza e restituisce una serie di numeri interi corrispondenti al resto in monete da 1 e 5 centesimi.Ad esempio, se il programma riceve N=101 e la sequenza 50, 20, 20, 20, produce in uscita 5, 1, 1, 1, 1
13. Realizzare un programma in grado di calcolare l'altezza media degli studenti di una classe, dando in input il numero di allievi della classe.
14. Realizzare un programma in grado di calcolare l'altezza media degli studenti di una classe, senza fornire inizialmente il numero di allievi della classe.
15. Progettare un algoritmo che legga una sequenza di valori numerici fino alla lettura di un valore 0 e scriva quanti valori sono stati letti e la loro somma.
16. Progettare un algoritmo che, dato un valore numerico numero, legga n valori e conti quanti di essi sono maggiori di numero scrivendo il risultato.
17. Progettare un algoritmo che, dato un valore numerico numero, legga n valori e conti quanti sono i valori maggiori di numero, quanti i valori uguali a numero e quanti quelli minori.
18. Progettare un algoritmo che, dato un valore numerico numero, legga n numeri e conti quanti di questi sono multipli di numero scrivendo il risultato.
19. Progettare un algoritmo che legga una sequenza di valori numerici fino a che la loro somma è minore di 100 e scriva la somma ottenuta e quanti sono i valori letti.
20. Progettare un algoritmo che, dati due valori numerici numMinore e numMaggiore, legga n valori e conti quanti di essi sono compresi tra numMinore e numMaggiore scrivendo il risultato.
21. Progettare un algoritmo che permetta di inserire un numero compreso nell'intervallo [1-1000] e, partendo da questo, sommi i primi cinque numeri che

sono divisibili sia per 3, sia per 5, visualizzando sia i numeri che rispettano il criterio, sia il risultato finale.

22. Progettare un algoritmo che prenda in input un valore numerico maggiore o uguale a 10 e restituisca la somma di tutte le cifre che lo compongono.
23. Progettare un algoritmo che indichi a video se un numero è o meno palindromo.

    Un numero palindromo è un numero che invertito restituisce lo stesso numero originale, ad esempio:

    a. -1221 è palindromo;
    b. 707 è palindromo;
    c. 11212 non è palindromo;
    d. 121 è palindromo;
    e. -12321 è palindromo;
    f. 1001 è palindromo;
    g. -744117 non è palindromo.

    Per verificare se un numero è palindromo potrebbe essere utile determinare l'inverso del numero dato, da confrontare con il numero originale.

    Per trovare l'inverso di un numero ricordati che puoi estrarre una cifra dopo l'altra con il % e spostarla di posizione moltiplicandola per 10.

    Progettare un algoritmo che, dato in input un numero intero positivo, visualizzi la somma della cifra più significativa e della cifra meno significativa del numero. Ad esempio:

    Input: 252 → Output: 4
    Input: 257 → Output: 9
    Input: 0 → Output: 0
    Input: 5 → Output: 10
    Input: -10 → Output: ERRORE

    Progettare un algoritmo che dato in input un numero intero positivo visualizzi la somma delle sue cifre pari.
24. Progettare un algoritmo che prenda in input due numeri nell'intervallo [10-99], e se hanno una cifra in comune visualizzi la stringa "I due numeri condividono cifre", altrimenti visualizzi "I due numeri non condividono cifre".
25. Progettare un algoritmo che, dato in input un numero intero positivo espresso in base 10, fornisca la sua conversione in base 2 (non si utilizzino gli array). Per visualizzare correttamente il numero binario si utilizzi la tecnica per determinare l'inverso di un numero consigliata nell'esercizio per individuare se il numero è palindromo.
26. Progettare un algoritmo che prenda in input un numero intero maggiore di 1 e visualizzi tutti i suoi fattori primi. Ad esempio, i fattori primi di 6 sono 1, 2, 3, 6.
27. Progettare un algoritmo che preso in input un numero intero positivo indichi se è o meno un numero perfetto. Un numero è perfetto se è uguale alla

somma dei suoi divisori propri, ad esempio, 6 è un numero perfetto in quanto è somma dei suoi divisori propri: 1 + 2 + 3 = 6.

28. Progettare un algoritmo che, leggendo n valori numerici, verifichi se essi sono forniti in ordine crescente o meno.
29. Progettare un algoritmo che, dato un valore numerico k, legga n coppie di valori e conti quante di queste coppie hanno come prodotto il valore k.
30. Dati n valori numerici in ordine crescente, progettare un algoritmo che scriva se i numeri forniti a partire dal secondo differiscono ognuno dal precedente di un valore costante. In caso affermativo l'algoritmo deve scrivere il valore della differenza, in caso negativo l'algoritmo deve scrivere il valore massimo delle differenze.
31. Una leggenda orientale narra di un matematico che, in cambio di alcuni servigi resi al re, chiese la seguente ricompensa: «un chicco di riso per la prima casella di una scacchiera, due chicchi di riso per la seconda casella di una scacchiera, quattro chicchi di riso per la terza casella... e così via per tutte le 64 caselle della scacchiera». Progettare un algoritmo che, a partire dal numero N di caselle che si intendono riempire, calcoli il numero complessivo di chicchi di riso che spettano come ricompensa.
32. Un'onda marina anomala dimezza la propria altezza ogni chilometro percorso e scompare raggiungendo un'altezza pari a zero quando l'altezza scende al di sotto del metro.
33. Progettare un algoritmo che calcoli, a partire dai valori dell'altezza iniziale h e dal numero di chilometri percorsi k, l'altezza raggiunta dall'onda.
34. Modificare l'algoritmo precedente in modo che, a partire dalla sola altezza iniziale dell'onda h, determini il numero di chilometri necessario prima che essa scompaia.
35. Nella disintegrazione atomica dei materiali radioattivi la massa perduta nel periodo di un anno è data dal prodotto della massa residua per una costante di decadimento caratteristica del tipo di materiale.
36. Progettare un algoritmo che calcoli, a partire dai valori della massa iniziale espressa in grammi, della costante di decadimento e del numero di anni trascorsi, la massa residua di materiale.
37. Modificare l'algoritmo precedente in modo che, a partire dalla massa iniziale espressa in grammi e dalla costante di decadimento, determini il numero di anni necessario prima che la massa residua di materiale sia inferiore a 1 g.
38. La popolazione di un particolare batterio raddoppia ogni ora. Progettare un algoritmo che, a partire dal numero di ore trascorse e dal valore espresso in «unità di carica batterica» della consistenza iniziale della popolazione batterica, ne calcoli la consistenza finale raggiunta.
39. La massa di un particolare materiale radioattivo dimezza ogni millennio. Progettare un algoritmo che, a partire dal numero di millenni trascorsi e dal

valore espresso in grammi della massa iniziale del materiale radioattivo, calcoli la massa finale residua.

40. Dato un valore numerico costante k (non necessariamente intero), l'N-esimo numero di Bernoulli è dato dalla somma dei primi N numeri interi elevati alla potenza k; per esempio per N = 5:
    1k + 2k + 3k + 4k + 5k
    Progettare un algoritmo che determini, a partire dai valori della costante k e del numero N, il numero di Bernoulli relativo.
41. La media geometrica di N numeri x1 , x2 , ..., xN è data dalla seguente formula:
    (x1 * x2 * ... * xN)1/N
    Progettare un algoritmo che calcoli la media geometrica di N numeri positivi inseriti dall'utente.
42. Il filosofo Zenone di Elea motivava il fatto che il moto è solo un'illusione con la seguente argomentazione: «dovendo percorrere una certa distanza si dovrà coprire con un primo spostamento metà della distanza, con un secondo spostamento metà della distanza rimanente, con un terzo spostamento metà della distanza ancora rimanente e così via senza arrivare mai a destinazione». Progettare un algoritmo che, data la distanza da percorrere e il numero di spostamenti effettuati, calcoli la distanza effettivamente coperta.
    In una acciaieria il semilavorato metallico grezzo viene prodotto con uno spessore di alcuni centimetri e viene successivamente lavorato passando per una serie di N laminatoi, ciascuno dei quali diminuisce lo spessore del 10%.
43. Progettare un algoritmo per determinare lo spessore del laminato a partire dallo spessore del semilavorato grezzo e dal numero di laminatoi presenti nel processo di lavorazione.
44. Modificare l'algoritmo precedente in modo che determini il numero di laminatoi necessari nel processo di lavorazione per ottenere un laminato di spessore definito a partire dallo spessore del semilavorato.
45. Un foglio di carta in formato A0 ha dimensioni 118,8 × 84 cm; a partire da questo un foglio in formato A1 ha il lato lungo uguale al lato corto del formato A0 (84 cm) e il lato corto uguale alla metà del lato lungo del formato A0 (118,8 cm : 2 = 59,4 cm). Per calcolare le dimensioni dei formati A2, A3, A4, ... si procede sempre nello stesso modo: il lato lungo è uguale al lato corto del foglio immediatamente più grande, mentre il lato corto è esattamente la metà del lato lungo del foglio immediatamente più grande e così via. Progettare l'algoritmo che calcola le dimensioni di un foglio in formato AN, dove N viene fornito come dato di ingresso.
46. Una popolazione di insetti ha un accrescimento mensile dato dalla seguente formula:
    k * P * (1 – P/M)

dove P è la popolazione di insetti, k è la costante di accrescimento, M è la massima popolazione sostenibile dall'ambiente locale.

47. Realizzare un algoritmo che abbia come dati il numero N dei mesi, la costante k, il massimo M, la popolazione iniziale Pi e che calcoli come risultato la popolazione di insetti trascorsi N mesi dal momento iniziale.
48. L'accrescimento della popolazione umana è guidato da una semplice legge matematica: l'incremento della popolazione tra un anno e il successivo è dato dal prodotto di una costante (il tasso di accrescimento) per la dimensione della popolazione. Realizzare l'algoritmo che consente di simulare anno per anno i valori della dimensione della popolazione a partire da:
    l'anno iniziale della simulazione;
    il valore della dimensione iniziale della popolazione;
    il valore del tasso di accrescimento;
    l'anno finale della simulazione.
49. Una pianta cresce ogni mese della metà di quanto è cresciuta il mese precedente (il primo mese cresce della metà dell'altezza iniziale). Progettare l'algoritmo che calcola l'altezza finale della pianta a partire dall'altezza iniziale e dal numero di mesi.
50. Nel 1593 il matematico dilettante francese François Viète approssimò il valore numerico di π con il seguente metodo, che fornisce risultati progressivamente più precisi al crescere del numero n di iterazioni:

$$\begin{cases} c_0 = 0 & p_0 = 2 \\ c_n = \sqrt{\dfrac{1 + c_{n-1}}{2}} & p_n = \dfrac{p_{n-1}}{c_n} \end{cases}$$

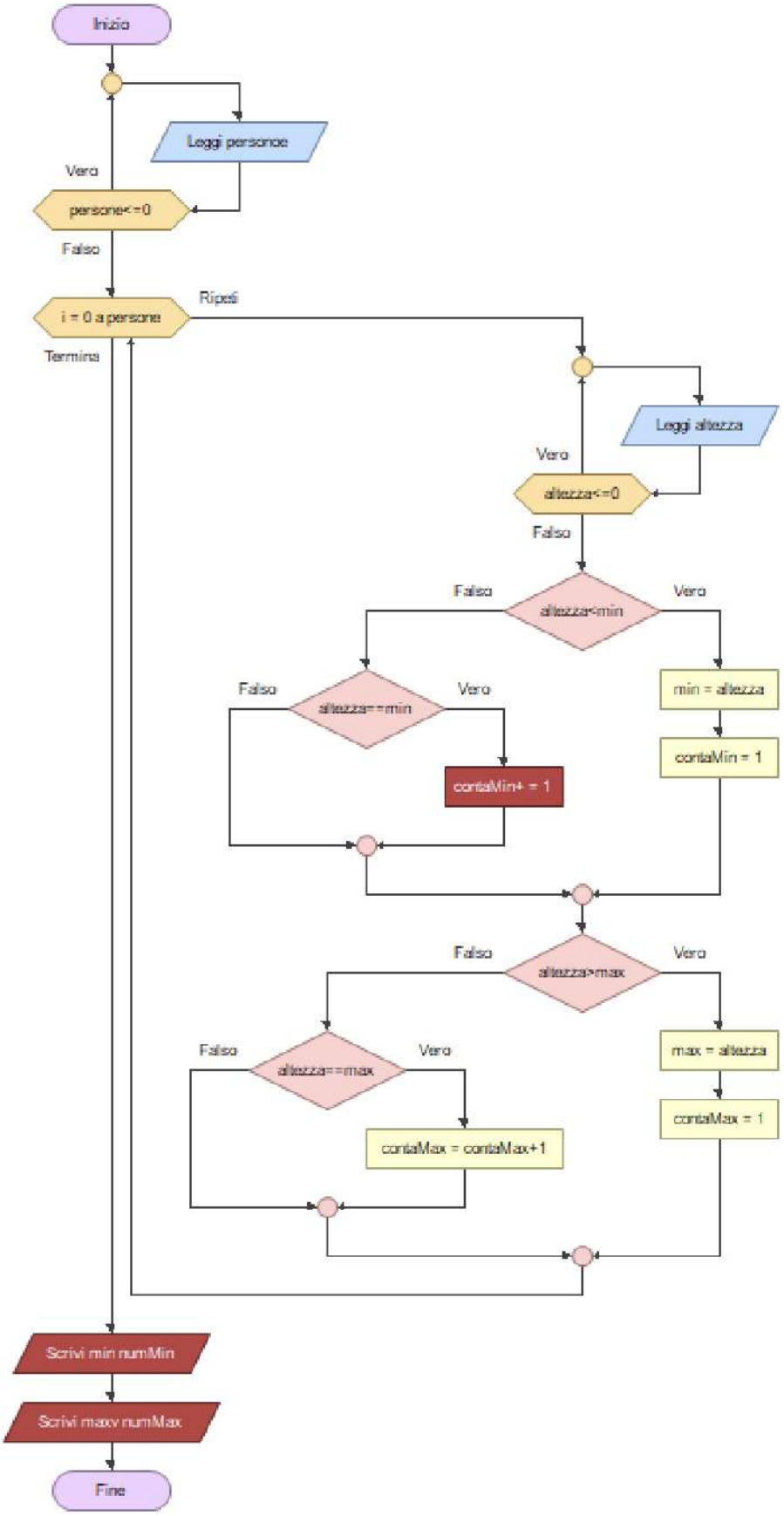
51. Realizzare un algoritmo che, dato il numero n di iterazioni da calcolare, produca la stima pn del valore di π nell'ipotesi di un esecutore che sia in grado di calcolare direttamente la radice quadrata di un numero.
52. La prima stima precisa del valore di π è stata fornita dal matematico cinese Tsu Chung-Chi nel V secolo d.C. utilizzando il seguente metodo iterativo:

$$x_0 = \sqrt{2}$$
$$x_{n+1} = \sqrt{2 - \sqrt{4 - x_n^2}}$$

    dove, a partire dal valore iniziale x0 , viene calcolato ogni volta il valore successivo xn + 1 a partire dal valore precedente xn; al termine del procedimento – dopo N iterazioni del calcolo – la stima del valore di π è data dalla seguente formula:
    s = 2 * xN
53. Progettare un algoritmo che, a partire dal numero di iterazioni N, calcoli una stima s del valore di π nell'ipotesi di un esecutore che sia in grado di calcolare direttamente la radice quadrata di un numero.

54. La società di assicurazioni “Io speriamo che me la cavo” ha stipulato N contratti di assicurazione su motociclette secondo la seguente formula:
se la moto ha cilindrata maggiore di 350, il costo dell'assicurazione è uguale a una quota fissa Q più € 30 per ogni mille euro del prezzo della moto;
altrimenti il costo dell’assicurazione è uguale alla quota fissa Q più € 20 per ogni mille euro del prezzo della moto.
55. Progettare un algoritmo in grado di fornire in output quanto ha incassato l’assicurazione.
56. Lo stipendio di un dipendente dell’azienda “Ma quanto mi costi” è formato da 3 parti: salario accessorio (A), stipendio base (B), compensi aggiuntivi (C). Sulla parte A si applica la trattenuta del 19%, sulla parte B il 16%, sulla parte C il 2%. Sul totale A+B+C viene trattenuto un ulteriore 0.5%.
Progettare un algoritmo che per ognuno dei dipendenti visualizzi il numero di matricola, il totale delle trattenute e lo stipendio finale netto. Si visualizzi infine il totale degli stipendi lordi e il totale delle trattenute di tutti i dipendenti.
57. Progettare un algoritmo che preveda l’inserimento delle altezze in centimetri di un gruppo di persone, e visualizzi l’altezza minima e il numero di persone che hanno un’altezza pari a quella minima. Si effettuino tutti i controlli necessari sui valori inseriti in input.
Il programma è utilizzato per determinare l’altezza minima e massima nell’ambito di un’applicazione statistica sulla crescita degli studenti italiani in età scolare, indicando anche il numero di persone che hanno la stessa altezza minima o massima.
Il programma prende in input il numero delle persone da analizzare e le altezze di ogni singolo individuo. Fornisce in output l’altezza minima e il numero di persone che hanno un’altezza pari al minimo, e l'altezza massima con il numero di persone che hanno l’altezza pari al massimo.
Il programma prevede che venga richiesto in input il numero di persone. Le variabili per contenere l’altezza minima e massima devono essere inizializzate a dei valori che permettono la loro immediata sostituzione. Inoltre sarà necessario inizializzare due variabili che contano il numero di persone con l'altezza minima e il numero di persone con l'altezza massima. Successivamente viene impostato un ciclo di controllo delle altezze con l'altezza minima. Se l’altezza inserita in input risulta essere inferiore al minimo la variabile del minimo dovrà essere sostituita con il nuovo minimo individuato, e il contatore delle persone che hanno quella altezza minima dovrà essere inizializzato nuovamente a zero. Se invece l'altezza inserita risulta essere pari al minimo bisognerà incrementare il contatore. Nel caso in cui l'altezza inserita fosse superiore al minimo quest’ultima non dovrà essere conteggiata. Il procedimento si ripeterà per l’altezza massima. Al termine il programma visualizza l'altezza minima, l'altezza massima, il numero di persone che hanno

un'altezza pari al minimo e il numero di persone che hanno un'altezza pari al massimo.



58. In una gara podistica si vogliono visualizzare il tempo migliore, il tempo peggiore e il tempo medio di percorrenza. Si preveda l'inserimento dei tempi di percorrenza dei diversi atleti effettuando tutti i controlli necessari sui valori inseriti in input. I tempi di percorrenza vengono forniti in secondi.
59. Progettare un algoritmo che permetta di assemblare uno scatolone di pacchi di farina usando pacchi da 5kg e 1kg. L'algoritmo deve prevedere l'inserimento dei seguenti dati:
    a. numero totale di kg da inserire in uno scatolone;
    b. numero di pacchi da 5kg disponibili;
    c. numero di pacchi da 1kg disponibili.
60. Uno scatolone può ritenersi assemblato se sono presenti almeno tutti i kg che può contenere, considerando la somma dei kg dei pacchi da 5kg e da 1kg. Non è però possibile dividere i pacchi di farina, quindi se lo scatolone può contenere 9kg di farina, e si hanno solo 2 pacchi da 5kg e nessuno da 1kg, non è possibile assemblare lo scatolone perché i pacchi da 5kg non possono essere divisi. Se invece si hanno a disposizione 1 pacco da 5kg e 5 pacchi da 1kg è possibile assemblare lo scatolone da 9kg perché avanza un pacco da 1kg intero e questo è considerato ammissibile.
61. Si visualizzi come output una scritta che indichi se lo scatolone è o meno stato assemblato.
62. Progettare un algoritmo che permetta di determinare il più grande numero primo fattore di un numero intero positivo dato. Si ricordi che per il teorema fondamentale dell'aritmetica un qualsiasi numero naturale maggiore di 1 o è un numero primo o si può esprimere come prodotto di numeri primi. Nel caso in cui il numero inserito sia minore o uguale ad 1 si richieda l'inserimento del numero, in quanto 0 e 1 non sono considerati primi.
    Esempi:
    Input: 21 → Output: 7 perché 7 è il più grande numero primo (3 * 7 = 21)
    Input: 217 → Output: 31 perché 31 è il più grande numero primo (7 * 31 = 217)
    Input: 13 → Output: 13 poiché 13 è il più grande numero primo (13 = 13)
    Input: 0 → Richiedere l'immissione dell'input
    Input: 45 → Output: 5 perché 5 è il più grande numero primo (3 * 3 * 5 = 45)
    Input: -1 → Richiedere l'immissione dell'input
63. Progettare un algoritmo che prende in input un numero maggiore o uguale a 5 e visualizzi un quadrato con le due diagonali, con una base e un'altezza pari al numero inserito in input.
    Ad esempio:

```
Input: 5 → Output:
*****
** **
* * *
** **
*****
```

```
Input: 8 → Output:
********
**    **
* *  * *
*  **  *
* *  * *
**    **
********
```

64. Un grande magazzino ha 4 reparti identificati dai numeri 1, 2, 3 e 4. La direzione decide di applicare degli sconti ai prodotti dei diversi magazzini, differenziandoli per magazzino. Si progetti un algoritmo in grado di richiedere in input le quattro percentuali di sconto da applicare ai prodotti dei vari reparti e, successivamente richieda in input N prodotto con il relativo reparto di appartenenza e il prezzo, e visualizzi per ciascun prodotto inserito il prezzo scontato.
65. Progettare un algoritmo in grado di svolgere le quattro operazioni fondamentali su due numeri.
    Visualizzare inizialmente il seguente menù:
    1) Somma
    2) Sottrai
    3) Moltiplica
    4) Dividi
    5) Esci
66. Se viene scelta una delle opzioni da 1 a 4 viene svolta l'operazione aritmetica richiesta sui due numeri visualizzando il risultato. Successivamente viene nuovamente visualizzato il menù e richiesto l'inserimento di due numeri.
    Se invece viene scelta l'opzione 5 il programma deve terminare, salutando cortesemente l'utente. Usare lo switch-case per il menù.
67. Progettare un algoritmo che preveda l'inserimento delle età degli studenti della tua classe e fornisca il numero di occorrenze del più grande.
68. Progettare un algoritmo che visualizzi la rendita annuale di un investimento effettuato presso la banca "Più soldi per tutti".
    La banca visualizza il piano di investimento usando il capitale iniziale in euro, la percentuale di interesse e il numero di anni dell'investimento.
    Il calcolo degli interessi si effettua tramite la seguente formula:
    `interessi = capitale * tasso / 100`
    e questi verranno sommati di anno in anno al capitale.
    Ad esempio se la somma dell'investimento è di 30000 euro al tasso del 12.5% per 10 anni, il piano annuale dell'investimento dovrà essere il seguente:

```
Anno | Interesse | Capitale
-----|-----------|----------
  1  |  3750.00  | 33750.00
  2  |  4218.75  | 37968.75
  3  |  4746.09  | 42714.84
  4  |  5339.35  | 48054.19
  5  |  6006.77  | 54060.96
```

```
6  |  6757.62  | 60818.58
7  |  7602.32  | 68420.90
8  |  8552.61  | 76973.51
9  |  9621.68  | 86595.19
10 |  10824.39 | 97419.58
```

69. Progettare un algoritmo che converta un numero binario in un numero in base 10. Il numero binario è rappresentato su N bit, e il valore di N dovrà essere fornito dall'utente. L'utente dovrà inserire le cifre del numero binario un bit alla volta, partendo dal bit meno significativo (ossia dal bit di peso 2^0). Il programma visualizzerà il numero in base 10 corrispondente (non si utilizzino gli array).
70. Progettare un algoritmo che calcoli il valore massimo e minimo di un insieme di N numeri inseriti da tastiera.
71. Progettare un algoritmo che analizzi una sequenza di numeri. I numeri dovranno essere inseriti da tastiera e l'algoritmo dovrà visualizzare i seguenti risultati:
    - quanti sono i numeri positivi, nulli e negativi;
    - quanti sono i numeri pari e dispari;
    - se la sequenza dei numeri inseriti è crescente, decrescente oppure ne crescente, nè decrescente.
      Si osservi che una sequenza è crescente se ogni numero è maggiore del precedente, è decrescente se ogni numero è minore del precedente, mentre non è crescente e neanche decrescente in tutti gli altri casi.
72. Progettare un algoritmo che calcoli il massimo comune divisore (MCD) di due numeri interi positivi. Il MCD è definito come il massimo tra i divisori comuni ai due numeri.
    Dati due numeri, n1 e n2, il MCD di n1 e n2 è il massimo tra i numeri che sono divisori (con resto uguale a zero) sia di n1 che di n2. In particolare si supponga che n1 sia minore di n2, il MCD è il massimo tra i numeri compresi tra 1 e n1 che sono divisori (con resto uguale a zero) sia di n1 che di n2.
73. Progettare un algoritmo che calcoli il minimo comune multiplo (MCM) di due numeri interi positivi.
    Dati due numeri interi n1 e n2, il minimo comune multiplo è il più piccolo numero m che è divisibile (con resto pari a zero) sia per n1 che per n2.
74. Progettare un algoritmo che chieda in input un numero maggiore di 1 e visualizzi un quadrato di asterischi ('*') di lato pari al valore numerico inserito.
    Ad esempio:

```
Input: 4 → Output:
****
****
****
****
```

75. Progettare un algoritmo che chieda in input un numero maggiore di 1 e visualizzi un triangolo rettangolo di asterischi ('*') di altezza e base pari al valore numerico inserito.
    Ad esempio:

```
Input: 4 → Output:
*
**
```

```
***
****
```

76. Progettare un algoritmo che chieda in input un numero maggiore di 1 e visualizzi un triangolo rettangolo di asterischi (‘*’) di altezza e base pari al valore numerico inserito, con il lato verticale spostato verso destra.
    Ad esempio:
    ```
    Input: 4 → Output:
       *
      **
     ***
    ****
    ```
77. Progettare un algoritmo che chieda in input un numero maggiore di 1 e visualizzi un quadrato i cui lati siano formati da asterischi (‘*’) di lato pari al valore numerico inserito.
    Ad esempio:
    ```
    Input: 5 → Output:
    *****
    *   *
    *   *
    *   *
    *****
    ```
78. Progettare un algoritmo che chieda in input un numero maggiore di 1 e visualizzi un triangolo rettangolo i cui lati siano formati da asterischi (‘*’) di altezza e base pari al valore numerico inserito.
    Ad esempio:
    ```
    Input: 5 → Output:
    *
    **
    * *
    *  *
    *****
    ```

Array

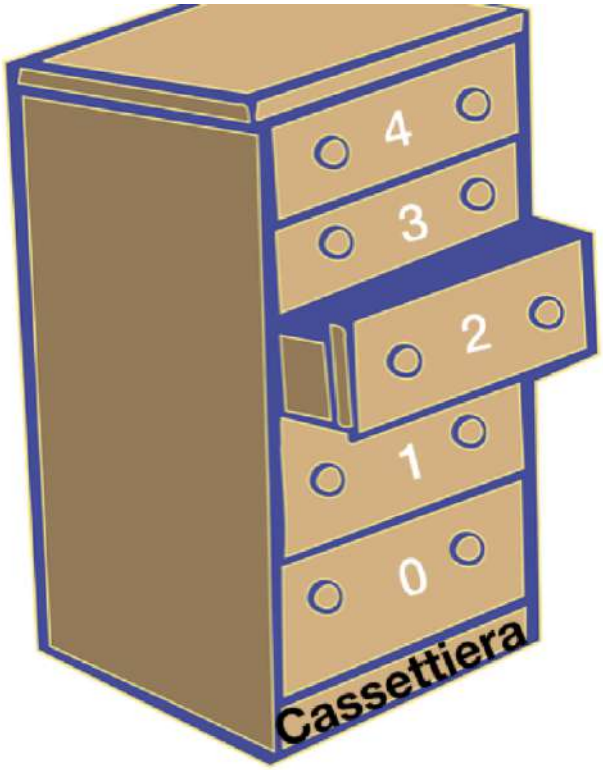
Un array è una struttura dati indicizzata che può contenere:

- dati primitivi,
- oggetti
- altri array

Un'array è caratterizzato da:

1. **Il nome** che individua la struttura dati composta da vari elementi.
2. **Un indice** che consente di accedere agli elementi dell'array.

Se una variabile la si assimila a una scatola che contiene un valore. L'array si può assimilare a una cassettiera in cui ogni cassetto contiene un valore. Ogni cassetto è numerato (indice) a partire da zero.

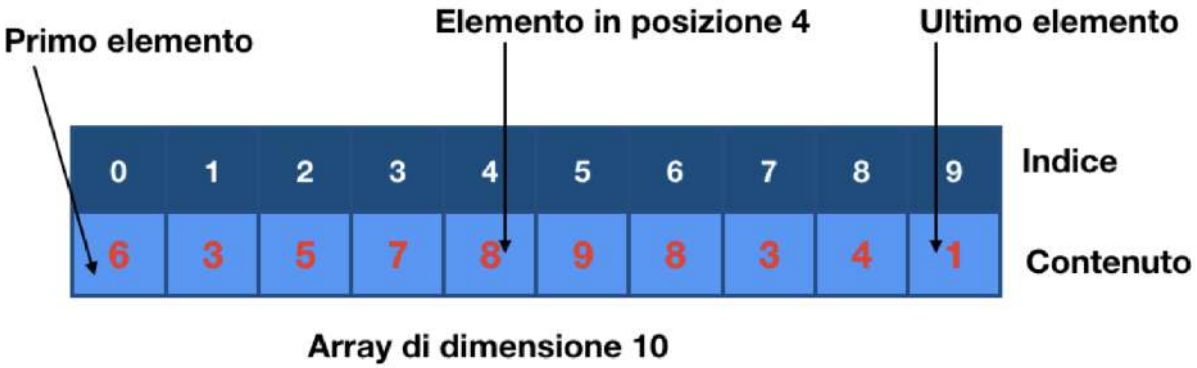


Cassettiera indica il nome dell'array e i numeri indicano l'indice.

Le principali caratteristiche di un Array sono:

- Gli elementi contenuti nell'array devono essere dello **stesso tipo**.
- La lunghezza è fissa
- la lunghezza di un array deve essere dichiarata in fase d'inizializzazione

- Il primo elemento dell'array si trova nella posizione 0, l'ultimo nella posizione n-1, dove n è la lunghezza dell'array.



Per utilizzare un array bisogna passare attraverso le fasi di:

1. Dichiarazione.
2. Creazione
3. Inizializzazione.

Dichiarazione

Come ogni variabile, un array deve essere dichiarato in Java. Questo può essere fatto in due modi equivalenti, ma il primo è più coerente con lo stile Java.

Per dichiarare un array è necessario posporre (oppure anteporre) una coppia di parentesi quadre all'identificatore.

Tipo[] nomeDellaVariabile;

oppure

Tipo nomeDellaVariabile[];

```
char[] caratteri; /* definizione  di un array di Caratteri */
int[] numeri;   /* definizione  di un array di interi */
```

```
Moto[] modelli; /*definizione dell'array di oggetti di tipo Moto */
```

Il valore **default** delle variabili di tipo array **è null**.

Dichiarare un oggetto non significa creare un oggetto!

int[] a;

- a non contiene l'oggetto
- a contiene il riferimento all'oggetto

Creazione

Un array è un oggetto speciale in Java e, in quanto tale, va istanziato. La sintassi è la seguente:

nomeVariabile = new TipoDiDato[n];

```
caratteri= new char[21];
/* inizializzazione di un array di caratteri */
numeri = new int[25];
/*  inizializzazione di un array di interi */
modelli = new Moto[4];
/* inizializzazione dell'array di oggetti di tipo Moto */
```

È obbligatorio specificare al momento dell'istanza dell'array la dimensione dell'array stesso.

Inizializzazione di un array e accesso ai suoi elementi

Il processo di creazione non ci fornisce un array vuoto, ma un array pieno di valori predefiniti. Ad esempio, per un array d'interi, questo è 0 e per un array di oggetti, il valore predefinito in ogni cella è null. Si accede a un elemento array (ad esempio, per impostarne il valore, o visualizzarlo sullo schermo o eseguire un'operazione con esso) tramite l'indice:

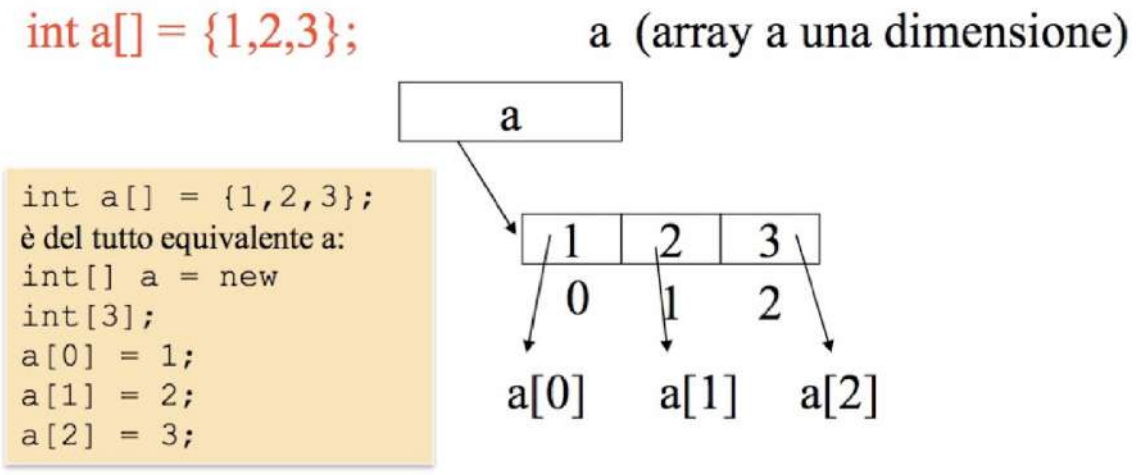
nomeDellaVariabile[indice]

Per inserire i dati un array, bisogna inserirli singolarmente in ogni elemento:

```
caratteri [0] = 'a';
caratteri [1] =  'b';
```

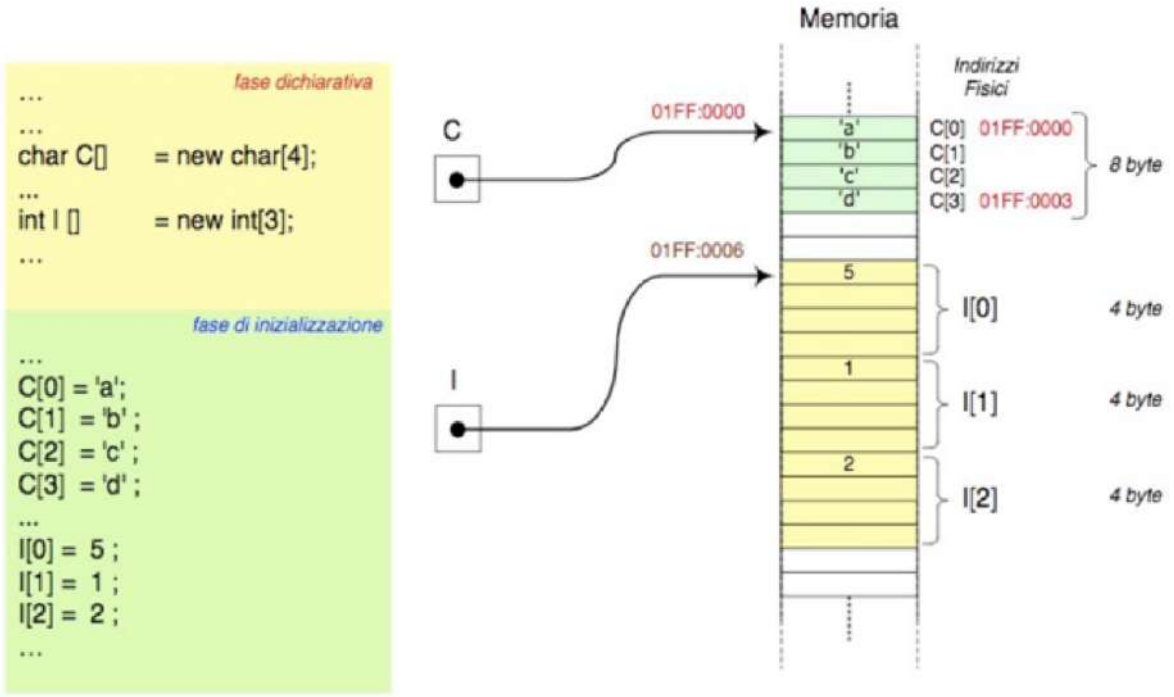
```
caratteri[20] = 'z';
numeri [0] = 1;
numeri[1] = 7;
numeri [24] = 56;
modelli[0]= new Moto( .....);
modelli[3]= new Moto( .....);
```

Oppure in modo più compatto durante la dichiarazione dell'array:



La variabile dichiarata a (**detta referenza all'oggetto**) contiene il riferimento necessario a trovare l'oggetto **puntato da a in memoria,** in pratica a contiene l'indirizzo di una locazione di memoria a partire dal quale è memorizzato l'oggetto.

```
char[] caratteri= {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'z'};
```



Gli array definiscono, una proprietà, chiamata length, che restituisce la dimensione effettiva dell'array stesso. Quindi:

caratteri.length

vale 21.

Per produrre tutti gli elementi di un array si esegue un ciclo dalla posizione zero alla posizione n-1:

```
s ProvaArray

blic static void main(String args[]) {
int x[] = new int[10];
for (int i=0; i<=9; i++) {
  x[i] = i;//riempiamo l'array con lo stesso valore
'indice
}
    for (int i=0; i<x.length; i++) {
  System.out.println("x["+i+"] = "+x[i]);
}
```

```
   // esempio di un Array di Stringhe
String stringa[] = {"primo","secondo","terzo"};
for (int i=0; i<=2; i++)
  System.out.println("stringa["+i+"] = "+stringa[i]);
}
```

1. Prima si dichiara "x[]" come nuovo oggetto Array di massimo 10 elementi.
2. Il primo ciclo for assegna a ogni elemento dell'Array il rispettivo numero "i".
3. Nel secondo ciclo stampa su schermo tutti gli elementi dell'array.

Il secondo Array è formato da elementi Stringa (Oggetti istanziati dalla Classe String).

Prova il codice precedente

Ciclo avanzato for-each

Poiché in un array l'attraversamento è un operazione comune, Java fornisce una sintassi alternativa che rende il codice più compatto. Ad esempio, considera un ciclo for che visualizza gli elementi di un array su righe separate:

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

Il ciclo si può scrivere in questo modo:

```
for ( int value: values) {
System.out.println (value);
}
```

Con questa variante del for non si devono specificare:

Il punto di partenza

La lunghezza dell'array

L'istruzione di modifica della condizione.

Java rileva che la variabile values è un array e assegna alla variabile value il contenuto di ciascun elemento dell'array.

Questa variante del for consente di scrivere meno codice quando si usano gli array.

Utilizzare gli array nei metodi

I metodi possono sia ricevere come argomento sia una variabile indicizzata che un intero array e possono restituire array.

Variabili indicizzate come argomenti di un metodo

Una variabile indicizzata di un array a, come **a [i]**, può essere utilizzata ogni volta che è possibile utilizzare una variabile del tipo base dell'array. Una variabile indicizzata può quindi essere un argomento di un metodo, così come ogni altra variabile dello stesso tipo base dell'array il valore della variabile a[i] non viene modificata nel metodo.

Il metodo:

```
public int somma(int a, int b){
return a+b;
}
```

Riceve come argomento due numeri interi, si può chiamare il metodo passando come parametro due interi indifferentemente che siano variabili o costanti intere o elementi di un array di interi;

```
int[] numero={1,2,3};
int a=5;
int b= somma(a,numero[1]);
```

Array come argomento di un metodo

Il modo con cui si specifica che l'argomento di un metodo è un array è simile al modo con cui si dichiara un array. Per esempio, il seguente metodo cerca accetta come argomento un qualsiasi array d'interi:

```
public class Esempio {
public  int cerca(int[] unArray,int valoreDaCercare) {
for(int i=0;i<unArray.length; i++)
if(unArray[i]==valoreDaCercare)
   return i;
return -1;
}
}
```

Quando si utilizza come parametro un array, è necessario indicare il tipo base dell'array, ma non si deve impostare la lunghezza dell'array stesso. È possibile utilizzare una sintassi alternativa per specificare che un array è un argomento di un metodo. Tale sintassi è simile a quella alternativa utilizzata in fase di dichiarazione di un array: è possibile specificare le parentesi quadre dopo il nome dell'array invece che dopo il tipo. La dichiarazione del precedente metodo diventa quindi:

public int cerca(int unArray[],int valoreDaCercare)

Nei metodi gli oggetti vengono passati per riferimento e quindi possono essere modificati dentro il metodo.

Ad esempio, il seguente metodo eleva al quadrato tutti gli elementi di un array di interi:

```
public class Main{
  public static void main(String args[]){
    int[] a={1,2,3,4,5,6,7,8,9};
    quadra(a);
    for(int i=0;i<a.length;i++)
```

```
      System.out.println(a[i]);
  }
  public static void quadra(int[] unArray){
    for (int i=1;i<unArray.length;i++)
      unArray[i]=(int)Math.pow(unArray[i],2);
    }
}
```

Codice

Operazioni sugli array

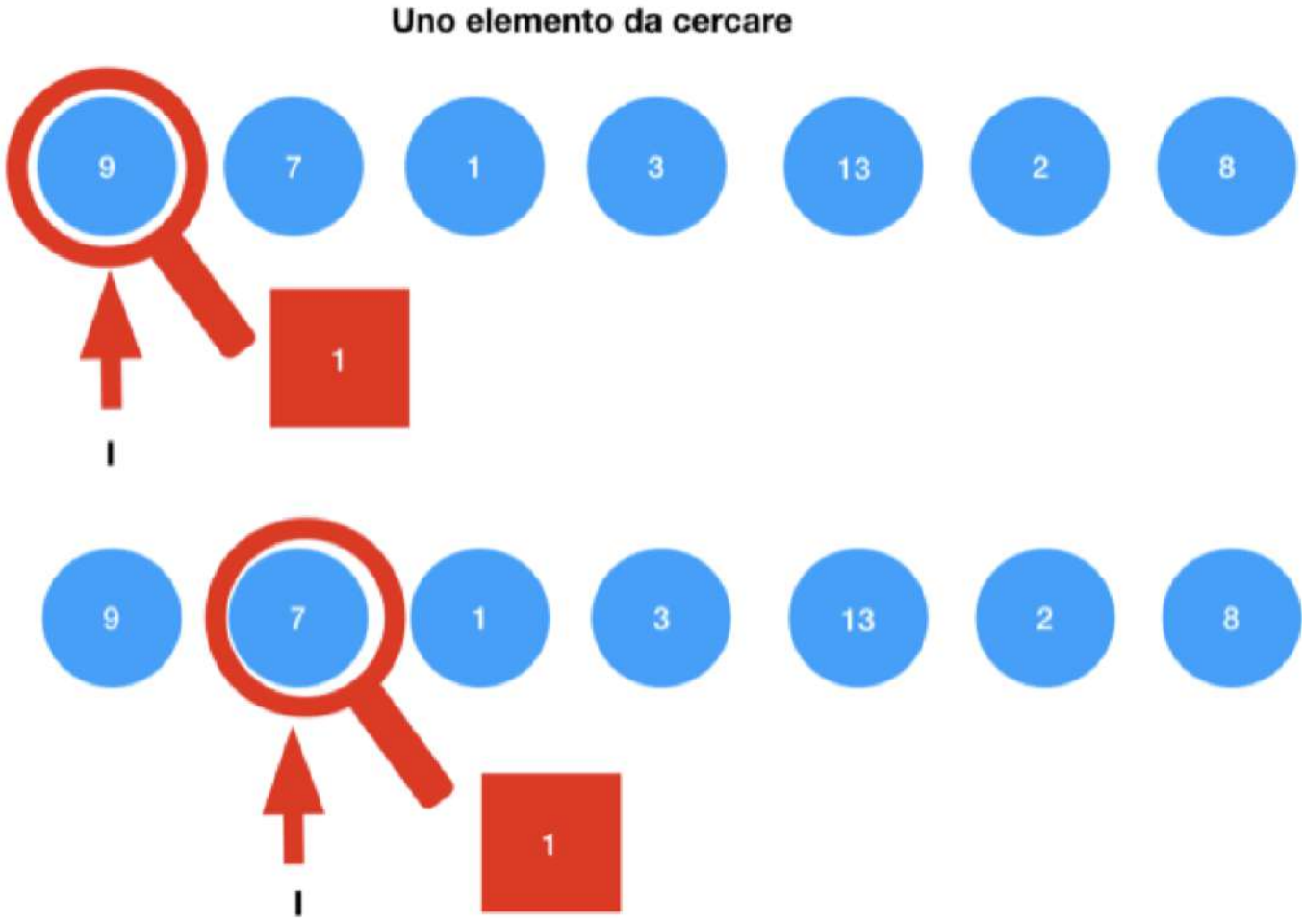
La ricerca sequenziale

La ricerca sequenziale o lineare è l'algoritmo di ricerca più semplice, consiste nel confrontare ogni elemento dell'array con l'elemento che si sta cercando.

Come cercare i calzini nella cassettiera. Si apre il cassetto zero e si vede se ci sono calzini, se non ci sono si apre il cassetto uno e così via fino alla fine. Quindi il procedimento è questo:

Si scorre l'array dall'inizio e si confronta l'elemento dell'array con il valore cercato:

1. se sono uguali si restituisce l'indice
2. altrimenti si prosegue fino alla fine
3. se si arriva alla fine restituisce -1 per segnalare che il valore cercato non è presente nell'array.




```
public  int cerca(int[] unArray,int valoreDaCercare) {
for(int i=0;i<unArray.length; i++)
if(unArray[i]==valoreDaCercare)
   return i;
return -1;
}
```

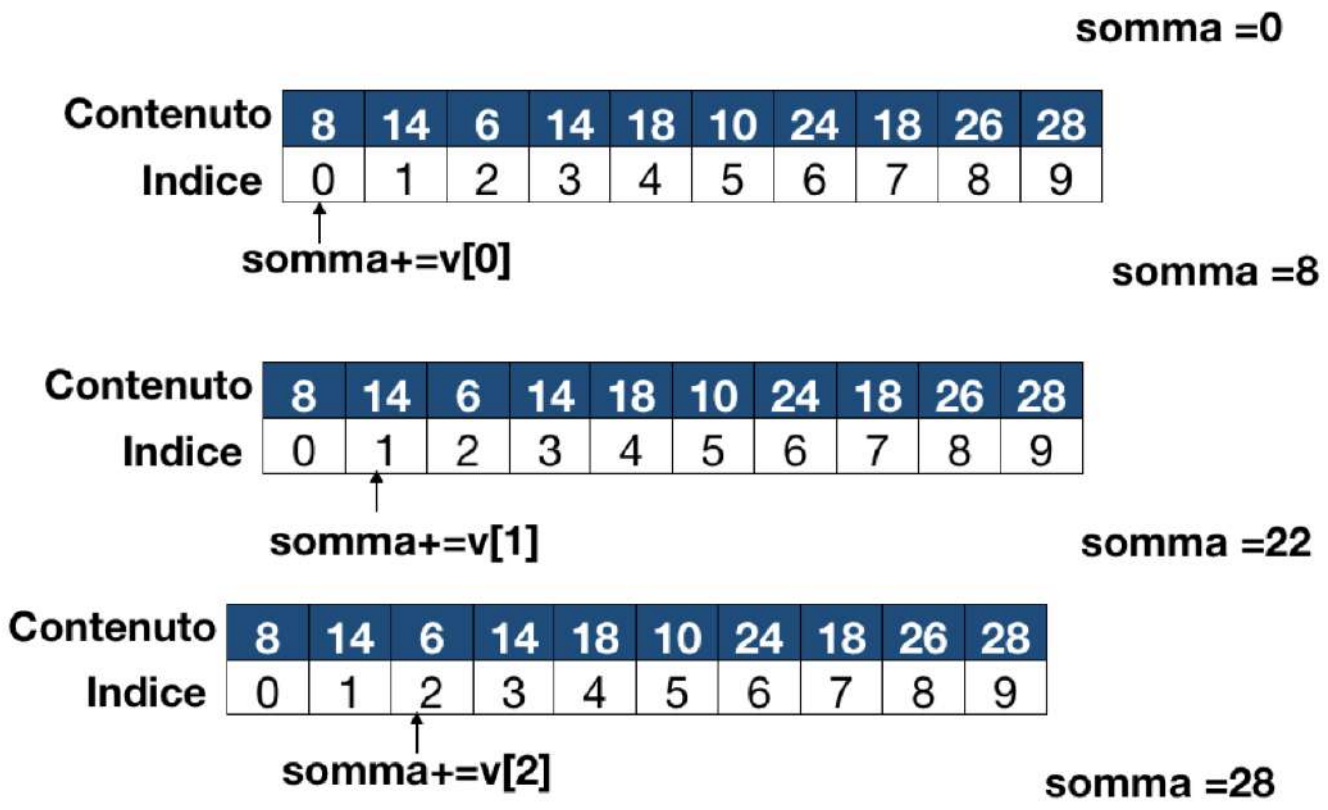
Riduzione

Riduzione significa ridurre l'array in un singolo valore, eseguendo sugli elementi di un array delle operazioni. Sono esempi di riduzione di un array:

- la somma o il prodotto dei suoi elementi (o solo di alcuni elementi che godono di una proprietà comune)
- la media
- il numero degli elementi che godono della stessa proprietà (pari, dispari, positivi, ecc…)

Esempio: dato un array scrivere una funzione che sommi tutti gli elementi dell'array.

Si risolve come per il calcolo della somma di una sequenza si inizializza a zero la variabile somma(accumulatore) e ogni volta si aggiunge un elemento dell'array



Quindi basta scorrere l'array e ogni volta sommare il valore corrispondente all'indice.

```
public int somma(int[] vettore) {
  int somma = 0;
  for (int indice = 0; indice < vettore.length; indice++) {
    somma += vettore[indice];
  }
```

```
  return somma;
}
```

Per il prodotto si procede allo stesso modo ma inizializzando prodotto al valore uno.

Ordinamento di un Array

Ordinare un array, significa, mettere gli elementi:

- In ordine alfabetico se sono stringhe.
- In ordine di grandezza se sono numeri.

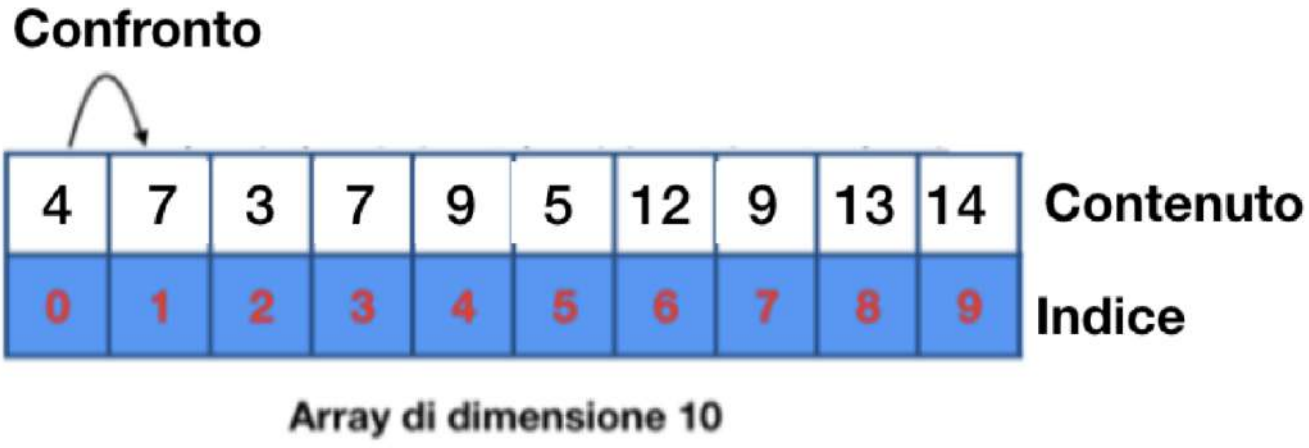
L'ordine può essere crescente o decrescente.

Esistono diversi algoritmi di ordinamento che adottano strategie diverse. Ma tutti si basano su due operazioni fondamentali: confronto e scambio.

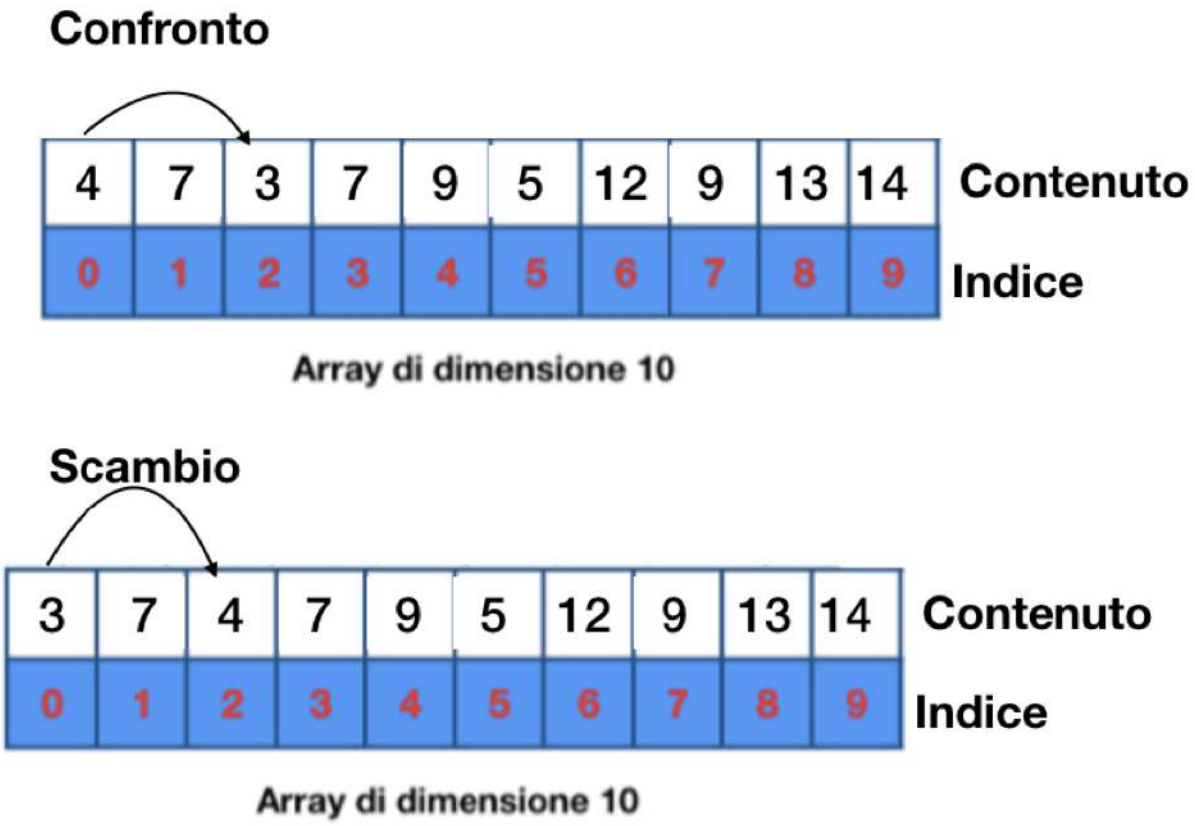
Ordinamento per sostituzione(exchange sort)

L'exchange sort è un semplice algoritmo di ordinamento efficace per Array di piccole dimensioni.

Consiste nel confrontare ogni elemento, a partire dal primo, con tutti gli altri:

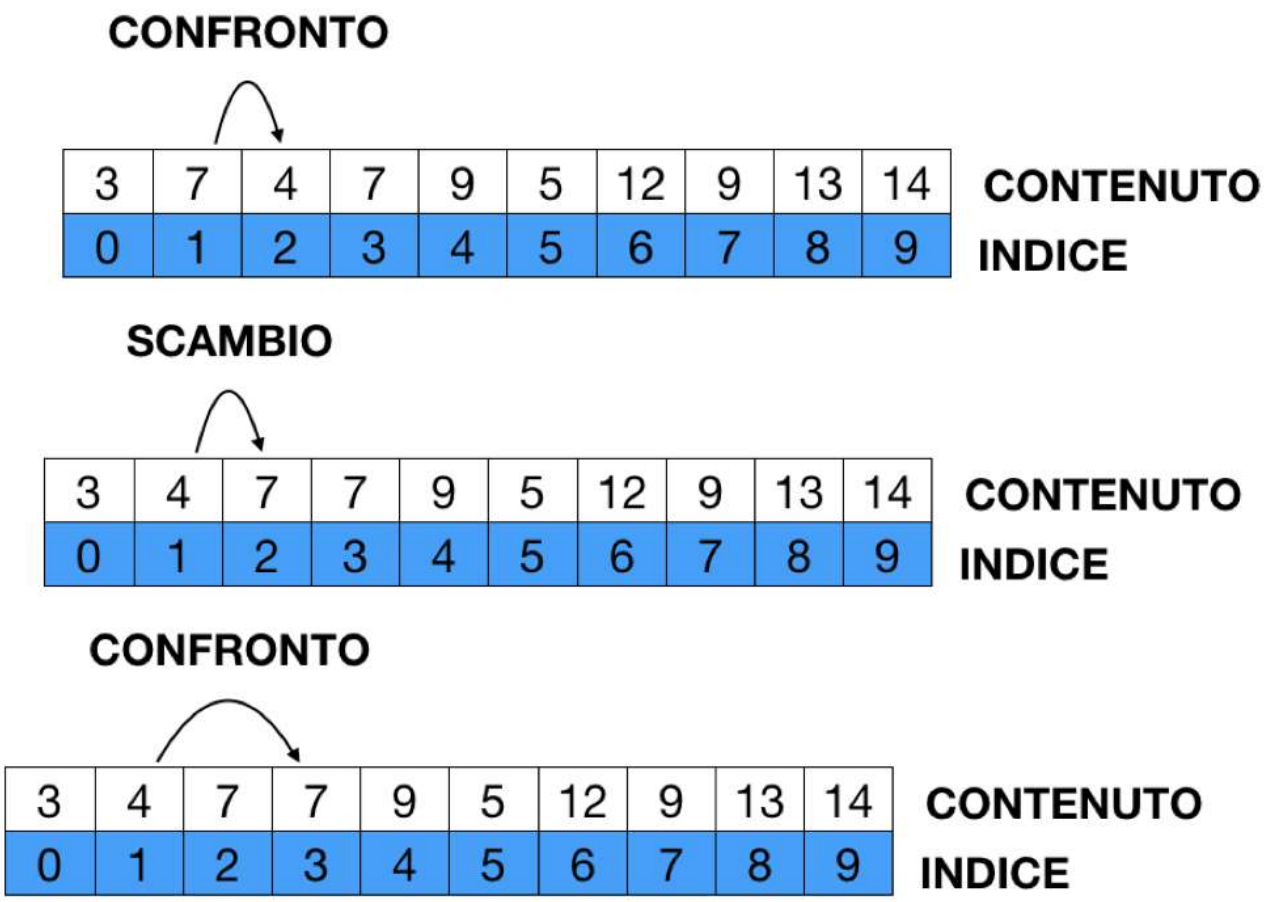


Se si incontra un elemento più piccolo si procede allo scambio tra i due.



Alla fine di tutti i confronti nella prima posizione si trova l'elemento minore.

Si ripete il procedimento per l'elemento nella seconda posizione, nella terza e così via fino all'ultimo elemento.



Se n è la dimensione dell'array per inserire il valore minimo nella prima posizione si hanno:

- n-1 confronti
- al massimo n-1 scambi.

Per il secondo elemento si controllano

- n-2 confronti,
- al massimo n-2 scambi

Per cui, alla fine, si hanno circa:

n(n-1)/2 confronti

un massimo di n(n-1)/2 scambi.

Il numero di confronti è costante anche in caso di parziale ordinamento degli elementi. Il numero di scambi dipende dal parziale ordinamento dell'array.

Per realizzare L' algoritmo si utilizzano due cicli annidati:

1. Il primo con un indice i che va da 0 a n-2
2. Il secondo con un indice j che va da i+1 a n-1

E all'interno del secondo ciclo un confronto tra vettore[i] e vettore[j] e se si verifica la condizione si effettua lo scambio tra gli elementi.

```
public void exchangeSort(int[] v) {
    int i, j;
    i = 0;
    int dim = v.length;
    while (i < dim - 1) {
        j = i + 1;
        while (j < dim) {
            if (v[i] > v[j]) {
                int tmp = v[i];
                v[i] = v[j];
                v[j] = tmp;
            }
            j++;
        }
        i++;
    }
}
```

Oppure usando il ciclo for

```
public void exchangeSort(int[] v) {
    int dim = v.length;
    for (int i = 0; i < dim - 1; i++) {
```

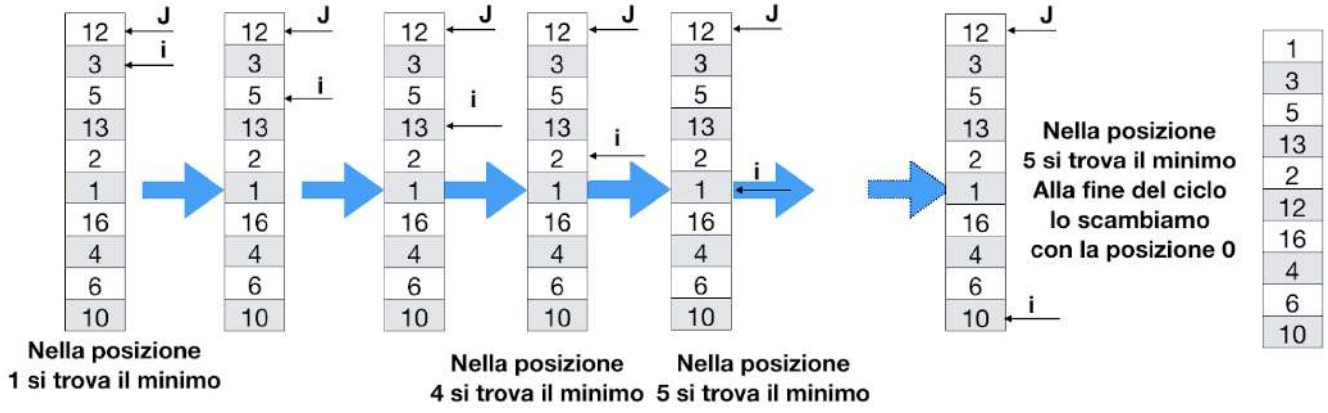
```
        for (int j = i + 1; j < dim; j++) {
            if (v[i] > v[j]) {
                int tmp = v[i];
                v[i] = v[j];
                v[j] = tmp;
            }
        }
    }
}
```

Si può osservare che essendo l'array un oggetto viene passato per riferimento quindi la modifica dentro la funzione si ripercuote anche fuori dalla funzione.

Selection Sort

L'algoritmo precedente non è efficace a causa degli elevati scambi. Dato che la strategia consiste nell'inserire l'elemento minore a sinistra lo si migliora:

1. Prima trovando l'elemento minimo
2. E solo a fine ciclo si fa lo scambio.



Per realizzare tale algoritmo si usano due cicli annidati:

1. Il primo con un indice j che va da 0 a n-2
2. Il secondo con un indice i che va da j+1 a n-1

Prima di entrare nel secondo ciclo si assegna a una variabile k il valore dell'indice j.

Nel ciclo interno si confronta vettore[i] con vettore[k] se si verifica la condizione si memorizza la posizione i in k.

Alla fine del ciclo se j e k sono diversi si effettua lo scambio tra gli elementi.

```
public void selectionSort(int[] v) {
  let dim = v.length;
  for (int j = 0; j < dim - 1; j++) {
   int k = j;
    for (int i = j + 1; i < dim; i++) {
      if (v[i] < v[k])
        k = i;
    }
    if (k != j) {
      int tmp = v[j];
      v[j] = v[k];
      v[k] = tmp;
    }
  }
}
```

La classe Arrays

L'algoritmo selection sort non è l'algoritmo di ordinamento più efficiente.

Quando l'efficienza è una caratteristica importante, è opportuno utilizzare un algoritmo più complesso ma anche più efficiente. Fortunatamente la classe Arrays del package java.util, definisce il metodo statico sort. Dato unArray, un array di valori primitivi od oggetti, l'istruzione:

```
Arrays.sort(unArray);
```

ordina gli elementi dell'intero array in senso crescente. Per ordinare la sola porzione di array compresa fra l'indice inizio e l'indice fine, basta scrivere:

```
Arrays.sort(unArray, inizio, fine);
```

La classe Arrays fornisce diverse versioni del metodo per gestire sia array di classi sia array di tutti i tipi primitivi. Se si vuole ordinare un array di oggetti con Arrays.sort, l'oggetto deve implementare l'interfaccia Comparable.

La classe Arrays contiene molti metodi (tutti statici) per manipolare gli array. Ci sono metodi per effettuare inizializzazioni, confronti, ricerche e ordinamenti, per trasformare un array in una lista e per ottenere una stringa dal contenuto dell'array (in modo da poter stampare il contenuto dell'array senza ricorrere ad un ciclo).

Per un array di numeri interi:

static void fill(int[] a, int val) assegna il valore specificato a ogni elemento dell'array;

public static boolean equals(int[] a, int[] a2) confronta i due array e restituisce true se sono uguali; due array sono considerati uguali se contengono gli stessi elementi nello stesso ordine.

static int binarySearch(int[] a, int key) cerca il valore indicato come key nell'array usando l'algoritmo di ricerca binaria;

static void sort(int[] a) ordina l'array;

static String toString(int[] a) restituisce una stringa che rappresenta il contenuto dell'array.

Esistono i metodi corrispondenti per gli altri tipi elementari e per il tipo Object.

Prova il Codice

Esercizi

1. Dato un array scrivere un metodo che sommi tutti gli elementi dell'array.
2. Progettare un programma che permetta d'indicare se uno studente è sufficiente o meno di una materia basandosi sulla media dei voti della materia. Caricare i voti della materia in un array.
3. Progettare un programma che converta un numero binario in un numero in base 10. Il numero binario è rappresentato su N bit, e il valore di N dovrà essere fornito dall'utente. L'utente dovrà inserire le cifre del numero binario un bit alla volta, partendo dal bit meno significativo (ossia dal bit di peso 2^0). Il programma visualizza il numero in base 10 corrispondente (si utilizzino gli array).
4. Progettare un programma che permetta d'indicare se uno studente è ammesso o non ammesso alla classe successive in base alla media finale. La media finale dovrà essere calcolata come media delle medie di tutte le materie e, se risulterà >= 6

allora lo studente sarà ammesso alla classe successiva, altrimenti lo studente non sarà ammesso alla classe successiva.

5. Progettare un programma che acquisisca da tastiera un vettore d'interi di dimensione N e calcoli minimo, massimo e media degli elementi.

   Si costruiscono tre metodi di riduzione una per il min una per il max e una per la media.

6. Dato un vettore di valori numerici, progettare un programma che conti quanti elementi di un array sono compresi tra un valore minimo e un valore massimo forniti da tastiera, visualizzando sia il conteggio, sia l'elenco dei valori numerici.

7. Progettare un programma che permetta di leggere 10 numeri interi e visualizzi la sequenza memorizzata senza le eventuali ripetizioni. Ad esempio, se nel vettore fossero memorizzati i valori: 15, 3, 5, 3, 11, 5, 15, 5, 15, 11, il programma dovrà visualizzare i valori 15, 3, 5, 11.

8. Progettare un programma che memorizzi in un array le ore di studio di uno studente per ogni giorno del mese. L'algoritmo deve calcolare e visualizzare il numero totale di ore passate a studiare nel corso del mese, il numero di giorni in cui lo studente ha studiato per più ore e il numero di giorni in cui lo studente ha studiato di meno.

9. In un array di 9 elementi sono memorizzate le presenze mensili degli studenti di una classe durante l'anno scolastico. Progettare un programma che determini:
   a) la media mensile di presenze nel corso dell'intero anno scolastico;
   b) il numero totale di presenze nel primo quadrimestre;
   c) il mese in cui si è registrato il numero massimo di presenze;
   d) il mese in cui si è registrato il numero minimo;
   e) la media delle presenze del secondo quadrimestre.
10. Progettare un algoritmo che permetta d'indovinare in 6 tentativi un codice numerico composto da sei cifre nell'intervallo [100000 - 999999). Il codice numerico deve essere generato come numero pseudo casuale e deve essere salvato in un array d'interi, cifra per cifra.
11. Progettare un algoritmo che preveda il caricamento dei nomi e delle età degli studenti della tua classe e visualizzi l'elenco dei nomi e delle età degli studenti maggiorenni.
    Sviluppare anche i seguenti punti:
    a. *visualizzazione del nome e l'età del più vecchio;*
    b. *visualizzazione del nome e l'età del del più giovane;*
    c. *visualizzare la media delle età;*
    d. *visualizzare l'elenco dei nomi e le età degli studenti che hanno un'età maggiore della media;*
    e. *visualizzare l'elenco dei nomi e le età che hanno un'età minore della media.*
12. Progettare un algoritmo che preveda il caricamento dei cognomi e delle età degli studenti della tua classe e visualizzi prima l'elenco dei cognomi degli studenti maggiorenni e poi l'elenco dei cognomi degli studenti minorenni.
13. Progettare un algoritmo che determini in un gruppo di persone quali risultano essere sottopeso, quali normopeso, quali in sovrappeso e quali obese. Si visualizzino gli elenchi delle persone che ricadono in ognuno dei gruppi. visualizzino gli elenchi delle

persone che ricadono in ognuno dei gruppi.

Si consideri che l'Indice di Massa Corporea (IMC) di una persona si determina con la seguente formula:

IMC = peso (kg) / altezza^2 (m^2)

e che l'Organizzazione Mondiale della Sanità individua le seguenti quattro categorie:

a) sottopeso: IMC inferiore a 19;
b) normopeso: IMC nell'intervallo [19, 24];
c) sovrappeso: IMC nell'intervallo (24, 30];
d) obeso: IMC superiore a 30.

14. Uno strumento di misura fornisce un dato ogni minuto nell'arco di un'ora. Per ovviare a possibili errori si vogliono elaborare i valori rilevati sostituendoli con una media a tre punti: ogni elemento viene sostituito dalla media di sé stesso, dell'elemento che lo precede e quello che lo segue. Per i due elementi estremi viene considerato due volte il valore dell'elemento stesso e una volta il successivo o il precedente nel caso si tratti rispettivamente del primo o dell'ultimo elemento. Progettare un algoritmo in grado di visualizzare i valori finali.

15.

Programmazione orientata agli Oggetti

Java è un linguaggio di programmazione orientato agli oggetti (OOP).

La OOP è una metodologia di programmazione che considera il programma come costituito da oggetti (o istanze) che possono interagire fra loro. il suo stato. I metodi rappresentano le funzionalità che l'oggetto mette a disposizion

In un programma, un oggetto può rappresentare un oggetto reale o una sua astrazione.

Gli oggetti Java come gli oggetti del mondo reale hanno due caratteristiche:

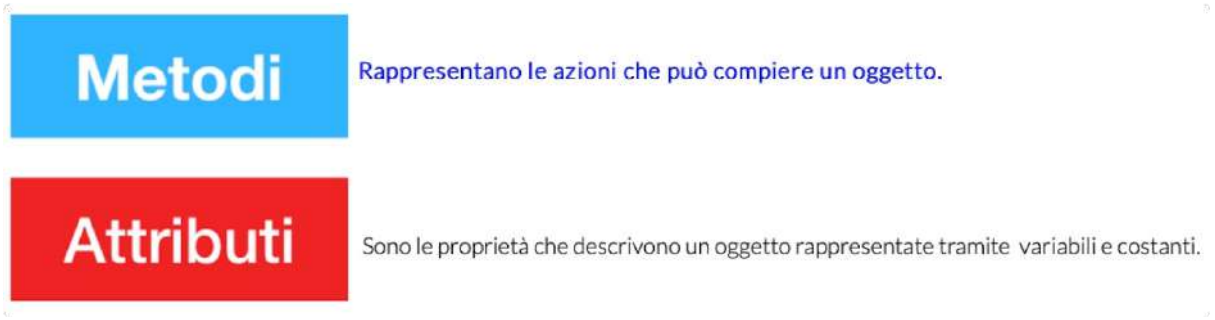
1. Attributi rappresentano gli elementi che caratterizzano l'oggetto, utili per descrivere le sue proprietà e definirne lo stato. Ad esempio, un oggetto Persona ha come attributi:
    - nome, cogome sesso, ecc..
2. Metodi ossia le azioni che può compiere:
    - studia, cammina, parla ecc...

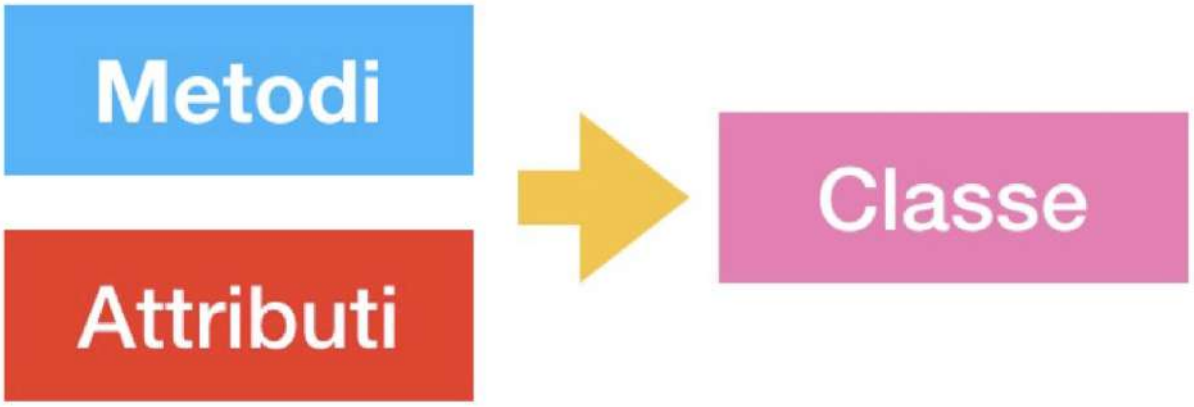
La programmazione basata sugli oggetti ha come obiettivo principale la creazione di nuovi tipi di dati denominati classi. Le classi sono progettate con lo scopo di modellare in astratto degli oggetti del mondo reale.

Che è l'elemento minimo di un programma Java.

classe

Rappresenta un tipo di dato composta da:





La programmazione a oggetti permette di:

- rappresentare un problema o delle entità reali attraverso oggetti software
- stabilire le relazioni che intercorrono tra le entità

Gli oggetti di uno stesso tipo condividono lo stesso tipo di dato. Tutti gli oggetti di una classe hanno gli stessi attributi e lo stesso comportamento.

Quando si definisce una classe, si costruisce il modello di un oggetto.

Ad esempio se si vuole creare un oggetto di tipo Moto.

Una moto è piuttosto complessa:

Se si sta creando un videogioco di corse, si ha bisogno di:

Velocità massima, caratteristiche di manovrabilità, colore. Questo è il modello di una moto per il gioco.

E i metodi accellera() frena ecc..

Se si realizza un programma per un negozio di moto le informazioni sono diverse:

Ad esempio targa anno 'immatricolazione prezzo ecc...

e i metodi: vendi() immatricola() ecc...

Il modello è diverso.

Per costruire un oggetto di tipo moto si costruisce una classe con:.

Attributi: il colore, la marca, la velocità, se è accesa.

Come si vede, queste caratteristiche possono descrivere le proprietà fisiche dell'oggetto, come il colore. Possono anche indicare lo stato dell'oggetto in un determinato momento, possiamo sapere se la moto è accesa o spenta.

Metodi. Si muove, Accelera, Frena, Si spegne

Nome classe: Moto
Attributi(Dati):
colore_____
marca_____ _____
velocità _____
targa _____
accesa_____
Metodi (azioni):
accelera
Come:
girando la manopola in alto
decelera
Come:
girando la manopola in basso

Classe moto, descrive una moto generica

Una classe specifica gli attributi, o dati, degli oggetti della classe. La definizione della classe Moto indica che un oggetto di tale classe ha cinque attributi:

- una stringa che indica il colore
- una Stringa indica la marca della moto
- un intero che indica la velocità
- una stringa che indica la targa
- un valore booleano che indica se il motore è acceso

La definizione di una classe non specifica il valore degli attributi, questi sono specifici dei singoli oggetti; la classe specifica solamente il tipo (di dato) di questi attributi.

Una classe, inoltre, specifica le azioni che possono essere svolte dagli oggetti e come queste azioni vengono svolte.

Per esempio, la classe Moto specifica quattro azioni:

- accelera
- decelera
- accendi
- spegni

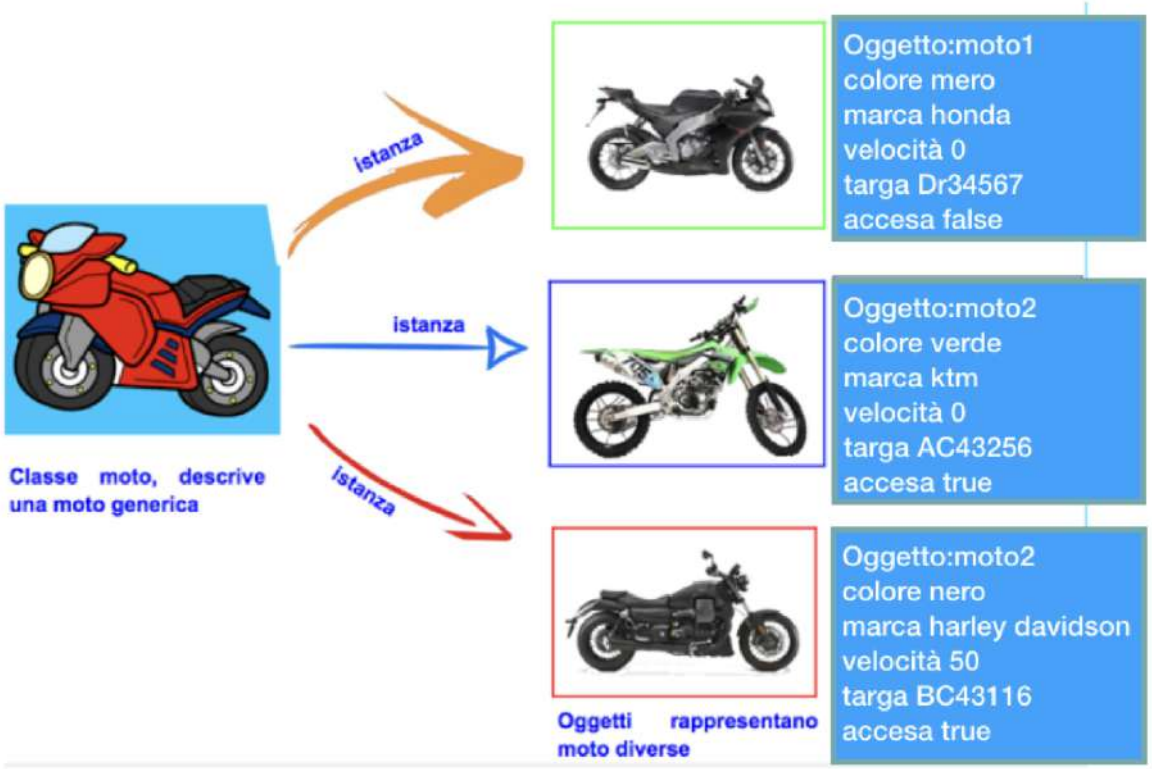
Queste azioni sono descritte all'interno della classe per mezzo di metodi.

Tutti gli oggetti di una classe hanno gli stessi metodi. Le definizioni dei metodi fanno parte della classe, essi descrivono il modo in cui gli oggetti svolgono le azioni.

La classe può essere immaginata come uno stampo dal quale vengono creati gli oggetti, tutti con gli stessi attributi e gli stessi metodi, un oggetto può esistere solo se esiste la relativa classe che ne descrive le caratteristiche e le funzionalità.

Istanze

Gli oggetti di questa classe rappresentano moto specifiche. Per creare oggetti si deve istanziare la classe, le varie istanze sono completamente indipendenti l'una dall'altra e quindi se si creano tre oggetti di tipo "Moto" si ottengono tre Moto(oggetti) diverse.



si dice che l'oggetto moto1 è un'istanza della classe Moto. Allo stesso modo l'oggetto moto2 è un'istanza della classe Moto. Quindi la stessa classe può generare più istanze che differiscono per il valore assunto dai suoi attributi, ma tutte possono utilizzare i metodi della classe.

La struttura base della dichiarazione di una classe in Java è la seguente:

```
class NomeClasse{
// attributi
// metodi
}
```

La parola chiave class serve per iniziare la dichiarazione di una classe ed è seguita dal nome della classe. Per convenzione, i nomi delle classi si indicano con la lettera iniziale maiuscola.

Tra le parentesi graffe si inserisce tutto il contenuto della classe, costituito dagli attributi e dai metodi. Naturalmente possono esistere classi formate da soli attributi oppure da soli metodi.

Variabili d'istanza

Le variabili d'istanza sono quelle variabile che sono definite all'interno di una classe(attributi), ma fuori dai metodi della classe stessa.

La sintassi per la dichiarazione di una variabile è la seguente:

[modificatori] tipo nome [= inizializzazione];

dove:

Modificatori: parole chiavi di Java che consentono di definire il livello di accesso degli attributi o dei metodi della classe (public, private, protected,final).

public	Specifica che l'attributo o il metodo è accessibile in qualunque blocco di codice anche esterno alla classe
private	**Specifica che** l'attributo o il metodo è accessibile soltanto all'interno della classe
protected	Specifica che l'attributo o il metodo è accessibile all'interno della classe ed all'interno delle classi che ereditano dalla nostra classe
default	**nessun valore (default):** Specifica che l'attributo o il metodo è accessibile solo all'interno della classe ed all'interno delle altre classi che fanno parte dello stesso package.

MODIFICATORE	STESSA CLASSE	STESSO PACKAGE	SOTTOCLASSE	OVUNQUE
public	SI	SI	SI	SI
protected	SI	SI	SI	NO
nessun modificatore	SI	SI	NO	NO
private	SI	NO	NO	NO

tipo: Data type della variabile.

nome: il nome della variabile

Gli attributi della la classe moto sono:

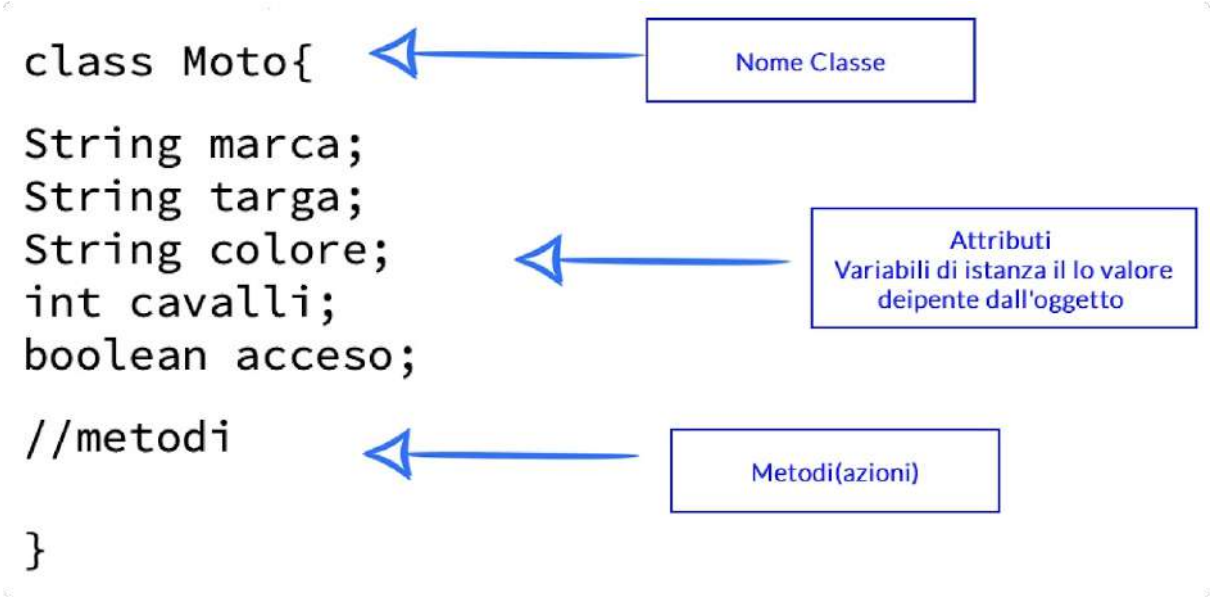
1. una Stringa indica la marca della moto

2. una stringa che indica la targa.

3. una stringa che indica il colore

4. un numero intero che indichi i cavalli

5. un valore booleano che indica se il motore è acceso o spento.

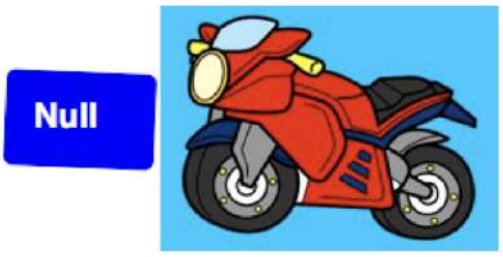


Per creare un oggetto si deve istanziare la classe:

Per prima cosa si definisce una variabile usando come data Type il nome della classe.

```
NomeClasse nomeVariabile

Moto ktm;
```

Questa istruzione dichiara che la variabile ktm è un riferimento (reference) a un oggetto della classe Moto.

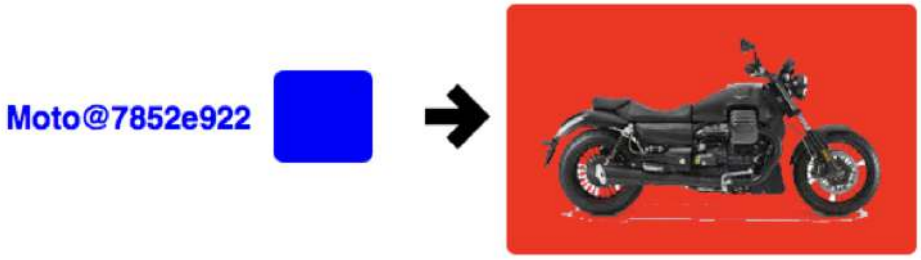


Usando l'operatore **new** si crea il nuovo oggetto.

ktm= new Moto();

new Moto ( );

crea e inizializza un nuovo oggetto il cui indirizzo viene quindi assegnato a ktm.



Le variabili d'istanza assumono i valori di default perché non gli è stato assegnato nessun valore.

Variabile	Valore
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000' (NULL)
boolean	false
Ogni tipo reference	null

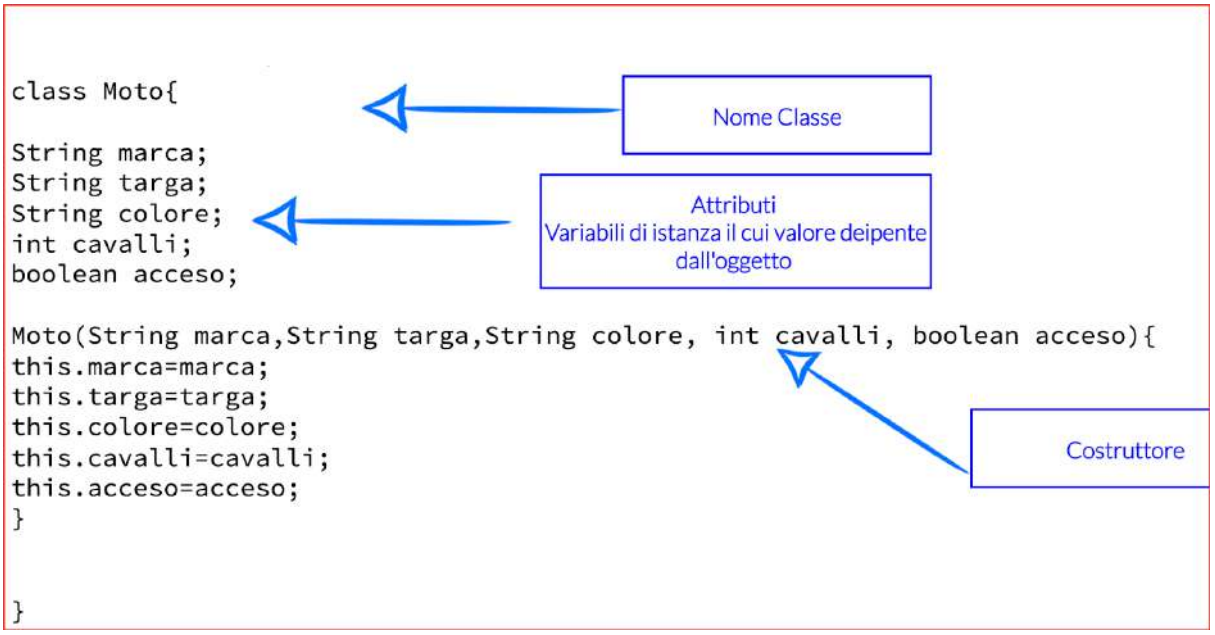
Quando si crea un oggetto java esegue il metodo costruttore. Se questo non è definito come nel nostro caso java crea un costruttore vuoto.

Costruttore

Quando si crea un oggetto di una classe utilizzando l'operatore new, si invoca un particolare tipo di metodo chiamato **costruttore**, **il quale deve avere lo stesso nome della classe**, e può avere dei parametri che utilizzerà per inizializzare, le variabili d'istanza se non inseriamo un costruttore java ne inserisce uno di default vuoto.

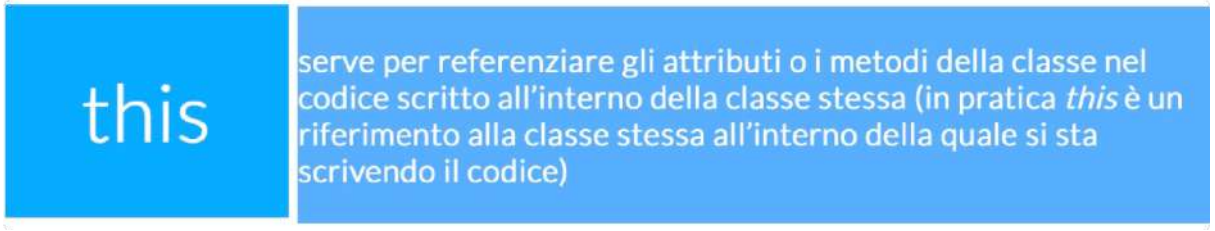
Quando si definisce un costruttore, non si specifica nessun tipo di ritorno. I costruttori forniscono un valore a tutte le variabili d'istanza, anche se non hanno un parametro per ognuna di esse. Se il costruttore non inizializza una particolare variabile d'istanza, lo farà Java, assegnando un valore di default. In ogni modo, quando si definisce un costruttore, è una normale pratica in programmazione assegnare esplicitamente un valore a tutte le variabili d'istanza.

La classe moto con il costruttore:



Il costruttore possiede i parametri d'input: String marca, String targa, String colore, int cavalli, boolean acceso (indicati tra le parentesi dopo il nome)

- il corpo del costruttore imposta il valori degli attributi assegnandogli i valori dei parametri d'ingresso.
- I parametri d'ingresso hanno lo stesso nome degli attributi.
- Per risolvere il conflitto di nomi tra gli attributi della classe Moto e i parametri d'ingresso viene utilizzata la parola chiave this.

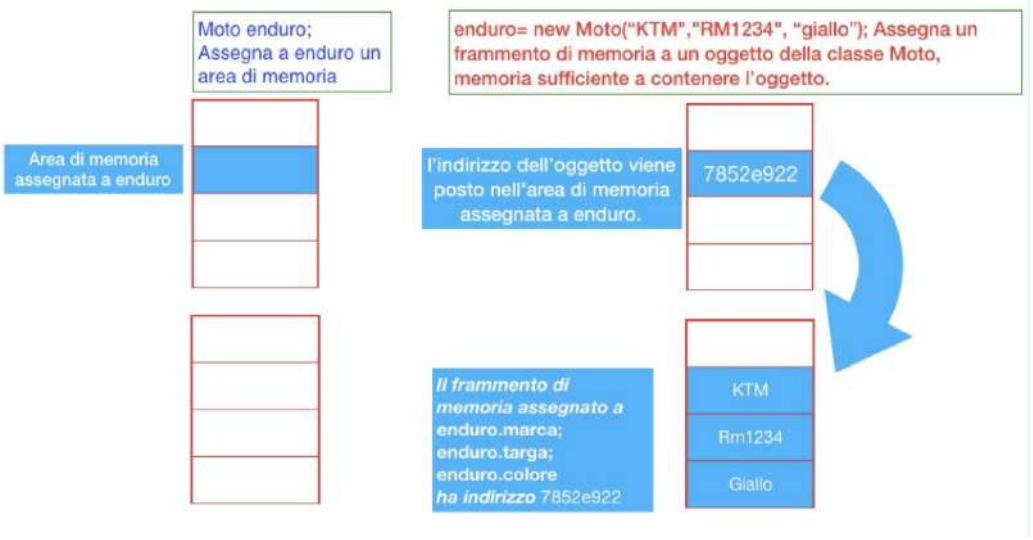


Quando si invoca il costruttore il compilatore alloca una quantità di memoria idonea a contenere l'oggetto e ritorna nella variabile ktm il suo riferimento. Si può dunque dire che il processo di creazione di un oggetto si attua in due passaggi:

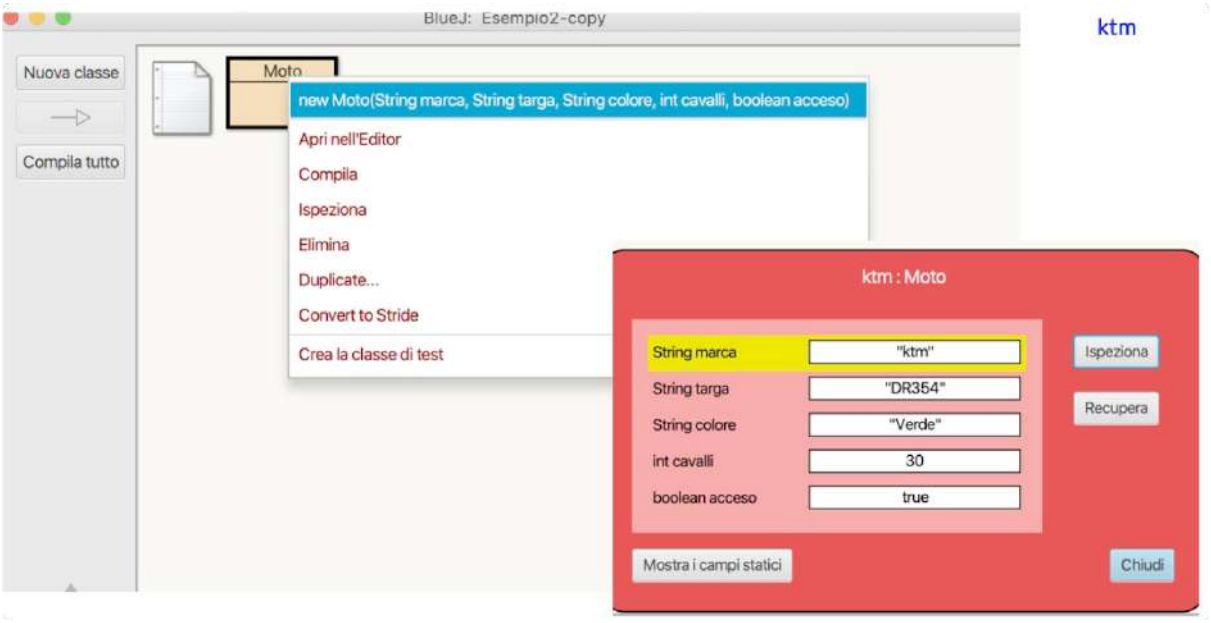
1. con una dichiarazione dove si dice che una variabile è di un certo tipo e che può contenere un oggetto di quel tipo;

2. con una definizione dove si alloca uno spazio di memoria che contiene l'oggetto creato (new) e il cui riferimento viene salvato nella variabile.

Un oggetto può avere al proprio interno più variabili, le sue variabili d'istanza.



L'oggetto km diventa:



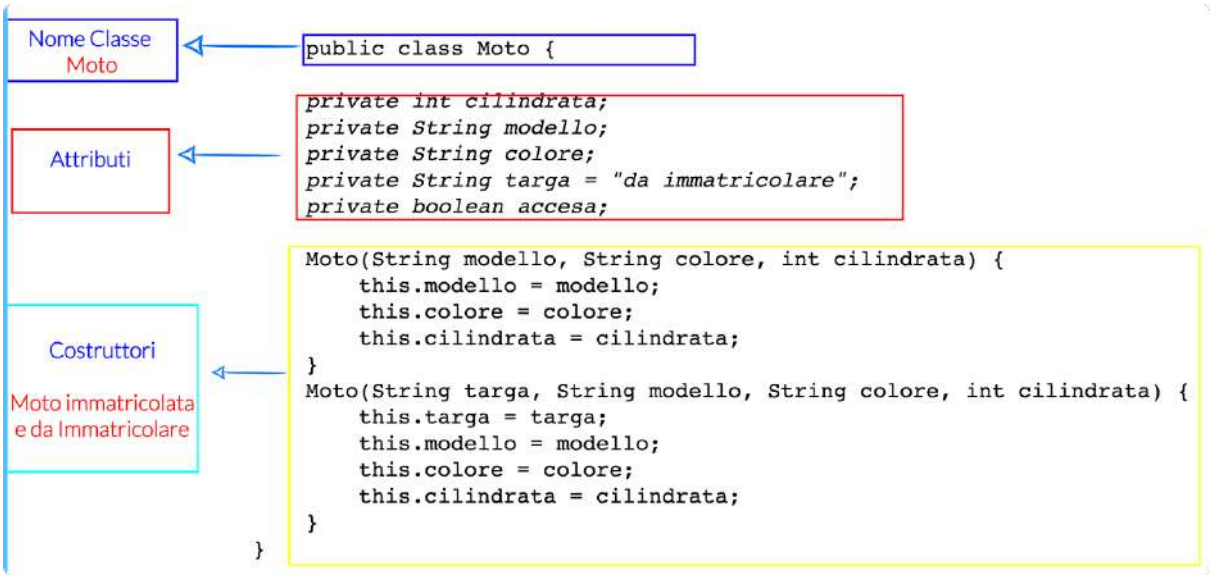
Ricapitolando:

1 Se non viene creato esplicitamente un costruttore, il compilatore provvederà in autonomia a crearne uno vuoto, definito costruttore di default.

2 Il costruttore ha lo scopo d'inizializzare le variabili d'istanza dell'oggetto creato, al fine di porle in uno stato consistente.

3 Se non si esplicita un valore da dare alle variabili d'istanza, il compilatore provvederà automaticamente a inizializzare con i valori di default.

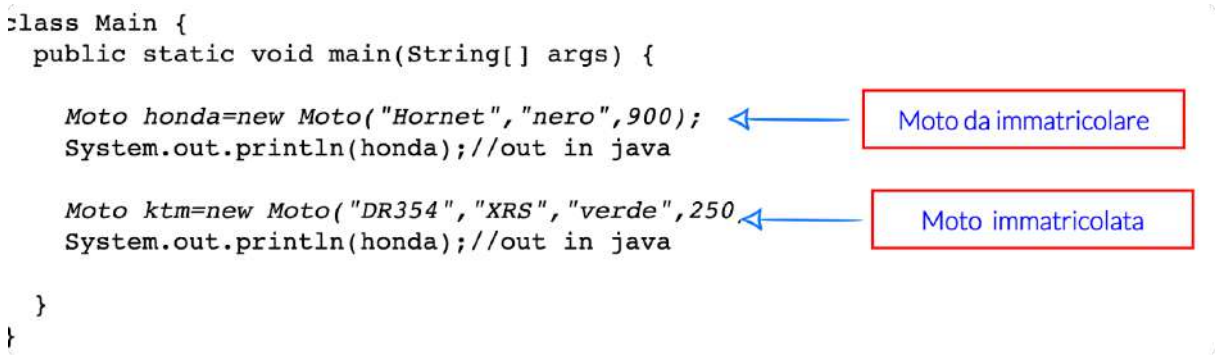
Inoltre i costruttori possono essere sovraccaricati, ovvero si possono scrivere più costruttori con parametri differenti per tipo, per numero e per posizione. Ciò consente di creare l'oggetto passando una varietà d'inizializzatori; a seconda del numero, tipo e ordine degli argomenti, il compilatore invocherà il costruttore corretto (con lo stesso numero, tipo e ordine dei parametri). L'utilizzo delle varie forme di costruttori ci permette d'introdurre un concetto tipico della programmazione object oriented: l'overloading di metodi. Si parla di overloading di metodi quando esistono nella stessa classe metodi che hanno lo stesso nome ma un differente numero di parametri.




```
public class Moto {

    private int cilindrata;
    private String modello;
    private String colore;
    private String targa = "da immatricolare";
    private boolean accesa;

    Moto(String modello, String colore, int cilindrata) {
        this.modello = modello;
        this.colore = colore;
        this.cilindrata = cilindrata;
    }
    Moto(String targa, String modello, String colore, int cilindrata) {
        this.targa = targa;
        this.modello = modello;
        this.colore = colore;
        this.cilindrata = cilindrata;
    }

}
```

Solitamente tutte le variabili d'istanza sono dichiarate private.

Per creare oggetti di tipo Moto creiamo una classe Main con il metodo static main.



Codice

Eseguendo il programma precedente si ottiene come output:

```
Moto@2a139a55
Moto@15db9742
```

I metodi:

print(oggetto) e println(oggetto)

stampano lo stato dell'oggetto **trasformandolo in una stringa**. Questo succede perché viene richiamato il metodo:

toString()

Della classe Object da cui tutte le classi Java ereditano.

Il metodo toString() ereditato, come rappresentazione dell'oggetto, restituisce una stringa con il nome della classe a cui appartiene l'oggetto seguito dall'indirizzo dell'oggetto in memoria.

Per ottenere una stringa più appropriata bisogna ridefinire il metodo.

Dichiarazione e implementazioni dei metodi

metodo toString

La dichiarazione o firma di un metodo è così composta:

modificatoreDiAccesso tipoDiRitono nomeMetodo(parametri metodo)

Per riscrivere il metodo toString della classe Object

La firma è:

public String toSting()

ovvero stiamo dichiarando che:

- è public, quindi accessibile in qualunque altra parte del codice (anche all'esterno della classe Moto)
- il metodo deve ritornare un valore di tipo String (parametro di output del metodo). Il codice del metodo deve terminare con **un'istruzione return** che restituisca al chiamante un valore di tipo String
- il nome del metodo è toSting
- (): il metodo non ha parametri d'ingresso. Le parentesi tonde sono infatti vuote

Dopo la firma del metodo segue il corpo del metodo delimitato dalle parentesi graffe:

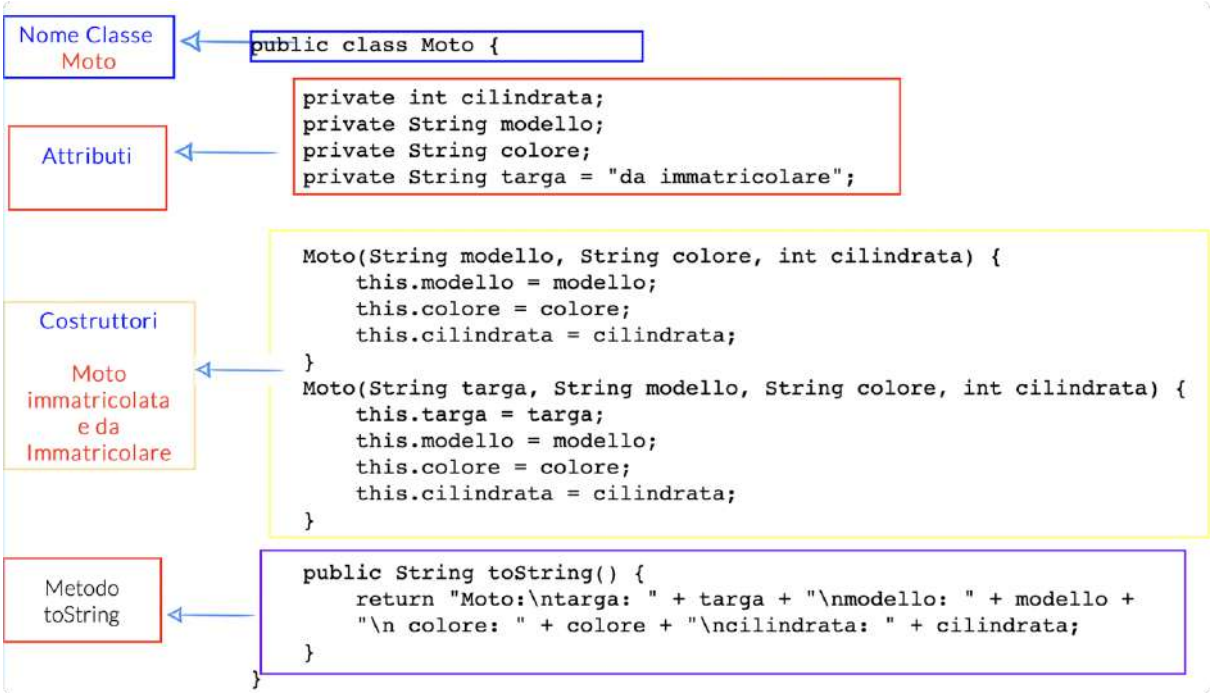
```
return "Moto: targa: "+targa +" modello: "+modello+" colore: "+colore+" cilindrata: "+cilindrata;
```

Il metodo esegue una sola istruzione: restituisce lo stato dell'oggetto.

Nota:

Non esiste una forma "standard" per la rappresentazione di un oggetto. Dipende molto dal significato della classe, dal suo contenuto e soprattutto da dove si intende usare la rappresentazione testuale.

La classe moto con il metodo toString.



Codice

Metodo set

L'oggetto Moto costruito è un oggetto immutabile. Una volta creato non può essere modificato in nessun modo. Per poterlo modificare si devono scrivere dei metodi chiamati **set** la cui firma è:

public void setAttributo(Tipo attributo)

Ad esempio:

```
public void setTarga(String targa) {
this.targa= targa;
}
```

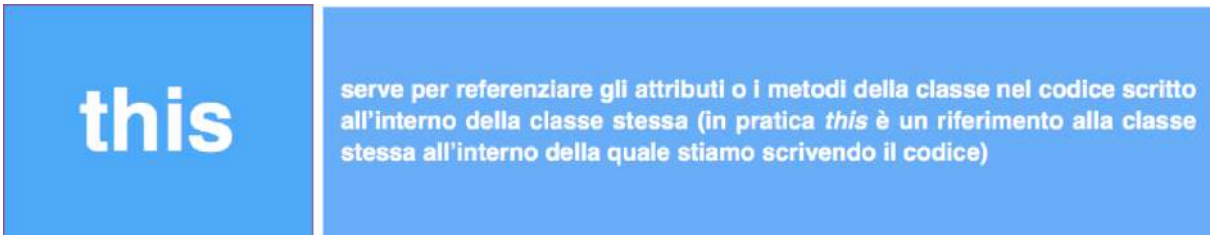
possiamo notare:

o il tipo di ritorno del metodo void. Questa è la parola chiave utilizzata in Java per indicare che il metodo non ritorna nulla (il metodo infatti serve per impostare un nuovo valore per la variabile colore, ma non deve restituire nulla)

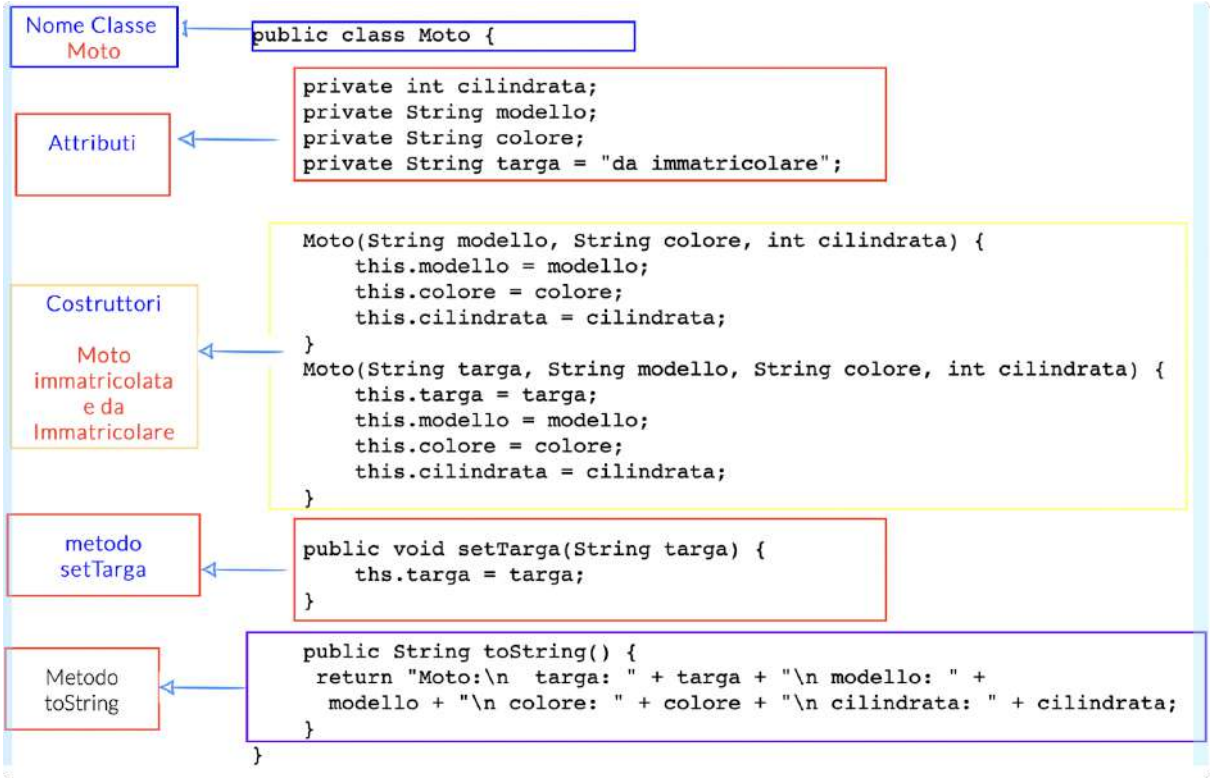
o Il nome formato dalla parola set seguito dal nome dell'attributo in maiuscolo (convenzione java):

o il metodo possiede un parametro di ingresso (o di input): String targa (indicato tra le parentesi dopo il nome del metodo)

o il corpo del metodo imposta il valore dell'attributo targa assegnandogli il valore del parametro di ingresso. Da notare che il parametro di ingresso si chiama anch'esso targa. Per risolvere il conflitto di nomi tra l'attributo della classe Moto ed il parametro di ingresso del metodo viene utilizzata la parola chiave **this.**



In modo analogo si creano metodi set per tutti gli attributi che si possono modificare della Classe.



Il Dot-Operator

Per accedere ai metodi o agli attributi di una classe java mette a disposizione il Il dot-Operator


```
class Main {
  public static void main(String[] args) {
    Moto honda = new Moto("Hornet", "nero", 900);
        System.out.println(honda);// out in java

        System.out.println(" Immatricoliamo la moto e inseriamo la targa");
        honda.setTarga("ad456");


        System.out.println(honda);
        }
}
```

Moto da immatricolare

Inseriamo la targa

Codice

Metodo get

Per ottenere i dati contenuti in una variabile d'istanza si usa il metodo get la cui firma è:

public Tipo getAttributo()

Ad esempio:

```
public String getTarga() {
return targa;
}
```

possiamo notare:

o Il tipo di ritorno è il tipo dell'attributo di cui si vuole sapere il valore. Nel nostro caso String

- Il nome è formato dalla parola get seguita dal nome dell'attributo in maiuscolo (convenzione java).
- Il metodo non possiede un parametro d'ingresso (o di input).
- Il corpo del metodo contiene il return dell'attributo.

In modo analogo si creano i metodi get per tutti gli attributi della Classe.

Normalmente tutti i metodi sono public. Se un metodo deve essere usato solo dagli altri metodi della sua classe, allora dovrebbe essere reso privato.Tutte le variabili d'istanza dovrebbero essere dichiarate private. In questo modo si costringe chi usa la classe ad accedere alle variabili d'istanza solo attraverso i metodi della classe. Questo permette alla classe di controllare tutte le attività di lettura e scrittura dei valori delle variabili d'istanza.

Scope

La portata e la durata delle variabili

Java consente di dichiarare le variabili all'interno di qualsiasi blocco(un blocco inizia con una parentesi graffa aperta e termina con una parentesi graffa chiusa) che definisce un ambito. Un ambito determina quali oggetti sono visibili ad altre parti del programma.

In Java, i due gli ambiti principali sono quelli:

1. **Definiti da un metodo.**
2. **Definiti da una classe.**

Le variabili dichiarate all'interno di un ambito non sono visibili (cioè, accessibili) al codice definito all'esterno di tale ambito. Quindi, quando si dichiara una variabile all'interno di un ambito, si localizza quella variabile proteggendola da accessi e/o modifiche non autorizzate. Una variabile dichiarata all'interno di un blocco è chiamata a variabile locale.

▪ Variabili locali: Sono create quando un metodo viene chiamato e cancellate dalla memoria quando il metodo termina.

```
package com.sistemi.sintassi;
public class variabili
{
 private static String classe ="Ciao io sono una variabile di Classe";
 public String istanza="Ciao io sono una variabile di Istanza";
 variabili(){
   visualizza();
    }
public static void main(String[] args)
   {
     String  locale = "Ciao io sono una variabile Locale";
    // stampa qualcosa…
         System.out.println(locale);
       System.out.println(classe );
       variabili a=new variabili();
      }
  public void visualizza(){
     System.out.println(istanza);
  }
}
```

La dichiarazione della variabile può avvenire ovunque all'interno di un blocco di codice, il quale è rappresentato da un gruppo d'istruzioni poste tra le parentesi graffe di apertura { e chiusura }, e dopo tale dichiarazione la variabile medesima può essere utilizzata. Se si creano blocchi annidati, la variabile del blocco più esterno è visibile all'interno del blocco interno, ma non vale il contrario.

```
12    public class BlocchiAnnidati {
13
14        public static void main(String[] args) {
15            int x = 20;
16            if (x < 20) {
17                int y = 11;
18                System.out.println("X " + x);
19            }
              System.out.println("Y " + y); // errore 'y' non è visibile
21        }
22    }
```

cannot find symbol
symbol: variable y
location: class BlocchiAnnidati

(Alt-Enter shows hints)

Parametri formali

Sono quelle variabili che vengono dichiarate all'interno delle parentesi tonde di un determinato metodo.

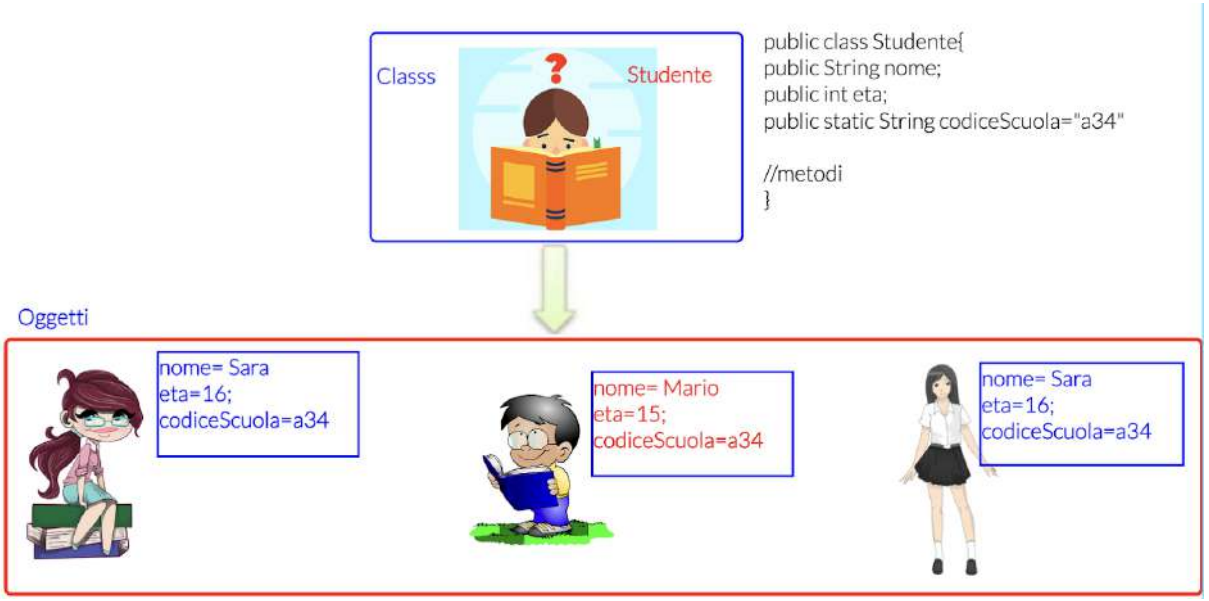
```
package com.sistemi.sintassi;
public class variabili
{
 private static String classe ="Ciao io sono una variabile di Classe";
 public String istanza="Ciao io sono una variabile di Istanza";
 variabili(){
   visualizza();
    }
public static void main(String[] args)          Parametri
   {
     String  locale = "Ciao io sono una variabile Locale";
    // stampa qualcosa…
        System.out.println(locale);
       System.out.println(classe );
       variabili a=new variabili();
      }
  public void visualizza(){
     System.out.println(istanza);
  }
}
```

Variabili di classe (static)

Le variabili di classe infine, comunemente dette anche static field o campi statici, sono variabili d'istanza ma nella loro definizione viene usata la keyword ‘static’.

static int v = 6;

Una variabile di classe è una variabile visibile da tutte le istanze di quell’oggetto e il suo valore non cambia da istanza a istanza, per questo appartiene trasversalmente a tutta la classe.



Codice

- **static applicata ad un attributo:** tutti gli oggetti della classe che vengono istanziati condividono lo stesso attributo.

Per accedere a un attributo static della classe si usa la sintassi **nomeClasse.nomeAttributo.**

- **static applicata a un metodo:** indica che il metodo è accessibile utilizzando direttamente il nome della classe e non necessita di un oggetto istanziato. La sintassi sarà quindi **nomeClasse.nomeMetodo**

	static data member	non-static data member
static methods	Yes	No
non-static methods	Yes	Yes

È possibile definire un blocco static che viene eseguito una sola volta (quando la classe viene portata in memoria) in tale blocco si inseriscono di solito delle operazioni per inizializzare le variabili static.

```
class moto {
static int numeroDiMoto;

static{
numeroDiMoto=1;
}
```

Esempio di Codice

Un'osservazione da fare riguarda la possibilità di conflitti di nome su di una variabile che ne alterano lo scope. Quando una variabile locale ed una globale hanno lo stesso nome (e sono contemporaneamente utilizzabili anche in base ai privilegi determinati dal qualificatore di accesso), la variabile locale ha sempre priorità su quella globale.
Esempio di conflitto di nome nello scope di una variabile

```
public class Scope {
static int numero=10;
    public static void main(String[] args) {

            stampaNumero();
            stampaNumero2();
    }
    public static void stampaNumero() {
        int numero = 5;
        System.out.println("Numero = " + numero);
}
public static void stampaNumero2() {
        System.out.println("Numero=" + numero);

    }
}
```

```
Output - JavaApplication28 (run)
run:
Numero = 5
Numero=10
BUILD SUCCESSFUL (total time: 0 seconds)
```

In esecuzione dal metodo **stampaNumero()** otteniamo la stampa della stringa "Numero= 5", **perché la variabile locale numero viene ridefinita col valore 5 ed ha priorità sulla variabile globale numero.** Invece il metodo **stampaNumero2()** da come output **"Numero= 10"**, in quanto in questo metodo la variabile numero non viene ridefinita e pertanto viene utilizzata **la variabile globale numero.**

.

Il paradigma della OOP

La programmazione a oggetti si basa su tre paradigmi fondamentali:

1. **1. Incapsulamento;**
2. **2. Ereditarietà;**
3. **3. Polimorfismo.**

Incapsulamento

Secondo il paradigma della OOP, l'incapsulamento prevede che tutto quello che riguarda un oggetto deve essere necessariamente definito al suo interno, l'utilizzatore dell'oggetto dovrebbe accedere agli attributi unicamente attraverso i metodi, evitando di accedervi direttamente.

Con l'incapsulamento si ottiene l'information hiding, ovvero il processo di nascondere il più possibile i dettagli d'implementazione al fine di ridurre la complessità di un oggetto. Il programmatore può quindi focalizzarsi sul nuovo oggetto senza preoccuparsi dei dettagli di implementazione, avendo dunque a che fare con classi più semplici.

Un oggetto dotato di buon incapsulamento possiede un'alta information hiding, poiché espone all'esterno pochi elementi. Quando un altro oggetto, nel programma, dovrà fare uso dell'oggetto ben incapsulato, utilizzerà quei pochi elementi visibili dall'esterno.

Inoltre, una classe nasconde sempre l'implementazione dei propri metodi, ovvero il codice che viene eseguito all'interno dei metodi. L'unica cosa visibile da parte del codice esterno alla classe è la firma del metodo. Potremmo dire che una classe mostra all'esterno "cosa fa, ma non come lo fa".

Chiunque può alzare la cornetta, comporre un numero telefonico e conversare con un'altra persona, ma pochi conoscono la sequenza dei processi scatenati da queste poche, semplici azioni. Evidentemente, per utilizzare il telefono, basta conoscere la sua interfaccia pubblica (costituita dalla cornetta e dai tasti), non la sua Implementazione interna.

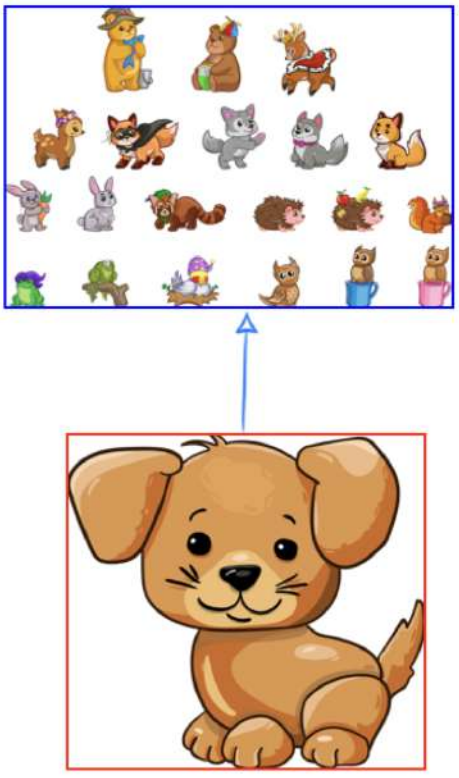
A livello d'implementazione ciò si traduce semplicemente nel dichiarare privati gli attributi di una classe e quindi inaccessibili fuori dalla classe stessa. L'accesso ai dati potrà essere fornito da un'interfaccia pubblica costituita da metodi dichiarati public.


```
class Moto {
//attributi
 static private int numeroMoto=4;
    private boolean motoreacceso;
    private String marca;
    private String colore;
    private String targa;
    //costruttore
    public Moto(String marca, String colore) {
      this.marca=marca;
      this.colore=colore;
      numeroMoto++;
    }
     /*metodi*/
    public void setTarga(String targa) {
        this.targa=targa;
    }
    void accendi() {
        if (motoreacceso == true) {
            System.out.println("Il motore e' gia' acceso");
        } else {
            motoreacceso = true;
            System.out.println("Ora il motore e' acceso");
        }
    }
}
```

Nulla vieta di utilizzare private, anche come modificatore di metodi, ottenendo così un incapsulamento funzionale. Un metodo privato infatti, potrà essere invocato solo da un metodo definito nella stessa classe, che potrebbe a sua volta essere dichiarato pubblico.

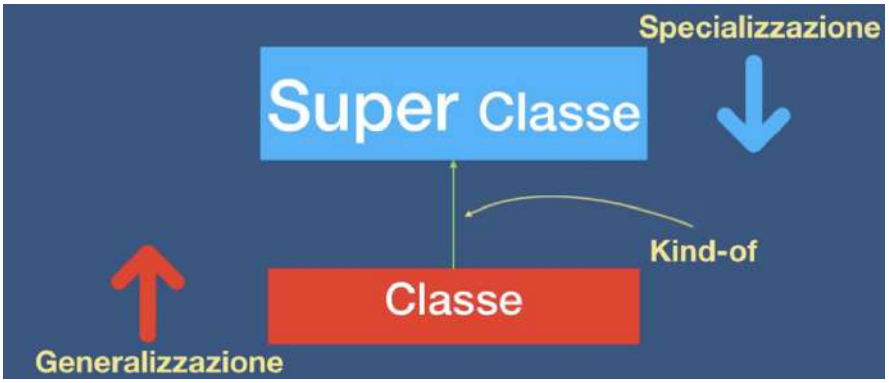
Ereditarietà

Il concetto di ereditarietà è ispirato a qualcosa che esiste nella realtà. Nel mondo reale noi classifichiamo tutto con classi e sottoclassi. Per esempio un cane è un animale, una moto è un veicolo, la chitarra è uno strumento musicale.



Questo permette di affermare che un certo concetto è un concetto particolare di un concetto più generale.

Questa relazione può essere letta in due modi dal generale al particolare e viceversa.

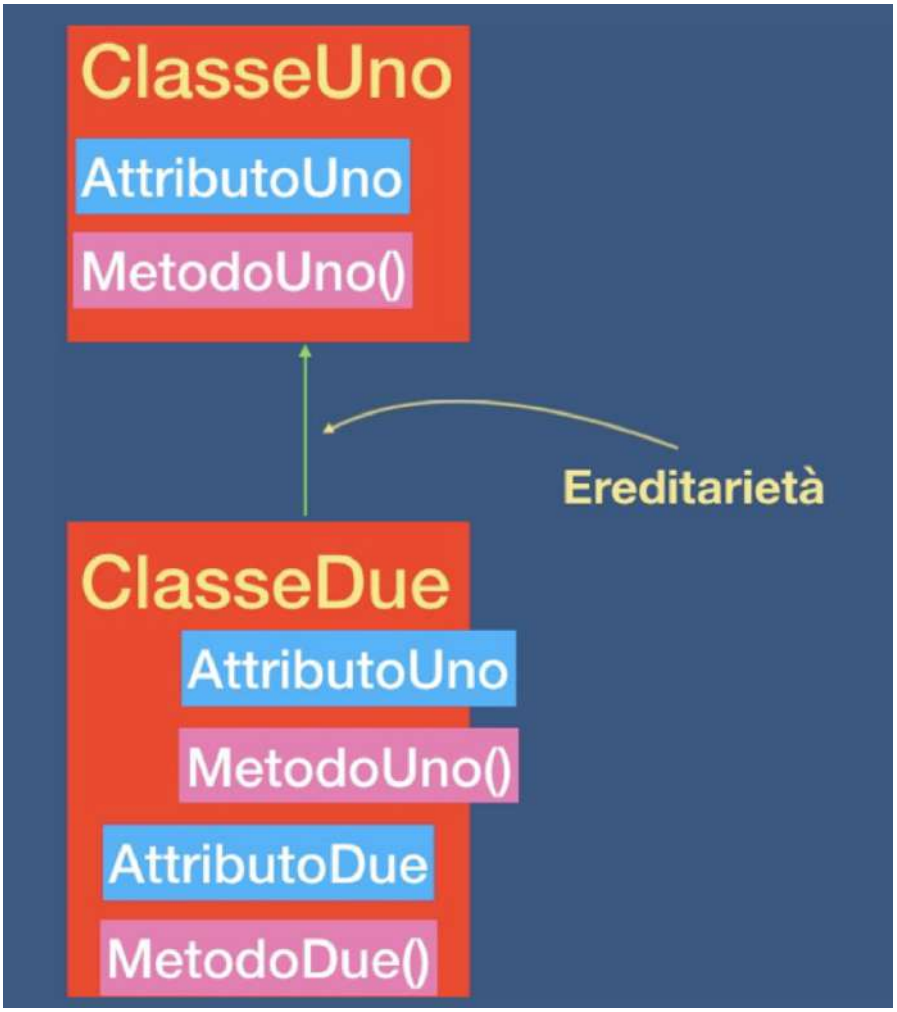


Questa è una relazione tra classi e non tra istanze.

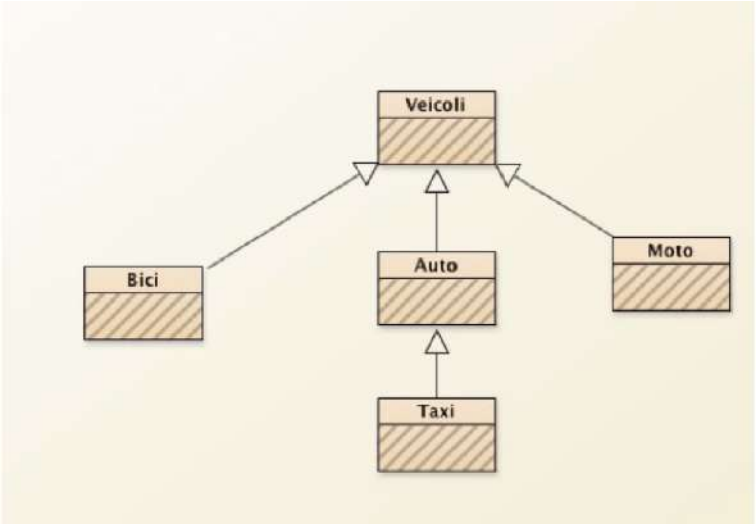
Perché ad esempio tutti i cani sono particolari tipi di Animali, ma non tutti gli animali sono cani.

La generalizzazione quando viene implementata in un linguaggio a oggetti porta al concetto di ereditarietà. Consideriamo due classi ClasseUno e ClasseDue ognuna con un suo attributo e un suo metodo:

Se ClasseDue è una sotto classe di ClasseUno allora eredita i metodi e gli attributi di ClasseUno.



Il risultato immediato è la possibilità di ereditare codice già scritto, e quindi gestire insiemi di classi collettivamente, giacché accomunate da alcune caratteristiche. L'ereditarietà, permette di creare delle gerarchie di classi.



Una classe Java **può avere un qualsiasi numero di classi figlie (e nipoti e così via) ma può essere figlia di una sola classe genitore,** ovvero **in Java non è consentita l'ereditarietà multipla.**

Le frecce indicano la relazione. In questo caso la classe Taxi è una sottoclasse della classe Auto, che è a sua volta una sottoclasse della classe Veicoli. La classe Taxi,

eredita tutti i membri pubblici e protetti sia della classe Auto sia della classe Veicoli. Infine, ricordiamo che tutte le classi Java derivano implicitamente dalla classe Object.

La parola chiave extends

In Java per indicare che una classe B eredita da una classe si usa la parola chiave extends, proprio perché B estende A. La sintassi è:

```
class SottoClasse extends SuperClasse {
//corpo della classe
}
```

Vediamo un semplice esempio di come i membri di una classe vengono ereditati da una sotto classe. Consideriamo due semplici classi:

```
Class SuperClasse{
String nome;
int numero;
}
```

```
Class SottoClasse extends SuperClasse{

}
```

Scriviamo una classe di test dove facciamo un'istanza della sottoclasse.

```
class Main {
  public static void main(String[] args) {
   SottoClasse sc= new SottoClasse();
    sc.nome="prova ereditarietà";
    sc.numero=1;
    System.out.println(sc.nome+" numero "+sc.numero);
  }
}
```

```
Output:
prova ereditarietà numero 1
```

Si può notare che usiamo un oggetto di SottoClasse per accedere agli attributi d'istanza di SuperClasse.

Codice
Nel caso precedente però non abbiamo **reso privati i due attributi contravvenendo** al principio dell'incapsulamento. Se provate a rendere privati i due attributi otterrete un errore di compilazione, questo perché:

Un oggetto della sottoclasse non deve accedere agli attributi privati della super Classe.

Per poter utilizzare questi attributi si scrive un metodo pubblico nella superclasse.

```
Class SuperClasse{
String nome;
int numero;
public void vediAttributi(){
System.out.println(nome+" "+numero;
 }
}
```

Se si esegue un test si vede che siccome le variabili non sono state inizializzate restituiscono il valore di default(null e zero). Questo problema lo si risolve con i costruttori.

```
class Main {
  public static void main(String[] args) {
    SottoClasse sc= new SottoClasse();
    sc.vistaAttributi();
  }
}
```

```
Output:
null  0
```

Ereditarietà e costruttori

Il costruttore è un metodo speciale, che possiede le seguenti proprietà:

1 ha lo stesso nome della classe;

2 non ha tipo di ritorno;

3 è chiamato automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe;

4 è presente in ogni classe;

5 Il compilatore introduce il “costruttore di default”, nel caso il programmatore non gliene abbia fornito uno in maniera esplicita;

Un’altra proprietà del costruttore è che nel caso di ereditarietà:

Il costruttore, della sottoclasse come prima istruzione, invoca sempre il costruttore della superclasse.

Quindi i costruttori vengono eseguiti in modo verticale.

Ad Esempio consideriamo le classi Moto e Veicoli così strutturate:

```
public class Veicoli {
public Veicoli()
{ System.out.println("Costruttore Veicoli");
}
}
```

```
public class Moto extends Veicoli{
public Moto()
{ System.out.println("Costruttore di una moto");
}
}
```

Se creiamo un oggetto Moto a= new Moto(); L’output risultante sarà:

Costruttore Veicoli

Costruttore di una moto

Il costruttore Moto() ha prima invocato il costruttore Veicoli() (superclasse) e poi è stato eseguito.

La parola chiave super

La parola chiave super è utilizzata in Java per riferirsi agli elementi della superclasse. In ogni costruttore, è sempre presente una chiamata al costruttore della superclasse tramite il reference super(). Per esempio nella classe Moto il costruttore verrà modificato dal compilatore nel seguente modo:

```
public class Moto extends Veicoli{
public Moto()
{
super();
```

```
System.out.println("Costruttore di una moto");
}
}
```

Supponiamo ora di dotare la superclasse Veicoli di un costruttore che imposti la variabile colore.

```
public class Veicoli {
public Veicoli (String colore) {
this.colore = colore;
}
}
```

Questa classe compilerà, ma Moto non compilerà più:

Questo succede perché, se un costruttore viene aggiunto esplicitamente, nessun altro costruttore di default viene implicitamente inserito. Quindi l'istruzione super() inserita implicitamente nella sottoclasse Moto prova a chiamare il costruttore della superclasse senza parametri. Per risolvere il problema bisogna modificare il costruttore di Moto in modo tale da fargli chiamare il costruttore della superclasse Veicoli che prende in input il colore:

```
public class Moto extends Veicoli {
public Moto(String colore){
```

```
super(colore);
System.out.println("Costruttore di una moto");
}
}
```

La chiamata al costruttore della superclasse mediante super() deve essere la prima istruzione di un costruttore e non potrà essere inserita all'interno di un metodo che non sia un costruttore.

codice

Come per l'operatore this, anche super, viene utilizzato sia per le variabili :

super.variabile

che per invocare altri metodi

super.metodo()

E' possibile riferirsi solo agli elementi contenuti nella superclasse, non è consentito pertanto utilizzare **super.super.** Se una super classe eredita degli elementi visibili, questi diventano degli elementi a tutti gli effetti della classe, di conseguenza si potranno utilizzare direttamente con un solo super.

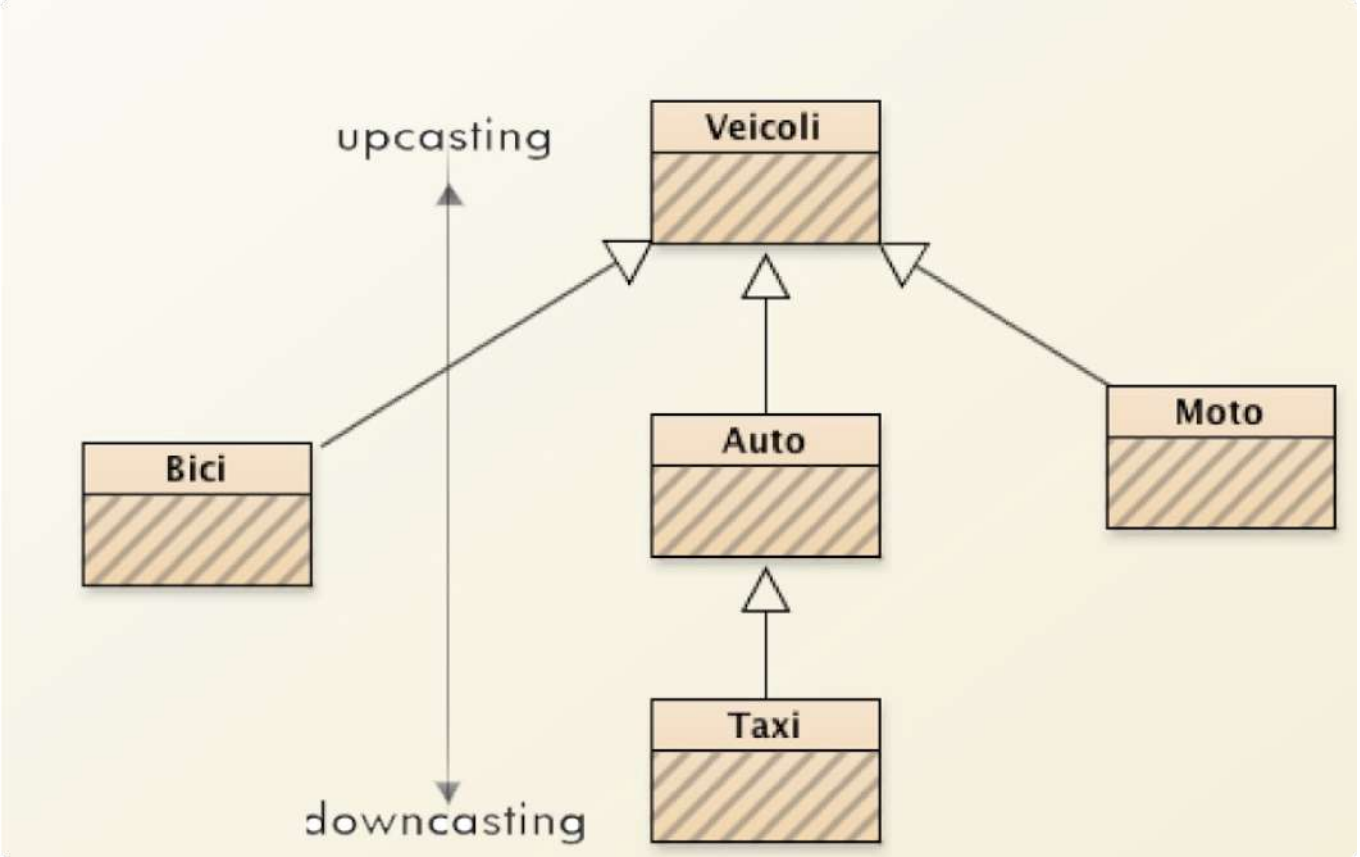
Gerarchie di classi

Una sottoclasse può essere superclasse di un proprio discendente, così si forma una gerarchia di classi:

Upcasting, downcasting

Upcasting consente a un oggetto di un tipo di sottoclasse di essere trattato come un oggetto di qualsiasi tipo di superclasse. L' Upcasting viene eseguito

automaticamente, mentre il **downcasting** deve essere fatto manualmente dal programmatore. Usiamo la gerarchia Veicoli per spiegare come funziona la gerarchia delle classi.



Auto e Moto sono entrambi Veicoli, quindi per l'ereditarietà, ha tutte le proprietà dei suoi antenati. Il che significa logicamente:

Se i veicoli hanno un proprietario allora anche le auto hanno un proprietario.

Ciò significa che non abbiamo bisogno di scrivere per ogni veicolo possibile, che ha un proprietario. Lo si scrive una volta, e ogni veicolo lo ottiene attraverso l'ereditarietà.

Col casting non **si sta modificando l'oggetto, lo si sta etichettando in modo diverso.** Se si crea un Auto e la si assegna a un Veicolo, l'oggetto è ancora un'auto, ma è trattato come qualsiasi altro veicolo e le sue proprietà sono nascoste fino a quando non vengono assegnate a un auto.

Diamo un'occhiata al codice dell'oggetto prima e dopo l'upcasting:

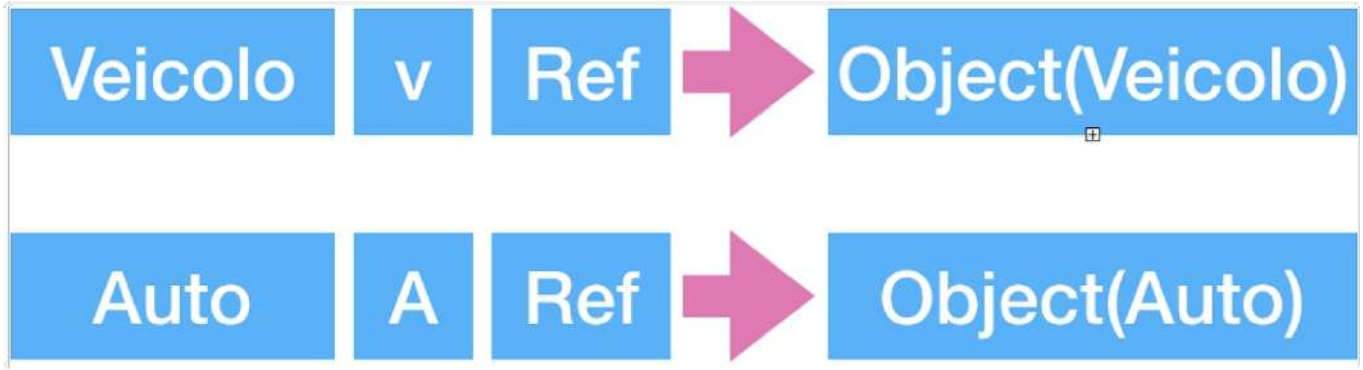
```
Auto miaAuto = new Auto();
```

```
Veicolo un Veicolo;
unVeicolo = miaAuto;
System.out.println(miaAuto);


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

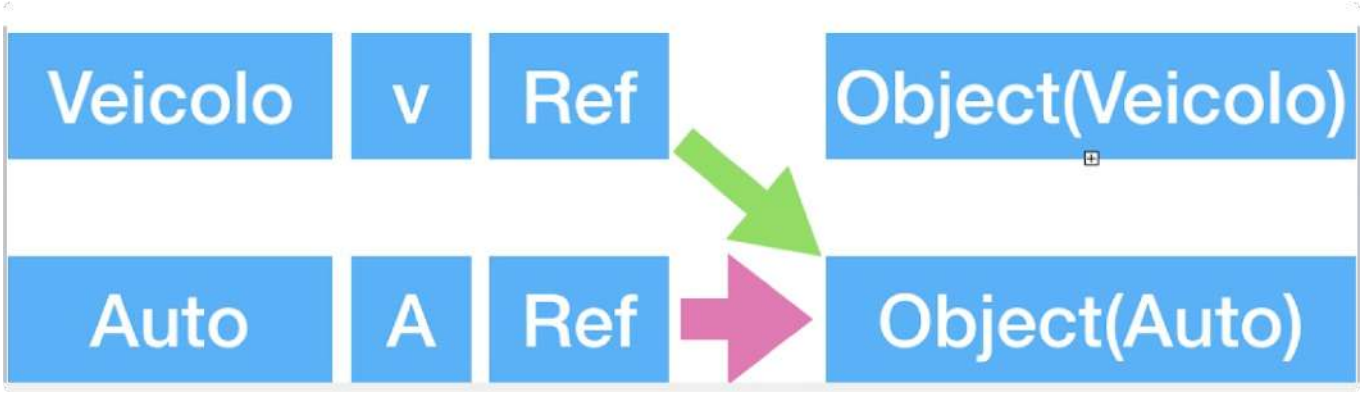
Auto@4aa298b7

Auto@400298b7

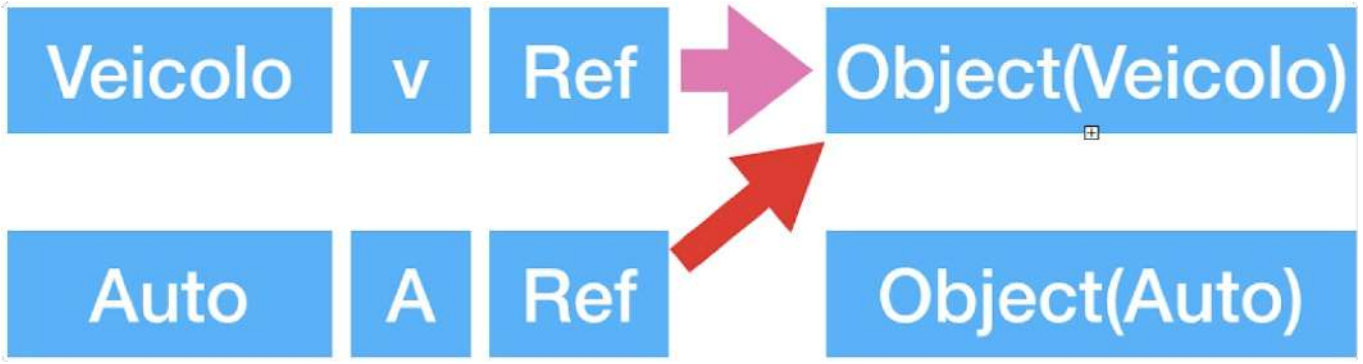
Auto dopo l'upcasting è esattamente la stessa Auto, non è diventata un Veicolo, è stato solo etichettato comeVeicolo. Questo è permesso, perché Auto è un Veicolo. Anche se sono entrambi dei Veicoli, Auto non può essere assegnato a una Moto.



È possibile assegnare a una variabile della superclasse (Veicolo) un oggetto della sottoclasse(Auto)



se proviamo a fare l'inverso otteniamo un errore:



non è possibile assegnare a una variabile della sotto classe (Auto) un oggetto della super classe (Veicolo)

Main. Java:25: error: incompatible types: Veicolo cannot be converted to Auto miaAuto-unVeicolo;

Il downcasting si deve sempre fare manualmente:

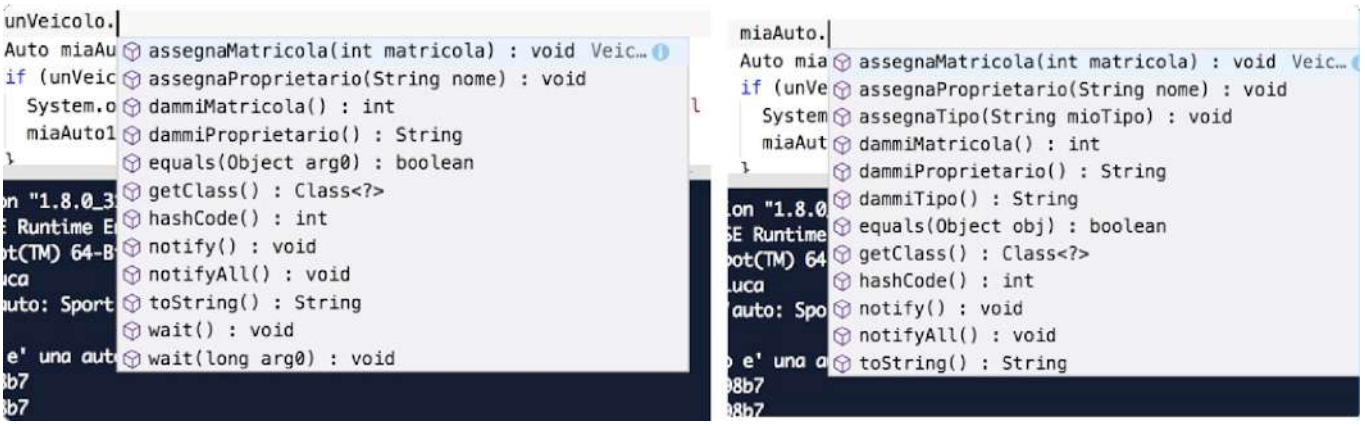
```
Auto miaAuto = new Auto();
Veicolo unVeicolo = miaAuto;//upcast automatico a Veicolo
Auto miaAuto = (Auto) unVeicolo;//downcasting manuale a Auto
```

Questo perché l'upcasting non può mai fallire. Ma se si ha un gruppo di Veicoli diversi e si vogliono assegnare tutti a un Auto, allora c'è una possibilità, che alcuni di questi Veicoli sono in realtà Moto, e il processo fallisce lanciando ClassCastException.

L'operatore instanceof

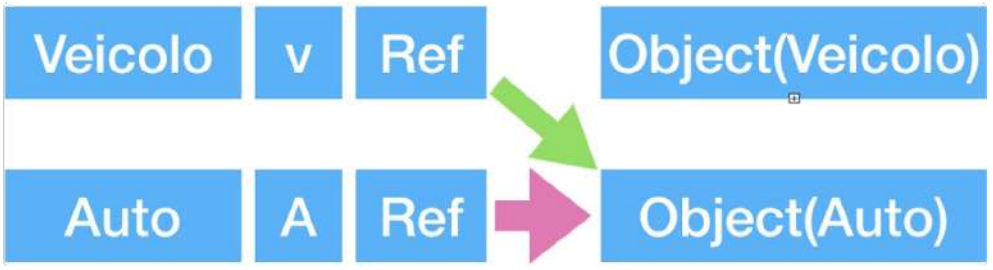
Per conoscere il tipo dell'oggetto contenuto in una variabile durante l'esecuzione del programma, è possibile utilizzare l'**operatore instanceof,** che indica se un determinato oggetto è di tipo specificato. Per esempio, per sapere se mioVeicolo è di tipo Auto, si può scrivere:

```
Auto miaAuto = new Auto();
Veicolo unVeicolo= miaAuto;
if (unVeicolo istanceof Auto) {// testare se il veicolo
è un Auto
System.out.println ("È un Auto! Ora si può ridurre a un
Auto") ;
Auto miaAuto= (Auto) unVeicolo;
}
```



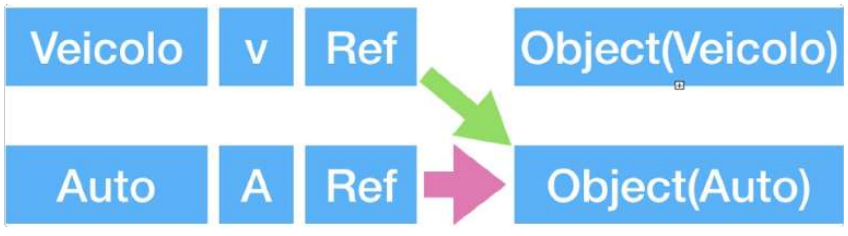
[Codice]

Se si usa un metodo della sotto classe da una variabile della superclasse che sta puntando a un oggetto della sottoclasse si ottiene un errore di compilazione, perché il metodo non è visibile.



Questo significa che la **variabile v anche se punta a un oggetto della sua sottoclasse** non può accedere ai suoi metodi.

Quindi è solo il Reference Type della variabile che determina quali sono i metodi visibili da variabile e non Object Type.

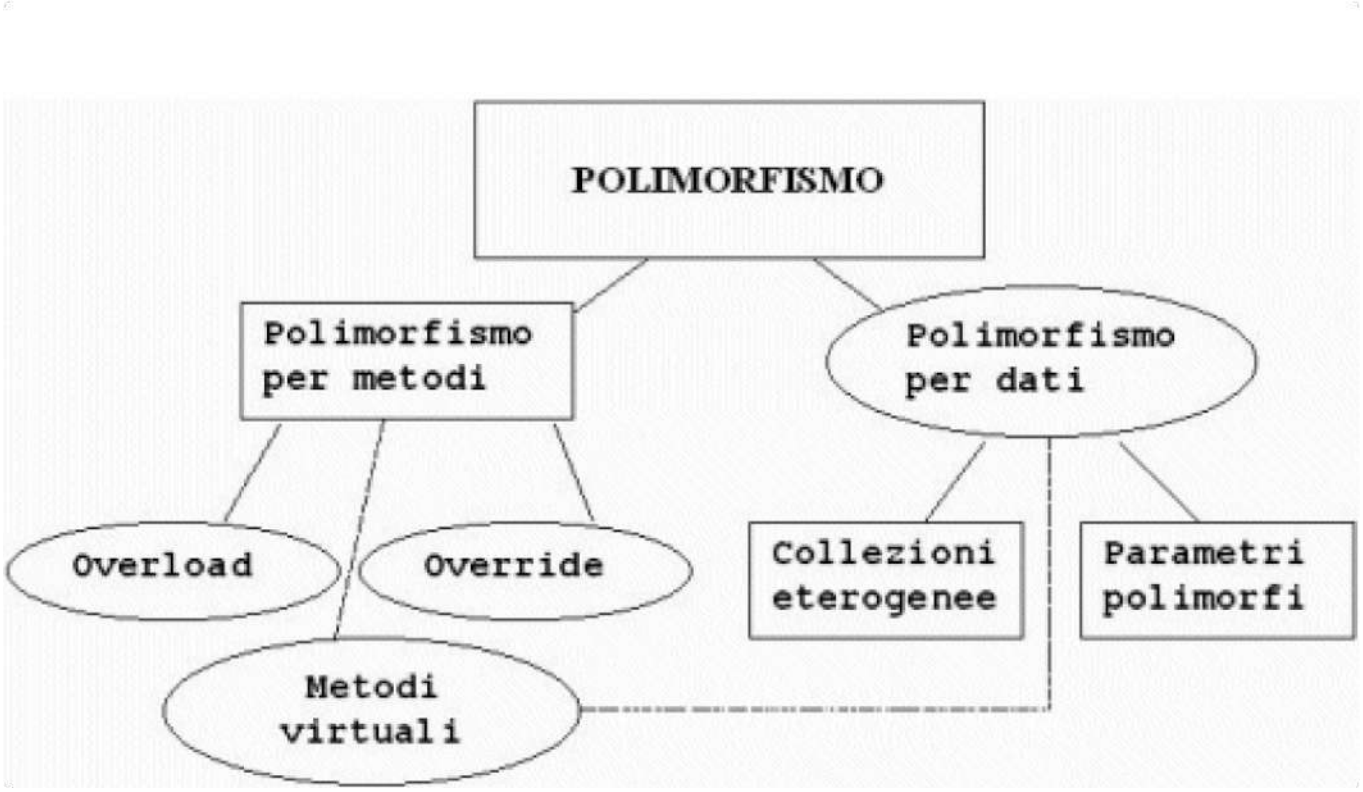


Polimorfismo

Il polimorfismo consente di riferirci con un unico termine a "entità" diverse. Ad esempio, sia un telefono

fisso sia un portatile permettono di telefonare, dato che entrambi i mezzi sono definibili come telefoni.

Telefonare quindi, può essere considerata un'azione polimorfica.



Polimorfismo per metodi

Il polimorfismo per metodi, ci permette di utilizzare lo stesso nome per metodi differenti. In Java esso trova una sua realizzazione pratica sotto due forme:

- **l'overload (che potremmo tradurre con "sovraccarico")**
- **l'override (che potremmo tradurre con "riscrittura").**

Overload

In Java un metodo è univocamente determinato non solo dal suo identificatore ma anche dalla sua lista di parametri, **cioè dalla sua firma**. Quindi, in una classe possono convivere metodi con lo stesso nome ma con differente firma. Per esempio potremmo assegnare lo stesso nome a due metodi che concettualmente hanno la stessa funzionalità, ma soddisfano tale funzionalità in maniera differente.

Presentiamo di seguito un banale esempio di overload:

```
public class Aritmetica {
public int somma(int a, int b) {
return a + b;
}
public float somma(int a, float b) {
return a + b;
}
public float somma(float a, int b) {
return a + b;
}
public int somma(int a, int b, int c) {
return a + b + c;
}
public double somma(int a, double b, int c) {
return a + b + c;
}
}
```

La lista dei parametri ha tre criteri di distinzione:

1. **tipale es**.: somma(int a, int b) è diverso da somma(int a, float b)

2. **numerico es.:** somma(int a, int b) è diverso da somma(int a, int b, int c)

3. **posizionale es.**: somma(int a, float b) è diverso da somma(float a, int b)

Override

L'override (che potremmo tradurre con "riscrittura") è il termine object oriented che viene utilizzato per descrivere la caratteristica, che hanno le sottoclassi, di ridefinire un metodo ereditato da una superclasse.

Non esisterà override senza ereditarietà. Una sottoclasse è sempre più specifica della classe che estende, e quindi potrebbe ereditare metodi che hanno bisogno di essere ridefiniti per funzionare correttamente nel nuovo contesto

```
public class Animale{
public void verso() {
System.out.printin("Grunt");
}
public class Ghepardo extends Animale{
public void verso() {
System.out.printIn("Groar!");
}
}
public class Muflone extends Animale {
public void verso() {
System.out.println("MO000!");
}
}
```

Per l'override si devono rispettare queste regole:

1. Il metodo riscritto nella sottoclasse deve avere la stessa firma (nome e parametri) del metodo della superclasse.

2. Il tipo di ritorno del metodo della sottoclasse deve coincidere con quello del metodo che si sta riscrivendo, o deve essere di un tipo che estende il tipo di ritorno del metodo della superclasse.

3. Il metodo ridefinito nella sottoclasse non deve essere meno accessibile del metodo originale della superclasse."

Il modificatore final

È possibile specificare che un metodo non può essere ridefinito nelle classi derivate. Per fare questo è sufficiente aggiungere il modificatore final alla dichiarazione del metodo, come mostrato di seguito:

public final void unMetodo()

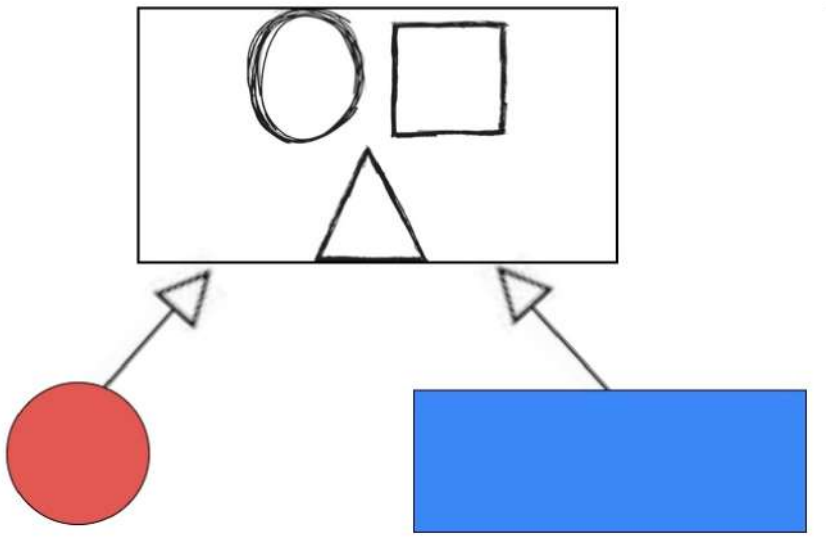
Un intera classe può essere dichiarata final. Le classi final non possono essere utilizzate come classi base di classi derivate. Ecco la sintassi per dichiarare una classe come final:

public final class unaClasse {final void myMethod()

Binding Dinamico

Supponiamo di dover progettare un insieme di classi che rappresentano diverse tipologie di figure geometriche: rettangoli, cerchi e così via. Ogni figura può essere un oggetto di una classe differente:

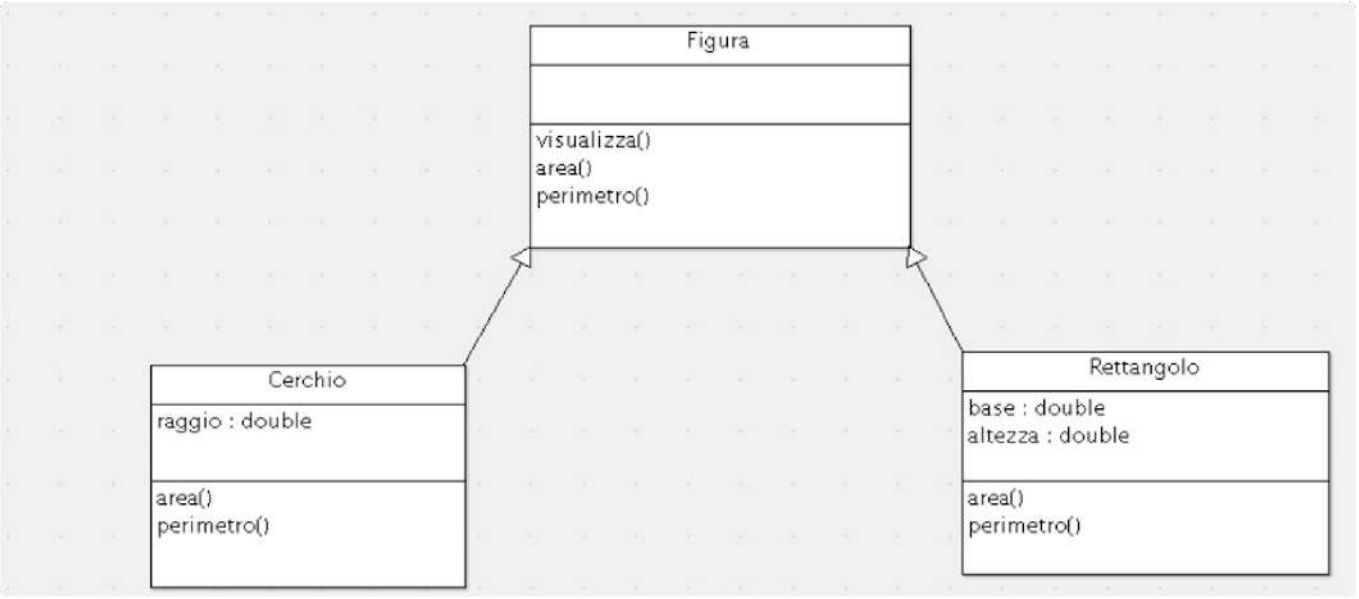
In un buon progetto software, queste classi dovrebbero tutte derivare da un'unica classe, il cui nome potrebbe essere Figura.



Si supponga di voler definire dei metodi che permettano di calcolare l'area e il perimetro di una figura geometrica. Ogni classe ha bisogno di un proprio metodo visto che deve eseguire operazioni diverse. Sapendo che i metodi appartengono a classi diverse, è possibile assegnare ai metodi lo stesso nome:

area()

perimetro()



Se **r è un oggetto di tipo Rettangolo** e **c è un oggetto di tipo Cerchio**, il comportamento dei metodi:

r.area()

r.perimetro()

e

c.area()

c.perimetro()

sarà diverso, perché corrisponde all'invocazione di due metodi ben distinti che hanno implementazioni differenti.

La classe base Figura può avere dei metodi utilizzabili da tutte le figure. Per esempio, la classe Figura può implementare un metodo che visualizza l'area e il perimetro della figura. Il metodo visualizza della classe Figura può utilizzare i metodi area e perimetro. Un implementazione della classe Figura potrebbe essere:

```
public class Figura {
  public void visualizza() {
    System.out.println("Area " + area());
    System.out.println("Perimetro" + perimetro());
  }

public double area() {
// Implementazione vuota; in Figura non si sa come implementarlo
// poiché dipende dalla specifica figura
return 0;
}

public double perimetro(){
  // Implementazione vuota; in Figura non si sa come implementarlo
  // poiché dipende dalla specifica figura
  return 0;
}
}
```

Le classiCerchio e Rettangolo

```
class Cerchio extends Figura {
  private double raggio;
  private final double PI = 3.14;

  Cerchio(double raggio) {
    this.raggio = raggio;
  }

  public double area() {
    return PI * raggio * raggio;
  }

  public double perimetro() {
    return 2 * PI * raggio;
  }
}
```

```
class Rettangolo extends Figura {
  private double altezza;
private double base;

Rettangolo(double base, double altezza){
this.base=base;
this.altezza=altezza;
}
public double area(){
return base*altezza;
}
public double perimetro(){
return 2*base+2*altezza;
  }
}
```

Quando si pensa a come usare il metodo visualizza, ereditato dalla classe Figura, per le dassi Rettangolo e Cerchio emergono delle complicazioni. Si consideri, per esempio, la classe Rettangolo che è una classe derivata da Figura e pertanto eredita il suo metodo visualizza che usa i metodi area e perimetro che sono implementati in

modo diverso per ogni tipo di figura. Il metodo visualizza è definito nella classe Figura mentre si vorrebbe che la sua esecuzione invocasse i metodi area e perimetro specifici di ogni figura. Se r è un'istanza della classe Rettangolo, si vuole che la seguente istruzione:

r. visualizza()

invochi i metodi area e perimetro della classe Rettangolo e non quella di Figura.

In Java tutto questo accade in modo automatico. Quando viene eseguito il metodo visualizza della classe Rettangolo, vengono invocati i corrispondenti metodi area e perimetro definiti nella classe Rettangolo perché Java utilizza un meccanismo conosciuto come **binding dinamico** (o dymmic binding o late binding).

BINDING DINAMICO: la risoluzione della chiamata ad un metodo ridefinito avviene dinamicamente in base al tipo dinamico dell'oggetto riferito e non in base al tipo statico della variabile referente

Figura r = new Rettangolo(2,3);

il tipo della variabile referente (Figura) è diverso (anche se compatibile) dal tipo dell'oggetto riferito (Rettangolo) il tipo dell'oggetto riferito si determina solo a run-time il metodo chiamato è il metodo area() di Rettangolo non il metodo area() di Figura.

```
class Main {
  public static void main(String[] args) {

   Figura r =new Rettangolo(2,3);
   Figura c = new Cerchio(3);
   Figura f=new Figura();
   System.out.println("------Rettangolo-------");
  r.visualizza();
  System.out.println("-----Cerchio-------");
  c.visualizza();
  System.out.println("------Figura-------");
  f.visualizza();
        }
```

```
    }
```

Codice

Metodi per cui il binding dinamico non viene applicato

Java non utilizza il binding dinamico per:

1. **i metodi privati**
2. **i metodi final**
3. **i metodi statici**

Nel caso dei metodi privati e final, l'assenza di binding dinamico non rappresenta un limite, perché comunque non sarebbe di alcuna utilità. Al contrario, l'assenza di binding dinamico per i metodi statici può essere significativa quando il metodo statico viene invocato utilizzando un oggetto chiamante invece che mediante il nome della classe.

Quando Java, non utilizza il binding dinamico, utilizza il binding statico. Nel caso del binding statico, la decisione su quale definizione di metodo debba essere eseguita viene presa durante la compilazione sulla base del tipo dell'oggetto chiamante.

Il Listato che segue mostra l'effetto del binding statico nel caso in cui venga invocato un metodo statico con un oggetto chiamante.

```
public class Figura {
  public void visualizza() {
    System.out.println("Area " + area());
    System.out.println("Perimetro" + perimetro());
  }

public double area() {
// Implementazione vuota; in Figura non si sa come implementarlo
// poiché dipende dalla specifica figura
return 0;
}

public double perimetro(){
  // Implementazione vuota; in Figura non si sa come implementarlo
```

```
  // poiché dipende dalla specifica figura
  return 0;
}
public static void disegna(){
  System.out.println("disegno figura");
}
}
```

Si noti che il metodo statico **disegna( )** , definito nella classe Figura, è stato ridefinito nella classe Rettangolo.

```
class Rettangolo extends Figura {
  private double altezza;
private double base;

Rettangolo(double base, double altezza){
this.base=base;
this.altezza=altezza;
}
public double area(){
return base*altezza;
}
public double perimetro(){
return 2*base+2*altezza;
  }
  public static void disegna(){
  System.out.println("disegno Rettangolo");
}
}
```

Nonostante questo, quando si ha un riferimento a un oggetto di tipo Rettangolo da una variabile di tipo Figura, il metodo disegna ( ) che viene eseguito è quello definito nella classe Figura e non quello definito nella classe Rettangolo.

Questo perché un metodo statico viene normalmente invocato utilizzando un nome di classe e non un oggetto chiamante. Purtroppo non è sempre così e alcune volte un metodo statico può avere un oggetto chiamante nascosto. Se si invoca un metodo statico dalla definizione di un metodo non statico senza utilizzare né un nome di classe, né un oggetto chiamante, l'oggetto chiamante implicitamente utilizzato è this.

Codice

Classi astratte

La classe Forma è stata progettata come una classe base per altre classi, come la classe Rettangolo, e se non c'è alcuna necessità d'istanziare oggetti di tipo Forma, i due metodi area() e perimetro(), non verranno mai utilizzato, poiché restituiscono sempre zero. Infatti non si può calcolare l'area o il perimetro di una Forma geometrica senza sapere di che forma si tratta.

Questi metodi sono stati definito all'interno della classe Forma esclusivamente per sfruttare il polimorfismo.

Tuttavia, invece di fornire una definizione "inventata" di un metodo che si pensa di ridefinire in una classe derivata, si può dichiarare il metodo astratto:

```
public abstract void area();
```

La sintassi per definire un metodo astratto prevede di far precedere all'intestazione del metodo la parola chiave abstract, di porre un punto e virgola alla fine dell'intestazione e di omettere il corpo del metodo.

Definire un metodo astratto significa posticipare la sua definizione al momento in cui si saprà effettivamente come definirla. Nel caso specifico, è come dire che: "ogni figura avrà un metodo area(), ma in questa classe non si sa come implementarlo".

Un metodo astratto deve essere ridefinito da ogni classe derivata dalla classe base astratta. Chiaramente, questo vale se la classe derivata sa come definirlo. Nell'ipotesi che la classe derivata sappia definirlo, includere un metodo astratto in una classe base è un modo per obbligare la classe derivata a definire un particolare metodo.

Java richiede che se una classe ha almeno un metodo astratto, la classe deve essere dichiarata astratta. Si fa ciò includendo la parola chiave abstract nell'intestazione della definizione della classe:

```
public abstract class Forma {
```

Una classe definita in questo modo è detta classe astratta. Le classi astratte non possono essere istanziate direttamente, ovvero non si possono creare i relativi oggetti. Nelle classi astratte si definiscono metodi tutti o in parte anch'essi astratti.

Sintassi:

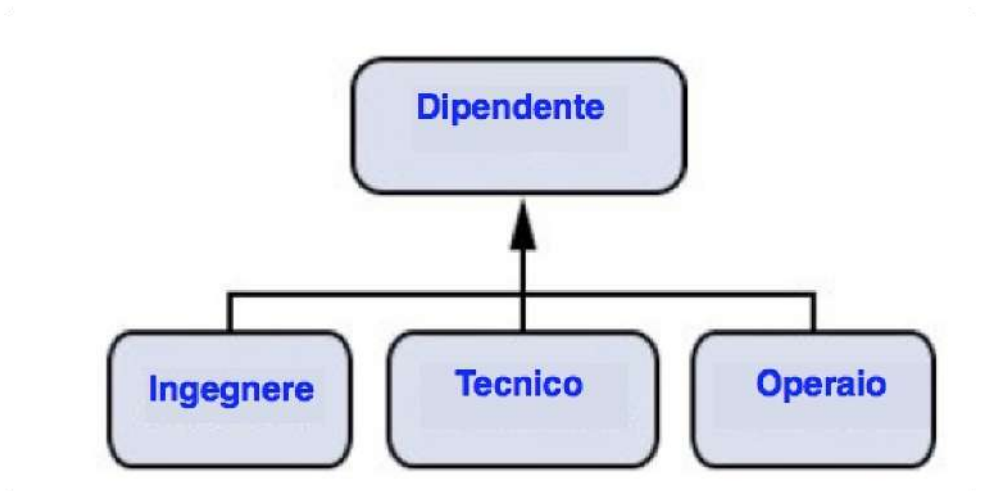
```
abstract class ClassName{
public abstract void abstractMethod();
}
```

Per definire classi e metodi astratti si usa la **keyword abstract.** Una sottoclasse di una classe astratta **deve obbligatoriamente** implementarne gli eventuali metodi astratti e, altrimenti essa stessa deve divenire classe astratta. Una classe abstract può avere anche variabili d'istanza e metodi non abstract.

```
public abstract class Figura {
  public void visualizza() {
    System.out.println("Area " + area());
    System.out.println("Perimetro" + perimetro());
  }
public abstract double area();
public abstract double perimetro();
}
```

Codice

Nella Figura mostriamo un altro esempio di ereditarietà con una classe astratta riferendosi a un'azienda dove si trovano impiegate diverse figure professionali.



La classe astratta Dipendente avrà un metodo astratto per il calcolo dello stipendio, nelle tre sottoclassi che da essa deriveranno (Ingegnere, Tecnico e Operaio) si dovrà definire in modo specializzato tale metodo.

```
public abstract class Dipendente
{
private String nome;
private String cognome;
public Dipendente(String nome, String cognome)
{
this.nome = nome;
this.cognome = cognome;
}
protected String getNome() { return nome; }
protected String getCognome() { return cognome; }

public String toString()
{
return cognome + " " + nome;
}
public abstract int calcoloStipendio(); // metodo astratto
}
```

Esempio:

• un Ingegnere, avrà uno stipendio mensile che sarà dato da un importo fisso più una percentuale,

• un Tecnico avrà uno stipendio mensile dato da un importo fisso più un quantum in base ai pezzi lavorati

• un Operaio avrà uno stipendio mensile dato da un importo a ore più una percentuale su un numero variabile di pezzi lavorati.

```
public class Ingegnere extends Dipendente {
  private int percentage;
  private int fisso;
  public Ingegnere(String n, String c, int p, int f) {
    super(n, c);
    setPercentage(p);
```

```
    setFisso(f);
  }
  public void setFisso(int f) // imposto il fisso come paga
  {
    fisso = f > 0 ? f : 0;
  }
  public void setPercentage(int p) // imposto la percentuale
  {
    percentage = p > 0 ? p : 0;
  }

  @Override
  public int calcoloStipendio() // calcolo specializzato del
guadagno
  {
    return fisso + (fisso * percentage / 100);
  }

  @Override
  public String toString() {
    return super.toString() + " guadagna € ";
  }
}
```

Il Listato evidenzia come la classe ingegnere sia una classe specializzata della classe base astratta Dipendente: infatti esegue l'override del metodo calcoloStipendio ereditato per il calcolo della paga.

Inoltre, poiché un Ingegnere è anche un Dipendente dal suo costruttore invochiamo il costruttore di Dipendente per inizializzare nome e cognome. In modo analogo possiamo creare le altre due classi.

```
class Main {
  public static void main(String args[]) {
        Dipendente e;
        Ingegnere eng = new Ingegnere("Mario", "Rossi", 10, 1000);
        Tecnico tec = new Tecnico("Paolo", "Canali", 800, 3);
        Operaio lab = new Operaio("Aldo", "Falco", 2, 44);
        e = eng; // ora è un Engineer
```

```
        System.out.print(e.toString() + e.calcoloStipendio());
        e = tec; // ora è un Technician
        System.out.print(" | " + e.toString() +
e.calcoloStipendio());
        e = lab; // ora è un Laborer
        System.out.println(" | " + e.toString() +
e.calcoloStipendio());
    }
}

Output:
Rossi Mario guadagna € 1100 | Canali Paolo guadagna € 815 | Falco
Aldo guadagna € 380
```

Dal Listato vediamo che nel metodo main si crea il riferimento e del tipo della classe astratta Dipendente e poi tanti riferimenti (eng, tec e lab) quante sono le classi da essa derivate. Successivamente assegniamo in sequenza tali riferimenti al riferimento e di tipoDipendente e da esso invochiamo, il metodo calcoloStipendio per visualizzare lo stipendio dell'oggetto che sta in quel momento referenziando.

Quindi una classe astratta si utilizza principalmente per definire entità astratte (o di alto livello) da cui è possibile definire altre classi che hanno una relazione gerarchica con la classe astratta.

Codice

Interfacce

Un'interfaccia è una sorta di classe astratta che dichiara, principalmente, dei metodi (è presente solamente la loro Firma) che le classi che la implementano devono poi definire. Pertanto, essa contiene una serie di metodi astratti (sono implicitamente abstract).

Sintassi:

```
modificatore-di-accesso interface MyInterface (
```

Sintassi per l'implementazione di un'interfaccia:

```
public class MyClass implements MyInterface{
```

```
public interface FormaGeometrica {
doble calcolaPerimetro();
double calcolaArea();
voi disegnaForma();
}
```

Un interface può anche contenere dei dati membro, che il compilatore tratta automaticamente come:

public final static

ossia come costanti di classe, per questo motivo bisogna inizializzare questi parametri.

Consideriamo l'interface FiguraGeometrica e una classe Cerchio che la implementi

```
interface FiguraGeometrica {
   double PIGRECO=3.14;
   public double calcolaArea();
    }

class Cerchio implements FiguraGeometrica {
  private double raggio;
  Cerchio(double raggio){
    this.raggio=raggio;
     }
public double calcolaArea(){
  return FiguraGeometrica.PIGRECO*raggio*raggio;
}
}
```

Nell'uso delle interfacce valgono le seguenti regole:

o Possiamo dichiarare una variabile indicando come tipo un'interfaccia

o Non possiamo istanziare un'interfaccia

Vediamo il codice che testa la classe cerchio:

- A una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia
- Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia.

```
public class Main {
  public static void main(String[] args) {
  FiguraGeometrica a;
  Cerchio b=new Cerchio(2);
  a=b;
   System.out.println("l'area del cerchio di raggio 2 è "+
a.calcolaArea());
   FiguraGeometrica c =new Cerchio(3);
  a=b;
   System.out.print("l'area del cerchio di raggio 3 è "+
c.calcolaArea());

  }
}
```

Codice

Metodi statici (Java 8)

Con Java 8 è possibile definire all'interno delle interfacce anche metodi statici. Quindi è possibile scrivere interfacce nel seguente modo:

```
public interface StaticMethodInterface {
static void metodostatico() {
System.out.println ("Metodo Statico Chiamato!") ;
}
```

e chiamare metodi statici direttamente dalle interfacce:

```
public class TestStaticMethodInterface{
public static void main(String args[])
StaticMethodInterface.metodoStatico();
```

```
}
}
```

Un'interfaccia come la precedente non ha bisogno di essere implementata, infatti i metodi statici di un'interfaccia non vengono ereditati. Interfacce di questo tipo servono semplicemente per definire metodi statici e pubblici. Ovvero definiscono funzioni.

Metodi di default e interfacce funzionali (Java 8)

Altra novità di Java 8 è la possibilità di dichiarare metodi concreti all'interno delle interfacce. Si parla di metodi di **default** perché vengono dichiarati usando come modificatore la parola chiave **default**. Per esempio consideriamo il seguente codice:

```
public interface Solista {
default void eseguiAssolo() {
//Scala maggiore in DO
System.out.println ("DO RE MI FA SOL LA SI");
}
```

In questo modo possiamo ereditare questo metodo in un'eventuale sottoclasse senza dover riscrivere il metodo (che non è astratto). La seguente classe compila senza problemi:

```
public class Musicista implements Solista {}
```

Il metodo esequiAssolo() sarà sempre possibile riscriverlo all'occorrenza. Avere però un'implementazione di default può essere in generale molto comodo.

È possibile scrivere più di un metodo di default in un'interfaccia, e questi possono convivere con metodi astratti e metodi statici. Il nome "interfaccia" in un certo senso ha perso di significato, anche se è sempre possibile usare le interfacce dichiarando solo metodi astratti. In particolare prendono il nome d'interfacce funzionali le interfacce che contengono un unico metodo astratto.

Quando si utilizza un'interfaccia?

Un'interfaccia si utilizza principalmente per definire metodi comuni a più tipi, i quali non hanno alcuna relazione gerarchica tra loro.

Esempio:

- le classi Prodotto e FormaGeometrica hanno bisogno di un metodo per la conversione degli attributi in una stringa XML
- l'interfaccia ConvertiDati definisce il metodo generaXML(), senza implementarlo
- le classi Prodotto e FormaGeometrica, possono implementare l'interfaccia Converti Dati attraverso l'utilizzo della chiave riservata implements.
- nelle classi Prodotto e formaGeometrica è necessario implementare il metodo generaXML() definito nell'interfaccia ConvertiDati.
- il contenuto del metodo generaXML() e l'XML generato, cambia in base alla classe che lo implementa

```
public class Prodotto implements ConvertiDati{
@Override
public String generaXML() {/* implementazione del
metodo*/
}
}
```

Interfacce e classi astratte, differenze

Le interfacce e le classi astratte sono elementi molto simili, e la loro similitudine è aumentata in Java8 che ha introdotto la possibilità di definire una implementazione, detta default, dei metodi dichiarati nelle interfacce.

	Interfacce	Classi astratte
Istanziabile	no	no
Fields	solo `static final`	sì
Costruttore	no	sì
Metodi statici	Java8+	sì
Dichiarazione metodi (virtual)	no	sì
Implementazione metodi	java8+ (con il qualificatore `default`)	sì

Entrambe non possono essere istanziate e possono dichiarare al loro interno metodi con o senza implementazione. Tuttavia ci sono delle caratteristiche che le

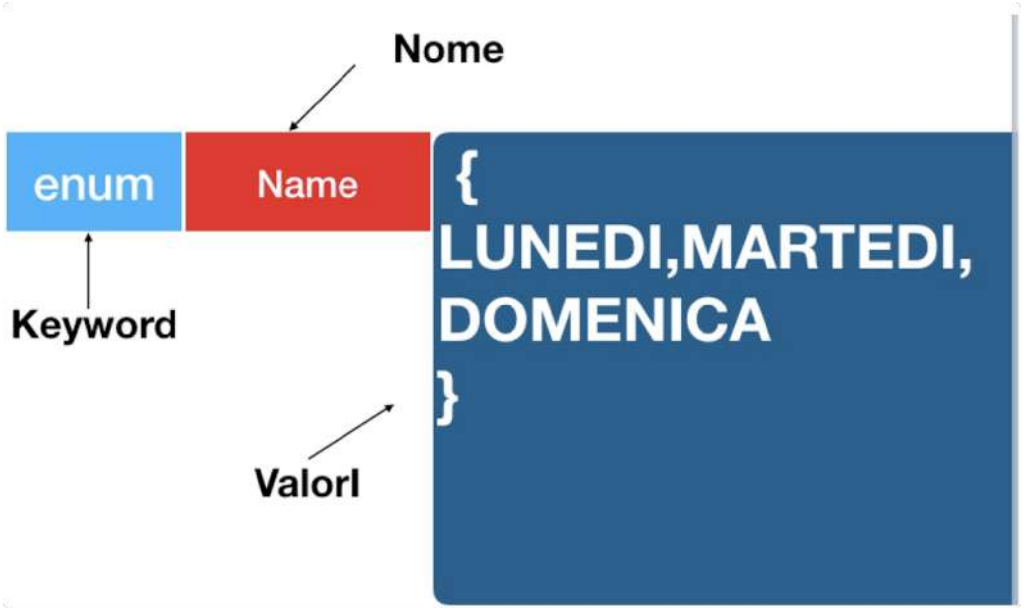
differenziano e che fanno la differenza, come per esempio le classi astratte possono dichiarare campi che sono non static e final, e dichiarare metodi public, protected e private. Invece con le interfacce tutti i campi sono automaticamente public, static e final e tutti i metodi che vengono dichiarati o definiti sono public. Un'altra importante caratteristica che le differenzia e che spesso è punto focale nella scelta di uno o dell'altro approccio, è il fatto che si può estendere una sola classe astratta, mentre si possono implementare tutte le interfacce che si vuole.

Quando utilizzare l'interfaccia e quando la classe astratta?

In base a quanto detto, cosa è meglio utilizzare? E in quali circostanze? 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. Si decide di utilizzare un'interfaccia se ci si trova nella situazione in cui alcune classi (assolutamente non legate fra di loro) si trovano a condividere i metodi di un'interfaccia, se si vuole specificare il comportamento di un certo tipo di dato (ma non implementarne il comportamento) o se si vuole avere la possibilità di sfruttare la "multiple inheritance".

Enum in Java

Le Enum definiscono un tipo di dati che può assumere un insieme limitato e fissato di valori. Ad esempio, un tipo di dati che rappresenta un giorno della settimana può assumere solo uno di 7 possibili valori ognuno con un determinato giorno. La sintassi per definire un enumerazione:



```
public enum Giorno {
  LUNEDI,
  MARTEDI,
  MERCOLEDI,
  GIOVEDI,
  VENERDI,
  SABATO,
  DOMENICA // opzionalmente può terminare con ";"
}
```

Possiamo definire una variabile di tipo Giorno come

Giorno giornoDellaSettimana;

Abbiamo una variabile che potrà contenere solamente un valore appartenente al set specificato nella definizione dell'enum Giorno e contemporaneamente avremo a disposizione anche i nomi simbolici (le costanti di prima) da usare nella scrittura del programma:

giornoDellaSttimana= Giorno. LUNEDI;

A ogni costante della classe viene assegnato in automatico un valore numerico in base all'ordine in cui sono stati scritti (nel nostro caso, a LUNEDI sarà assegnato zero, a MARTEDI uno, a DOMENICA sei).

Questo valore indica la posizione occupata all'interno della classe.

Tecnicamente parlando in Java una enum è una classe come le altre ma che "implicitamente" estende sempre la classe java.lang.Enum. Esistono metodi per passare da indice numerico a valore enumerato, e viceversa.

Per passare da valore enumerato a indice, si usa il seguente metodo della classe Enum

public int ordinal()

restituisce un intero che rappresenta l'indice dell' enum.

Per l'operazione inversa, si usa il seguente metodo statico, che ogni classe enumerata possiede automaticamente (non appartiene alla classe Enum)

public static E[] values ()

restituisce un array contenente tutti i possibili valori di E.

Quindi, per ottenere il valore di posto i-esimo, è sufficiente accedere all'elemento i-esimo dell'array restituito da values.

valueOf (str)

restituisce la costante di enumerazione i cui valori corrispondono alla stringa passata in str.

Codice

Il fatto che le enum siano a tutti gli effetti classi, apre la possibilità di aggiungere dentro di essi metodi, attributi e costruttori. Esaminiamo un po di sintassi:

1. **I costruttori devono essere privati** e non possono essere chiamati esplicitamente, sono unicamente a disposizione del compilatore.
2. **La lista dei valori** nel caso in cui siano definiti dei costruttori non deve essere considerata, come abbiamo fatto fino a ora, come una lista di etichette ma come una forma compatta per istruire il compilatore a costruire determinate istanze della classe e assegnare loro un nome simbolico. Non ci sono restrizioni circa i metodi e gli attributi che possono essere inclusi nel corpo di un enum.

```
public enum Giorno {
  LUNEDI(1),
  MARTEDI(2),
  MERCOLEDI(3),
  GIOVEDI(4),
  VENERDI(5),
  SABATO(6),
  DOMENICA(7);
```

```
  private int valore;
  private  Giorno(int valore){
    this.valore=valore;
      }
      public int getValore(){
        return valore;
      }
}
```

Giorno giornoDellaSttimana=Giorno.LUNEDI;

Quando definiamo la variabile tutti i costruttori vengono invocati con il valore predefinito. E quindi possiamo accedere a questo valore tramite il metodo:

getValore();

Lo stesso risultato si ottiene:

Giorno.LUNEDI.getValore();

Codice

Eccezioni

Un'eccezione è una situazione imprevista (errore) che non viene rilevata durante la fase di compilazione, ma si presenta durante l'esecuzione di un'applicazione. Un errore di questo tipo impedisce la normale esecuzione del programma e dev'essere gestita, altrimenti il programma s'interrompe.

In Java è possibile gestire le eccezioni utilizzano all'interno del codice delle parole chiavi:

1. try

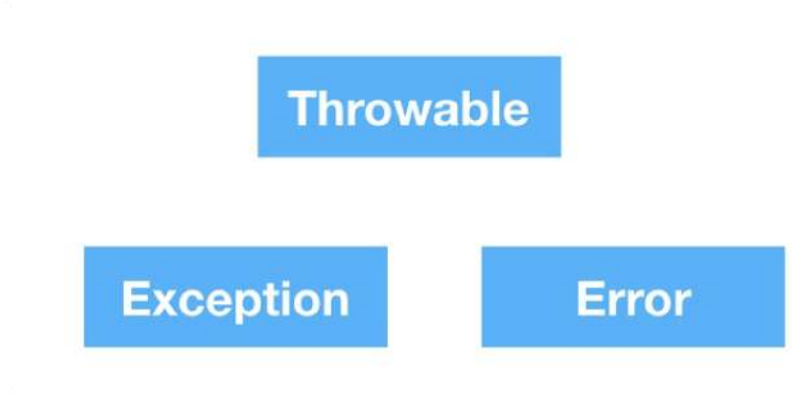
2. catch

3. finally

4. throw

5 throws

Sarà anche possibile **creare eccezioni personalizzate** e decidere non solo come, ma anche in quale parte del codice gestirle, grazie a un meccanismo di propagazione molto potente. Questo concetto è implementato nella libreria Java mediante la classe Exception (sotto classe di Throwable) e le sue sotto classi.



Exception: Rappresenta la superclasse di tutte quelle eccezioni gestibili ad esempio:

L'eccezione che si può verificare quando in un programma si esegue una divisione per zero. Tale operazione non è eseguibile.

Error: Rappresenta una situazione imprevista non dipendente da un errore commesso dallo sviluppatore. A differenza delle eccezioni, gli errori non sono

gestibili. Un esempio di errore che potrebbe causare un programma è quello relativo alla terminazione delle risorse di memoria. Questa condizione non è gestibile.

Un'ulteriore categorizzazione delle eccezioni è data dalla divisione delle eccezioni in:

1. **unchecked:** Ci si riferisce alle **RuntimeException** (e le sue sottoclassi) Eccezioni non controllate (unchecked) Sono dovute al codice scritto nel nostro programma e in linea teorica potrebbero essere evitate. Rappresentano i «famosi» bug software. Queste eccezioni non vengono verificate dal compilatore e possono accadere solo durante l'esecuzione del programma.
2. **checked exception:** Tutte le altre. Sono dovute a eventi che si verificano esternamente al software (esempio l'accesso a un file inesistente). Si chiamano controllate in quanto il compilatore verifica che vengano indicate e intercettate nel software. Se non vengono indicate, si hanno degli errori di compilazione. Se si utilizza un metodo che lancia una checked exception senza gestirla da qualche parte, la compilazione non andrà a buon fine. Da qui il termine checked exception (in italiano "eccezioni controllate".

Il fatto che sia la classe Exception sia la classe Error estendono una classe che si chiama "lanciabile" (Throwable) è dovuto al meccanismo con cui la JVM reagisce quando s'imbatte in una eccezione-errore.

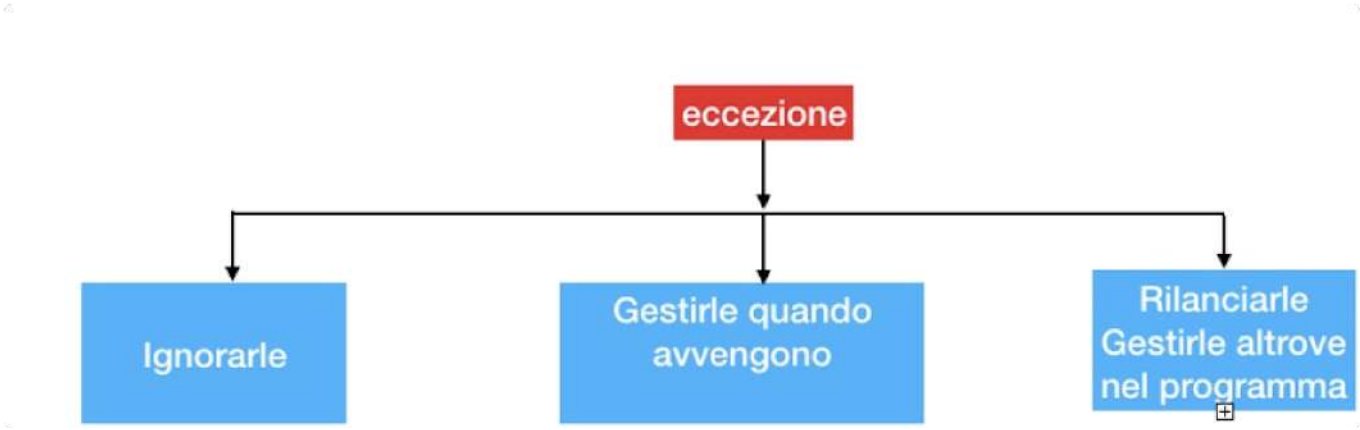
Se un blocco di codice genera un'eccezione durante il runtime, la JVM istanzia un oggetto dalla classe eccezione relativa al problema e lancia l'eccezione appena istanziata (tramite la parola chiave throw). Se il nostro codice non cattura (tramite la parola chiave catch) l'eccezione, il gestore automatico della JVM interrompe il programma generando in output informazioni dettagliate su ciò che è accaduto.

Supponiamo che durante l'esecuzione un programma provi a eseguire una divisione per zero tra interi. La JVM istanzierà un oggetto di tipo **ArithmeticException (**inizializzandolo opportunamente) e lo lancerà. In pratica è come se la JVM eseguisse le seguenti righe di codice:

```
ArithmeticException ex = new ArithmeticException();
throw exc;
```

Tutto avviene dietro le quinte e sarà trasparente allo sviluppatore.

Ci sono 3 modi di gestire l'eccezioni:



Se un'eccezione è ignorata da un programma, questo terminerà producendo un messaggio opportuno con l'indicazione delle chiamate di metodi che hanno portato all'eccezione dell'errore della linea in cui l'eccezione si è verificata.

Esempio: Divisione per zero EVITARE!!!!!

```
public class Main
{
  public static void main(String args[])
  {
      int x = 3;
      int y = 0;
      int z = x/y;
      System.out.println("z=" + z);
  }
}
```

Main Exception in thread "main" java.lang.ArithmeticException: / by zero
at Main.main(Main.java:7)

Gestire le Eccezioni (processare l'eccezione quando accade)

Nell'esempio precedente l'eccezione genera un messaggio molto esplicativo dal momento che sono stati evidenziati:

1. Il tipo di eccezione (java.lang.ArithmeticException);

2. Un messaggio descrittivo (/ by zero);

3. Il metodo in cui è stata lanciata l'eccezione (Main.main):

4. Il file in cui è stata lanciata l'eccezione (Main.java):

5. La riga in cui è stata lanciata l'eccezione (:7).

L'unico problema è che il programma è terminato prematuramente. Utilizzando le parole chiave try e catch sarà possibile gestire l'eccezione in maniera personalizzata:

Racchiudiamo il codice da controllare dentro un blocco che inizia con la parola chiave try.

```
public class Main
{
  public static void main(String args[])
  {
    try
    {
      int x = 3;
      int y = 0;
      int z = x/y;
```

```
      System.out.println("z=" + z);
    }
    catch (ArithmeticException ex)
    {
      System.out.println("Si e' verificata un'eccezione -> siamo
nel blocco catch");
      //ex.printStackTrace();
    }
    System.out.println("siamo fuori dal blocco catch");
  }
}
```

Se durante l'esecuzione del programma tutto va bene i blocchi catch vengono saltati e il programma prosegue normalmente.

Se nel blocco try viene sollevata un'eccezione (nell'esempio ArithmeticException) verrà catturata nel blocco catch, e viene eseguito il codice contenuto nel blocco.

Si possono inserire una serie di blocchi catch in modo da gestire una serie di eccezioni.

Il blocco catch deve dichiarare un parametro (come se fosse un metodo), il Data Type di questo parametro deve essere una classe derivata dalla classe throwable.

Per reperire informazioni sull'eccezione sollevata si usa il metodo printStackTrace() che visualizza messaggi informativi identici a quelli visualizzati quando l'eccezione non è gestita, ma senza interrompere il programma. È fondamentale che si dichiari, tramite il blocco catch, un'eccezione del tipo giusto.

Codice

Come per i metodi, anche per i blocchi catch i parametri possono essere polimorfi. Per esempio, il seguente frammento di codice:

```
int a = 10;
int b = 0;
try{
   int c= a/b:
   System.out.printin(c);
```

```
} catch (Exception ex){
    ex.printStackTrace();
}
```

contiene un blocco catch che cattura qualsiasi tipo di eccezione, essendo **Exception la superclasse** da cui discende ogni altra eccezione. Il reference ex è in quest'esempio un parametro polimorfo. È possibile far seguire a un blocco try più blocchi catch, come nel seguente esempio:

```
int a = 10;
int b= 0;
try{
  int c= a/b;
  System.out.println(c);
} catch (ArithmeticException ex) {
   System.out.printin("Divisione per zero...");
}catch (NullPointerException ex){
   System.out.println("Reference nullo...");
}catch (Exception ex) {
   ex.printStackTrace();
}
```

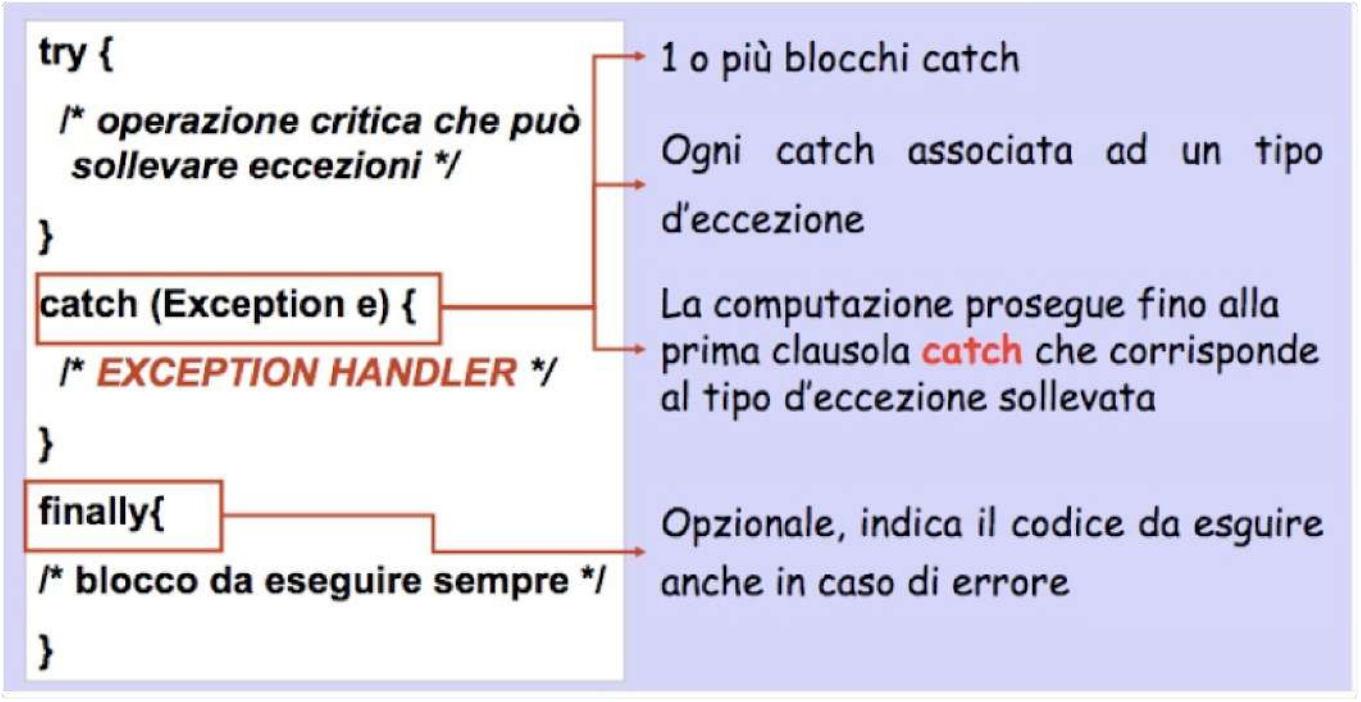
In questo modo il programma risulta più robusto e gestisce diversi tipi di eccezioni. Nel peggiore dei casi (ovvero se il blocco try lanciasse un'eccezione non prevista) l'ultimo blocco catch gestisce il problema.

finally

È possibile far seguire un blocco try, oltre che da blocchi catch, da un altro blocco definito dalla keyword finally. Ciò che è definito in un blocco finally viene sempre eseguito:

- Sia se viene lanciata un'eccezione.
- Sia se non viene lanciata nessuna eccezione.

Per esempio, è possibile utilizzare un blocco finally quando esistono operazioni critiche che devono essere eseguite in qualsiasi caso.




```
public class Eccezione {
public static void main(String args[]) {
int a = 10;
int b =0;
try{
  int c = a/b;
  System.out.println(c);
}catch (ArithmeticException exc) {
    System.out.println( "Divisione per zero...");
}catch (Exception exc){
    exc.printstackTrace);
}finally {
  System.out.println( "Operazione terminata");
      }
}
}
```

L'output del programma è: Divisione per zero... Operazione terminata.

Se invece b è diversa da zero(b= 2) :

Output: Operazione terminata.

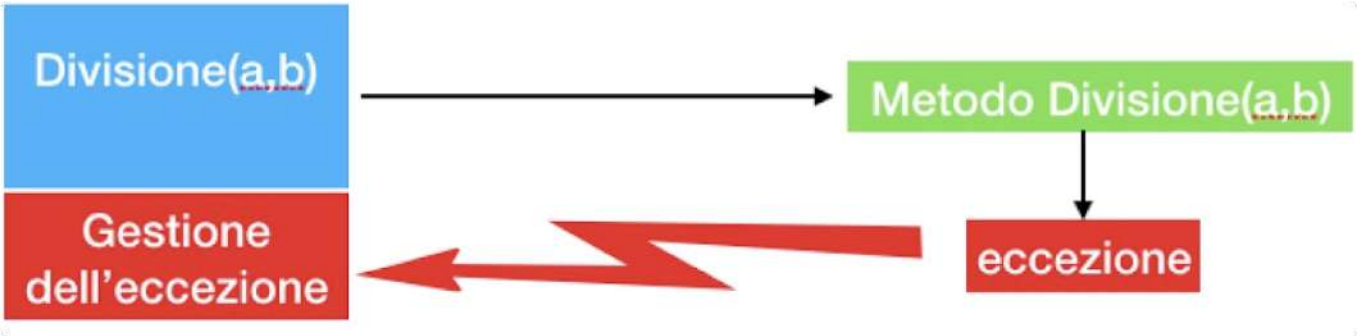
È possibile anche far seguire direttamente a un blocco try un blocco finally, che viene eseguito sia in presenza che in assenza di un'eccezione. Comunque, se si verifica

un'eccezione, non essendo gestita con un blocco catch, il programma terminerebbe in modo anomalo.

Propagazione: l'istruzione throws

Il costrutto **try - catch** permette di gestire i problemi localmente, nel punto preciso in cui sono stati generati.

Tuttavia, in un'applicazione stratificata, può essere preferibile fare in modo che le classi periferiche lascino rimbalzare l'eccezione verso gli oggetti chiamanti, in modo da delegare la gestione dell'eccezione alla classe che possiede la conoscenza più dettagliata del sistema.



L'istruzione **throws,** se è presente nella firma di un metodo, consente di propagare l'eccezione al metodo chiamante, in modo da delegarne a esso la gestione:

```
public void divisione(int a, int b) throws
ArithmeticException {
int c=a/b:
}
```

La chiamata al metodo divisione, definito con clausola throws. dovrà essere posta all'interno di un blocco try - catch.

```
public void aritmetica(int a, int b){
try{
  int c=divisione (a,b);
}catch(ArithmeticException e){
//codice eccezione;
}
}
```

```
public int divisione(int a, int b) throws ArithmeticException {
int c=a/b;
return c;
}
```

In alternativa, il metodo chiamante potrà a sua volta delegare la gestione dell'eccezione mediante una throws.

```
public void aritmetica(int a, int b) throws ArithmeticException{
  int c=divisione (a,b);
}

public int divisione(int a, int b) throws ArithmeticException {
int c=a/b;
return c;
}
```

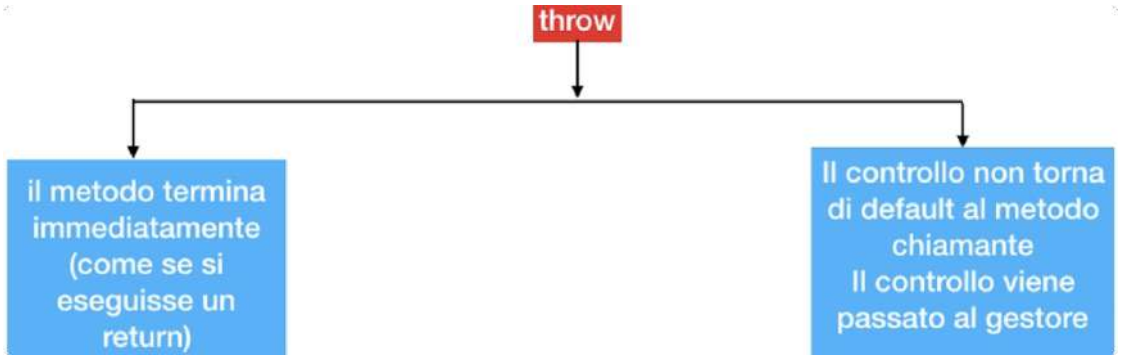
Codice

Lancio di eccezioni: il costrutto throw

Finora si è visto come gestire metodi che possono generare eccezioni, o in alternativa come delegare la gestione delle stesse a un metodo chiamante. Ma cosa si deve fare se si desidera generare in prima istanza un'eccezione?

Se vogliamo lanciare un'eccezione durante l'esecuzione del programma possiamo utilizzare la parola chiave **throw**.

L'istruzione throw richiede come argomento un oggetto Throwable o una sua sottoclasse. È possibile utilizzare throw all'interno di un blocco catch, qualora si desideri ottenere sia la gestione locale di un'eccezione sia il suo inoltro all'oggetto chiamante.



```java
public class Main {
  public static void main(String[] args) {
    int a= 10;
    int b= 0;
    try{
      int c= divisione(a,b);
      System.out.println(c);
    }catch(Exception e){
       System.out.println(e.getMessage() );
    }finally{
        System.out.println("Programma terminato");
    }


  }
  public static int divisione(int a,int b) throws Exception{
    int c;
    if(b==0)
     throw new Exception("il denominatore deve essere diverso da zero");
    else
    c=a/b;
    return c;
  }
}
```

Codice

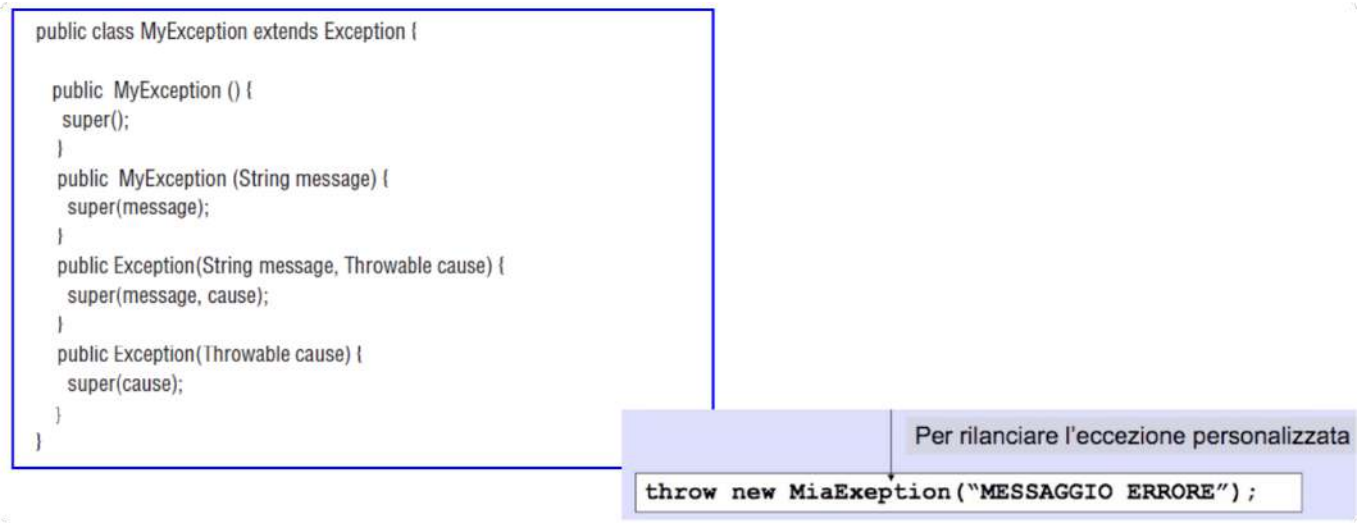
Eccezioni definite dall'utente

Nonostante l'enorme varietà di eccezioni già presenti in Java, il programmatore può facilmente crearne di proprie, qualora desideri segnalare condizioni di eccezione tipiche di un proprio programma. Per creare una nuova eccezione è sufficiente dichiarare una sottoclasse di Exception (o di una qualsiasi altra eccezione esistente) e ridefinire uno o più dei seguenti

costruttori:

- **Exception():** crea un'eccezione.
- **Exception(String message):** crea un'eccezione specificando un messaggio diagnostico.
- **Exception(Throwable cause)**: crea un'eccezione specificando la causa.
- **Exception(String message, Throwable cause):** crea un'eccezione specificando un messaggio diagnostico e una causa.

Tuttavia la JVM non può lanciare automaticamente la nostra MiaException, la JVM, infatti, sa quando lanciare una ArithmeticException ma non sa quando lanciare una MiaException. In tal caso sarà compito dello sviluppatore lanciare l'eccezione usando la parola chiave throw.



Codice

Stringhe

Tipo di dati String

Una stringa è una **sequenza immutabile di caratteri**. I caratteri sono rappresentati in memoria usando la codifica **UFT-16** e i simboli **Unicode**. La codifica prevede che un singolo carattere occupi 16 bit. Sono esempi di stringhe:

- una frase
- il codice fiscale
- un numero di telefono

Una stringa non è un array di caratteri ma un oggetto definito nella classe;

java.lang.String

Per dichiarare e assegnare una stringa si possono usare i metodi:

String Literals

```
String txt= "Ciao Mondo";
```

Usando le virgolette doppie e non gli apici singoli che rappresentano un singolo caratteri.

Oppure si possono **usare i costruttori** della classe String():

```
String txt= new String();
```

Stringa null.

```
String txt= new String("Ciao Mondo");
```

Crea una stringa uguale a quella passata come parametro;

```
String txt= new String(char[] caratteri);
```

Crea una stringa da un array di caratteri,

String txt= new String(char[] c, int offset,int count);

Crea una stringa composta da un sottoinsieme di caratteri contenuti nell'array c. Tale sottoinsieme è costruito fornendo l'indice di partenza inclusivo e il numero di caratteri.

Per inserire caratteri speciali all'interno di una stringa si fa ricorso al carattere di escaping:

\(backslash)

Ad esempio, per inserire un ritorno a capo si utilizza la sequenza:

\n

Questo meccanismo consente di inserire lo stesso delimitatore di stringa all'interno della stringa:

Sequence	Character represented
\0	The NUL character (\u0000).
\b	Backspace (\u0008).
\t	Horizontal tab (\u0009).
\n	Newline (\u000A).
\v	Vertical tab (\u000B).
\f	Form feed (\u000C).
\r	Carriage return (\u000D).
\"	Double quote (\u0022).
\'	Apostrophe or single quote (\u0027).
\\	Backslash (\u005C).
\x*xx*	The Latin-1 character specified by the two hexadecimal digits *xx*.
\u*xxxx*	The Unicode character specified by the four hexadecimal digits *xxxx*.

La classe String **è immutabile**, cioè lo stato di un'istanza di quella classe non può essere modificato dopo che è stata creata. Per comprendere questo concetto vediamo come le stringhe vengono salvate in memoria.

Se noi creiamo più istanze di una determinata classe, ognuna contiene un reference a una determinata area di memoria indipendentemente dal contenuto.

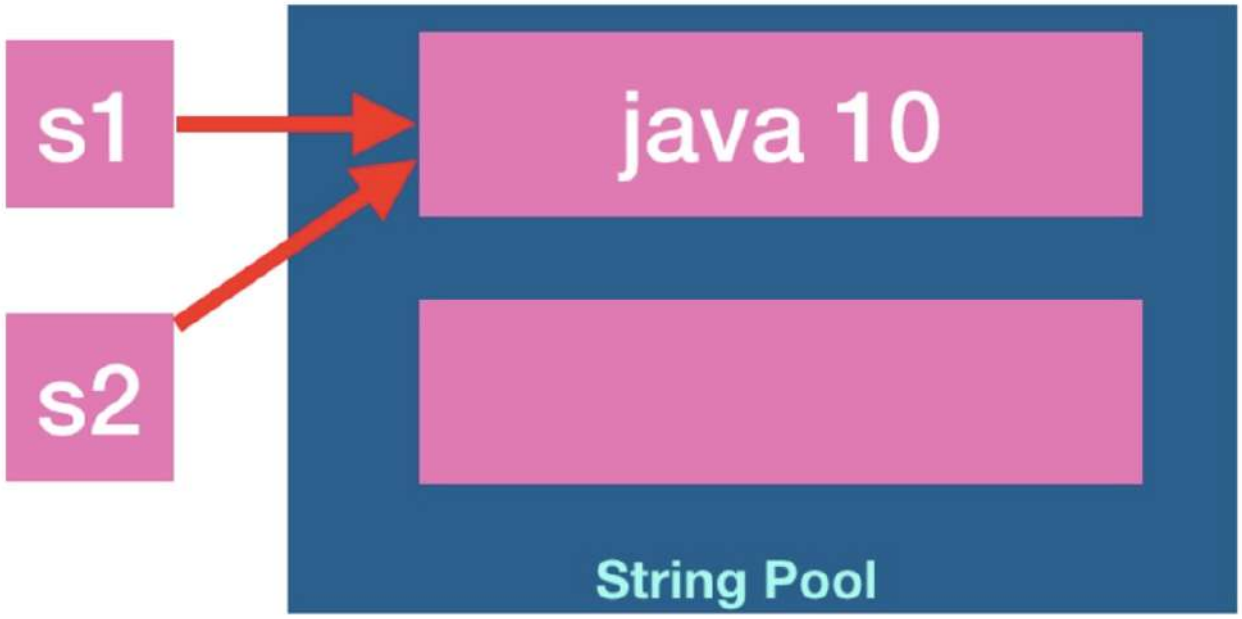
```
Libro l1= new Libro("java 10");
```

```
Libro l2= new Libro("java 10");
```

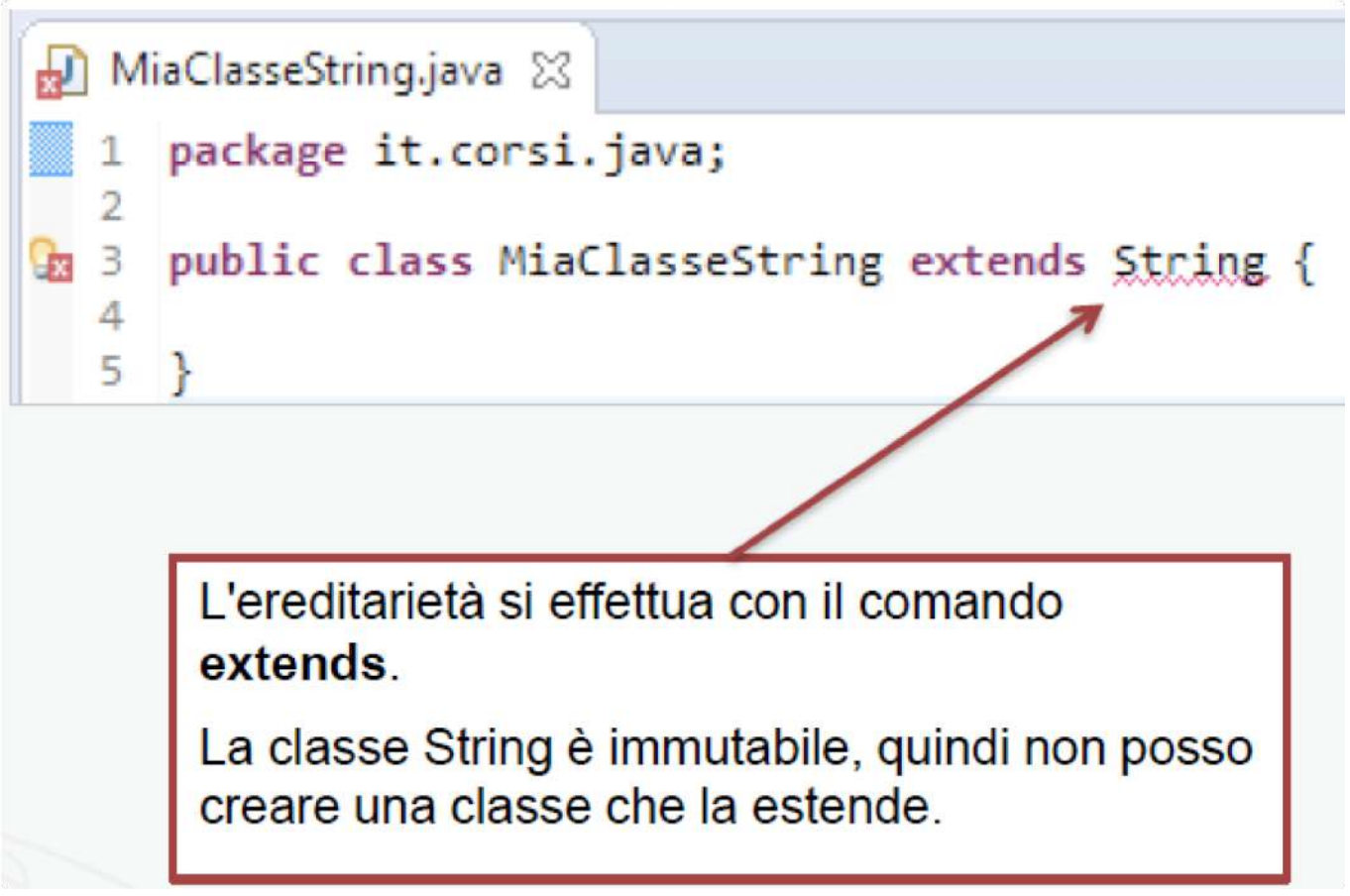
Quando abbiamo a che fare con le stringhe java gestisce un'area di memoria particolare (String Pool) che contiene i valori delle stringhe create. Se istanziamo due oggetti string che contengono lo stesso valore questi puntano alla stessa area di memoria.

```
String s1= new String("java 10");
```

```
String s2= new String("java 10");
```



La classe String non è ereditabile:



Essendo String una classe possiede dei metodi che permettono di manipolarla:

lista completa dei metodi

Lunghezza della stringa

Il metodo:

length() restituisce la lunghezza di una stringa:

String txt="ciao mondo";

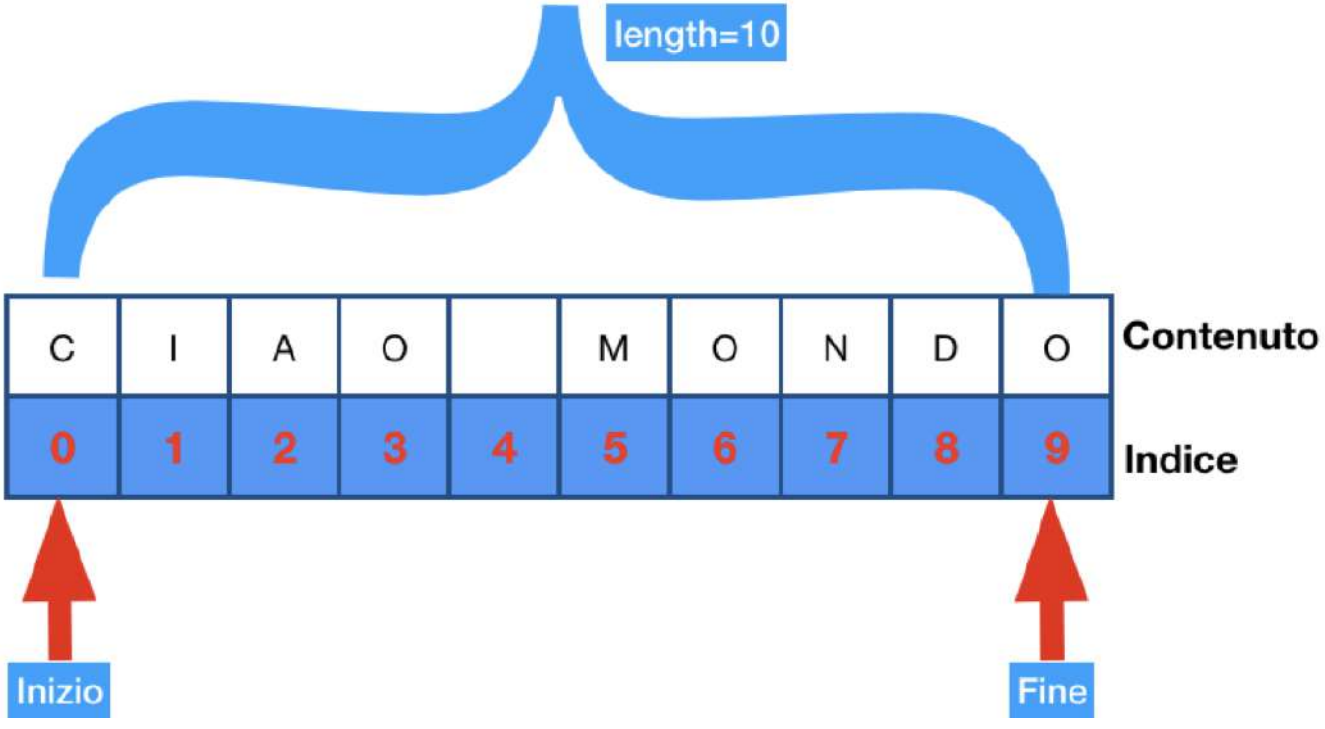
int i=txt.length();

a i viene assegnato il valore 10

Estrazione di caratteri da una stringa

charAt(int posizione):

Restituisce il carattere della posizione specificata se non viene trovato alcun carattere, restituisce una stringa vuota.



String str="Ciao modo"

str.charAt(5); restituisce il carattere 'M'

codePointAt(int index)

Restituisce il codice Unicode in corrispondenza dell'indice specificato.

String str="Ciao modo"

str.codePointAt(5);

restituisce 77

Ricerca di una stringa

string.indexOf(int ch)

- restituisce la posizione della prima occorrenza del carattere ch all'interno dell'oggetto stringa (restituisce -1 se non trovato);

string.indexOf(int ch, int start)

Parametri:

- **ch:** richiesto carattere da cercare
- **start:** indice di partenza

string.indexOf(String s)

Restituisce la prima occorrenza della sottostringa s

string.indexOf(String s, int start)

Restituisce la prima occorrenza della sottostringa s a partire da start

string.lastIndexOf(String s)

- restituisce la posizione dell'ultima occorrenza della sottostringa s all'interno dell'oggetto stringa (restituisce -1 se non trovato).:

string.lastIndexOf(String s, int start)

Parametri:

- **s:** richiesto stringa da cercare
- **start:** indice di partenza contando all'indietro

string.lastIndexOf(int ch)

- restituisce la posizione dell'ultima occorrenza di uno specifico valore all'interno dell'oggetto stringa (restituisce -1 se non trovato).:

string.lastIndexOf(int ch, int start)

Parametri:

- **ch:** richiesto carattere da cercare
- **start:** indice di partenza contando

Concatenazione

La concatenazione consiste nell'unire più stringhe in un'unica stringa. La concatenazione si può effettuare in due modi:

1. **Utilizzando l'operatore +.**
2. **Utilizzando il metodo concat.**

A una stringa è possibile concatenare anche numeri.

```
String str1 = " ciao ";
String str2 = " mondo ";
String str3 = str1.concat(" ", str2);
str3= str3+" "+123;
```

Trasformazione

Si hanno a disposizione diversi metodi che consentono di effettuare trasformazioni delle stringhe. Ad es. Si può trasformare una stringa:

- In minuscolo **toLowerCase()**.
- In maiuscolo **toUpperCase().**
- Eliminare spazi iniziali e finali **trim().**

```
String txt=" Programmare in java ";
String s=txt.trim();
String t=txt.toUpperCase();
String w=txt.toLowerCase();
```

Tutti i metodi non modificano la stringa di partenza ma creano una nuova stringa. Per modificare la stringa di partenza si deve riassegnare

```
String txt=" Programmare in java ";
txt=txt.trim();
txt=txt.toUpperCase();
```

Estrazione (sotto stringa) substring

substring(int start):

Restituisce la sottostringa che va dal carattere che si trova all'indice indicato in start fino alla fine della stringa.

substring(int start, int end):

restituisce la sottostringa che va dal carattere che si trova all'indice indicato in start fino al carattere che si trova all'indice indicato in end-1.

Codice

Confronto

La classe String mette a disposizione due metodi per effettuare il confronto tra stringhe:

equals(String s):

effettua un confronto tra due stringhe e ritorna true se sono uguali, false altrimenti.

Questo metodo è case sensitive (cioè "parola" e "Parola" sono diverse).

equalsIgnoreCase(String s):

effettua un confronto tra due stringhe e ritorna true se sono uguali, false altrimenti. Questo metodo non è case sensitive (cioè "parola" e "Parola" sono uguali).

Esempio: equals

```
String str1 = "Stringa di test";
String str2 = "stringa di TEST";
System.out.println(str1.equals(str2));
Output: false
```

Esempio: equalslenoreCase

```
String str3 = "Stringa di test";
String str4 = "stringa di TEST";
System.out.println(str3.equalslgnoreCase(str4));
Output: true
```

Sostituzione del contenuto in una stringa

Il metodo replace () sostituisce tutte le occorrenze di un valore specificato con un altro valore in una stringa:

stringa.replace(String searchvalue, String newvalue)

Parametri

- **searchvalue: Il valore da sostituire**
- **newvalue: Il nuovo valore**

Il metodo replace () non modifica la stringa su cui è chiamato. Restituisce una nuova stringa.

Per impostazione predefinita, il metodo replace () è case sensitive.

Per sostituire solo la prima occorrenza si utilizza il metodo:

stringa.replaceFirst(String regex, String newvalue)

Sostituisce la prima sottostringa di questa stringa che corrisponde all'espressione regolare data con la sostituzione data.

```
public class Main {
  public static void main(String[] args) {
    String txt="Ho un giubbotto blu e un maglione blu";
    System.out.print(txt.replaceFirst("blu","rosso"));

  }
}

Output:
Ho un giubbotto rosso e un maglione blu
```

Trasformare una stringa in un Array

split() - divide una stringa sulla base di un separatore e restituisce un array; vediamo un esempio:

```
String miaStringa = "divido la stringa in base agli spazi vuoti!";
String[] array=miaStringa.split(" "));
```

Se il separatore è omesso, l'array restituito conterrà l'intera stringa nell'indice [0].

Se il separatore è "", l'array restituito sarà un array di singoli caratteri:

```
String text="ciao mondo";
String[] array=text.split("");
```

Output

['c', 'i', 'a', 'o', ' ', 'm', 'o', 'n', 'd', 'o']

Metodi utili

- valueOf(valore): restituisce il valore primitivo Nota: questo metodo viene solitamente chiamato automaticamente da JavaScript dietro le quinte e non esplicitamente nel codice.
- startsWith(String prefix): metodo che ritorna true se la stringa inizia con il prefisso indicato, false altrimenti.
- endsWith(String suffix): metodo che ritorna true se la stringa finisce con il suffisso indicato, false altrimenti.

Esempio:

```
String str2 = "Stringa 2";
boolean starts = str2.startsWith("Str"); // ritorna true
boolean ends = str2.endsWith("Str"); // ritorna false
int numString = String.valueOf(28);
```

Il metodo:

str1.compareTo(str2)

restituisce un numero che indica se str1 viene prima, dopo, oppure è uguale a str2:

- Restituisce numero negativo se str1 è ordinato prima di str2
- Restituisce 0 se le due stringhe sono uguali
- Restituisce 1 se str1 è ordinato dopo str2

compareToIgnoreCase(String str)

Confronta le stringhe ignorando maiuscole e minuscole.

string.repeat(count)

crea una nuova stringa ripetendo il valore di string count volte.

Programmazione generica

La programmazione generica è stata introdotta in Java a partire dalla versione 5, e permette di scrivere:

- classi,
- interfacce
- metodi

in forma parametrica, generici, ovvero che compiono una medesima operazione su un insieme **di tipi di dato differenti**.

L'utilizzo più massiccio dei tipi generici è sicuramente contestuale all'utilizzo delle collection.

Se si vuole realizzare una classe (contenitore) per tipi diversi si può scrivere la classe nel seguente modo:

```
public class Contenitore {
  private Object object;
  public void setObject(Object object) {
    this.object = object;
  }
  public Object getObject() {
    return object;
  }
}
```

Il parametro object può assumere come valore qualsiasi oggetto.

La classe di Test:

```
public class Test {
  public static void main(String[] args) {
    Contenitore a = new Contenitore();
    a.setObject("ciao mondo");
    System.out.println((String) a.getObject());// cast da Object a String
    a.setObject(123);
    System.out.println((int) a.getObject());// cast da Object a int
     }
}
```

Output:

```
ciao mondo
123
```

Questa semplice classe non è facile da gestire. Infatti, una volta recuperato l'oggetto object mediante il metodo getObiect() si è obbligati a convertirlo per usarlo. Questa operazione è potenzialmente pericolosa, perché potrebbe anche recuperare un tipo diverso da quello che ci si aspetta.

Nella classe Test inseriamo volontariamente un errore:

```
public class Test {  public static void main(String[] args)
{
    Contenitore a = new Contenitore();
    a.setObject("ciao mondo");
    System.out.println((String) a.getObject());// cast da
Object a string
    a.setObject(123);
    System.out.println((int) a.getObject());// cast da
Object a int
    a.setObject(143);
    System.out.println((String) a.getObject());// casting
errato produce un errore
  }
}
```

In fase di compilazione non viene rilevato nessun problema, ma se la si esegue si ha un eccezione.

ciao mondo

123

Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String

at Test.main(Test.java:9)

Generics e tipi parametro

Per rendere il codice precedente più robusto la si può "renderla generica" aggiungendo alla definizione dei parametri usando le **parentesi angolari** che circondano gli identificatori dei tipi.

Sintassi:

class Nome <T1, T2… Tn> {

}

La classe contenitore diventa

```
public class Contenitore <T>{
private T object;
public void setObject(T object) {
    this.object = object;
}
public T getObject() {
   return object;
}
}
```

Si noti che ora al posto di Obiect c'è il **tipo parametrico T**, che non rappresenta un tipo esistente ma è un segnaposto **per qualsiasi data Type.** Questo verrà sostituito con un tipo reale nel momento in cui verrà istanziato un Contenitore.

Una classe per il test.

```
public class Main {
  public static void main(String[] args) {
    Contenitore<String> a = new Contenitore<String>();
    a.setObject("ciao mondo");
    System.out.println(a.getObject());//senza cast
```

```
    Contenitore<Integer> b = new Contenitore<Integer>();
    b.setObject(123);
    System.out.println(b.getObject());
     }
}
```

In questo modo è come se si sostituisce al parametro T prima la classe String e poi la classe Integer in tutta la definizione della classe, Quindi il metodo setObject() accetterà solo quella classe Se si passa un altro tipo si ottiene un errore in compilazione.

```
public class Main {
  public static void main(String[] args) {
    Contenitore<Integer> b = new Contenitore<Integer>();
    b.setObject(123);
    System.out.println(b.getObject());

    String valore=b.getObject());
     }
}
```

```
Main.java:7: error: incompatible types: Integer cannot be converted to String
    String valore= b.getObject();
                               ^
1 error
```

Errore ottenuto in compilazione.

Per convenzione quando si dichiara un tipo parametro si usa un identificatore costituito da una sola lettera maiuscola, che dovrebbe rappresentare l'iniziale di un nome simbolico (nel caso di T significa "Type"). In particolare la libreria standard usa spesso:

- E per "Element",
- K per "Key".
- N per "Number"
- T per "Type"
- V per "Value",
- S. U. V per il secondo, terzo e quarto tipo.

Ricapitolando la sintassi di una classe generica:

```
class NomeClasse<parametri>{
//codice
}
```

per istanziarla si usa la sintassi:

```
NomeClasse<parametri>variabile = new NomeClasse<>()
```

Facciamo un altro esempio usando la classe contenitore su delle classi fatte da noi invece che sulle classi String e Integer.

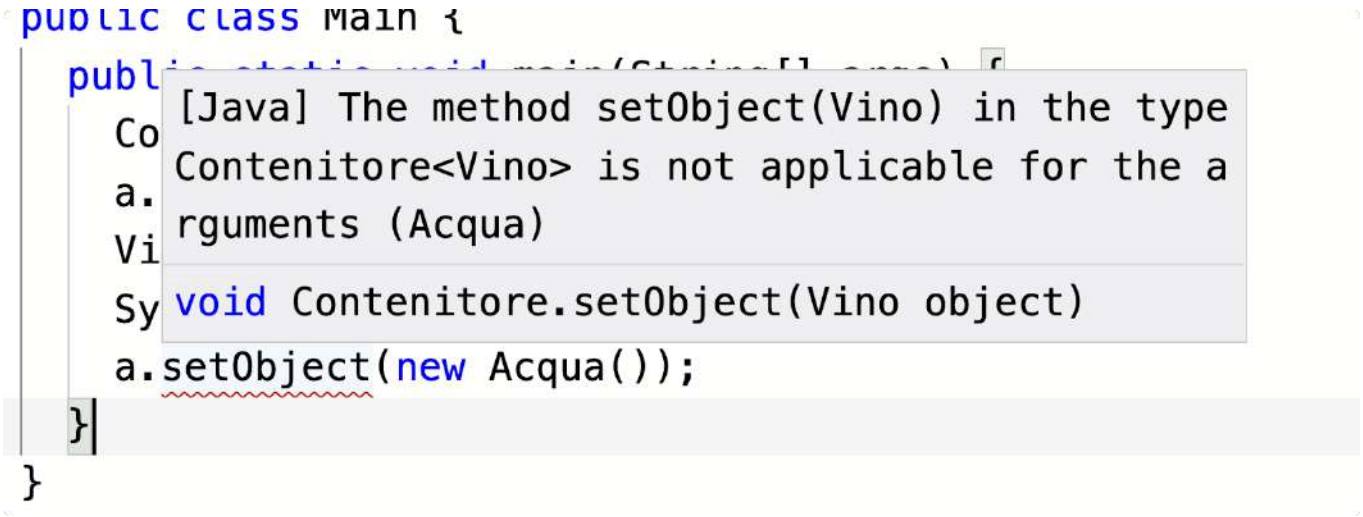
```
public class Acqua {
    public String toString(){
        return " una bottiglia d'acqua";
    }
}
```

```
public class Vino{
       public String toString(){
        return " una bottiglia di vino";
    }
 }
```

```
public class Main {
  public static void main(String[] args) {
    Contenitore<Vino> a = new Contenitore<>();
    a.setObject(new Vino());
    Vino vino=a.getObject();
```

```
    System.out.println(vino);//
    }
}
```

Se si usano altri tipi si ottengono errori in compilazione:



Codice

Classi con più parametri generici

Una classe generica può contenere più tipi di parametri riscriviamo la classe contenitore con due tipi di parametri.

```
public class Contenitore <T,V>{
private T t;
private V v;
Contenitore(T t,V v){
  this.t=t;
  this.v=v;
}
public T getT() {
   return t;
}
public V getV() {
   return v;
}
}
```

La classe contenitore si istanzia passando due parametri che possono essere: due oggetti di tipo diverso, o due oggetti dello stesso tipo il funzionamento è identico.

```
public class Main {
public static void main(String [] args) {
Contenitore<String,Integer> a= new Contenitore("java ",17);
System.out.println(a.getT()+a.getV());
Contenitore<String,String> b= new Contenitore("linguaggio
","java");
System.out.println(b.getT()+b.getV());
}
}
```

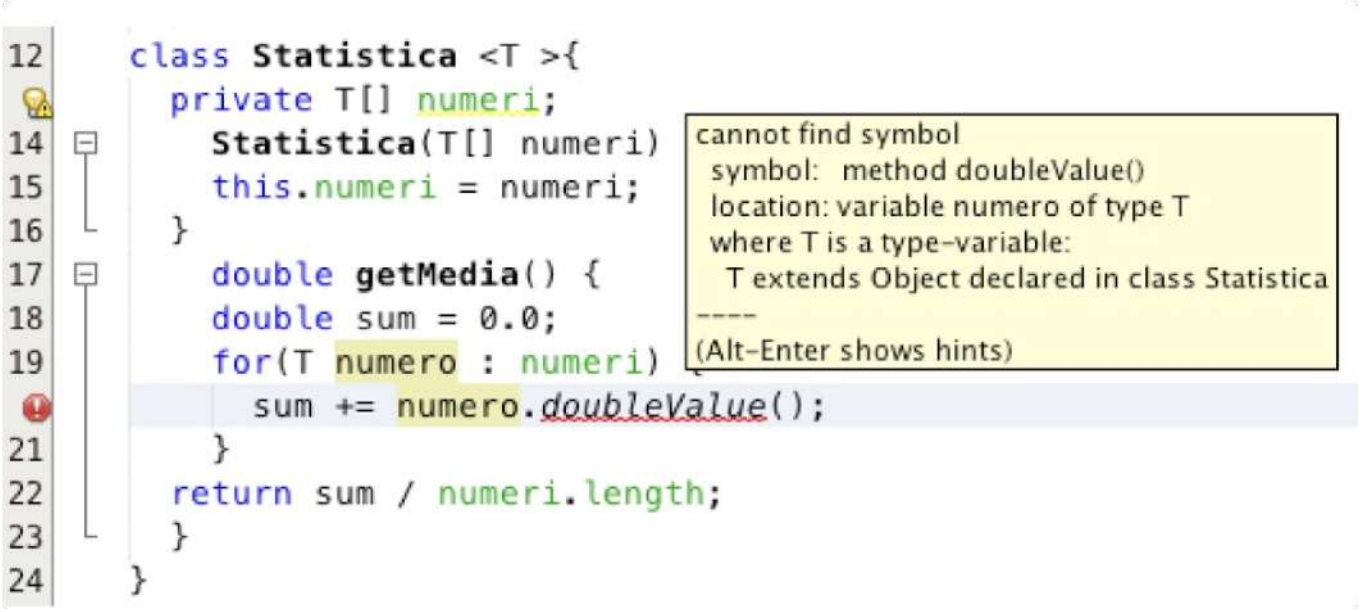
Codice

Parametri di tipo delimitati (bounded Types)

Esistono situazioni in cui si ha la necessità di applicare un limite al parametro. Scriviamo una classe Generica che si occupa di calcoli.

```
class Statistica <T>{
  private T[] numeri;
    Statistica(T[] numeri) {
    this.numeri = numeri;
  }
    double getMedia() {
    double sum = 0.0;
    for(T numero : numeri) {
      sum += numero.doubleValue();
    }
  return sum / numeri.length;
  }
}
```

Quando si compila la classe si ha un errore:



perché il metodo **doubleValue** non viene riconosciuto in quanto è un metodo **della classe Number mentre T è un tipo Object.**

Si deve fare in modo che la classe accetti solo parametri **di tipo numerico**. Per far questo si deve estendere il parametro con la classe Number, così da porre un limite superiore.

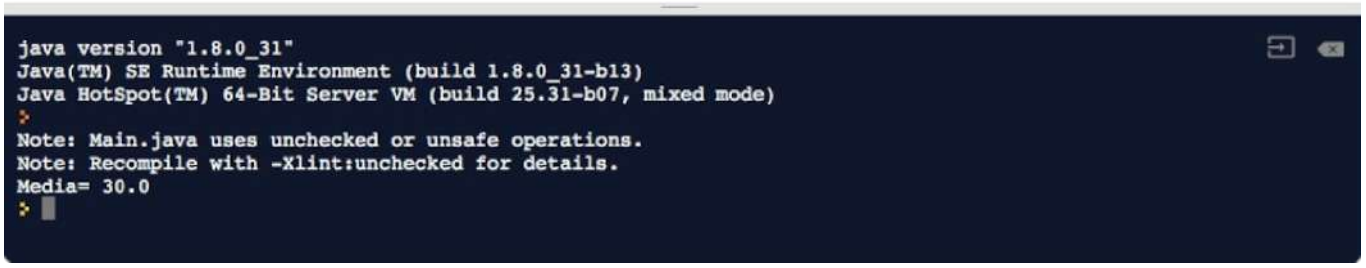
```
class Statistica <T extends Number>{
  private T[] numeri;
    Statistica(T[] numeri) {
    this.numeri = numeri;
  }
    double getMedia() {
    double sum = 0.0;
    for(T numero : numeri) {
      sum += numero.doubleValue();
    }
  return sum / numeri.length;
  }
}
```

Abbiamo specificato che il parametro **T deve essere sostituito da un tipo compatibile con Number**.

Creiamo la classe di Main per il test:

```
public class Main {
public static void main(String [] args) {
  Statistica <Integer>s;
  s = new Statistica(new Integer[]{10,20,30,40,50});
  System.out.println("Media= "+s.getMedia());

}
}
```

Output:



Codice

Metodi generici

Un metodo è detto generico quando accetta parametri **di diversi Data Type** su cui esegue lo stesso algoritmo.

Se non vi fosse la possibilità di scrivere il metodo in forma generica, per adempiere allo stesso scopo si dovrebbe ricorrere al meccanismo dell'overloading dei metodi, che comporta la necessità di scrivere tanti metodi che eseguono lo stesso compito su tipi di dato differenti.

Sintassi:

modificatore <parametri> tipoDiRitorno nomeMetodo(attributi){

//corpo}

Prima del tipo di ritorno, si scrivono dentro le parentesi angolari < > la lista dei parametri generici separati dalla virgola (.)

Esempio:

```
public static <E extends Number> void print(E[] a) {
    for (E e : a)
      System.out.print(e + "  ");
  }
```

Con questo metodo si può stampare un array di qualsiasi tipo.

```
public class Main {
  public static void main(String[] args) {
    Integer[] a = { 1, 2, 3, 4 };
    Double[] b = { 1.4, 2.5, 3.6, 4.7 };

    System.out.println("un array di interi ");
    print(a);
    System.out.println("un array di double ");
    print(b);
  }
public static <E> void print(E[] a) {
    for (E e : a)
      System.out.print(e + "  ");
   System.out.println();
  }
}
```

Output:

```
un array di interi
1  2  3  4
un array di double
1.4  2.5  3.6  4.7
```

Consideriamo il metodo generico per terminare il massimo tra tre parametri:

```
public static <T extends Comparable<T>> T max(T a, T b, T c) {
    T tmp;
    tmp = a;
    if (b.compareTo(tmp) > 0) {
      tmp = b;
    }
    if (c.compareTo(tmp) > 0) {
      tmp = c;
    }
    return tmp;
  }
```

Nel listato il metodo max che ha, nella sezione dei parametri di tipo formali. Un parametro di tipo T che estende (extends) un'interfaccia generica di tipo **Comparable**. Se si devono estendere più classi o più interfacce (nei generici la keyword extends si usa anche per le interfacce) occorre utilizzare il carattere &.

Codice

Costruttore Generico

Un costruttore di una classe può essere generico anche se la sua classe non lo è. Ad esempio possiamo inserire nella classe Contenitore un costruttore parametrizzato:

```
class Contenitore {
private Obiect t;
<T> Contenitore(T t){
this.t=t;
}
}
```

La dichiarazione del parametro T avviene tra il modificatore di accesso e la dichiarazione del costruttore.

Creiamo una classe vuota e una di prova.

```
class ClasseVuota {
}
```

```
1   public class Main {
2     public static void main(String[] args) {
3       Contenitore a = new Contenitore("Ciao mondo");
4       System.out.println(a.getT());
5       Contenitore b = new Contenitore(true);
6       System.out.println(b.getT());
7       Contenitore c = new Contenitore(new ClasseVuota());
8       System.out.println(c.getT());
9
10    }
11  }
```

```
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
Ciao mondo
true
ClasseVuota@4aa298b7
```

Codice

Interfacce generiche

Si possono realizzare anche interfacce generiche:

```
interface MiaInterfaccia<T>{
public void mioMetodo(T t);
}
```

Il metodo è stato solo dichiarato e sarà implementato all'interno di una classe che implementa l'interfaccia.

```
class MiaClasse <T> implements MiaInterfaccia<T>{
public void mioMetodo(T t){
   System.out.println(t);
 }
}
```

Se una classe implementa un'interfaccia generica allora anche la classe deve essere generica, e deve avere lo stesso numero di Type parametri e questi devono essere compatibili con quelli dell'interface.

La classe Test:

```
public class Main {
  public static void main(String[] args) {
    MiaInterfaccia<String> a = new MiaClasse<>();
    MiaInterfaccia<Integer> b = new MiaClasse<>();
   a.mioMetodo("java");
   b.mioMetodo(8);
  }
}
```

Codice

L'INTERFACCIA COMPARABLE

Abbiamo visto che la classe Arrays del package java.util, definisce il metodo statico sort in grado di ordinare un array di valori primitivi o di oggetti, l'istruzione:

Arrays.sort(unArray);

ordina gli elementi dell'intero array in senso crescente.

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    int[] interi={5,7,8,9,1,3,2,10};
    String[] lista={"pane","pasta","acqua","frutta"};
    System.out.println("array di numeri interi non ordinato");
    System.out.println(Arrays.toString(interi));
      System.out.println("array di stringe non ordinato");
    System.out.println(Arrays.toString(lista));
    Arrays.sort(interi);
    Arrays.sort(lista);
System.out.println("array di numeri interi  ordinato");
    System.out.println(Arrays.toString(interi));
      System.out.println("array di string   ordinato");
    System.out.println(Arrays.toString(lista));
```

```
  }
}
```

Se si vuole ordinare un array di oggetti con Arrays.sort, l'oggetto deve implementare l'interfaccia Comparable.

L'interfaccia Comparable contiene l'intestazione del solo metodo compareTo, che deve essere quindi definito per ogni classe che implementi l'interfaccia:

public int compareTo(Object a)

L'interfaccia consente di specificare come un oggetto vada confrontato con un altro definendo quando uno dei due "viene prima", "viene dopo" o "è uguale" all'altro. Il metodo compareTo dovrà restituire:

- un numero negativo, se l'oggetto sul quale è chiamato "viene prima" del parametro dell'altro;
- zero, se l'oggetto sul quale è chiamato "è uguale" al parametro dell'altro;
- un numero positivo, se l'oggetto sul quale è chiamato "viene dopo" il parametro dell'altro.

A partire da java 5.0 l'interfaccia Comparable è di tipo parametrico:

```
public interface Comparable<T> {
  int compareTo(T other);
}
```

Il parametro specifica il tipo degli oggetti che una certa classe accetta per fare confronti; **di solito si tratta della stessa classe.**

Il vantaggio è che con la genericità non c'è bisogno di usare un cast per convertire un parametro di tipo Object nel tipo desiderato.

Come esempio, si consideri la classe persona

```
public class Persona{
```

```
   private String nome;
  private String cognome;
  public Persona(String cognome, String nome){
    this.nome=nome;
    this.cognome=cognome;
  }
  public String toString(){
    return cognome+";  "+nome ;
  }
}
```

E si vuole ordinare in modo naturale cognome nome.

```
import java.util.*;
public class Persona implements Comparable<Persona>{
  private String nome;
  private String cognome;
  public Persona(String cognome, String nome){
    this.nome=nome;
    this.cognome=cognome;
  }
  public String toString(){
    return cognome+";  "+nome ;
  }
    @Override
    public int compareTo(Persona o) {
        if(cognome.compareToIgnoreCase(o.cognome)==0){
         return nome.compareTo(o.nome);//se i cognomi sono uguali
ordina per nome
        }
       return cognome.compareToIgnoreCase(o.cognome);
    }
}
```

La classe Test:

```
public class Main {
```

```
  public static void main(String[] args) {
    Persona[] persone=new Persona[4];
    persone[0]=new Persona("Fantozzi","Ugo");
    persone[1]=new Persona("Fantozzi","Pina");
    persone[2]=new Persona("Rossi","Mario");
    persone[3]=new Persona("Bianchi","Anna");
      System.out.println("array di persone ordinato");
    Arrays.sort(persone);
    for(Persona p:persone)
System.out.println(p);
  }
}
```

Output:

```
array di persone ordinato
Bianchi;  Anna
Fantozzi;  Pina
Fantozzi;  Ugo
Rossi;  Mario
```

Codice

L'INTERFACCIA COMPARATOR

Quando si implementa Comparable si sta definendo un ordinamento naturale infatti si scrive all'interno della classe un ordinamento. E' possibile specificare solo un criterio di ordinamento. Quando si ha bisogno di un ordinamento particolare (o di più di un ordinamento) allora si usa Comparator. È il caso di persone che si potrebbero voler ordinare non solo per cognome e nome ma anche per età, per altezza.

L'interfaccia Comparator

definisce un solo metodo:

```
public interface Comparator<T>{
int compare(T o1, T o2);
}
```

L'interfaccia Comparator deve essere implementata in una classe distinta, dalla classe degli oggetti da comparare. Questo significa che è possibile creare più classi per definire un numero arbitrario di criteri di comparazione alternativi o comunque più particolari rispetto all'ordinamento "naturale" espresso tramite Comparable.

Riprendiamo la classe Persona aggiungendo come attributo l'anno di nascita:

```
import java.util.*;
public class Persona implements Comparable<Persona>{
  private String nome;
  private String cognome;
  private int annoNascita;
  public Persona(String cognome, String nome,int
annoNascita){
    this.nome=nome;
    this.cognome=cognome;
    this.annoNascita=annoNascita;
  }
  public String toString(){
    return cognome+";  "+nome+" ; "+annoNascita ;
  }
  public int getAnnoNascita() {
        return annoNascita;
    }
    @Override
```

```
    public int compareTo(Persona o) {
        if(cognome.compareToIgnoreCase(o.cognome)==0){
         return nome.compareTo(o.nome);//se i cognomi
sono uguali ordina per nome
        }
       return cognome.compareToIgnoreCase(o.cognome);
    }
}
```

La classe per il confronto

```
import java.util.Comparator;
public class ConfrontaAnno implements Comparator<Persona> {
  @Override
  public int compare(Persona persona1, Persona persona2) {
    int r;
    if (persona1.getAnnoNascita() < persona2.getAnnoNascita()) {
      r = +1;
    } else if (persona1.getAnnoNascita() >
persona2.getAnnoNascita()) {
      r = -1;
    } else {
      r = 0;
    }
    return r;
  }
}
```

E la classe Main:

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    Persona[] persone=new Persona[4];
    persone[0]=new Persona("Fantozzi","Ugo",1945);
    persone[1]=new Persona("Fantozzi","Pina",1952);
    persone[2]=new Persona("Rossi","Mario",2001);
    persone[3]=new Persona("Bianchi","Anna",2000);
      System.out.println("array di persone ordinato per cognome e
```

```
nome");
    Arrays.sort(persone);
    for(Persona p:persone)
System.out.println(p);
System.out.println("array di persone ordinato per anno di
nascita");
    Arrays.sort(persone,new ConfrontaAnno());
    for(Persona p:persone)
System.out.println(p);
  }
}
```

Output:

```
array di persone ordinato per cognome e nome
Bianchi;  Anna ; 2000
Fantozzi;  Pina ; 1952
Fantozzi;  Ugo ; 1945
Rossi;  Mario ; 2001
array di persone ordinato per anno di nascita
Rossi;  Mario ; 2001
Bianchi;  Anna ; 2000
Fantozzi;  Pina ; 1952
Fantozzi;  Ugo ; 1945
```

Codice

Collection

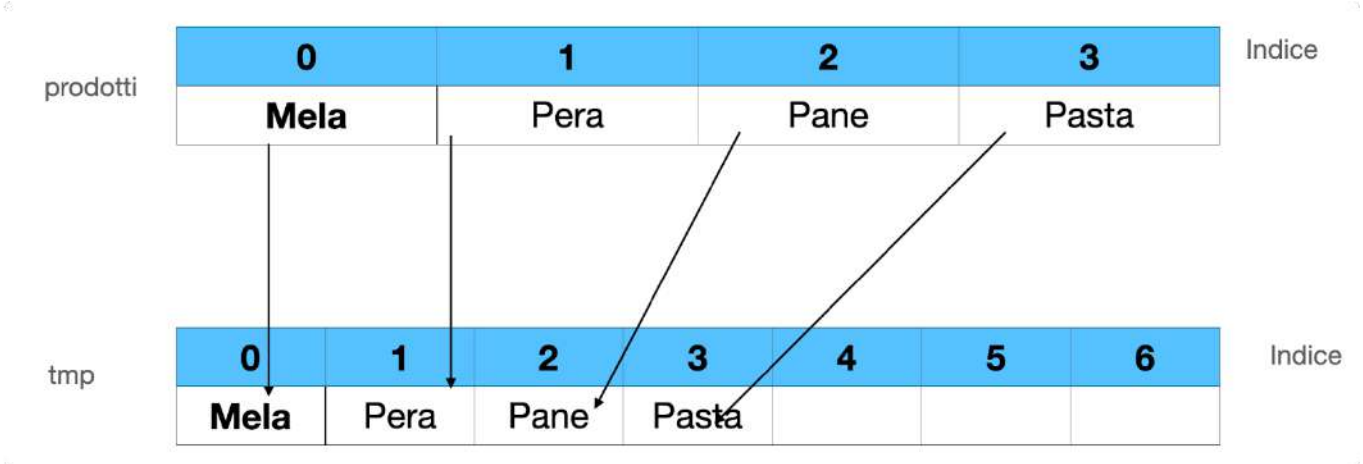
In Java, la lunghezza di un array non può essere cambiata. Per esempio, se si scrive un programma che registra in un array i prodotti presenti in un magazzino. Si potrebbe chiedere all'utente il numero di prodotti da memorizzare e poi creare l'array utilizzando la seguente istruzione:

```
Prodotto[] prodotti = new Prodotto[numeroProdotti];
```

Ma cosa accade se l'utente inserisce numeroProdotti ma poi decide d'inserire un altro prodotto?

Non esiste alcuna possibilità d'incrementare le dimensioni dell'array. Per fare questo si deve creare un nuovo array più grande:

copiare gli elementi dall'array originale al nuovo array



e poi rinominare il nuovo array come prodotti. Per esempio, le seguenti istruzioni raddoppiano le dimensioni dell'array:

```
class Main {
  static int inseriti = 0;
  static Scanner tastiera = new Scanner(System.in);

  public static void main(String[] args) {
    System.out.println("Inserisci il numero di prodotti");
    int numeroProdotti = tastiera.nextInt();
    tastiera.nextLine();
    Prodotto[] prodotti = new Prodotto[numeroProdotti];
    for(int i=0;i<prodotti.length;i++)
    prodotti=inserisci(prodotti);
    System.out.println("lista
prodotti\n"+Arrays.toString(prodotti));
   prodotti=inserisci(prodotti);
    System.out.println("Nuova lista
prodotti\n"+Arrays.toString(prodotti));

  }

  public static Prodotto[] inserisci(Prodotto[] p) {
    System.out.println("Inserisci Prodotto");
    String nome = tastiera.next();
    if (inseriti < p.length) {
      p[inseriti] = new Prodotto(nome);
      inseriti++;
    } else {
      Prodotto[] tmp = new Prodotto[2 * p.length];
      for (int i = 0; i < p.length; i++)
        tmp[i] = p[i];
      tmp[inseriti] = new Prodotto(nome);
      p = tmp;
      inseriti++;
    }
    return p;
  }
}
```

Output:

```
Inserisci il numero di prodotti
2
Inserisci Prodotto
Mela
Inserisci Prodotto
Pera
lista prodotti
[Mela, Pera]
Inserisci Prodotto
Pane
Nuova lista prodotti
[Mela, Pera, Pane, null]
```

Codice

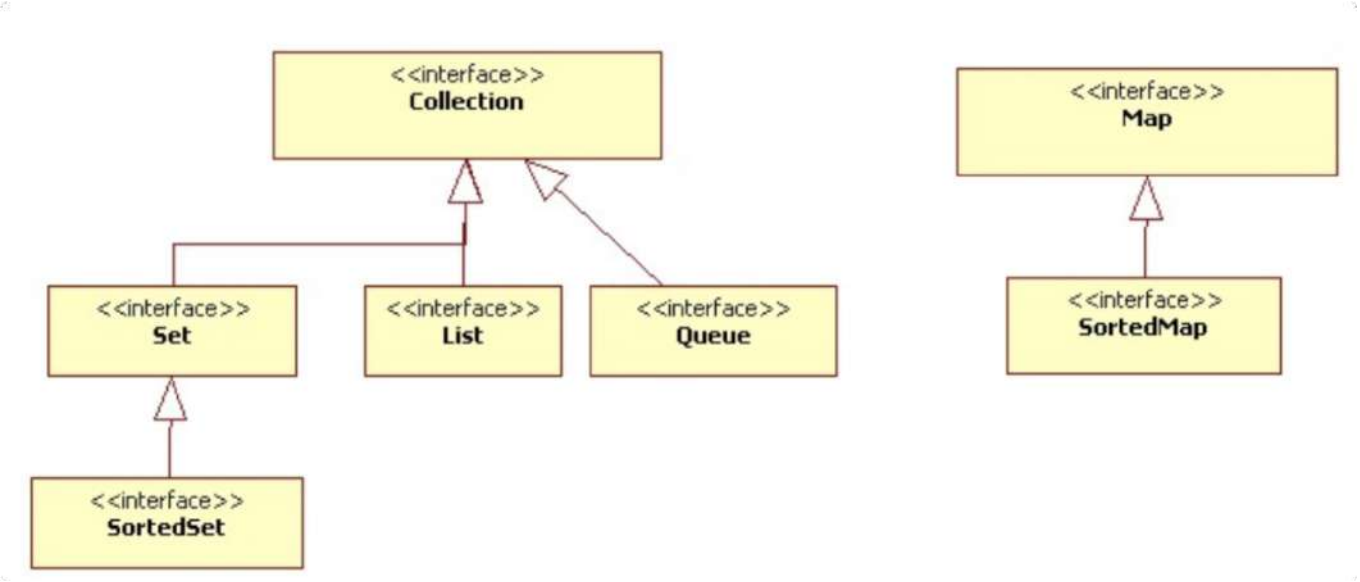
Per andare oltre questo limite, Java mette a disposizione, **nel framework collection (JFC)** una serie strumenti che consentono di gestire liste di oggetti di lunghezza variabile. In Java si hanno due concetti distinti:

1.Collection : un gruppo di singoli elementi

2. Map: un gruppo di coppie di oggetti chiave-valore

Collection<T>

Costituisce l'interfaccia più generica della libreria Java che contiene le classi per le collezioni. Questa interfaccia descrive le operazioni di base che devono essere implementate da tutte le classi di tipo collezione. Ci sono molte classi predefinite che implementano l'interfaccia Collection<T> ed è possibile definirne di nuove. Un metodo scritto per lavorare su un parametro di tipo Collection<T> funzionerà anche con tutte queste classi. Inoltre, i metodi dell'interfaccia Collection<T> garantiscono la possibilità di utilizzare contemporaneamente diverse classi di tipo collezione.



- Collection: nessuna ipotesi su elementi duplicati o relazioni d'ordine
- List: introduce l'idea di sequenza
- Set: introduce l'idea di insieme di elementi quindi senza duplicati
- SortedSet: Insieme ordinato
- Map: introduce il concetto di mappa cioè di insieme che associa chiavi (identificatori univoci) a valori

Il collections framework (package java.util) è costituito dai seguenti componenti architetturali:

1. **interfacce**, rappresentate dai tipi di dati astratti che modellano le strutture di dati quali per esempio: Set, List, Queue, Map e Deque,
2. **le classi concrete** che realizzano le interfacce: HashSet, ArrayList, HashMap, TreeSet, TreeMap, LinkedList. LinkedHashSet. LinkedHashMap
3. **algoritmi,** rappresentati dalle operazioni che possiamo effettuare sulle strutture dati, come: ricerca (searching), ordinamento (sorting), mescolamento (shuffling) implementati in appositi metodi statici della classe java.util.Collections
4. **costrutti di attraversamento** delle collezioni, rappresentati dal costrutto for avanzato e da un oggetto definito iteratore (Iterator).

Interfacce	Implementazioni			
	Hash Table	Resizable Array	Balanced Tree	Linked List
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

Sono presenti almeno due implementazioni per ogni interfaccia:

Implementazioni primarie HashSet ArrayList HashMap

TreeSet e TreeMap implementano SortedSet e SortedMap

COSTRUTTORI

Nonostante non sia espressamente richiesto dall'interfaccia, qualunque classe che implementi l'interfaccia Collection<T> dovrebbe avere almeno due costruttori:

1. un costruttore senza argomenti che crea un oggetto di tipo Collection<T> vuoto
2. un costruttore con un parametro di tipo Collection<? extends T> che crea un oggetto di tipo Collection<T> contenente gli stessi elementi dell'argomento.

Un'interfaccia non si può istanziare, ecco perché bisogna istanziare una delle classi che implementano l'interfaccia:

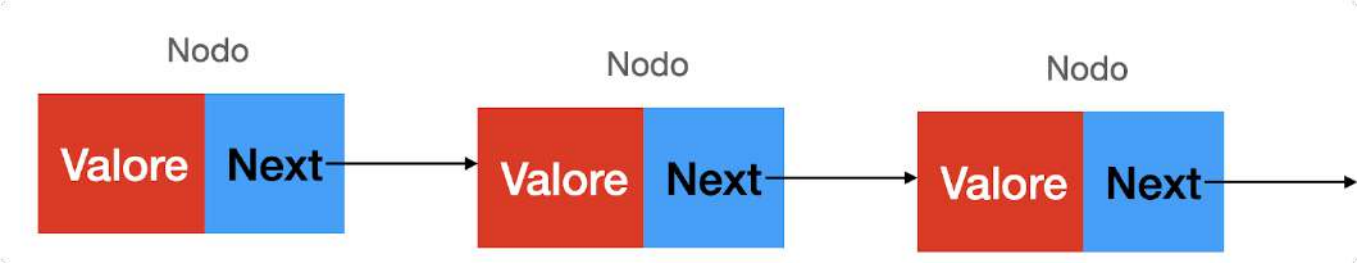
'ArrayList() o LinkedList().

Tale istanza può cambiare la propria lunghezza durante l'esecuzione del programma.

La classe ArrayList si basa comunque su array. Di fatto, per estendere la capacità del suo array, ArrayList utilizza la tecnica utilizzata precedentemente per estendere l'array prodotti.

```
Collection<E> list = new ArrayList<>();
```

La classe LinkedList, è sviluppata avvalendosi della struttura di dati di tipo lista concatenata. Una lista concatenata è composta da un certo numero di nodi, ciascuno dei quali viene creato in runTime e contiene un riferimento al nodo successivo.



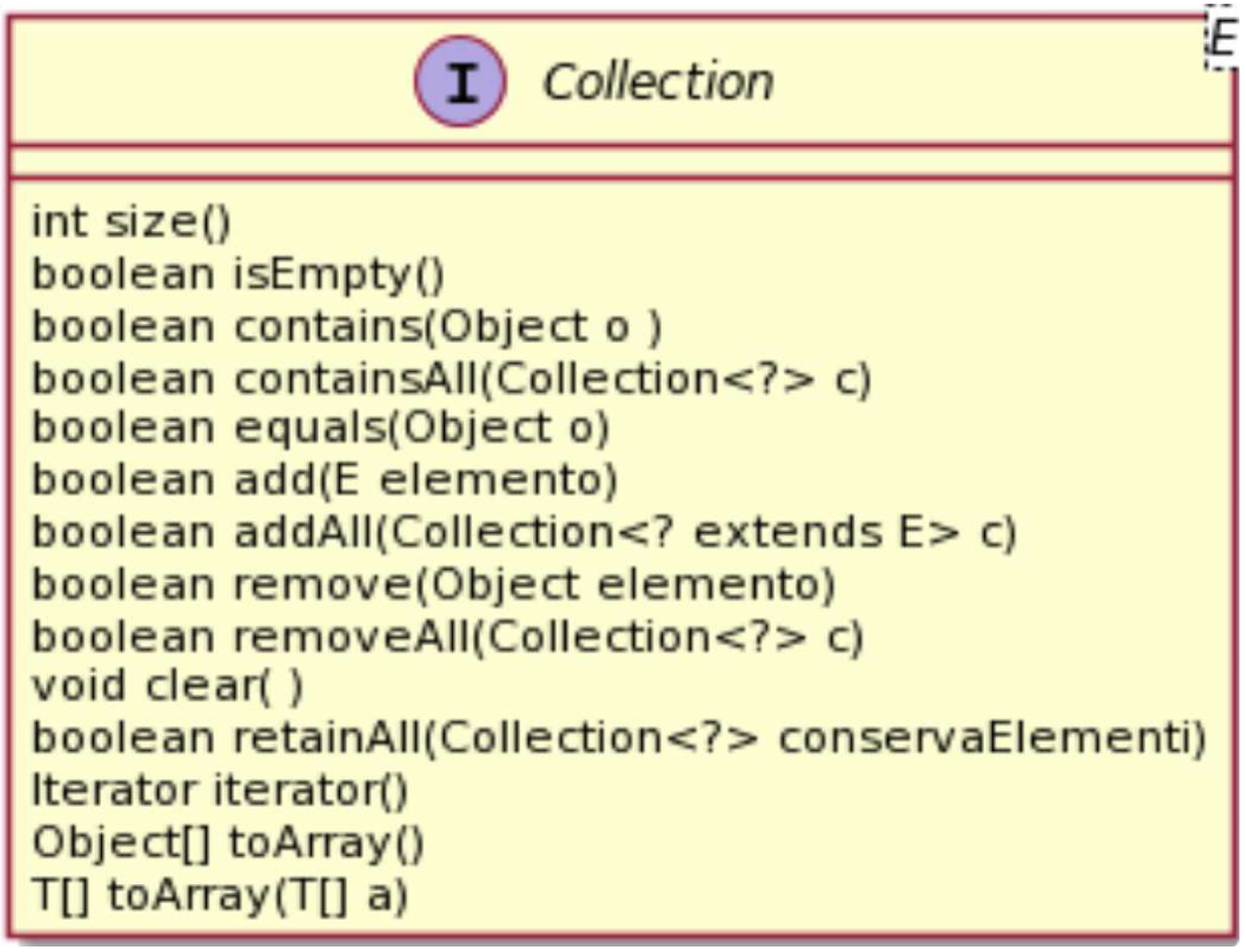
Lista Concatenata

Alcune liste concatenate, poi, soddisfano anche la seguente proprietà:

Ciascun Nodo contiene anche un collegamento al Nodo precedente questa viene detta lista doppiamente concatenata.

Lista doppiamente concatenata.

```
Collection<E>lista= new LinkedList<>();
```



public int size() Restituisce il numero di elementi contenuti nell'oggetto chiamante.

public **boolean isEmpty()** Restituisce true se l'oggetto chiamante è vuoto, altrimenti restituisce false.

public **boolean contains(Object o)** Restituisce true se l'oggetto chiamante contiene almeno un'istanza di o.

public boolean containsAll(Collection<?> c) Restituisce true se l'oggetto chiamante contiene tutti gli elementi in **c**. Per ogni elemento in **c**, il metodo usa elemento .equals per determinare se elemento è contenuto nell'oggetto chiamante.

public boolean equals(Object altro) Questo è il metodo equals per la collezione, non per gli elementi in essa contenuti. Sovrascrive il metodo equals ereditato.

public boolean add(E elemento) Garantisce che l'oggetto chiamante contenga l'elemento specificato. Restituisce true se l'oggetto chiamante è stato modificato dalla chiamata. Restituisce false se l'oggetto chiamante non ammette elementi duplicati e contiene già elemento; inoltre, restituisce false anche se l'oggetto chiamante non viene modificato per qualunque altro motivo.

public boolean addAll(Collection<? extends E> collezioneDaAggiungere) Garantisce che l'oggetto chiamante contenga tutti gli elementi in collezioneDaAggiungere. Restituisce true se l'oggetto chiamante è stato modificato dalla chiamata, altrimenti restituisce false.

public boolean remove(Object elemento) Rimuove una singola istanza dell'elemento specificato dall'oggetto chiamante. Restituisce true se l'oggetto chiamante conteneva l'elemento, altrimenti restituisce false.

public boolean removeAll(Collection<?> collezioneDaRimuovere) Rimuove dall'oggetto chiamante tutti gli elementi che sono contenuti anche in collezioneDaRimuovere. Restituisce true se l'oggetto chiamante è stato modificato, altrimenti restituisce false.

public void clear( ) Rimuove tutti gli elementi dall'oggetto chiamante.

public boolean retainAll(Collection<?> conservaElementi) Mantiene nell'oggetto chiamante tutti gli elementi contenuti anche nella collezione conservaElementi. In altre parole, rimuove dall'oggetto chiamante tutti gli oggetti non contenuti nella collezione conservaElementi. Restituisce true se l'oggetto chiamante è stato modificato dalla chiamata, altrimenti restituisce false.

Le caratteristiche dell'interfaccia Collection:

• non definisce l'ordine in cui sono memorizzati gli elementi

• non definisce se ci possono essere elementi duplicati

• non può contenere tipi primitivi ma solamente oggetti.

Per inserire tipi primitivi è necessario effettuare il boxing. Dalla versione 1.5 di Java, il boxing viene fatto automaticamente (si parla di autoboxing).

Nell'esempio usiamo la Classe ArrayList ma lo stesso funziona con La classe LinkedList

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedList;
public class Main {
  public static void main(String args[]) {
    Collection<String> list = new ArrayList<>();
    System.out.println(String.format("%40s %25s %35s",
"Istruzione", "Lista", "size"));
    System.out.println(String.format("%s",

"-----------------------------------------------------------------
-----------------------------------------------"));
    stampaList("List<String> list = new ArrayList<>();", list);
    list.add("Arance");
    stampaList("list.add(\"Arance\");", list);
    list.add("Pere");
    stampaList("list.add(\"Pere\");", list);
    list.add("Mele");
    stampaList("list.add(\"Mele\");", list);
    list.add("Melone");
    stampaList("list.add(\"Melone\");", list);
    list.add("Mele");
    stampaList("list.add(\"Mele\");", list);
    list.remove("Mele"); // eliminamo l'elemento Mele
    stampaList("list.remove(\"Mele\");", list);
Collection<String>lista= new LinkedList<>(list);
    stampaList("Collection<String>lista= new LinkedList<>(list);",
lista);

    list.clear();// cancella la lista
    stampaList("list.clear();", list);
  }

  private static void stampaList(String a, Collection<String>
list) {
    System.out.println(String.format("%40s %30s %30s ", a, list,
list.size()));
  }
```

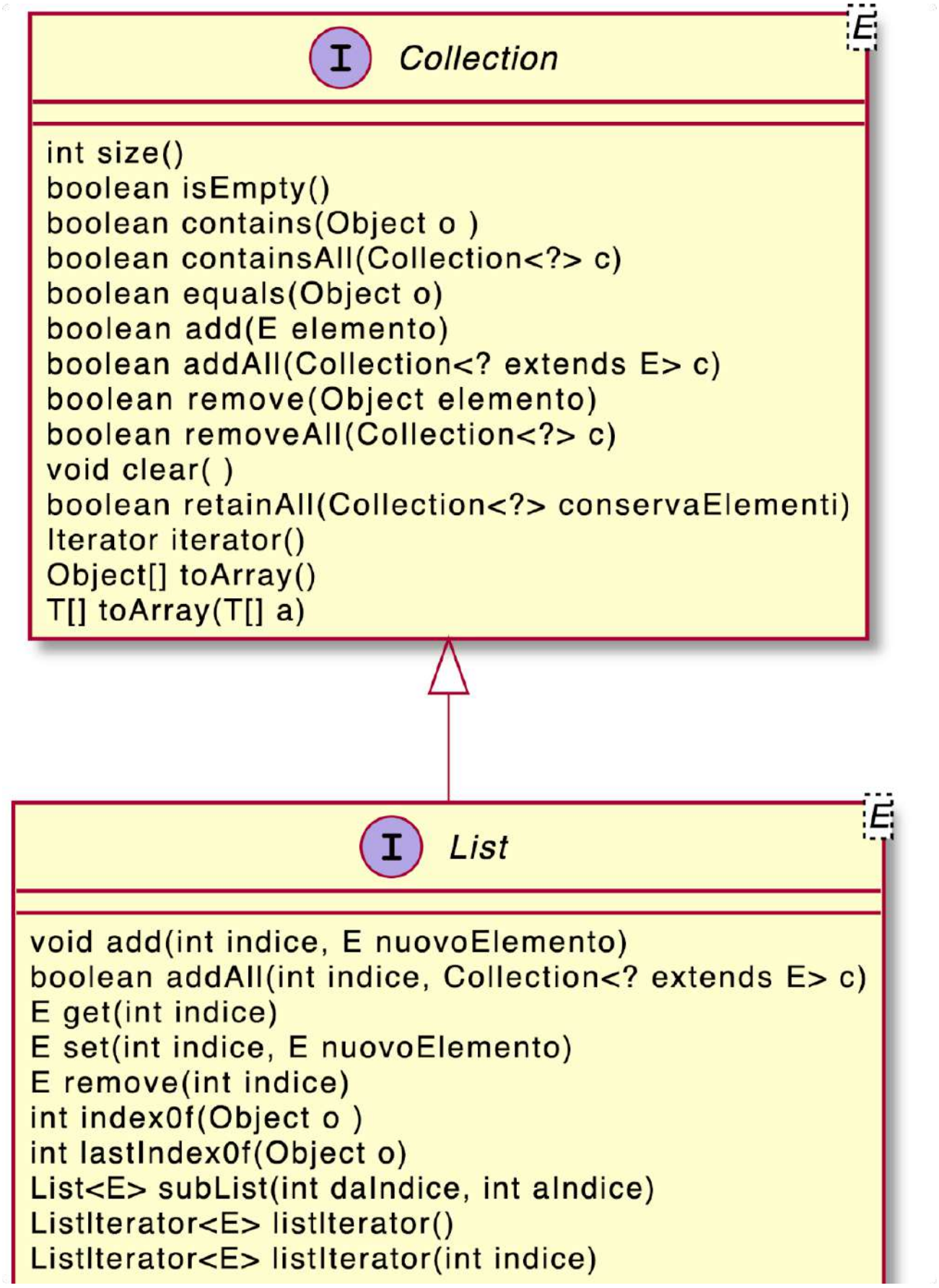
```
}
```

Output:

```
                          Istruzione                  Lista                              size
-----------------------------------------------------------------------------------------------
  List<String> list = new ArrayList<>();                 []                               0
                    list.add("Arance");                  [Arance]                         1
                      list.add("Pere");                  [Arance, Pere]                   2
                      list.add("Mele");                  [Arance, Pere, Mele]             3
                    list.add("Melone");                  [Arance, Pere, Mele, Melone]     4
                      list.add("Mele");                  [Arance, Pere, Mele, Melone, Mele]       5
                   list.remove("Mele");                  [Arance, Pere, Melone, Mele]     4
Collection<String>lista= new LinkedList<>(list);         [Arance, Pere, Melone, Mele]               4
                         list.clear();                   []                               0
```

[Codice:]

Interfaccia List



L'interfaccia List estende l'interfaccia Collection con l'aggiunta di alcuni metodi che operano con indici ed ha le seguenti caratteristiche:

• gli oggetti sono ordinati in base all'ordine di inserimento

• può contenere duplicati

• consente di aggiungere elementi specificando l'indice (ad es. è possibile inserire un elemento nella posizione 5)

- consente di ottenere gli elementi specificando l'indice

Metodi una aggiunta ai metodi di Collection

public void add(int indice, E nuovoElemento) Inserisce nuovoElemento alla posizione indice nella lista di elementi dell'oggetto chiamante. L'elemento che si trovava alla posizione indice e tutti i successivi vengono spostati di una posizione.

public boolean addAll(int indice, Collection<? extends E> collezioneDaAggiungere) Inserisce tutti gli elementi di collezioneDaAggiungere nella lista di elementi dell'oggetto chiamante a partire dalla posizione indice. L'elemento originariamente alla posizione indice e i successivi vengono spostati più avanti. Gli elementi sono aggiunti nello stesso ordine in cui sono forniti da un iteratore di collezioneDaAggiungere.

public E get(int indice) Restituisce l'elemento indicato dall'indice.

public E set(int indice, E nuovoElemento) Imposta l'elemento alla posizione indice a nuovoElemento. Viene restituito l'elemento che si trovava originariamente in quella posizione.

public E remove(int indice) Rimuove l'elemento alla posizione indice dell'oggetto chiamante. Sposta gli elementi successivi a sinistra di una posizione (sottrae 1 ai loro indici). Restituisce l'elemento rimosso.

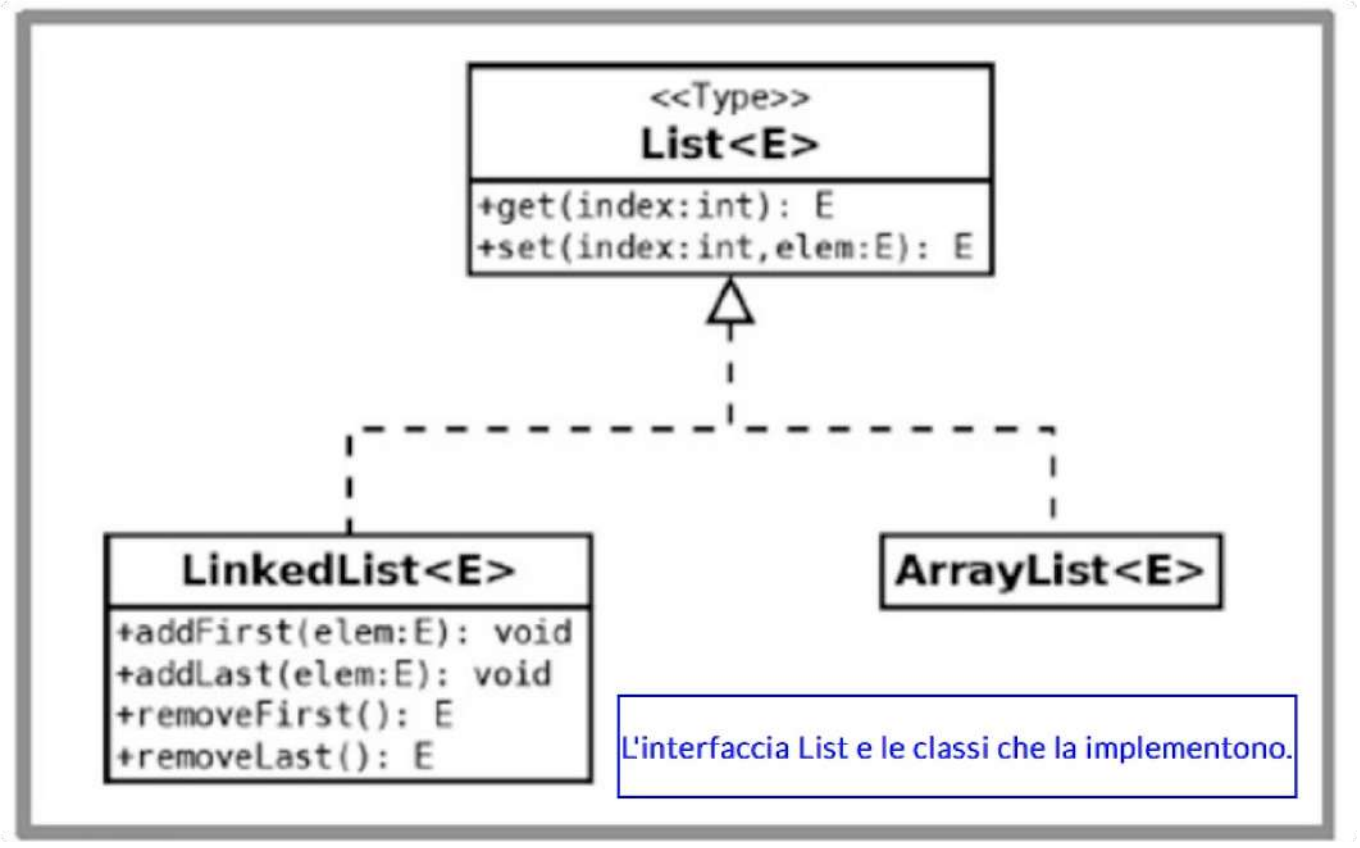
public int index0f(Object obiettivo) Restituisce l'indice del primo elemento uguale a obiettivo. Utilizza il metodo equals dell'oggetto obiettivo per verificare l'uguaglianza. Restituisce -1 se obiettivo non viene trovato.

public int lastIndex0f(Object obiettivo) Restituisce l'indice dell'ultimo elemento uguale a obiettivo. Utilizza il metodo equals dell'oggetto obiettivo per verificare l'uguaglianza. Restituisce -1 se obiettivo non viene trovato.

public List<E> subList(int daIndice, int aIndice) Restituisce una vista degli elementi alle posizioni comprese tra daIndice a aIndice dell'oggetto chiamante; l'oggetto alla posizione daIndice è incluso, quello alla posizione aIndice (se presente) è escluso. La vista è composta di riferimenti all'oggetto chiamante; le modifiche alla vista quindi modificano potenzialmente l'oggetto chiamante. L'oggetto restituito è di tipo List<T>, ma non è necessario che sia dello stesso tipo dell'oggetto chiamante. Se daIndice coincide con aIndice, viene restituito un oggetto List<E> vuoto.

ListIterator<E> listIterator() Restituisce un iteratore per l'oggetto chiamante (gli iteratori per le collezioni sono trattati successivamente).

ListIterator<E> listIterator(int indice) Restituisce un iteratore per l'oggetto chiamante che parte da indice. Il primo elemento restituito dall'iteratore è quello alla posizione indice.



La classe ArrayList realizza l'interfaccia List mediante un array, mentre la classe LinkedList realizza l'interfaccia List usando la struttura lista concatenata. Aggiungere e rimuovere elementi in una lista concatenata è un'operazione efficiente. Visitare in sequenza gli elementi di una lista concatenata è efficiente, ma accedervi in ordine casuale non lo è.

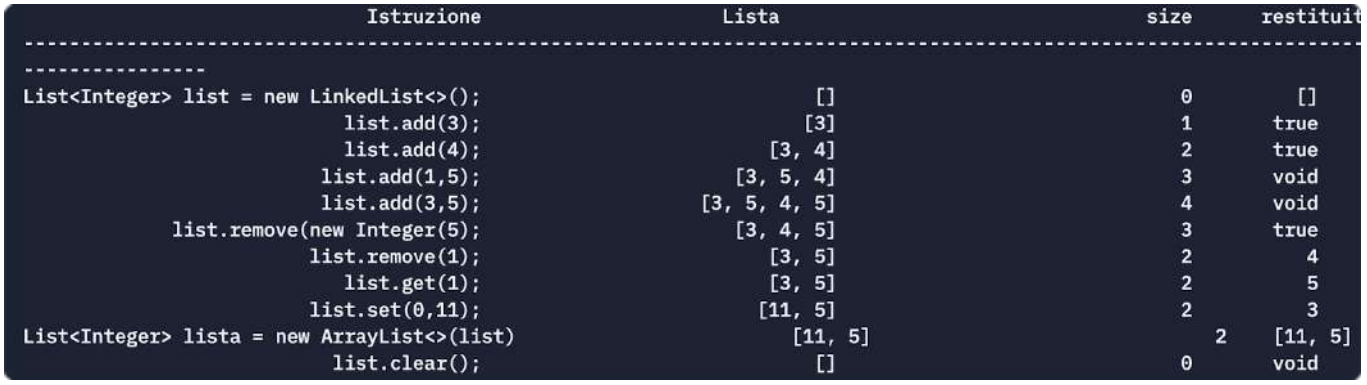
```
public class Main {
```

```
  public static void main(String args[]) {
  List<Integer> list = new LinkedList<>();
    System.out.println(String.format("%40s %25s %35s %15s ",
"Istruzione", "Lista", "size","restituito"));
    System.out.println(String.format("%s",

"-----------------------------------------------------------------
-----------------------------------------------------------------
----------------"));
  stampaList("List<Integer> list = new LinkedList<>();",
list,"[]");
    String b=""+list.add(3);
    stampaList("list.add(3);", list,b);
    b=""+list.add(4);
    stampaList("list.add(4);", list,b);
   list.add(1,5);
    stampaList("list.add(1,5);", list,"void");
    list.add(3,5);
    stampaList("list.add(3,5);", list,"void");
    b=""+list.remove(new Integer(5)); // elimina l'elemento 5
    stampaList("list.remove(new Integer(5);", list,b);
    b=""+ list.remove(1); // elimina l'elemento indice 1
    stampaList("list.remove(1);", list,b);
    b=""+ list.get(1); // restituisce l'elemento indice 1
    stampaList("list.get(1);", list,b);
    b=""+ list.set(0,11); // modifica l'elemento di indice 0
    stampaList("list.set(0,11);", list,b);
    List<Integer> lista = new ArrayList<>(list);
    stampaList("List<Integer> lista = new ArrayList<>(list)",
lista,lista.toString());
   list.clear();// cancella la lista
    stampaList("list.clear();", list,"void");

  }

  private static void stampaList(String a, List<Integer>
list,String b) {
    System.out.println(String.format("%40s %30s %30s %10s ", a,
list, list.size(),b));
  }
```

```
}
```

Codice

Output:



Istruzione	Lista	size	restituit
List<Integer> list = new LinkedList<>();	[]	0	[]
list.add(3);	[3]	1	true
list.add(4);	[3, 4]	2	true
list.add(1,5);	[3, 5, 4]	3	void
list.add(3,5);	[3, 5, 4, 5]	4	void
list.remove(new Integer(5);	[3, 4, 5]	3	true
list.remove(1);	[3, 5]	2	4
list.get(1);	[3, 5]	2	5
list.set(0,11);	[11, 5]	2	3
List<Integer> lista = new ArrayList<>(list)	[11, 5]	2	[11, 5]
list.clear();	[]	0	void

La classe ArrayList<E>

La classe ArrayList del Framework JCF implementa l'interfaccia List mediante un array dinamico (i cui elementi possono aumentare o diminuire a runtime), che consente l'accesso in tempo costante a qualunque elemento a partire dal suo indice. Ogni oggetto di tipo ArrayList ha:

- una capacity che rappresenta lo spazio allocato per contenere gli elementi dell'array
- una size che rappresenta il numero di elementi che esso effettivamente contiene.

Quando il numero di elementi aggiunti supera la capacità dell'array, quest'ultima viene automaticamente incrementata e lo stesso array viene ricreato.

Come tutte le altre classi del JCF, la classe ArrayList è parametrica.

Quando si crea un oggetto della classe ArrayList<E>, occorre specificare il tipo dei suoi elementi, che andrà a sostituire il parametro di tipo, E. Ad esempio:

ArrayList<String> lista= new ArrayList<>();

ArrayList<Prodotto> listaProdotti= new ArrayList<>();

L'unico vincolo posto al tipo d’elemento è che non può essere un tipo primitivo, come int (ma può essere la corrispondente classe involucro, Integer).

Le caratteristiche di un ArrayList sono:

• è un array ridimensionabile, dinamicamente. Cresce le sue dimensioni per accogliere nuovi elementi e riduce le dimensioni quando gli elementi vengono rimossi.

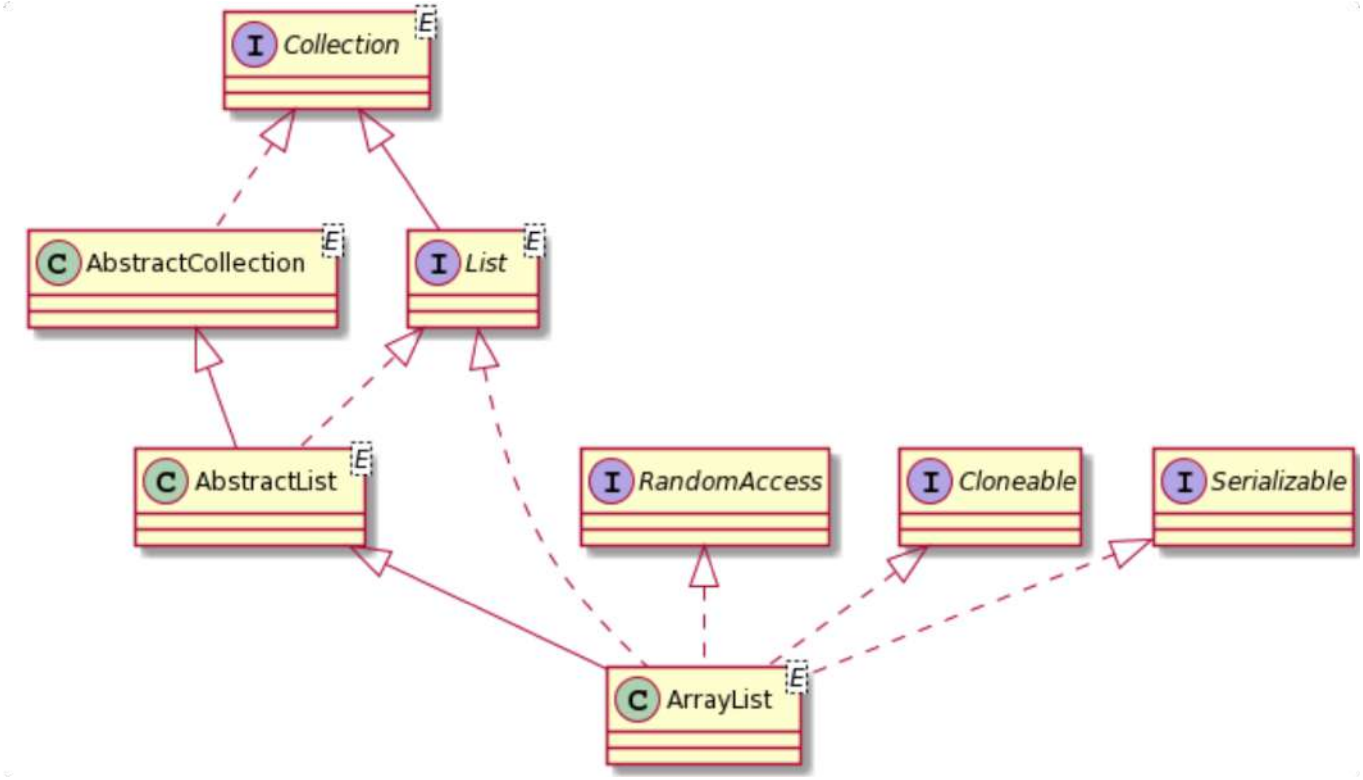
- Utilizza internamente un'array per memorizzare gli elementi, permette di recuperare gli elementi dal loro indice.
- Consente valori duplicati e nulli.
- Mantiene l'ordine di inserimento degli elementi.
- Non è possibile creare un ArrayList di tipi primitivi è necessario utilizzare tipi boxed come Integer, Character, Boolean etc.
- Non è sincronizzato. Se più thread tentano di modificare un ArrayList allo stesso tempo, il risultato finale sarà non deterministico. È necessario sincronizzare esplicitamente l'accesso a un ArrayList se più thread lo modificheranno.

Costruttori

- public ArrayList<E>() crea un'istanza della classe ArrayList vuota in cui non è specificata la capacità iniziale
- public ArrayList<E>(int initial Capacity) crea un'istanza della classe ArrayList in cui è specificata la capacità iniziale
- public ArrayList(Collection<? extends E> c) Costruisce un'arrayList contenente gli elementi della raccolta fornita, nello stesso ordine in cui sono memorizzati in essa. Questo oggetto di tipo ArrayList ha una capacità iniziale pari al 110% della dimensione della raccolta copiata..

Metodi

La classe ArrayList implementa i metodi astratti definiti nelle interfacce Collection, List e nelle interfacce Serializable e Cloneable,RandomAccess e delle classi astratte AbstractCollection e AbstractList



Object clone(). Restituisce una copia superficiale dell'istanza di ArrayList .

void ensureCapacity(int minCapacity) Aumenta la capacità di questa istanza di ArrayList , se necessario, per garantire che possa contenere almeno il numero di elementi specificato dall'argomento di capacità minima.

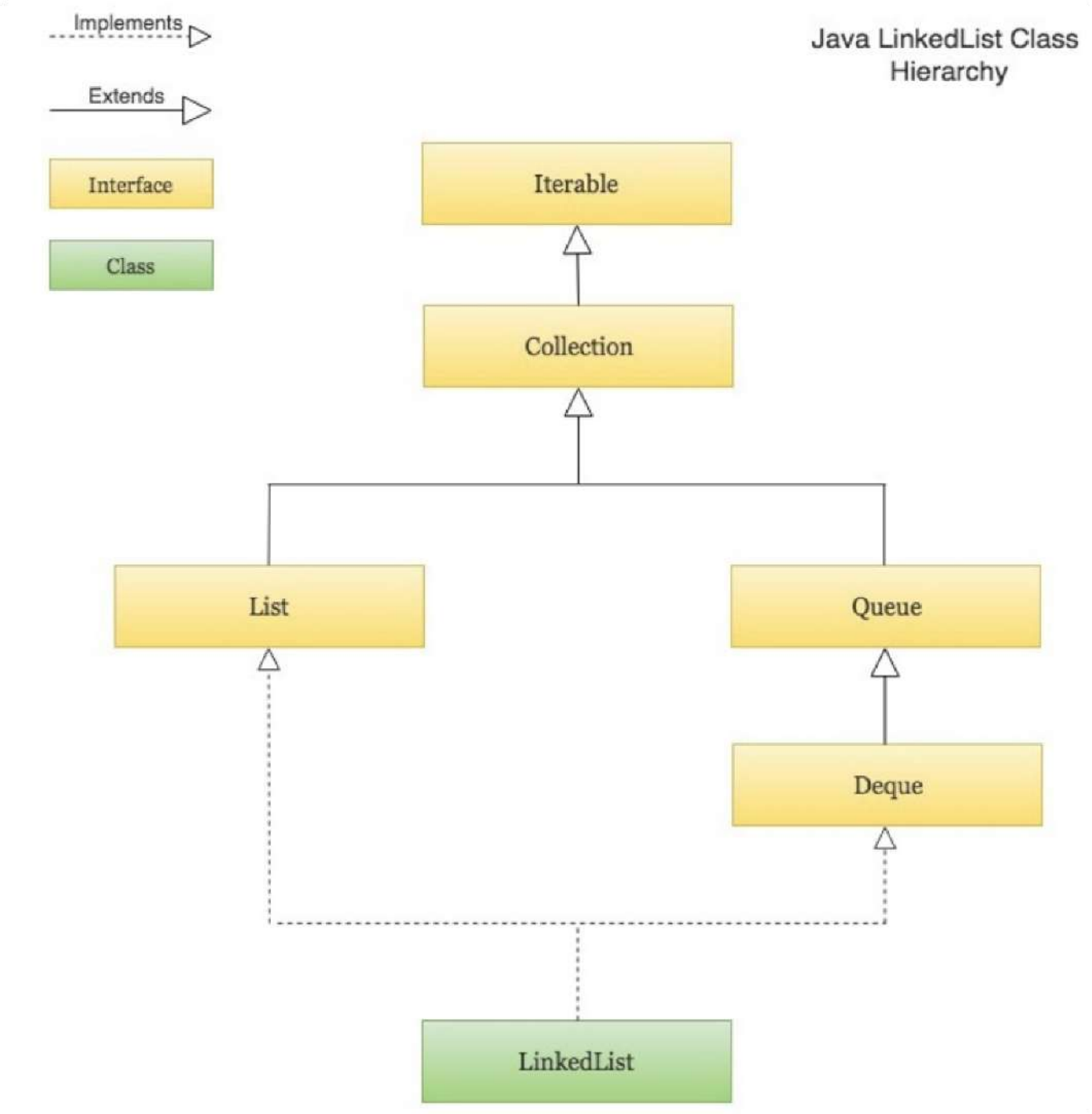
int lastIndexOf(Object o) Restituisce l'indice dell'ultima occorrenza dell'elemento specificato in questo elenco o -1 se questo elenco non contiene l'elemento.

void sort(Comparator<? super E> c) Ordina questo elenco in base all'ordine indotto dall'oggetto specificato Comparator.

void trimToSize() Ritaglia la capacità di questa istanza di ArrayList in modo che corrisponda alla dimensione corrente dell'elenco.

La classe LinkedList<E>

La classe LinkedList, è sviluppata avvalendosi della struttura di dati di tipo lista concatenata.



Di seguito sono riportati alcuni punti chiave:

- LinkedList mantiene l'ordine di inserimento degli elementi.
- LinkedList può avere valori duplicati e nulli.

• linkedList implementa Queue Deque interfacce Pertanto, può anche essere usata come Queue, Deque o Stack.

• LinkedList non è thread-safe. È necessario sincronizzare esplicitamente le modifiche simultanee a LinkedList in un ambiente a più thread

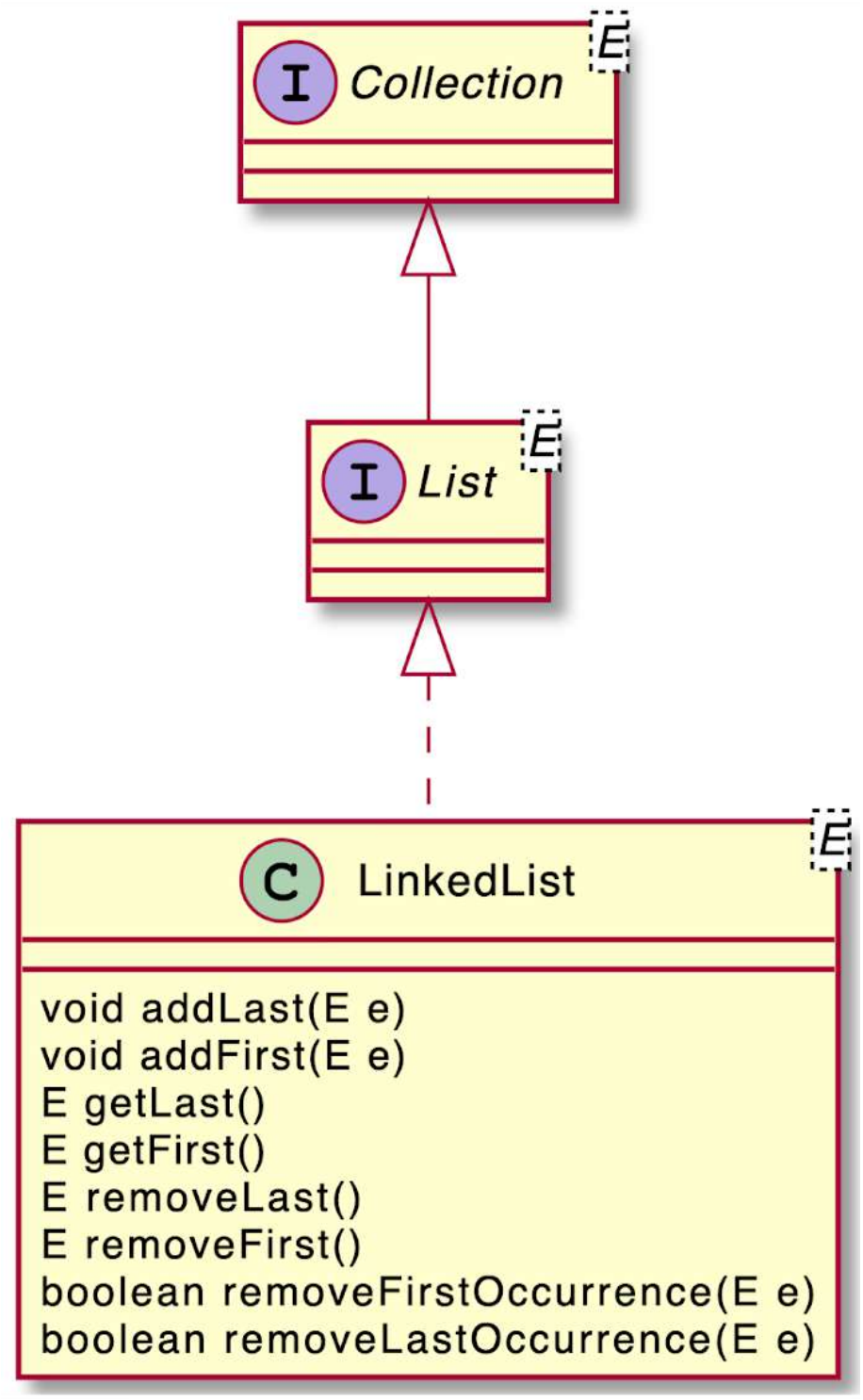
Costruttori

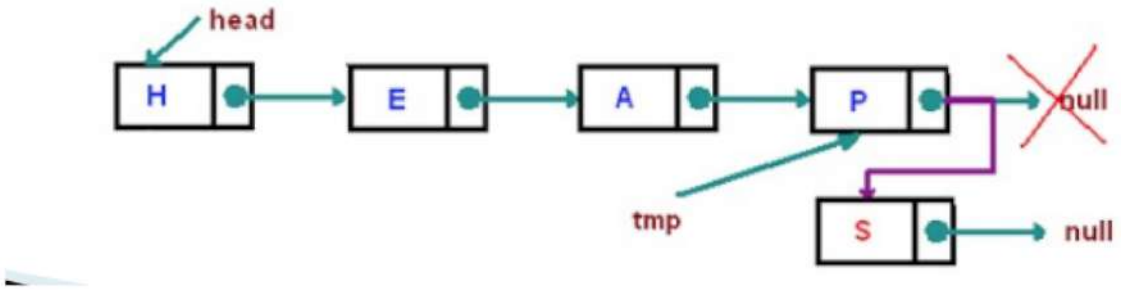
LinkedList() Costruisce una lista vuota.

LinkedList(Collection<? extends E> c) Costruisce una lista contenente gli elementi della raccolta specificata, nell'ordine in cui vengono restituiti dall'iteratore della raccolta.

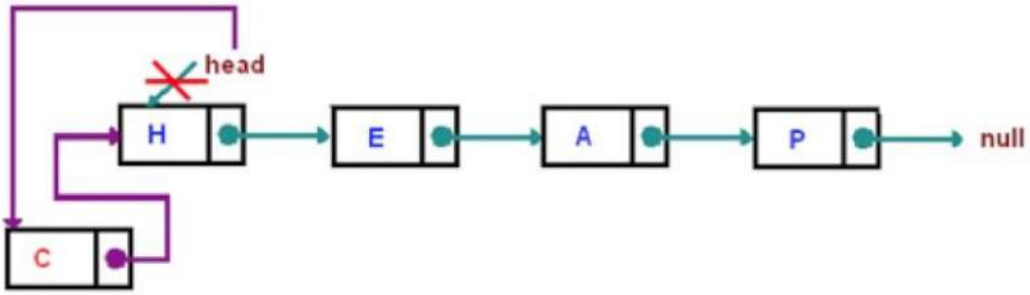
Metodi

Usandola come Lista si usano tutti i metodi visti per List più alcuni metodi che sono propri della classe

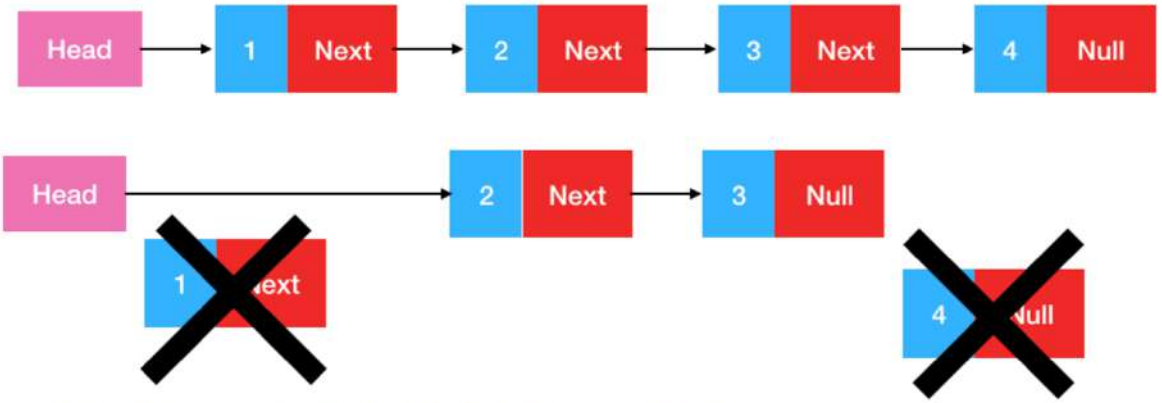

I Collection
E
I List
E
C LinkedList
E
void addLast(E e)
void addFirst(E e)
E getLast()
E getFirst()
E removeLast()
E removeFirst()
boolean removeFirstOccurrence(E e)
boolean removeLastOccurrence(E e)



addLast appende il nodo alla fine della lista : lista.addLast(s)



addFirst crea un nodo e lo aggiunge all'inizio della lista: lista.addFirst(c)

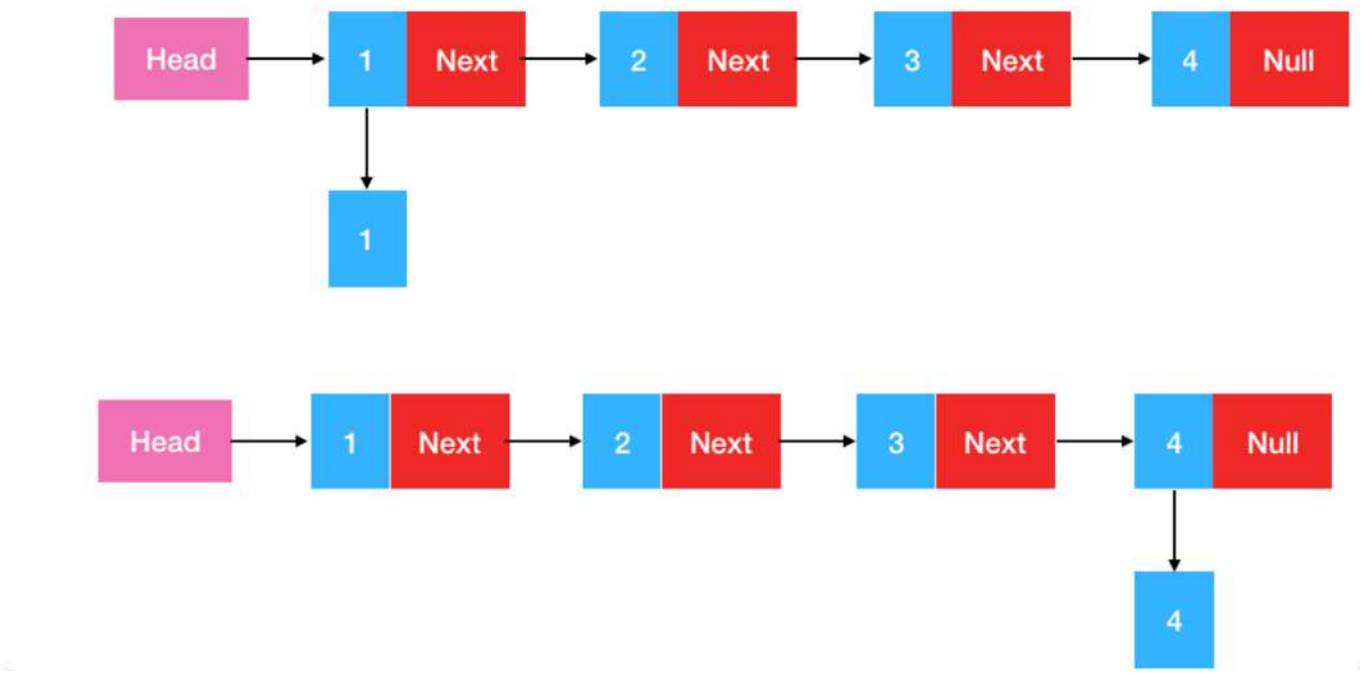


removeFirst elimina un nodo all'inizio della lista: lista.removeFirst()

removeLast elimini il nodo alla fine della lista : lista.removeLast()

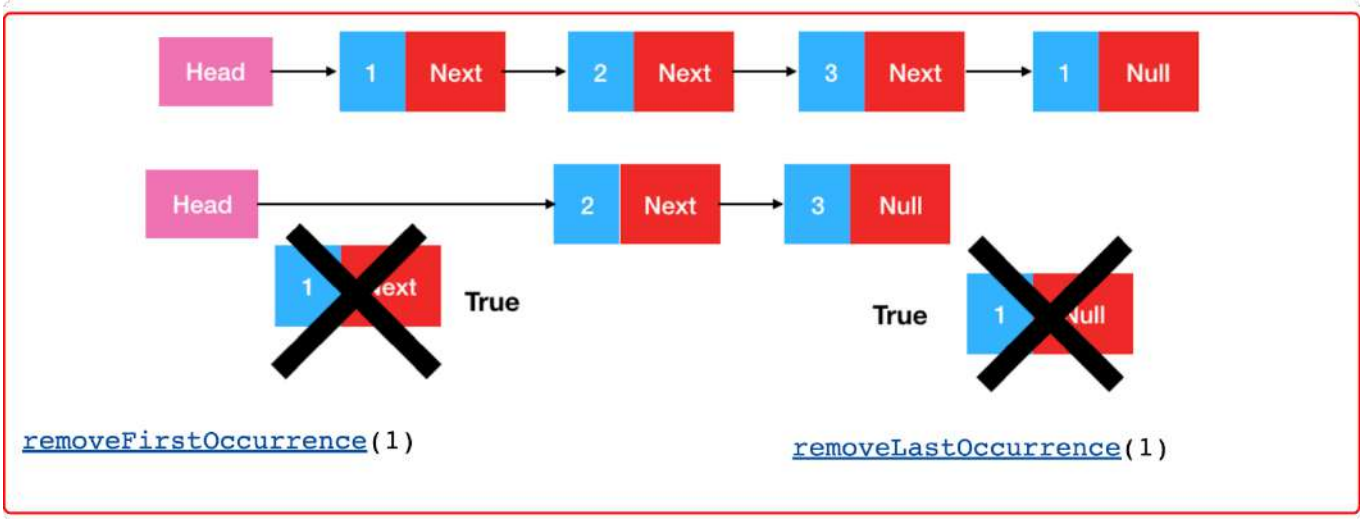
[Codice]

E	**getFirst()**: Restituisce il primo elemento in questa lista.
E	**getLast()**: Restituisce l'ultimo elemento in questa lista.



boolean **removeFirstOccurrence(E e)** Rimuove la prima occorrenza dell'elemento specificato nella lista (quando si attraversa dalla testa alla coda).

boolean **removeLastOccurrence(E e)** Rimuove l'ultima occorrenza dell'elemento specificato dalla lista (quando si attraversa dalla testa alla coda).



ARRAYLIST E LINKEDLIST A CONFRONTO

La decisione se scegliere la classe ArrayList oppure la classe LinkedList, dipende dalle performance e dalle operazioni più frequentemente effettuate dall'applicazione sviluppata. Una LinkedList è più efficiente nei casi di inserimento e cancellazione degli elementi, perché la lista viene modificata agendo solo sul riordinamento dei collegamenti tra i nodi, un ArrayList lo è nelle operazioni di accesso e ottenimento degli elementi tramite 'indicizzazione, perché si procede in modo arbitrario (direttamente alla posizione indicata) e non sequenziale.

	Vantaggi	Svantaggi
Array	• Semplicità di gestione • Accesso diretto agli elementi	• Gestione statica del numero degli elementi • Difficoltà delle operazioni di inserimento e cancellazione di elementi con criteri di ordinamento
Lista	• Gestione dinamica del numero degli elementi • Facilità nelle operazioni di inserimento e cancellazione di elementi anche in presenza di criteri di ordinamento	• Maggiore complessità di gestione rispetto all'*array* • Struttura ad accesso strettamente sequenziale

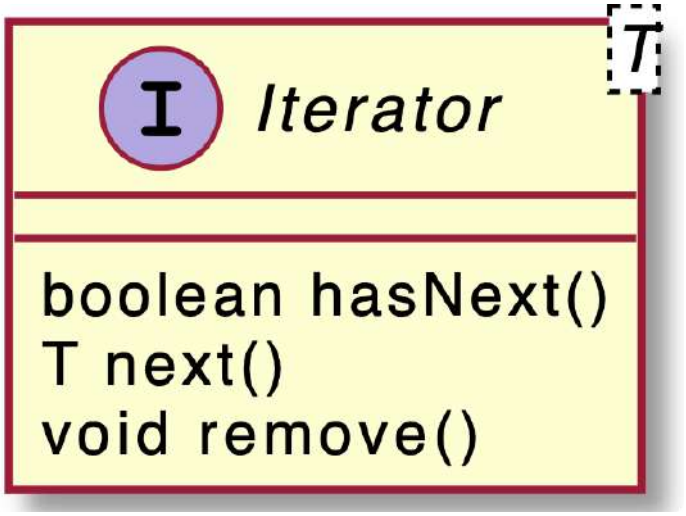
L'interfaccia Iterator<T>

Iterator è un'interfaccia utilizzata del framework Collections che permette di iterare su una collezione. Qualunque classe implementi l'interfaccia Collection<T> offre il metodo iterator( ), la chiamata al metodo iterator() sulla collezione utilizzata, ne restituisce un'implementazione concreta.

L'iteratore si può definire come un cursore che seleziona in maniera sequenziale gli elementi della collezione.

L'interfaccia Iterator<T> fa parte del package java.util.

Metodi



boolean hasNext() ritorna true se la collezione ha un successivo elemento. Genera un'eccezione NoSuchElementException se non esiste l'elemento successivo.

T next() ritorna il successivo elemento della collezione.

void remove() rimuove l'ultimo elemento ritornato dall'iteratore che deve essere usato solo durante un ciclo sugli elementi, altrimenti Java genera l'eccezione

Questo metodo può essere chiamato solo una volta per ogni chiamata a next. Se la collezione è stata modificata senza utilizzare remove, il comportamento dell'iteratore non è specificato.

Genera una IllegalStateException se il metodo next non è ancora stato chiamato o se il metodo remove è già stato chiamato dopo l'ultima chiamata di next.

Genera una UnsupportedOperationException se l'operazione di rimozione non è supportata da questo Iterator<T>.

Tutte le eccezioni citate sono del tipo non controllato, quindi non è necessario gestirle in un blocco catch o dichiararle in una clausola throws.

L'eccezione NoSuchElementException appartiene al package java.util, che deve essere quindi importato se il codice utilizza questa classe. Tutte le altre eccezioni appartengono al package java. lang e quindi non richiedono l'importazione di package aggiuntivi.

I passi per iterare una lista sono:

1. creiamo un ciclo while: la condizione da verificare è hasNext() che ritorna true finché non arriviamo all'ultimo elemento della lista
2. il metodo next/) ritorna l'elemento successivo

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    numbers.add(13);
    numbers.add(18);
    numbers.add(25);
    numbers.add(40);
    System.out.println("------ la lista contiene--------");
    System.out.println(numbers);
    Iterator<Integer> numbersIterator = numbers.iterator();
    while (numbersIterator.hasNext()) {
      Integer num = numbersIterator.next();
      if (num % 2 != 0) {
        numbersIterator.remove();
      }
    }
    System.out.println("------ la lista contiene--------");
    System.out.println(numbers);
  }
}
```

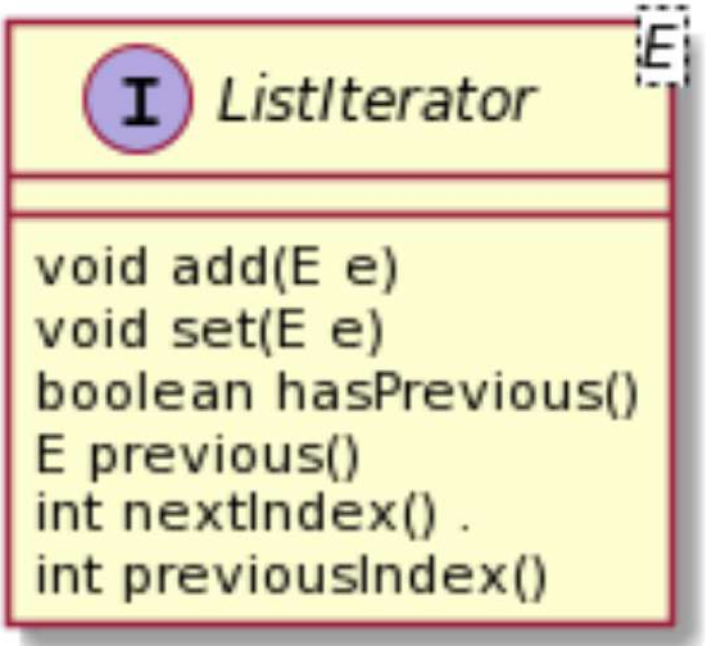
Output:

```
------ la lista contiene--------
[13, 18, 25, 40]
------ la lista contiene--------
[18, 40]
```

l'interfaccia ListIterator<E>

L'interfaccia ListIterator<E> estende Iterator<T>. Un ListIterator<T> ha tutti i metodi di un Iterator<E>, più altri che abilitano nuove funzionalità: un ListIterator<T> può muoversi lungo la lista degli elementi della collezione in entrambe le direzioni e offre metodi, come set e add, che possono essere utilizzati per modificare gli elementi della collezione. Un oggetto di tipo ListIterator è disponibile solo per gli oggetti di tipo List.

Metodi

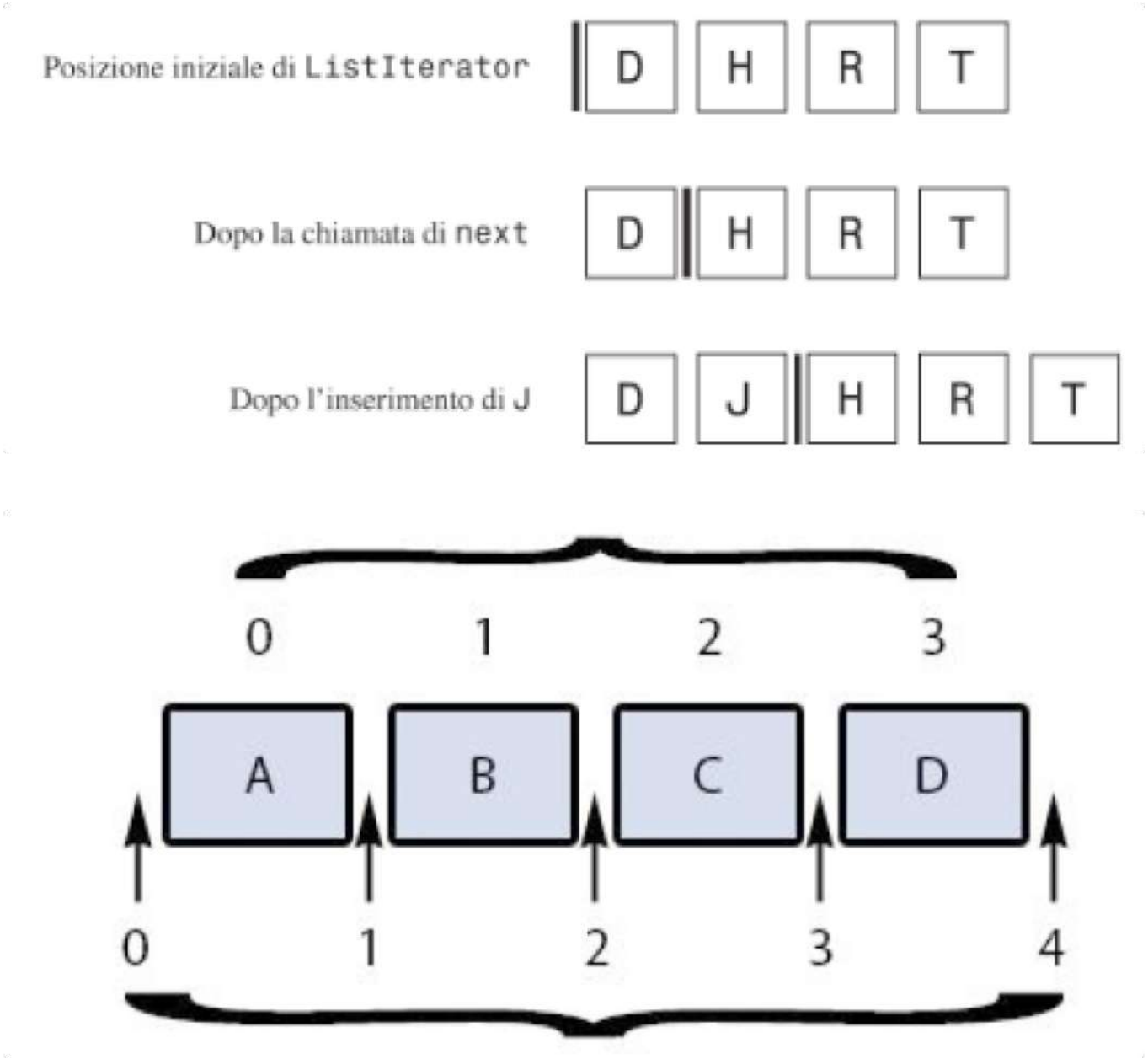


• void add(E e) aggiunge l'elemento indicato dal parametro nella lista corrente. Se nella lista sono presenti altri elementi, l'elemento è inserito prima del successivo elemento che sarebbe ritornato dal metodo next e dopo il precedente elemento che sarebbe ritornato dal metodo previous.

• void set(E e) modifica l'ultimo elemento ritornato dai metodi next o previous con l'elemento specificato dal parametro e.

- boolean hasPrevious() ritorna true se la collezione ha un precedente elemento. E previous() ritorna il precedente elemento della collezione.
- int nextIndex() ritorna l'indice del prossimo elemento che sarebbe ritornato dal metodo next.
- int previousIndex() ritorna l'indice del precedente elemento che sarebbe ritornato dal metodo previous.

CURSORI: Un oggetto di tipo ListIterator ha un cursore che rappresenta una sorta di indicatore per la posizione dove l'iteratore si troverà in un determinato momento durante l'iterazione. Questa posizione non sarà mai sull'elemento da processare, ma sarà sempre nel mezzo di due elementi.



Per spiegare la Figura vediamo che cosa accade se invochiamo i seguenti metodi a partire dal cursore posizionato all'indice 2:

- **hasNext** ritorna true perché c'è l'elemento C:

- **hasPrevious** ritorna true perché c'è l'elemento B:

- **next** ritorna l'elemento C:

- **previous** ritorna l'elemento B:

- **nextIndex** ritorna l'indice 2:

- **previousIndex** ritorna l'indice 1

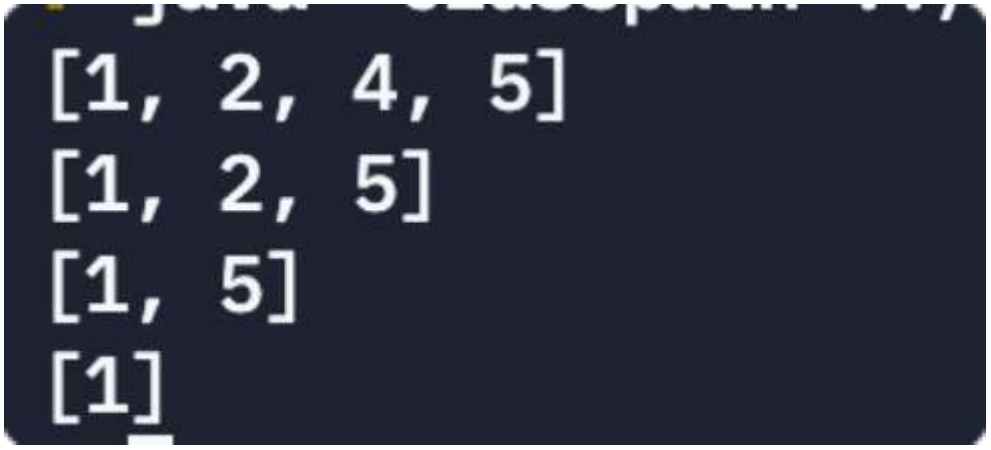
- **add** inserisce un elemento tra l'elemento C e l'elemento B e prima del cursore:
- **set** modifica l'elemento C se **prima è invocato il metodo next**. Altrimenti modifica l'elemento B **se prima è invocato il metodo previous**;
- remove elimina l'elemento C se prima è invocato il metodo next, altrimenti elimina l'elemento B se prima è invocato il metodo previous.

ATTENZIONE I metodi set e remove agiscono sempre sull'elemento corrente ritornato dal metodo next o previous, mentre il metodo add agisce sulla posizione corrente del cursore.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    for(int i=1;i<6;i++)
    numbers.add(i);
    ListIterator<Integer> iteratore = numbers.listIterator();
    int i=2;
    while(numbers.size()!=1){
     i--;
    while (iteratore.hasNext()) {
      iteratore.next();

      if (i % 3== 0 && numbers.size()!=1){
         iteratore.remove();
         System.out.println(numbers);
      }
      i++;
    }

     i--;
```

```
        while (iteratore.hasPrevious()) {
      iteratore.previous();
      if (i % 3== 0&&numbers.size()!=1){
      iteratore.remove();
      System.out.println(numbers);
      }
      i++;
       }
    }
    }
}
```

Output:



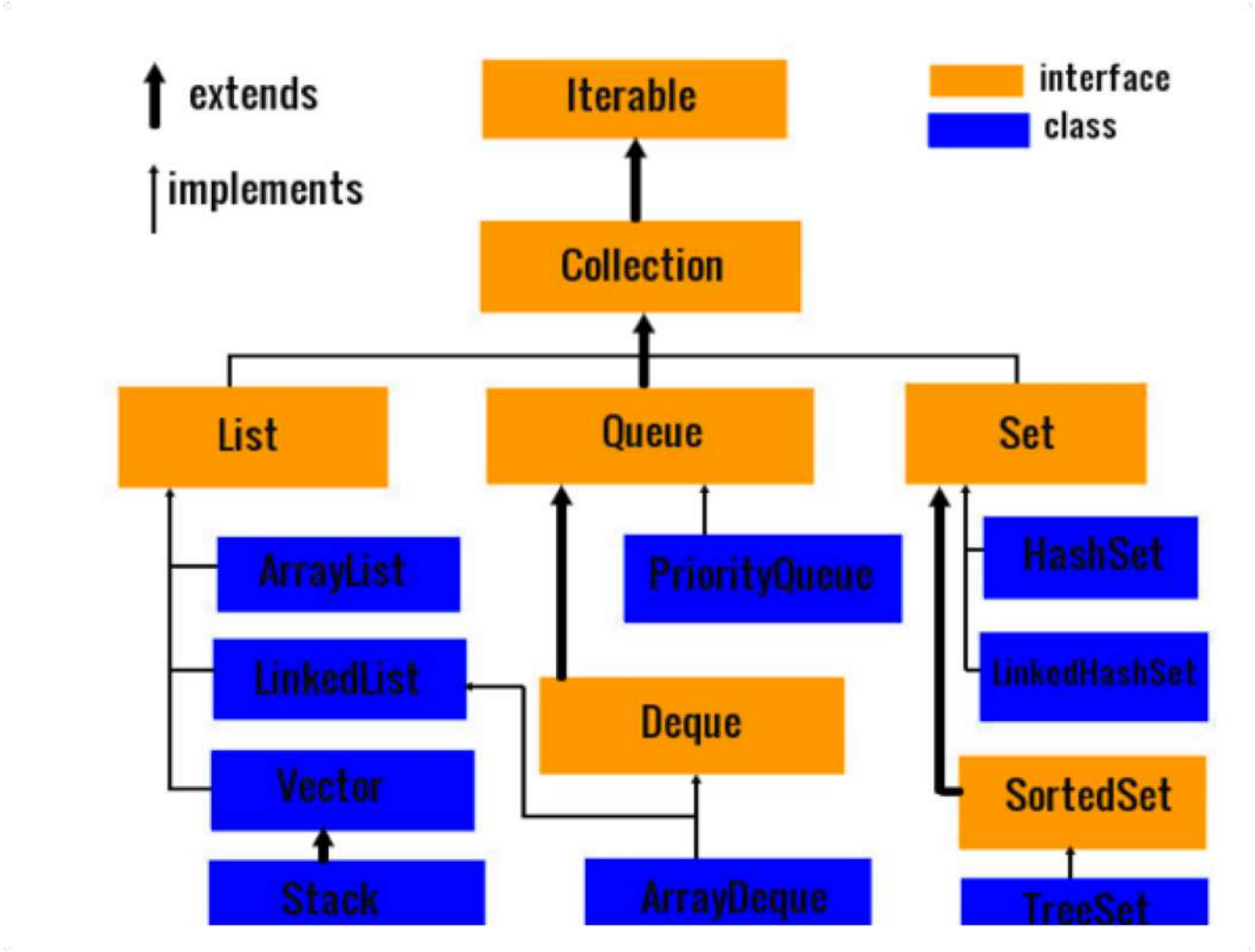
Le Code

Una coda o queue è una sequenza finita di elementi in cui:

1. gli inserimenti possono avvenire soltanto alla fine della sequenza (all'estremità detta back);

2. le rimozioni possono avvenire soltanto all'inizio della sequenza (all'estremità detta front).

Una coda impone un ordine cronologico ai propri elementi: tra gli elementi presenti in coda, il primo elemento che è stato accodato all'estremità back sarà il primo elemento ad essere rimosso, dall'estremità front; il secondo elemento che è stato accodato sarà il secondo ad essere rimosso; e così via. Questa proprietà, che definisce le code, si chiama **"First In, First Out" (FIFO)**, cioè: chi è entrato per primo, uscirà per primo.



L'interfaccia Queue<E>

Queue specializza Collection introducendo l'idea di coda di elementi da sottoporre a elaborazione

- ha una nozione di posizione (testa della coda)
- l'interfaccia di accesso si specializza:

o remove() estrae l'elemento "in testa" alla coda, rimuovendolo

o element () lo estrae senza rimuoverlo

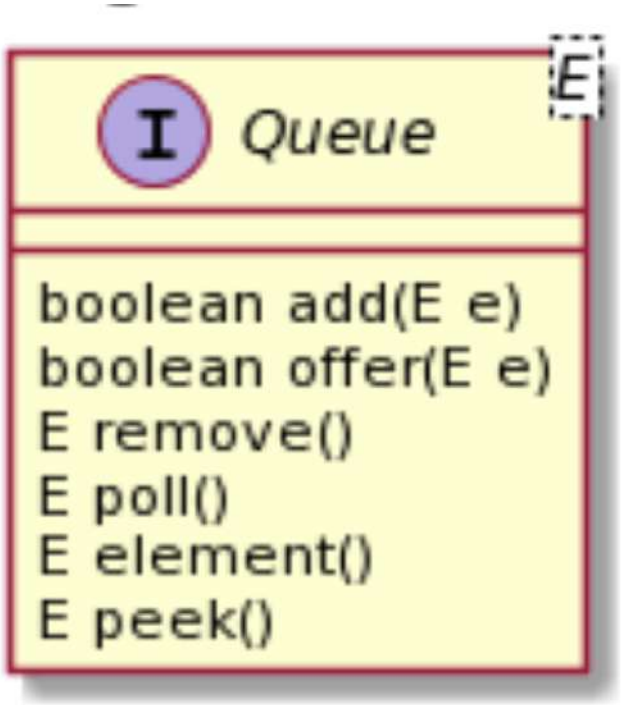
Ognuno di questi metodi è presente in due formati differenti:

1. se l'operazione fallisce un formato lancia un'eccezione

2. l'altro restituisce un valore speciale (per esempio null o false).

In particolare quando parliamo di valore speciale ci riferiamo alla situazione in cui il metodo restituisce o l'oggetto stesso appena aggiunto o recuperato, oppure un booleano (come nel caso del metodo offer()). Quindi, a seconda dell'esigenza, lo sviluppatore può usufruire di un metodo piuttosto che di un altro.

Metodi



Il metodo add(E e) se fallisce nell'aggiungere un elemento lancia una unchecked exception.

Il metodo offer(E e) inserisce un elemento ritornando true o false qualora l'operazione di inserimento riesca oppure no.

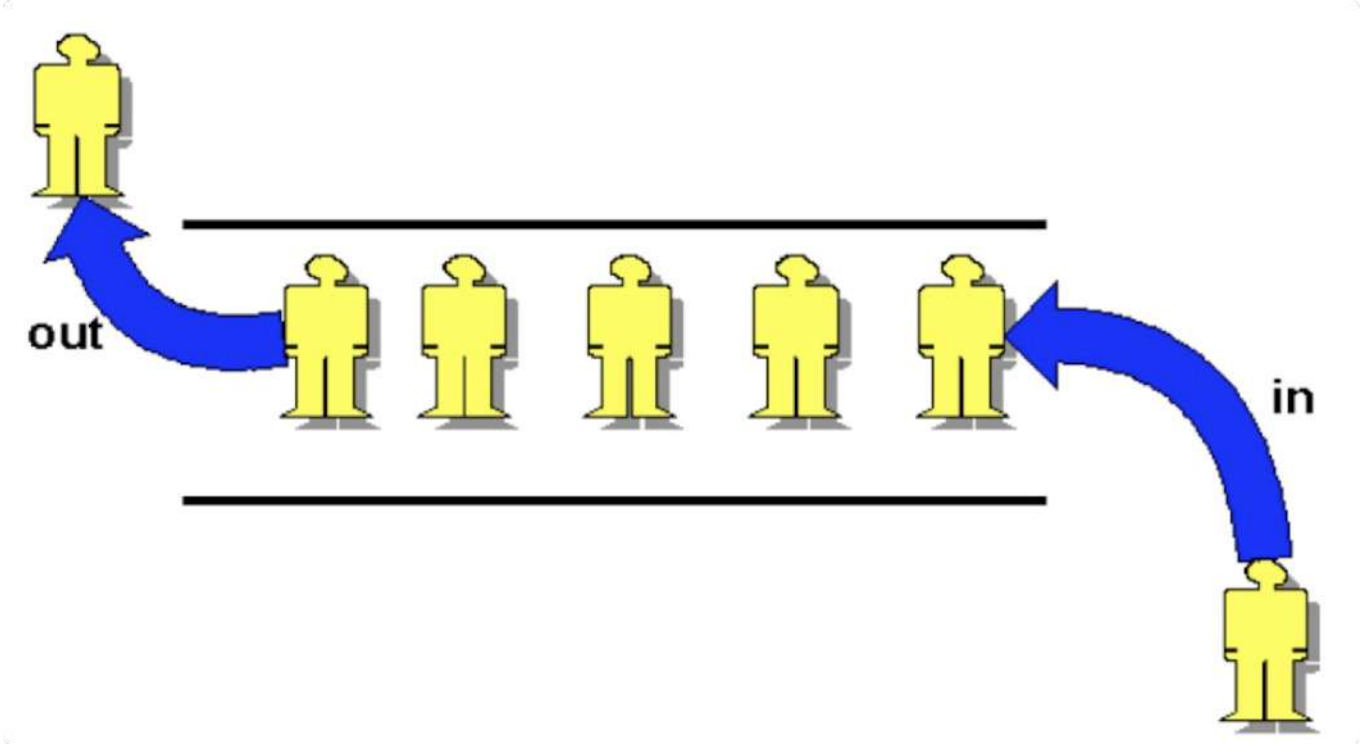
I metodi remove() e poll() ritornano e rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo poll() ritorna null, mentre remove() lancia un'eccezione. Il metodo poll() restituisce un riferimento all'oggetto rimosso in caso di successo.

I metodi element() e peek() invece ritornano ma non rimuovono l'elemento che si trova in testa alla coda.

Nel caso in cui non ci sia niente alla testa della coda, il metodo peek() ritorna null. Mentre element() lancia un'eccezione. Il metodo peek() ritorna un riferimento all'oggetto rimosso in caso di successo.

Tipo di operazione	Metodo che lancia un'eccezione	Metodo che ritorna un valore speciale
Inserimento	`add(e)`	`offer(e)`
Rimozione	`remove()`	`poll()`
Recupero	`element()`	`peek()`

In genere, le cose vengono usate nella modalità FIFO (first-in-first-out).

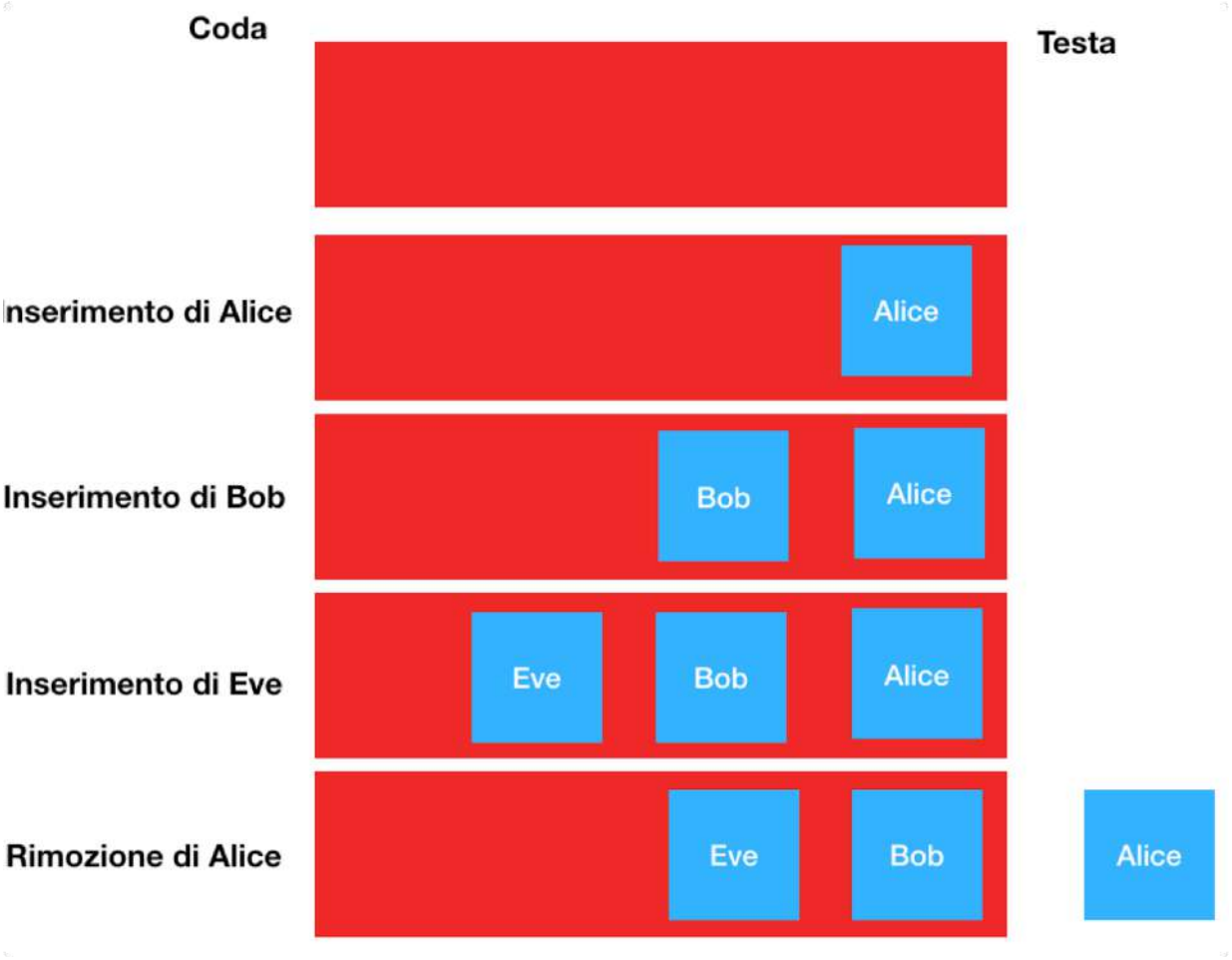


Altri tipi di code possono utilizzare regole di posizionamento differenti, ad esempio le PriorityQueue che in cima alla coda avranno quegli elementi con una maggior priorità di essere estratti.

L'interfaccia Queue viene implementata tramite le classi:

LinkedList: sviluppata avvalendosi della struttura di dati di tipo lista doppiamente collegata, già vista come implementazione del tipo List.

ArrayDeque: sviluppata avvalendosi della struttura di dati di tipo array dinamico che cresce o decresce a seconda delle necessità. Inoltre, non ha restrizioni di capacità e gli elementi null non sono ammessi




```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;

public class Main {
  public static void main(String[] args){
   Queue<String> coda=new LinkedList<>();
  print(coda);
   //inserimento valori nella coda
   coda.add("Alice");
   print(coda);
   coda.add("Bob");
   print(coda);
   coda.add("Trudi");
   print(coda);
   coda.offer("Eve");
   print(coda);
   coda.offer("mallory");
  print(coda);
  //estrazione della testa
   System.out.println("Dalla coda esce "+coda.poll());
    System.out.println("in coda ci sono "+coda.size()+" persone esse sono:");
    Iterator<String> itera=coda.iterator();
    //(scorrere la coda senza estrarre
    while(itera.hasNext())
```

```
        System.out.println(itera.next());
    while(!coda.isEmpty())
        System.out.println("Dalla coda esce "+coda.poll());
    System.out.println("in coda ci sono "+coda.size()+" persone");
  }
  public static void print(Queue coda){
    System.out.println("in coda ci sono "+coda.size()+" persone "+coda);
  }
}
```

Output:

```
in coda ci sono 0 persone []
in coda ci sono 1 persone [Alice]
in coda ci sono 2 persone [Alice, Bob]
in coda ci sono 3 persone [Alice, Bob, Trudi]
in coda ci sono 4 persone [Alice, Bob, Trudi, Eve]
in coda ci sono 5 persone
Dalla coda esce Alice
in coda ci sono 4 persone esse sono:
Bob
Trudi
Eve
mallory
Dalla coda esce Bob
Dalla coda esce Trudi
Dalla coda esce Eve
Dalla coda esce mallory
in coda ci sono 0 persone
```

Codice

La Classe PriorityQueue<E>

Un'implementazione di Queue definita dalla classe PriorityQueue, ordina i propri elementi a seconda del proprio ordinamento naturale (definito mediante l'implementazione dell'interfaccia Comparable) o a seconda di un oggetto Comparator associato al momento della creazione. Attenzione che "usando un Iterator per iterare su di essa, non è garantito che i suoi elementi vengono iterati nell'ordine che ci si aspetta. Infatti per ragioni prestazionali, i suoi elementi sono gestiti in background senza rispettare l'ordinamento, utilizzando una lista. Il consiglio per iterare gli elementi ordinati è quello di usare un'istruzione come la seguente:

Arrays.sort(coda.toArray0));

Dove coda è un reference a un oggetto PriorityQueue. Infatti abbiamo prima trasformato in array la priority queue, e poi ordinato i suoi elementi mediante il metodo statico sort() della classe di utilità Arrays.

Costruttori:

1. PriorityQueue(): Crea un PriorityQueue con la capacità iniziale predefinita che ordina i suoi elementi secondo il loro ordinamento naturale.

```
PriorityQueue<E> coda = new PriorityQueue<>();
```

2. PriorityQueue(Collection<E> c): crea un PriorityQueue contenente gli elementi nella raccolta specificata.

```
PriorityQueue<E> coda= new PriorityQueue<>(Raccolta<E> c);
```

1. PriorityQueue(Comparator<E> comparator): crea un PriorityQueue con la capacità iniziale predefinita che ordina i suoi elementi in base al comparatore specificato.

```
PriorityQueue<E> coda = new PriorityQueue<>(Comparator<E> comparatore);
```

1. PriorityQueue(int initialCapacity) : crea un PriorityQueue con la capacità iniziale specificata che ordina i suoi elementi in base al loro ordinamento naturale.

```
PriorityQueue<E> coda = new PriorityQueue<>(int initialCapacity);
```

1. PriorityQueue(int initialCapacity, Comparator<E> comparator): crea un PriorityQueue con la capacità iniziale specificata che ordina i suoi elementi in base al comparatore specificato.

```
PriorityQueue<E>  coda = new PriorityQueue(int capacity, Comparator<E> c);
```

1. PriorityQueue(PriorityQueue<E> c) : crea un PriorityQueue contenente gli elementi nella coda di priorità specificata.

PriorityQueue<E> coda = new PriorityQueue(PriorityQueue<E> c);

1. PriorityQueue(SortedSet<E> c) : crea un PriorityQueue contenente gli elementi nel set ordinato specificato.

PriorityQueue<E> coda = new PriorityQueue<>(SortedSet<E> c);

```
import java.util.Iterator;
import java.util.PriorityQueue;
import java.util.Queue;
public class Main {
   public static void main(String[] args){
  Queue<String> coda= new PriorityQueue<>();
    System.out.println("in coda ci sono "+coda.size()+" persone" + coda);
   //inserimento valori nella coda
      coda.add("bob");
      System.out.println("in coda ci sono "+coda.size()+" persone " + coda);
     coda.add("eve");
     System.out.println("in coda ci sono "+coda.size()+" persone " + coda);
     coda.offer("trudy");
     System.out.println("in coda ci sono "+coda.size()+" persone " + coda);
     coda.offer("mallory");
     System.out.println("in coda ci sono "+coda.size()+" persone " + coda);
   coda.add("alice");
     System.out.print("in coda ci sono "+coda.size()+" persone ");
     System.out.println(coda);
     System.out.println("L'estrazione avviene in ordine alfabetico, in base al
compareTo()");
     while(!coda.isEmpty())
        System.out.println("Dalla coda esce "+coda.poll());
     System.out.println("in coda ci sono "+coda.size()+" persone");
   }
}
```

Output:

```
in coda ci sono 0 persone[]
in coda ci sono 1 persone [bob]
in coda ci sono 2 persone [bob, eve]
in coda ci sono 3 persone [bob, eve, trudy]
in coda ci sono 4 persone [bob, eve, trudy, mallory]
in coda ci sono 5 persone [alice, bob, trudy, mallory, eve]
L'estrazione avviene in ordine alfabetico, in base al compareTo()
Dalla coda esce alice
Dalla coda esce bob
Dalla coda esce eve
Dalla coda esce mallory
Dalla coda esce trudy
in coda ci sono 0 persone
```

Costruiamo una coda di persone munite di un codice che ne determina la priorità.

```
public class Persona implements Comparable<Persona>{
  private String nome;
  private Codice codice;
```

```
public Persona(String nome, Codice codice){
    this.nome=nome;
    this.codice=codice;

  }

  public String toString(){
    return nome +" Codice "+codice ;
  }
    @Override
    public int compareTo(Persona o) {
        return codice.compareTo(o.codice);
    }
}
```

Il codice è inserito in una Enum

```
public enum Codice {
```

```
    Rosso,Giallo,Bianco
}
```

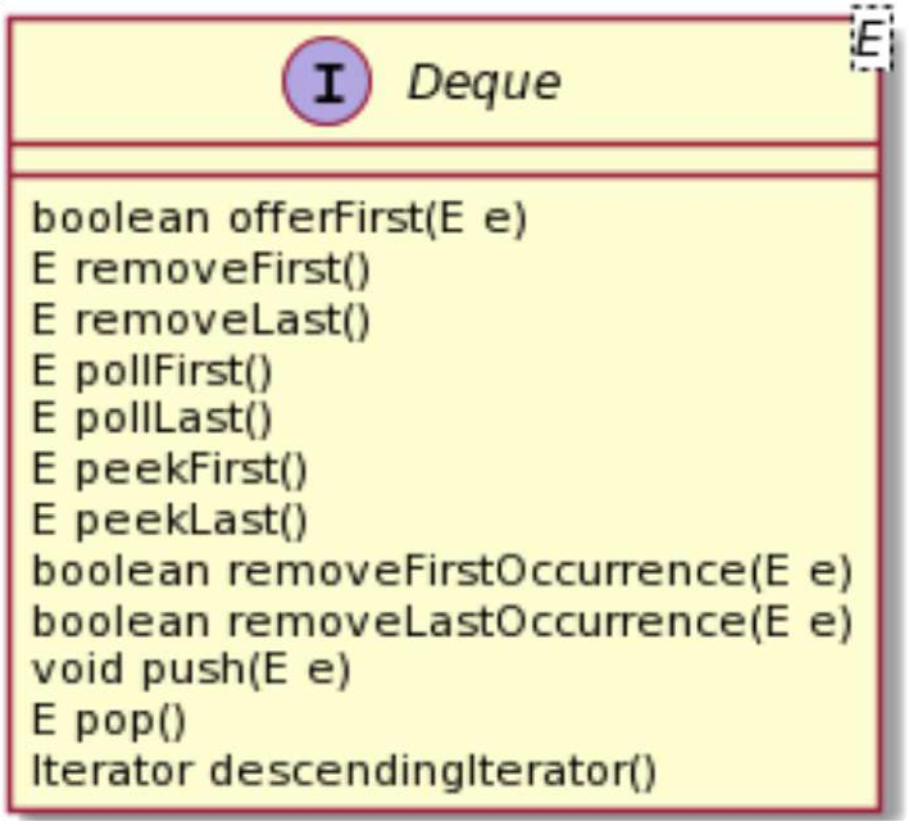
```
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
  public static void main(String[] args) {
    Queue<Persona> coda = new PriorityQueue<>();
    Codice c;
    System.out.println("in coda ci sono " + coda.size() + "
persone" + coda);
    // inserimento valori nella coda
    c = Codice.Bianco;
    coda.add(new Persona("bob", c.Bianco));
    System.out.println("in coda ci sono " + coda.size() + "
persone " + coda);
    coda.add(new Persona("eve", c.Giallo));
    System.out.println("in coda ci sono " + coda.size() + "
persone " + coda);
    coda.offer(new Persona("trudi", c.Rosso));
    System.out.println("in coda ci sono " + coda.size() + "
persone " + coda);
    coda.offer(new Persona("mallory", c.Rosso));
    System.out.println("in coda ci sono " + coda.size() + "
persone " + coda);
    coda.add(new Persona("alice", c.Giallo));
    System.out.print("in coda ci sono " + coda.size() + " persone
");
    System.out.println(coda);
    System.out.println("L'estrazione avviene in ordine alfabetico,
in base al compareTo()");
    while (!coda.isEmpty())
      System.out.println("Dalla coda esce " + coda.poll());
    System.out.println("in coda ci sono " + coda.size() + "
persone");
  }
}
```

output:

```
in coda ci sono 0 persone[]
in coda ci sono 1 persone [bob Codice Bianco]
in coda ci sono 2 persone [eve Codice Giallo, bob Codice Bianco]
in coda ci sono 3 persone [trudy Codice Rosso, bob Codice Bianco, eve Codice Giallo]
in coda ci sono 4 persone [trudy Codice Rosso, mallory Codice Rosso, eve Codice Giallo, bob Codice Bianco]
in coda ci sono 5 persone [trudy Codice Rosso, mallory Codice Rosso, eve Codice Giallo, bob Codice Bianco, alice Codice Giallo]
L'estrazione avviene in ordine alfabetico, in base al compareTo()
Dalla coda esce trudy Codice Rosso
Dalla coda esce mallory Codice Rosso
Dalla coda esce alice Codice Giallo
Dalla coda esce eve Codice Giallo
Dalla coda esce bob Codice Bianco
in coda ci sono 0 persone
```

Codice

L'interfaccia Deque <E>

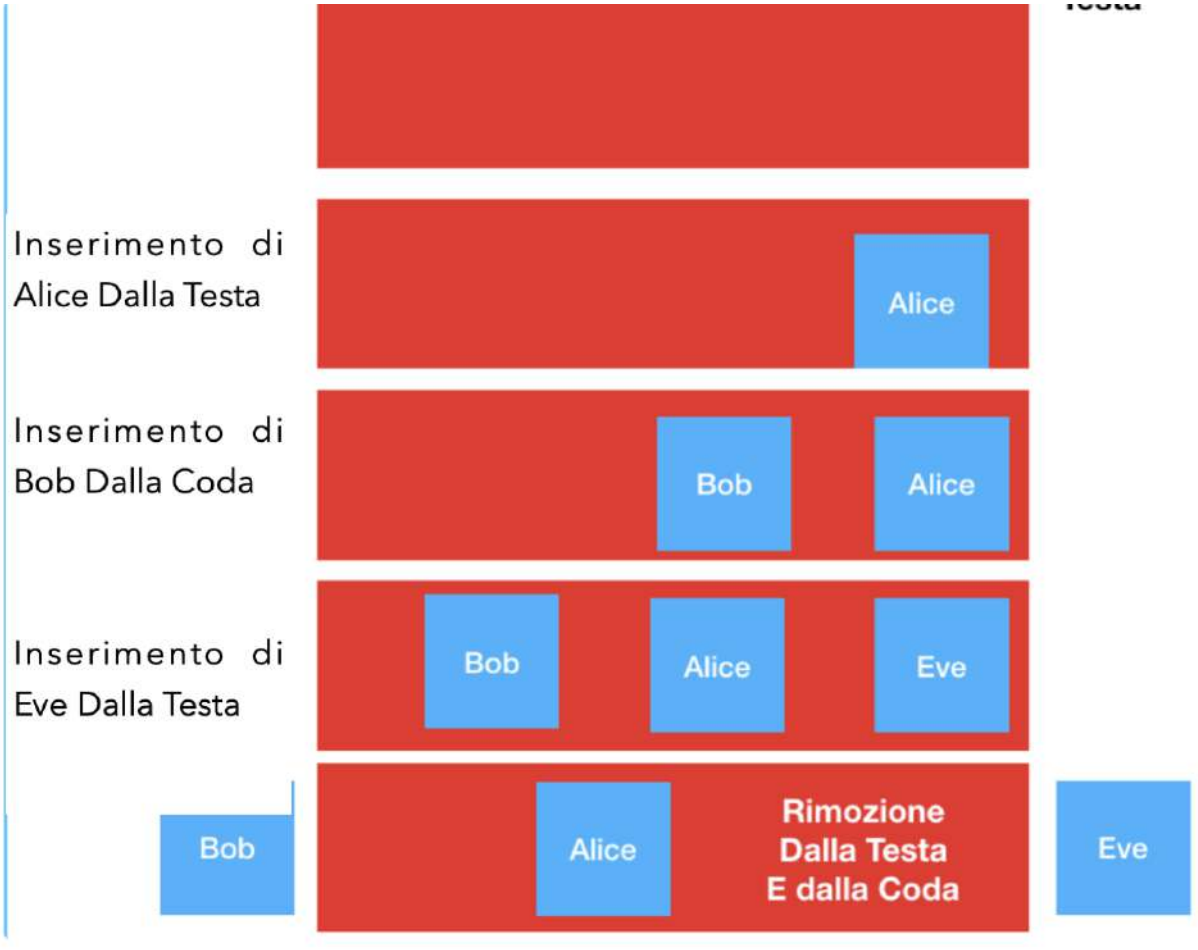


Interfacce Deque(che si pronuncia "deck") è una collezione lineare che supporta l'inserimento e la rimozione dei suoi elementi da entrambe le estremità. Può essere quindi utilizzata come una coda FIFO o LIFO a seconda della necessità. Come per Queue, le funzionalità di questa interfaccia sono duplicate per lanciare eccezioni o ritornare valori speciali. La seguente tabella riassume i principali metodi definiti in Deque.

OPERATION	END	EXCEPTION METHOD	SPECIAL-RETURN METHOD
enqueue	Front	`addFirst`	`offerFirst`
	Back	`addLast`	`offerLast`
dequeue	Front	`removeFirst`	`pollFirst`
	Back	`removeLast`	`pollLast`
peek	Front	`getFirst`	`peekFirst`
	Back	`getLast`	`peekLast`

Poiché l'interfaccia Deque estende l'interfaccia Queue i metodi ereditati da questa si comportano come operazioni FIFO. Quindi chiamando:

- add o offer si inserisce un elemento in coda.
- remove o poll si rimuove un elemento dalla testa della coda.
- element o peek si visualizza l'elemento in testa alla coda.



Deque può essere utilizzata come stack. Pertanto, include i metodi di stack, vale a dire push e pop.

Deque può essere implementata tramite le Classi concrete:

LinkedList: sviluppata avvalendosi della struttura di dati di tipo lista doppiamente collegata, già vista come implementazione del tipo Lista.

ArrayDeque: sviluppata avvalendosi della struttura di dati di tipo array dinamico che cresce o decresce a seconda delle necessità. Inoltre, non ha restrizioni di capacità e gli elementi null non sono ammessi.

La classe ArrayDeque<E>

Un ArrayDeque è un tipo speciale di array espandibile che ci consente di aggiungere o rimuovere un elemento da entrambi i lati. Un'implementazione di ArrayDeque può essere utilizzata come Stack (Last-In-First-Out) o Queue (First-In-First-Out).

Costruttori

ArrayDeque(): Costruisce un array vuoto con una capacità iniziale sufficiente per contenere 16 elementi.

ArrayDeque(**Collection**<? extends **E**> c): Costruisce un Array contenente gli elementi della raccolta specificata, nell'ordine in cui vengono restituiti dall'iteratore della raccolta.

ArrayDeque(int numElements): Costruisce un array vuoto con una capacità iniziale sufficiente per contenere il numero specificato di elementi

```
import java.util.Deque;
import java.util.Iterator;
import java.util.ArrayDeque;
public class Main {
   public static void main(String[] args){
   Deque<String> coda=new ArrayDeque<>();
   //inserimento valori nella coda
   coda.add("alice");
   coda.add("bob");
   coda.add("trudi");
   System.out.println("La prima della coda è: "+coda.peek()+"
```

```
"+coda);
   //si inserisce alla testa della coda
   System.out.println("carol salta la coda e diventa la prima: ");
   coda.addFirst("carol");
   System.out.println("La prima della coda è: "+coda.peek()+"
"+coda);
   coda.offer("eve");
   coda.offer("mallory");
   System.out.println("in coda ci sono "+coda.size()+" persone
"+coda);
  //estrazione della testa
   System.out.println("Dalla coda esce "+coda.poll()+ "  "+coda);

    System.out.println(coda.pollLast()+" lascia la coda  "+coda);
    System.out.println("in coda ci sono "+coda.size()+" persone
esse sono: visualizzandole in ordine inverso");
    Iterator<String> itera=coda.descendingIterator();
    //(scorrere la coda senza estrarre

    while(itera.hasNext())
         System.out.println(itera.next());
    while(!coda.isEmpty())
        System.out.println("Dalla coda esce "+coda.poll());
     System.out.println("in coda ci sono "+coda.size()+"
persone");
   }
}
```

Output:

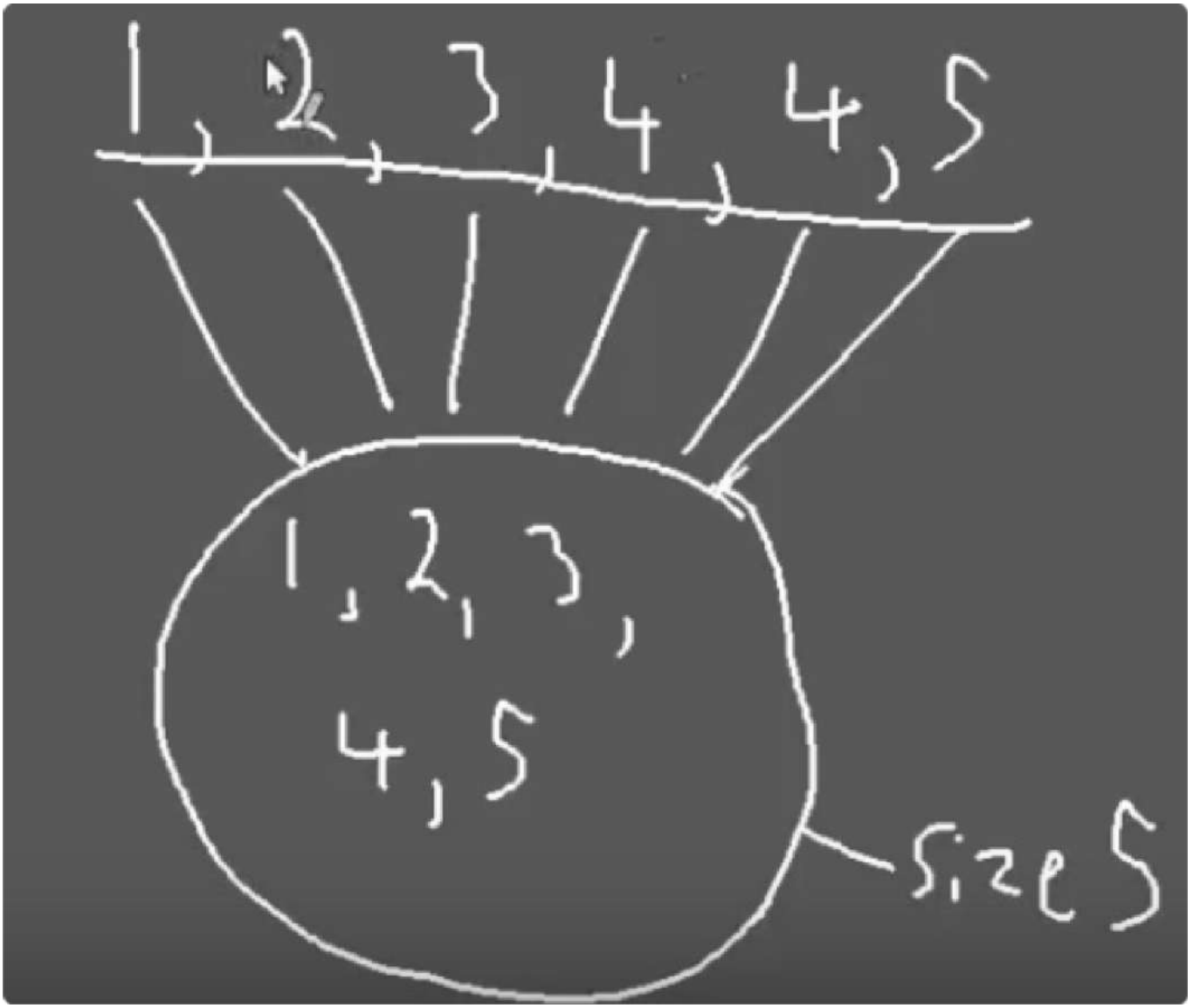
```
La prima della coda è: alice [alice, bob, trudi]
carol salta la coda e diventa la prima:
La prima della coda è: carol [carol, alice, bob, trudi]
in coda ci sono 6 persone [carol, alice, bob, trudi, eve, mallory]
Dalla coda esce carol  [alice, bob, trudi, eve, mallory]
mallory lascia la coda  [alice, bob, trudi, eve]
in coda ci sono 4 persone esse sono: visualizzandole in ordine inverso
eve
trudi
bob
alice
Dalla coda esce alice
Dalla coda esce bob
Dalla coda esce trudi
Dalla coda esce eve
in coda ci sono 0 persone
```

codice

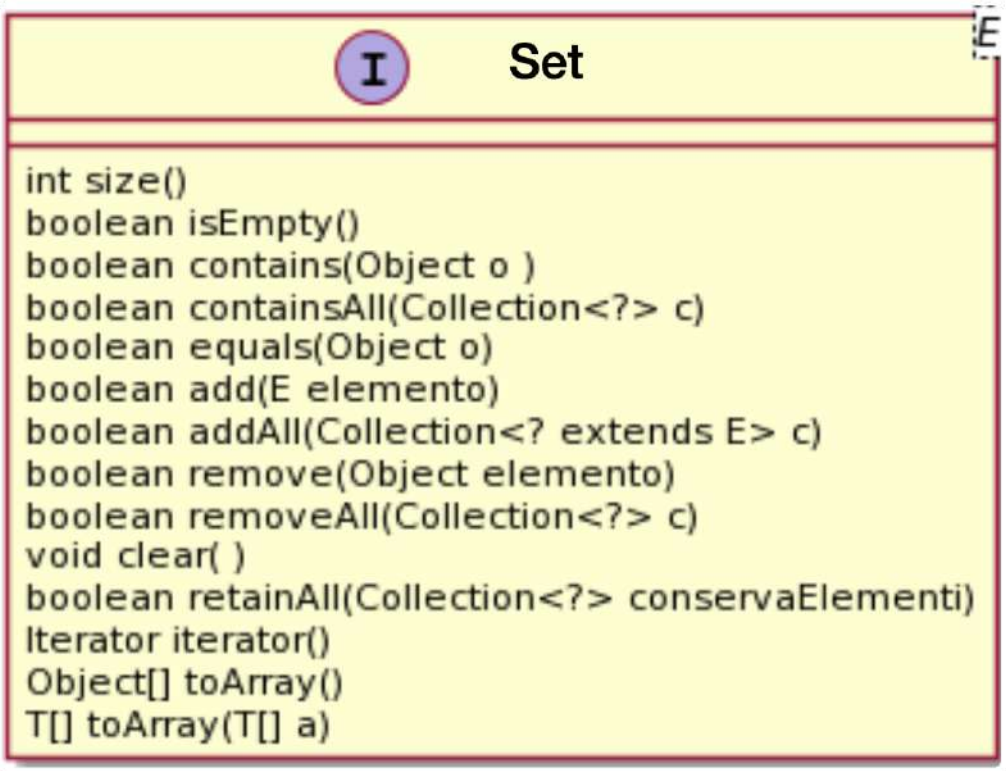
Interface Set<E>

L'interfaccia iava.util.Set definisce il tipo di dato Insieme. Le caratteristiche di questa collezione sono:

- non permette duplicati
- si perde il concetto di accesso posizionale ossia non si può richiedere di accedere all'elemento in posizione 10.



L'interfaccia Set non definisce alcun metodo aggiuntivo rispetto all'interface Collection, l'unica differenza tra le interfacce Collection e Set consiste nel fatto che in una raccolta che implementi Set non è consentita la presenza di elementi duplicati, un vincolo che riguarda le specifiche del costruttore di default e del metodo add.



Le operazioni possibili sono:

- test sulla presenza di un elemento;
- inserzione di una nuovo elemento:
- eliminazione di un elemento.

Metodi

boolean contains(Object) restituisce true se l'elemento è presente, false altrimenti.

boolean add(E element): inserisce l'elemento element, viene reso true se l'elemento è stato aggiunto, false se era già presente.

boolean remove (E element): rimuove l'elemento element se presente, viene reso true se l'elemento è stato eliminato, false se non era presente.

boolean addAll(Collection c): aggiunge gli elementi della Collection c se non presenti, (implementando così una variante della unione), viene reso true se l'insieme è stato modificato.

boolean removeAll(Collection c): rimuove gli elementi della Collection c se presenti. (implementando così una variante della differenza), viene reso true se l'insieme è stato modificato.

boolean retainAll(Collection c): conserva solo gli elementi della Collection c che sono presenti, (implementando così una variante della intersezione), viene reso true se l'insieme è stato modificato.

Inoltre, tra gli altri, vi sono i metodi: clear(), isEmpty(), size(), equals0), iterator(), dall'ovvio significato.

Un'implementazione dell'interfaccia Set è data dalla classe **HashSet.**

La parola 'hash"sì riferisce al fatto che la classe HashSet è implementata utilizzando una tabella hash. Una tabella hash (o mappa hash) è una struttura dati che contiene oggetti. Un oggetto è memorizzato in una tabella hash associandogli una **chiave (key}** univoca. Una tabella hash per un dato tipo di chiave è composta da:

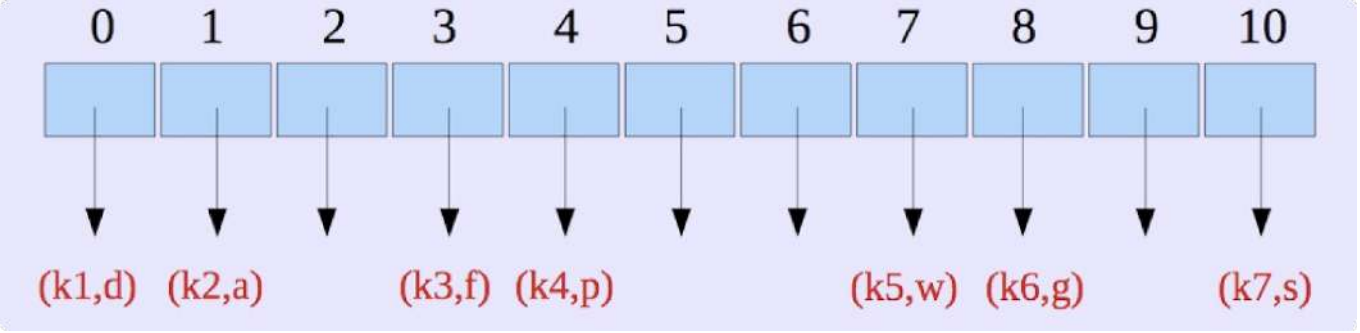
- un array (chiamato bucket array)
- una funzione hash h

I bucket array

Un bucket array per una tabella hash è un array A di lunghezza N. Ogni singola cella di A è considerata come un «bucket» (secchio), cioè come un insieme delle coppie chiave-valore. Il numero N definisce la capacità dell'array.

Se si considera che le chiavi sono dei numeri interi, uniformemente distribuiti nell'intervallo [0, N - 1], allora questo bucket array è proprio quello che occorre per implementare una tabella hash. Una entry e con chiave k è semplicemente inserita nel bucket A[k] (Figura). In questo caso le ricerche, gli inserimenti e le cancellazioni nel bucket array richiedono solo un tempo O(1).

Non sempre le chiavi sono interi nell'intervallo [0,N-1].

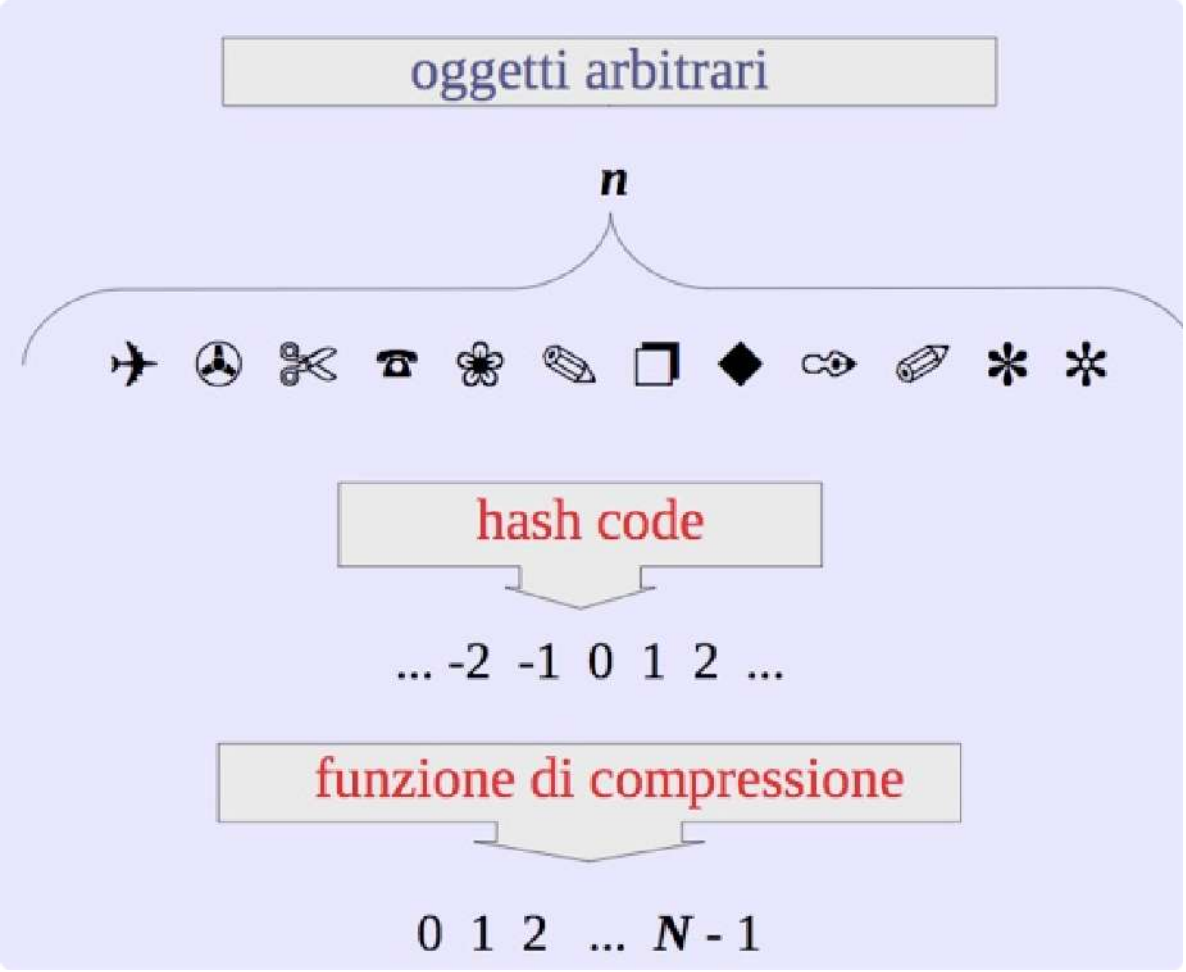


Funzioni hash

La seconda parte della struttura di una tabella hash è la funzione hash h: che trasforma ogni chiave k della mappa di origine in un numero intero compreso nell'intervallo [0, N - 1]. Avendo a disposizione questa funzione il metodo del bucket array può essere applicato a chiavi arbitrarie. L'idea quella è di utilizzare il valore della funzione hash. h(k), come indice nel bucket array, A, al posto della chiave k. In altre parole, si memorizza la entry (k,v) nel bucket A(h(k)).

Naturalmente, nell'ipotesi che esistano due o più chiavi aventi lo stesso valore hash, allora due entry distinte saranno mappate nello stesso bucket in A. In questo caso si dice che si verifica una collisione. Seguendo la convenzione di Java, per determinare il valore di una funzione hash, h(k), sono necessari due passi:

1. il primo è la trasformazione della chiave k in un numero intero, detto codice hash;
2. il secondo è l'applicazione della cosiddetta funzione di compressione, ovvero la trasformazione del codice hash ottenuto nel passo precedente in un numero intero, compreso nell'intervallo ([0, N-1]), degli indici di un bucket array.



La classe HashSet<E>

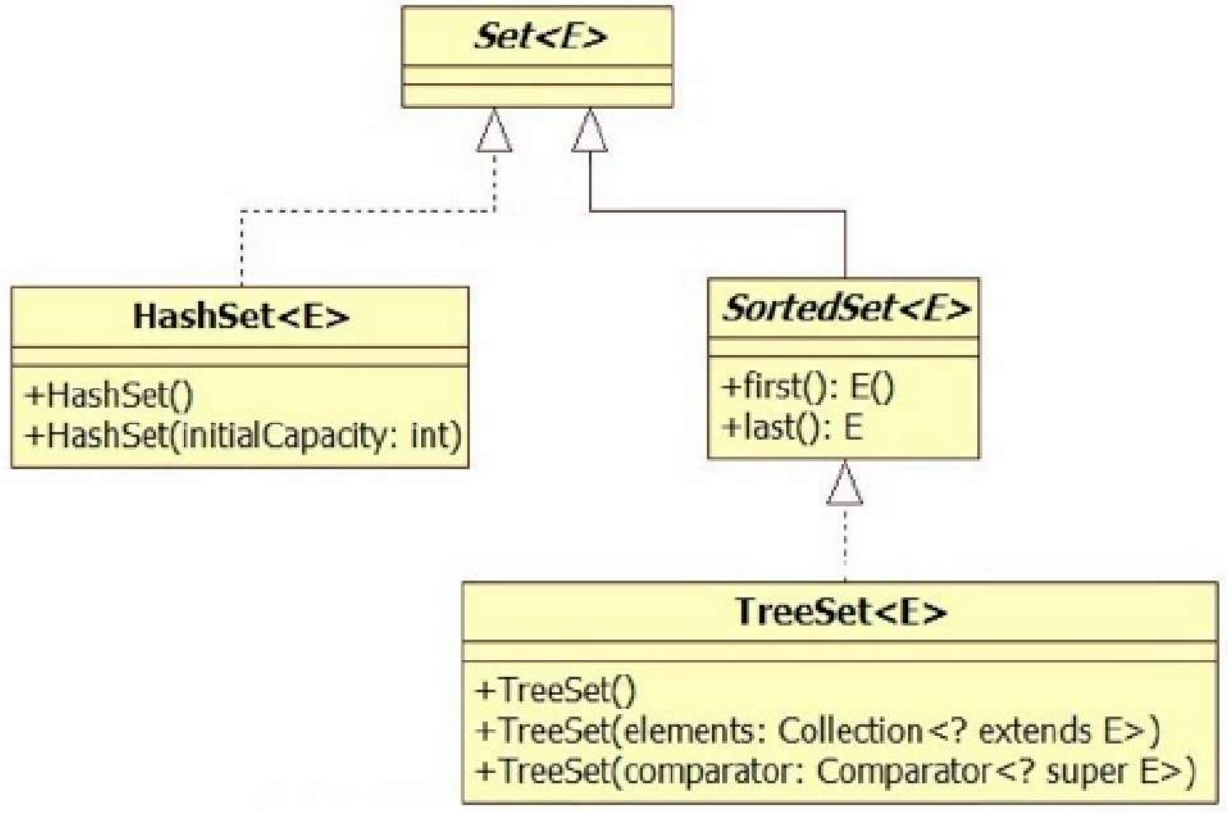
La classe HashSet implementa naturalmente tutti i metodi dell'interfaccia Set e non aggiunge altri metodi, a parte i costruttori.

Costruttori

HashSet() Crea un nuovo HashSet vuoto

HashSet(Collection<? extends T> c) Crea un nuovo insieme contenente tutti gli elementi di c. Se c è nuli, genera un'eccezione NullPointerException.

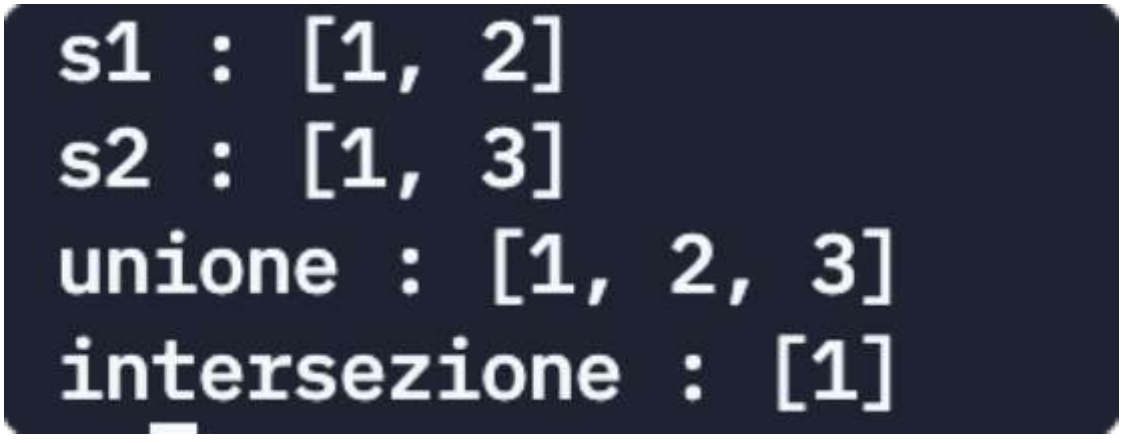
HashSet(int dimensioneiniziale) Crea un nuovo insieme vuoto della capacità specificata. Se la dimensione iniziale è minore di 0, genera un'eccezione IllegalArgumentException.




```
import java.util.HashSet;
import java.util.Iterator;
```

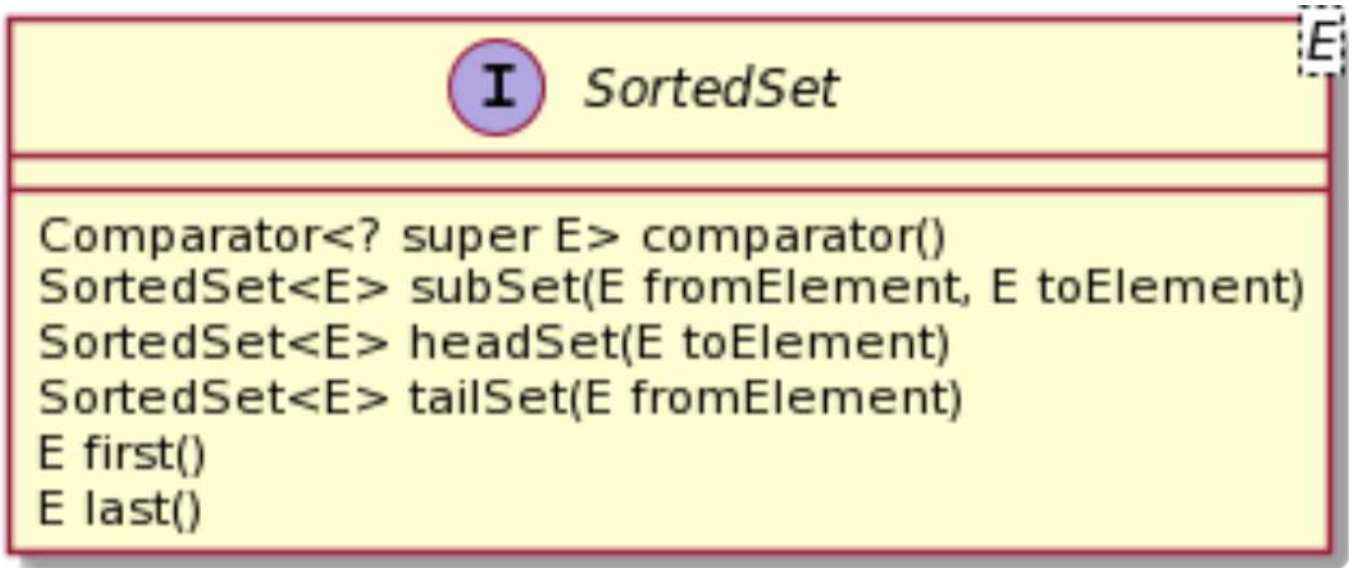
```java
import java.util.Set;
public class Main {
  public static void main(String[] args) {
    Set<Integer> s1 = new HashSet<>();
    s1.add(1);
    s1.add(2);
    System.out.println("s1 : "+s1);
    Set<Integer> s2 = new HashSet<>();
    s2.add(1);
    s2.add(3);
    System.out.println("s2 : "+s2);
    Set<Integer> union = unione(s1,s2);
    System.out.println("unione : "+union);
    Set<Integer> intersection = intersezione(s1,s2);
    System.out.println("intersezione : "+intersection);
    }
  // Calcola l'unione tra s1 ed s2, il metodo non modifica s1 ed s2
   private static Set<Integer> unione(Set<Integer> s1, Set<Integer> s2) {
    Set<Integer> res = new HashSet<Integer>(s1);
    res.addAll(s2);
    return res;
  }
    // Calcola l'intersezione di s1 ed s2, il metodo non modifica s1 ed s2
    private static Set<Integer> intersezione(Set<Integer> s1, Set<Integer> s2) {
      Set<Integer> s3=new HashSet<>();
      Iterator<Integer> a= s1.iterator();
      while(a.hasNext()){
        int i= a.next();
    if(s2.contains(i))
          s3.add(i);}
          return s3;
    }
}
```

Output:



L'interfaccia SortedSet<E>

L'interfaccia SortedSet rappresenta una collezione di elementi senza duplicati, ordinati secondo un ordinamento ascendente naturale oppure secondo uno arbitrario fornito mediante un oggetto comparatore (Comparator).



Estende, di fatto, l'interfaccia Set fornendo i seguenti ulteriori metodi:

Comparator<? super E> comparator() ritorna l'oggetto comparatore utilizzato per l'ordinamento degli elementi di un insieme. Se non è stato fornito alcun oggetto comparatore, il metodo ritornerà null.

E first() ritorna l'elemento di un insieme al primo posto nell'ordinamento.

E last() ritorna l'elemento di un insieme all'ultimo posto nell'ordinamento.

SortedSet<E> subSet(E fromElement, E toElement) ritorna un sottoinsieme di un insieme formato dagli elementi compresi nel range indicato dal parametro fromElement (incluso) e dal parametro toElement (escluso).

SortedSet<E> headSet(E toElement) ritorna un sottoinsieme di un insieme formato dagli elementi che, secondo l'ordinamento, sono posti prima dell'elemento fornito dal parametro toElement (escluso).

SortedSet<E> tailSet(E fromElement) ritorna un sottoinsieme di un insieme formato dagli elementi che secondo l'ordinamento, sono posti dopo o a partire dall'elemento fornito dal parametro fromElement (incluso).

La Classe TreeSet<E>

TreeSet, è un Set implementato interiormente come un albero di ricerca bilanciato. Questa classe estende l'interfaccia SortedSet perché gli elementi vengono inseriti in modo ordinato secondo un ordinamento naturale della classe o non naturale.

Di conseguenza gli elementi presenti nel TreeSet devono implementare l'interfaccia Comparable o l'interfaccia Comparator. Nello specifico gli elementi inseriti in un TreeSet formano un albero rosso/nero e indipendentemente dall'inserimento, l'iterarotre restituisce gli oggetti in modo ordinato.

Costruttori

In TreeSet abbiamo a disposizione diversi costruttori, tra cui:

• **public TreeSet():** Consente di creare un TreeSet vuoto;

• **public TreeSet (Collection c):** Consente di creare un TreeSet con gli elementi contenuti nella Collection passata come argomento;

• **public TreeSet (Comparator c):** Consente di assegnare agli elementi da ordinare una relazione d'ordine non naturale.

Iterando con un iteratore su TreeSet, gli elementi verranno restituiti in maniera ordinata.

Metodi

boolean add(E e) Aggiunge l'elemento specificato a questo insieme se non è già presente.

boolean addAll (Collection<? extends E> c) Aggiunge tutti gli elementi della collezione specificata a questo set.

E ceiling(E e) Restituisce l'elemento minimo in questo set maggiore o uguale all'elemento dato o null se non esiste un tale elemento.

void clear () Rimuove tutti gli elementi da questo set.

Object clone() Restituisce una copia superficiale di questa TreeSet istanza.

Comparator<? super E> comparator () Restituisce il comparatore usato per ordinare gli elementi in questo set, o null se questo insieme usa l'ordinamento naturale dei suoi elementi.

boolean contains (E e) Restituisce true se questo insieme contiene l'elemento specificato.

Iterator<E> descendingIterator () Restituisce un iteratore sugli elementi in questo set in ordine decrescente.

Navigableset<> descendingset ( ) Restituisce una vista di ordine inverso degli elementi contenuti in questo Set

E first() Restituisce il primo (più basso) elemento attualmente in questo set.

boolean isEmpty() Restituisce true se questo insieme non contiene elementi.

Iterator<E> iterator () Restituisce un iteratore sugli elementi in questo set in ordine crescente.

E last () Restituisce l'ultimo (il più alto) elemento attualmente in questo set.

boolean remove (E e) Rimuove l'elemento specificato da questo set se è presente.

int size() Restituisce il numero di elementi in questo set (la sua cardinalità).

NavigableSet<E> tailSet (E fromElement, boolean inclusive) Restituisce una vista della porzione di questo insieme i cui elementi sono maggiori di (o uguali a se inclusive è vero) fromElement.

```
public class Main {

  public static void main(String[] args) {
    SortedSet<String> set = new TreeSet<>();
    set.add("pane");
    set.add("frutta");
    set.add("pasta");
    set.add("vino");
```

```
    set.add("birra");
    set.add("acqua");
    System.out.println("lista: " + set);
    Iterator<String> iterator = set.iterator();
    while (iterator.hasNext())
      System.out.println(iterator.next());
  }
}
```

Output:

```
lista: [acqua, birra, frutta, pane, pasta, vino]
acqua
birra
frutta
pane
pasta
vino
```

Oppure inserendo un oggetto con Comparatore:

```
public class Main {
   public static void main(String[] args) {
     SortedSet<Persona> set = new TreeSet<>();
Persona p1 = new Persona( "Bianchi", "Mario","Via Firenze
1");
Persona p2 = new Persona("Rossi","Giorgio",  "Via Roma 2");
set.add(p1);
set.add(p2);
set.add(new Persona("Rossi","Ambra",  "Via Milano 3"));
Iterator<Persona> iterator = set.iterator();
while(iterator.hasNext())
System.out.println(iterator.next().toString());
```

```
    }
}

public class Persona implements Comparable <Persona>{
  private final String cognome;
  private final String nome;
  private final String indirizzo;
Persona(String cognome,String nome, String indirizzo){
    this.cognome=cognome;
    this.nome=nome;
    this.indirizzo=indirizzo;
}
        @Override
  public int compareTo(Persona o) {
    int r = cognome.compareToIgnoreCase(o.cognome);
    if (r == 0) {
      r = nome.compareToIgnoreCase(o.nome);
    }
    return r;
  }
   @Override
    public String toString() {
        return "Persona{" + "cognome=" + cognome + ", nome="
+ nome + ", indirizzo=" + indirizzo + '}';
    }
}
```

Output:

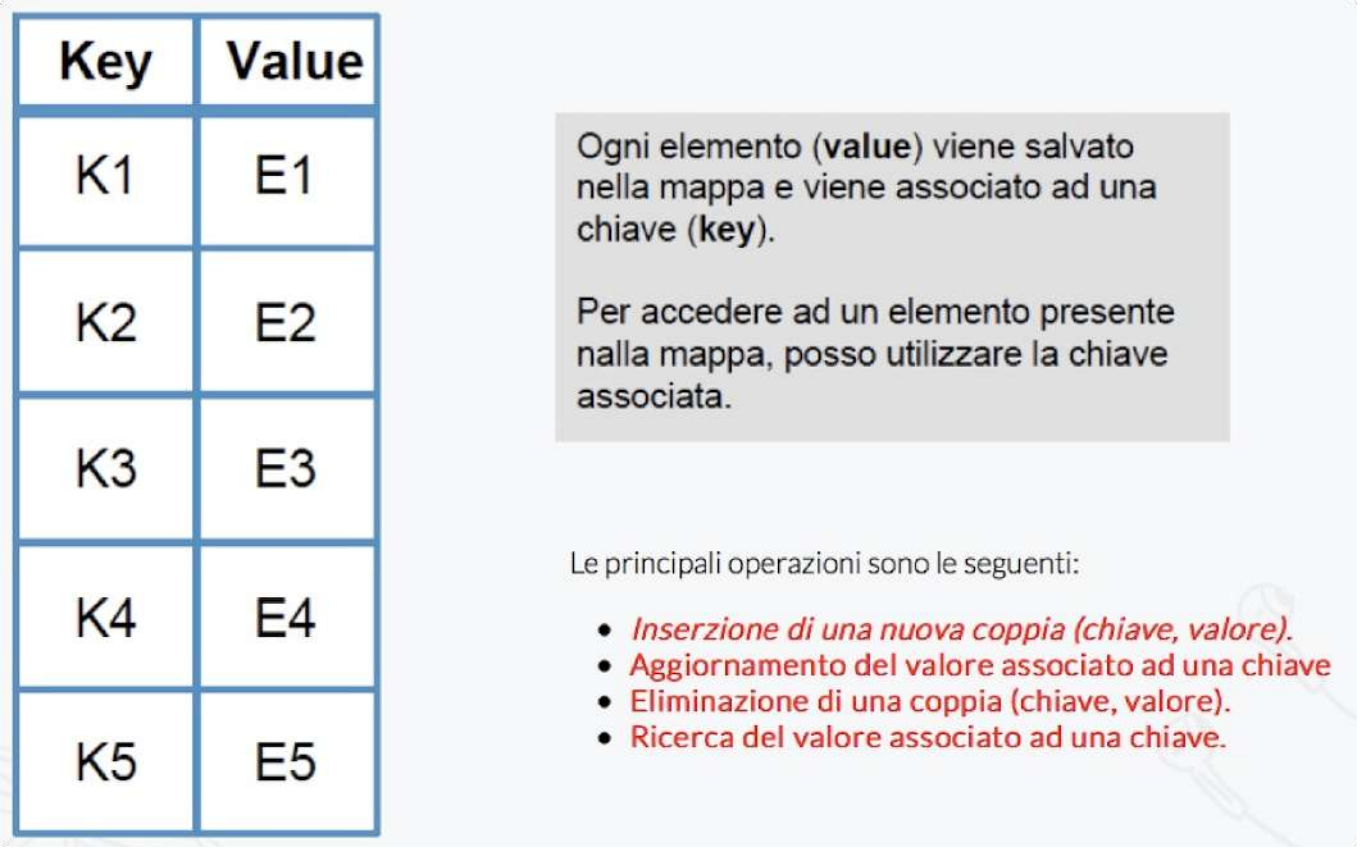
```
Persona{cognome=Bianchi, nome=Mario, indirizzo=Via Firenze 1}
Persona{cognome=Rossi, nome=Ambra, indirizzo=Via Milano 3}
Persona{cognome=Rossi, nome=Giorgio, indirizzo=Via Roma 2}
```

Le mappe

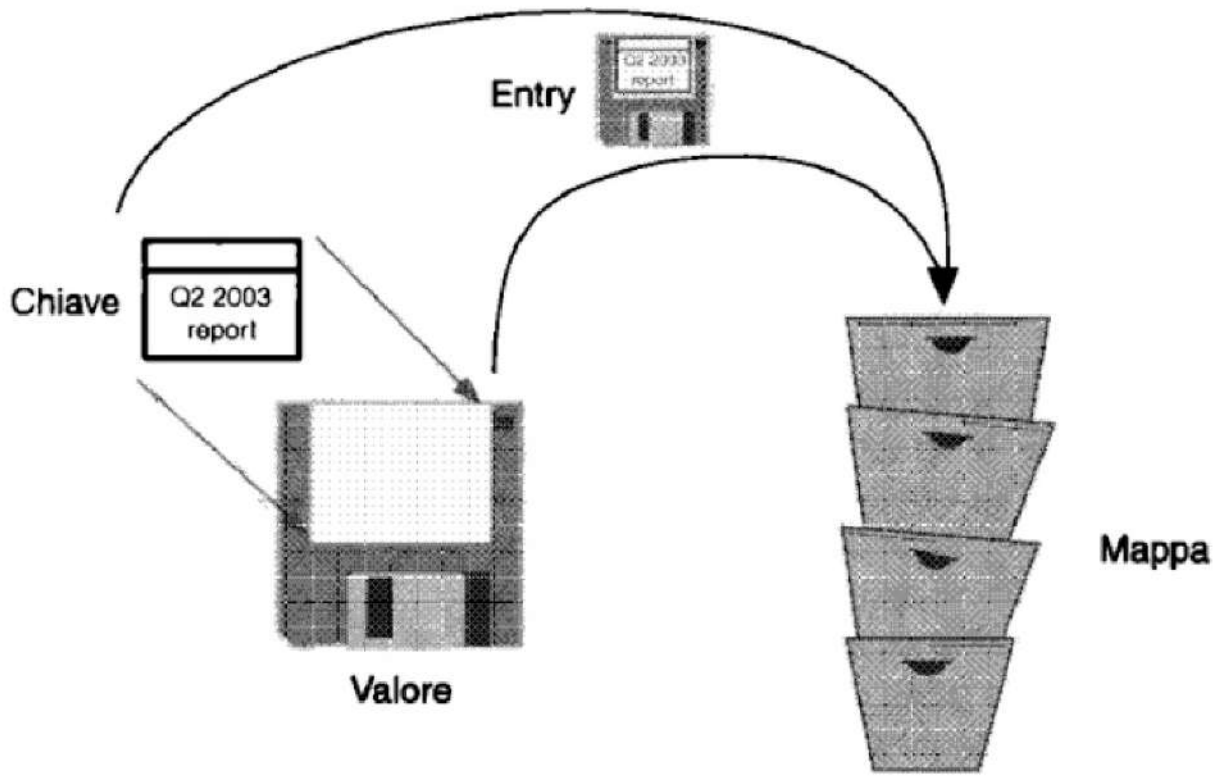
Una mappa permette di memorizzare degli elementi che possono essere trovati velocemente mediante delle chiavi. In particolare, una mappa memorizza coppie chiave-valore (k,v), chiamate entry, in cui:

- k: è la chiave.
- v è il valore a essa corrispondente.

Inoltre ogni chiave è unica e così la corrispondenza tra chiavi e valori definisce una funzione (mapping). In una mappa che memorizza dei record studente (con nomi, indirizzi e classi frequentate dagli studenti), la chiave può essere costituita dal numero ID dello studente.



Le chiavi (etichette) sono assegnate a valori (dischetti) da un utente. Le entry che ne risultano (dischetti etichettati) sono inserite in una mappa (contenitore). Le chiavi possono essere utilizzate in seguito per trovare oppure cancellare i valori.



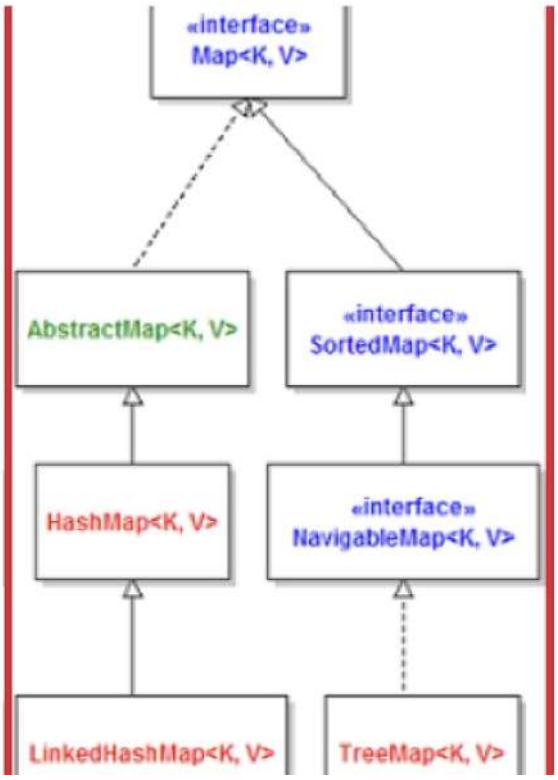
Per gestire le mappe, Java mette a disposizione una serie strumenti che consentono di gestire liste di oggetti dinamiche e non ordinate. Le map sono composte da:

- interfacce che definiscono i metodi utili per la manipolazione delle collezioni di oggetti
- classi implementano le interfacce

Le classi e interfacce fanno parte del package java.util. Tra i metodi per la manipolazione delle mappe di oggetti troviamo quelli che consentono la ricerca rapida di elementi all'interno della mappa.

L'interfaccia **Map<K.V>** definisce tutti i metodi che devono essere implementati per la corretta gestione delle mappe. Le classi principali che implementano l'interfaccia Mao sono:

HashMap - Hashtable - Properties - TreeMap

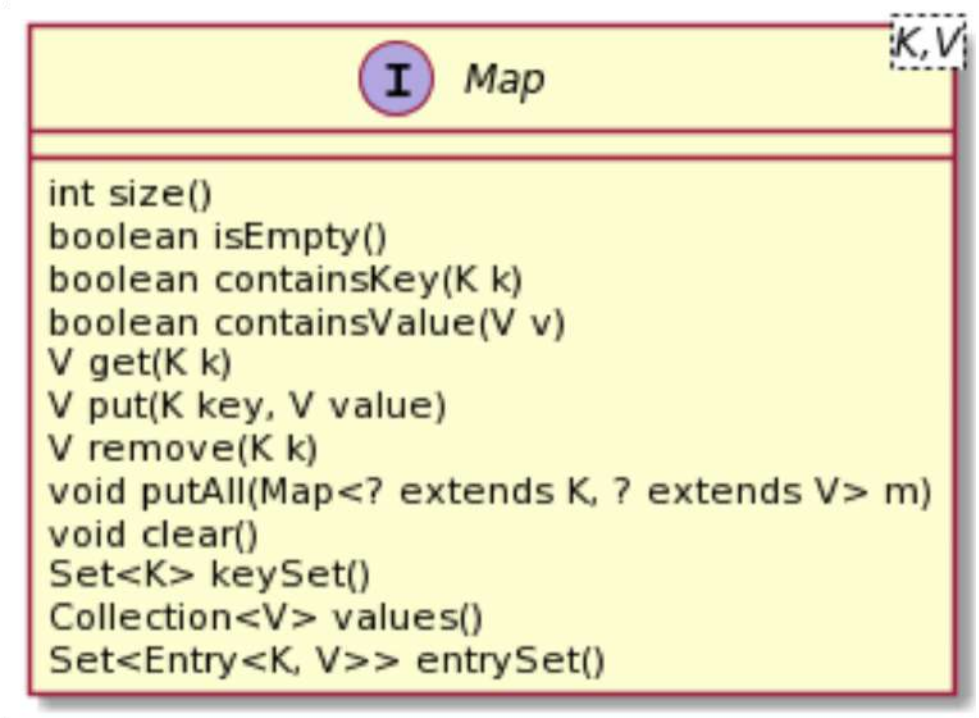


Un dizionario è un esempio di mappa:

o la chiave è la parola che viene definita

o Il valore è costituito dalla sua definizione e dall'etimologia

Interface Map<K,V>



Parametri:

K - il tipo di chiavi gestite da questa mappa

V - il tipo di valori mappati

Di seguito, alcuni metodi dell'interfaccia Map:

V put(K k,V v) Associa, alla mappa, il valore specificato alla chiave specificata. Se la mappa conteneva già un'associazione per la chiave, il vecchio valore viene sostituito. Ritorna il valore precedentemente associato a key, oppure null se non vi era alcuna associazione riguardante key.

boolean containsKey(K key) Determina se questo oggetto di tipo Map contiene un'associazione per la chiave specificata.

Viene lanciata un eccezione:

ClassCastException se la chiave specificata non può essere confrontata con le chiavi presenti nella mappa.

NullPointerException se la chiave specificata è uguale a null se questa mappa usa l'ordinamento naturale, oppure il suo comparatore non consente l'uso di chiavi uguali a null.

boolean containsValue(V value) Determina se questo oggetto di tipo Map contiene almeno un'associazione che coinvolga il valore specificato.

V get(K key) Restituisce il valore associato alla chiave specificata all'interno di questo oggetto di tipo Map, altrimenti, restituisce null.

Lancia un eccezione:

ClassCastException se la chiave specificata non può essere confrontata con le chiavi presenti nella mappa.

NullPointerException se la chiave specificata è uguale a null e questa mappa usa l'ordinamento naturale, oppure il suo comparatore non consente l'uso di chiavi uguali a null.

V remove(Object key) Elimina da questo oggetto di tipo Map l'associazione relativa alla chiave specificata, se tale associazione esiste. Ritorna il valore precedentemente associato alla chiave specificata, se tale chiave era associata a un valore; altrimenti, restituisce null.

Lancia un eccezione:

ClassCastException se la chiave specificata non può essere confrontata con le chiavi presenti nella mappa.

NullPointerException se la chiave specificata è uguale a null e questa mappa usa l'ordinamento naturale, oppure il suo comparatore non consente l'uso di chiavi uguali a null.

Set<Entry<k,V>> entrySet() Restituisce un oggetto di tipo Set contenente le coppie chiave-valore presenti in questo oggetto di tipo Map.

Set<K> keySet() : Restituisce un insieme di tutte le chiavi di una mappa ad albero

Collection<V> values(): Restituisce una collezione di tutti i valori contenuti nella mappa

L'interfaccia Map non ha il metodo iterator(), per cui non è possibile scandire un oggetto di tipo Map se non tramite una "vista" come quella generata dal metodo entrySet(). A questo scopo, l'interfaccia Map ha un'interfaccia interna pubblica. Entry, che dispone dei metodi getKey() e getValue().

Sono definiti anche i metodi keySet() e values(), che consentono, rispettivamente, di effettuare una scansione di un oggetto di tipo Map visto come insieme di chiavi o come raccolto di valori. Il termine "raccolta di valori" è più adeguato di "insieme di valori", dal momento che ci possono essere valori duplicati.

La classe HashMap

La classe HashMap è un'implementazione dell'interfaccia Map. Sviluppata avvalendosi della struttura di dati di tipo mappa a sua volta implementata come tabella hash. Non garantisce alcun ordinamento e consente valori e chiavi nulle.

Un'istanza di HashMap ha due parametri che ne influenzano le prestazioni:

1. Capacità iniziale: il numero di voci nella tabella hash al momento della creazione.

2. load factor: è una misura di quante voci è possibile inserire nella la tabella hash prima che la sua capacità venga aumentata automaticamente.

Quando il numero di voci nella tabella hash supera il prodotto del fattore di carico per la capacità corrente, la tabella hash viene rehashed (ovvero, le strutture di dati interne vengono ricostruite) in modo che la tabella hash abbia circa il doppio del numero di bucket.

Come regola generale, il load factor predefinito (.75) offre un buon compromesso tra costi di spazio e tempo.

Il numero previsto di voci nella mappa e il suo fattore di carico dovrebbero essere presi in considerazione quando si imposta la sua capacità iniziale, in modo da ridurre al minimo il numero di operazioni di rehash.

La classe HashMap<K, V> implementa tutti i metodi dell'interfaccia Map<K, V>. Gli unici metodi in più sono i costruttori.

public HashMap() Crea una nuova mappa vuota con capacità iniziale 16 e fattore di carico 0.75.

public HashMap(int capacitaIniziale) Crea una nuova mappa vuota con capacità iniziale specificata e fattore di carico 0.75. Genera una IllegalArgumentException se capacitaIniziale è negativa.

public HashMap(int capacitaIniziale, float fattoreDiCarico) Crea una nuova mappa vuota con capacità iniziale e fattore di carico specificati. Genera una IllegalArgumentException se capacitaIniziale è negativa o fattoreDiCarico è non positivo.

public HashMap(Map<? extends K, ? extends V> m) Crea una nuova mappa contenente le stesse associazioni della mappa m. La capacità iniziale è impostata alle dimensioni di m e il fattore di carico a 0.75. Genera una NullPointerException se m è null.

Le principali operazioni sulle mappe sono le seguenti:

- Inserzione di una nuova coppia (chiave, valore).
- Aggiornamento del valore associato ad una chiave.
- Eliminazione di una coppia (chiave, valore).
- Ricerca del valore associato ad una chiave.

Nella tabella è mostrato l'effetto di una serie di operazioni su una mappa inizialmente vuota, che deve contenere entry con chiavi intere e valori consistenti in un singolo carattere.

Operazione	Output	Mappa
isEmpty()	true	Vuota
put(5,A)	null	{(5,A)}
put(7,B)	null	{(5,A), (7,B)}
put(2,C)	null	{(5,A), (7,B), (2,C)}
put(8,D)	null	{(5,A), (7,B), (2,C), (8,D)}
put(2,C)	C	{(5,A), (7,B), (2,C), (8,D)}
get(7)	B	{(5,A), (7,B), (2,C), (8,D)}
get(4)	null	{(5,A), (7,B), (2,C), (8,D)}
get(2)	C	«5,A), (7,0), (2,C), (8.D)}
size()	4	{(5,A), (7,B), (2,C), (8,D)}
remove(5)	A	{(7,B), (2,C), (8,D)}
remove(2)	E	{(7,B), (8,D)}
get(2)	null	{(7,B), (8,D)}
IsEmpty()	false	{(7,B), (8,D)}

Esempio: gestire una classifica con una mappa. Immaginiamo di voler gestire una classifica dei Piloti di Formula 1 attraverso una mappa in cui ciascuna entry sia rappresentata da:

1. Una chiave che identifica il posizionamento del Pilota.
2. Un valore (il nome del Pilota).

```
import java.util.HashMap;
import java.util.Map;
public class Main {

    public static void main(String[] args) {
        Map<Integer, String> classifica = new HashMap<>();
        classifica.put(1, "Senna");
        classifica.put(2, "Prost");
        classifica.put(3, "Mansell");
        classifica.put(4, "Lauda");
        classifica.put(5,"Patrese");
```

```
        System.out.println("Classifica");
       for(Integer key:classifica.keySet())
        System.out.println(key+"  "+classifica.get(key));
    }
}
```

Output:


```
public static void main(String[] args) {
 Map<Integer, String> classifica = new HashMap<>();
 classifica.put(1, "Senna");
 classifica.put(2, "Prost");
 classifica.put(3, "Mansell");
 classifica.put(4, "Lauda");
 classifica.put(5, "Patrese");
 System.out.println("Size of Map  " + classifica.size());
 System.out.println("La Map contiene l'elemento con key=1  " +
classifica.containsKey(1));
System.out.println("La Map contiene l'elemento con valore=Senna "
+ classifica.containsValue("Senna"));
```

```
System.out.println("La Map è Vuota? " + classifica.isEmpty());
System.out.println("l'elemento con chiave 2 rimosso dalla Mappa è
" + classifica.remove(2));
System.out.println("Size of Map  " + classifica.size());
classifica.clear();
 System.out.println("Size of Map  " + classifica.size());
  }
}
```

Output:

```
Size of Map  5
La Map contiene l'elemento con key=1  true
La Map contiene l'elemento con valore=Senna   true
La Map è Vuota? false
l'elemento con chiave 2 rimosso dalla Mappa è  Prost
Size of Map  4
Size of Map  0
```

Per sapere se all'interno di una "mappa" è contenuta una data chiave o un dato valore, si usano rispettivamente i due metodi "containsKey()" e "containsValue()", il primo dei quali riceve come parametro la chiave. Mentre il secondo il valore, e ritorna "true" se presente, altrimenti "false".

Iterare le HashMap

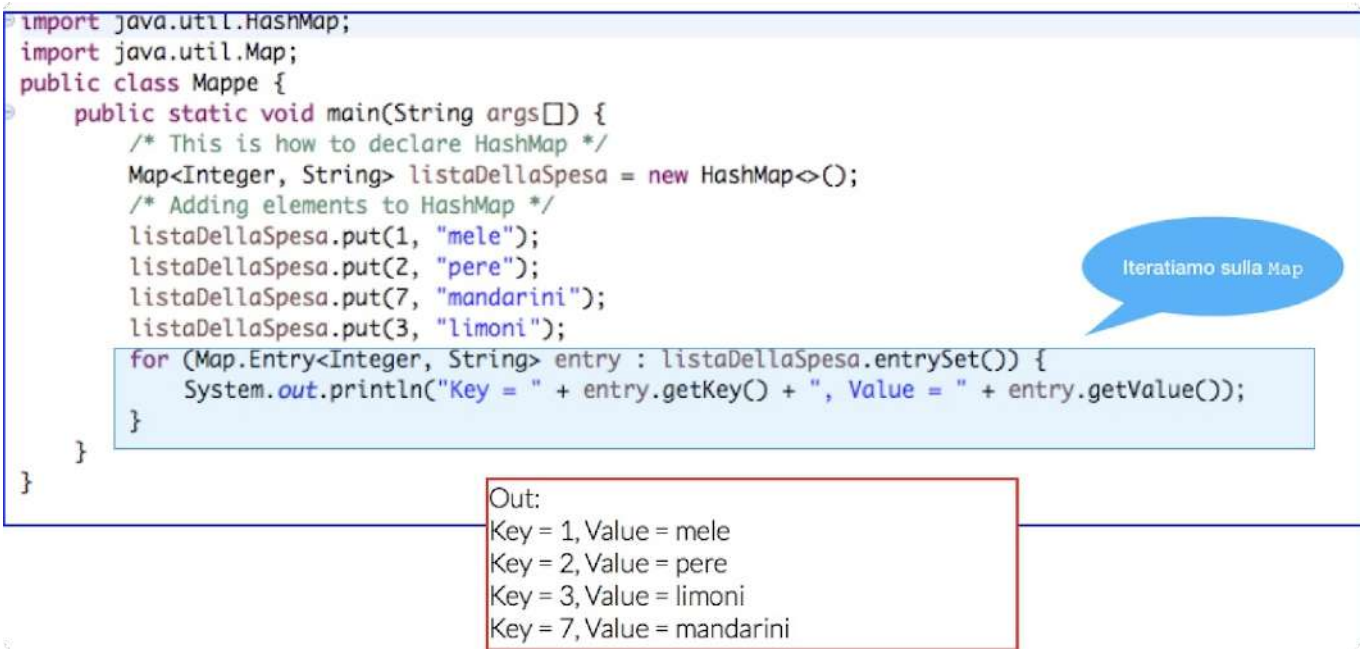
Esistono diversi modi per iterare una Map in Java. Vediamo questi metodi considerando i vantaggi e i svantaggi. Considerando che tutte le mappe in Java implementano l'interfaccia Map, le seguenti tecniche funzionano per qualsiasi implementazione mappa (HashMap, TreeMap, LinkedHashMap, Hashtable, etc.)

Metodo 1 : Iterazione delle voci utilizzando un ciclo For-Each.

Questo è il metodo più comune ed è preferibile nella maggior parte dei casi. Dovrebbe essere usato se si ha bisogno della coppia chiave-valore.

```
Map<K, v> map = new HashMap<>0:
for (Map.Entry<K,V> entry :map. entrySet()) {
System.out.println("Key ="+ entry. getKey()+", Value = "+
entry.getvalue));
```

```
}
```

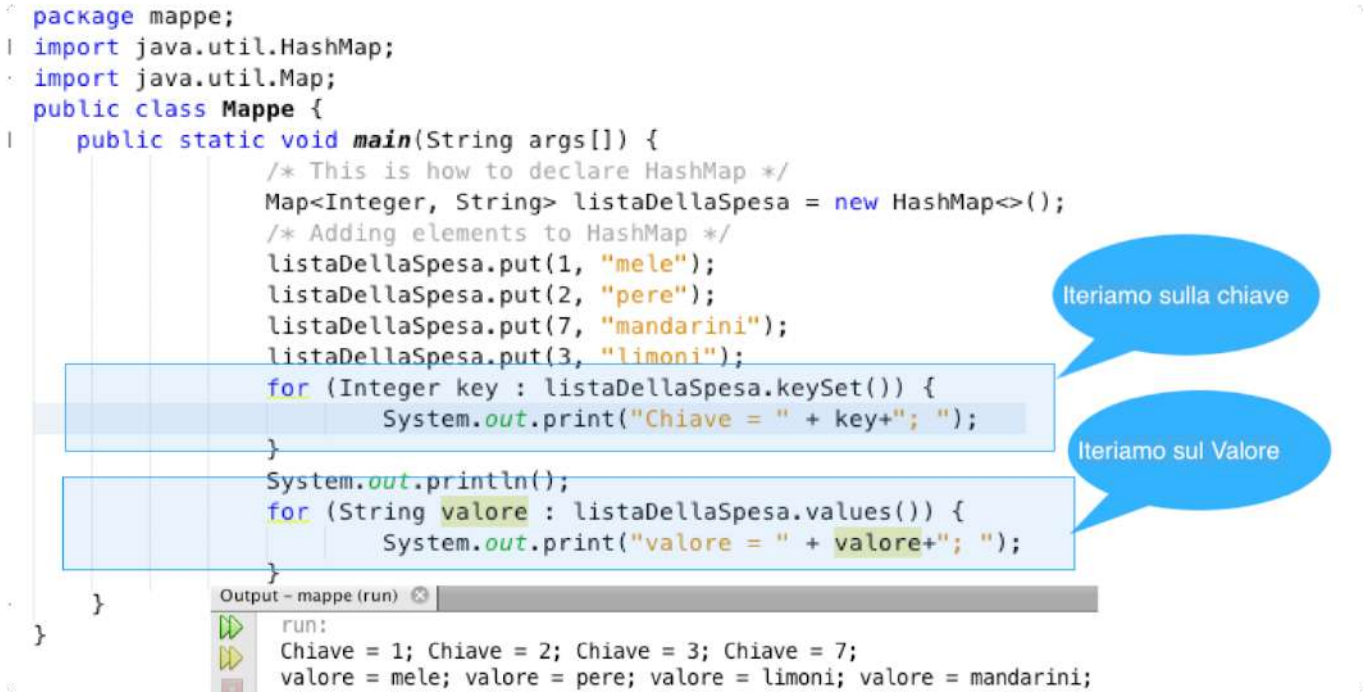


Metodo 2: Iterazione su chiavi o valori utilizzando un ciclo For-Each.

Se si ha bisogno solo di chiavi o solo dei valori si puoi eseguire ('iterazioni su keySet o value invece di entrySet.

```
Map<K,V> map =new HashMap<>();
for(K key : map.keySet()){
System.out.println("Key = "+ key);
}
 for(V value : map.values()) {
System.out.println("Value = "+ value);
}
```

Questo metodo offre un leggero vantaggio in termini di prestazioni rispetto entrySet (circa il 10% più veloce) ed è più pulito.



Metodo 3: Tramite iterator

Senza Generics

```
Map map =newHashMap() ;
Iterator itera = mappa.entrySet().iterator():
while(itera.hasNext()){
Map. Entry coppia = (Map.Entry) itera.next();
System.out.println("Chiave: "coppia. getKey)+", valore:
"+coppia.getValue());
}
```

```java
package mappe;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class Mappe {
   public static void main(String args[]) {
            /* This is how to declare HashMap */
            Map listaDellaSpesa = new HashMap<>();
            /* Adding elements to HashMap */
            listaDellaSpesa.put(1, "mele");
            listaDellaSpesa.put(2, "pere");
            listaDellaSpesa.put(7, "mandarini");
            listaDellaSpesa.put(3, "limoni");
            Set set = listaDellaSpesa.entrySet();
            Iterator iterator = set.iterator();
              while (iterator.hasNext())
              {
                  Map.Entry chiaveValore = (Map.Entry) iterator.next();
                  Integer key= (Integer)chiaveValore.getKey();
                  String valore=(String)chiaveValore.getValue();
                  System.out.println("Key ="+key+"  valore = " + valore);
              }
   }
}
```

Iteratore senza Generics

Uso di Generics:

```java
Map<K, V> map =newHashMap<>():
Iterator<Map. Entry<K, V>> entries = map.entrySet
().iterator();
while(entries.hasNext()){
Map. Entry<K, V> entry =(Map.Entry<K,V>)entries.next();
System.out.println("Key "+ entry.getkey()+", Value"+
entry. getValue());
}
```

```java
package mappe;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class Mappe {
   public static void main(String args[]) {
            /* This is how to declare HashMap */
            Map<Integer, String> listaDellaSpesa = new HashMap<>();
            /* Adding elements to HashMap */
            listaDellaSpesa.put(1, "mele");
            listaDellaSpesa.put(2, "pere");
            listaDellaSpesa.put(7, "mandarini");
            listaDellaSpesa.put(3, "limoni");
            Set set = listaDellaSpesa.entrySet();
            Iterator <Map<Integer,String>> iterator = set.iterator();
              while (iterator.hasNext())
              {
                  Map.Entry<Integer,String> chiaveValore = (Map.Entry<Integer,String>) iterator.next();
                  System.out.println("Key ="+chiaveValore.getKey()+"  valore = " + chiaveValore.getValue());
              }
   }
}
```

Iteratore Con Generics

Creato l'oggetto map si costruisce l'iteratore della collezione mappa.entrySet() richiamando il metodo "iterator().

Con il ciclo while si itera finché la collezione avrà elementi (hasNext() ritorna "true" se ci sono ancora elementi). Si crea un nuovo oggetto di tipo Map.Entry e tramite casting esplicito gli si assegna ciò che ritorna il metodo next() dell'iteratore.

Si stampa il contenuto della collezione attraverso i metodi "getKey()" e "getValue()" che ritornano rispettivamente chiave e valore.

```
public class Main {
  public static void main(String[] args) {
    Map<Integer, Character> alfa = new HashMap<>();
    System.out.println("la mappa alfa di dimensione " +
alfa.size());

    alfa.put(1, 'a');
    alfa.put(2, 'b');
    alfa.put(7, 'g');
    alfa.put(3, 'd');
    alfa.put(4, 'c');
    alfa.put(5, 'e');
    alfa.put(21, '1');
    alfa.put(8, 'h');
    alfa.put(9, 'i');

    // Usiamo iterator per visualizzare
    Set set = alfa.entrySet();
    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {
      Map.Entry chiave_valore = (Map.Entry) iterator.next();
      System.out.println("La coppia chiave valore è: " +
chiave_valore);
    }
    /* Get values based on key */
    char var = alfa.get(2);
    System.out.println("Il valore di chiave 2 : " + var);
    System.out.println(" ");
    /* rimuoviamo il valore 21 */
    alfa.remove(21);
    // modifichiamo te posizioni 3 e 4
    alfa.put(3, 'c');
    alfa.put(4, 'd');
```

```
    System.out.println("la Mappa dopo l'aggiornamento :");
    Set set2 = alfa.entrySet();
    Iterator iterator2 = set2.iterator();
    while (iterator2.hasNext()) {
      Map.Entry entry2 = (Map.Entry) iterator2.next();
      System.out.print("Key is: " + entry2.getKey() + " Value is:
");
      System.out.println(entry2.getValue());
    }
  }
}
```

Codice

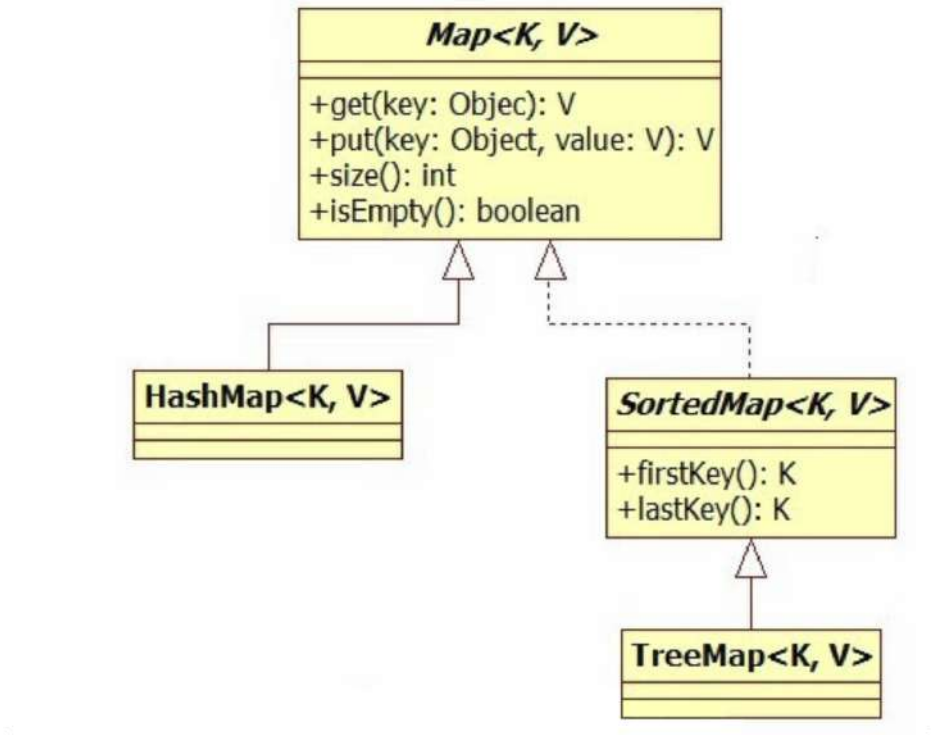
Ordinamento HashMap

HashMap non conserva alcun ordine per impostazione predefinita. Se c'è una necessità, dobbiamo ordinarla in modo esplicito in base al requisito. Vediamo come ordinare HashMap per :

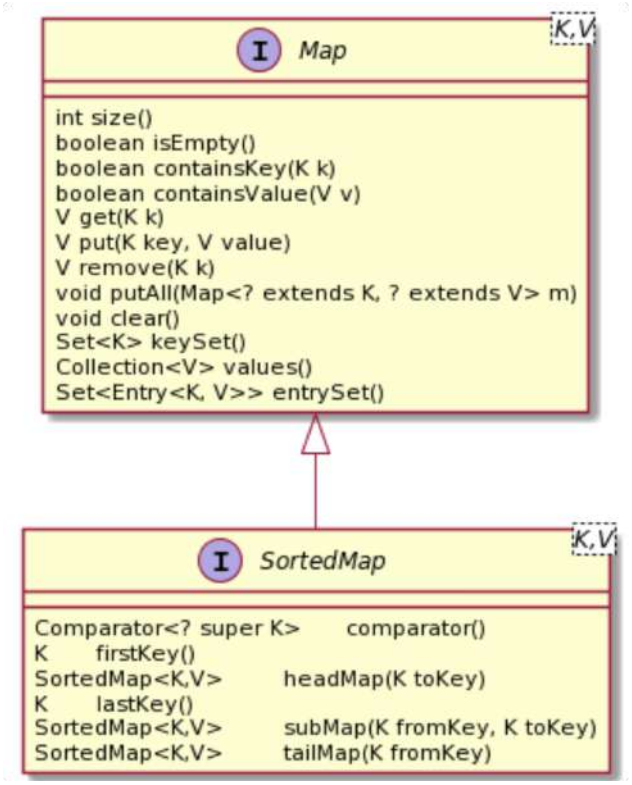
- chiavi usando TreeMap
- •valori usando Comparator .

HashMapOrdinamento per chiavi

Nel caso di una TreeMap le chiavi devono essere oggetti confrontabili. Questo requisito si ottiene facendo in modo che la classe che rappresenta le chiavi implementi l'interfaccia Comparable, oppure utilizzando un costruttore di TreeMap che accetti come input un oggetto di una classe che implementi l'interfaccia Comparator.



Interface SortedMap<K,V)



Comparator<? super K> comparator(): Restituisce il comparatore utilizzato per ordinare le chiavi in questa mappa, o null se questa mappa utilizza l'ordinamento naturale delle sue chiavi.

SortedMap<K,V> headMap(K toKey): Restituisce una vista della porzione di questa mappa le cui chiavi sono rigorosamente minori di toKey.

SortedMap<K,V> subMap(K fromKey, K toKey): Restituisce una vista della porzione di questa mappa le cui chiavi vanno da fromKey, inclusivo, a toKey, esclusivo.

S**ortedMap<K,V> tailMap(K fromKey):** Restituisce una vista della porzione di questa mappa le cui chiavi sono maggiori o uguali a fromKey.

K firstKey(): Restituisce la prima chiave (più bassa) attualmente in questa mappa.

K lastKey(): Restituisce l'ultima chiave (più alta) attualmente in questa mappa.

La Classe TreeMap<K,V>

Questa è la classe concreta che permette di implementare una mappa ordinata utilizzando come chiavi classi che producono oggetti confrontabili.

public class TreeMap extends AbstractMap implements NavigableMap, Cloneable, Serializable

Costruttori

TreeMap(): Crea una mappa vuota che verrà ordinata utilizzando l'ordine naturale delle sue chiavi.

TreeMap(Comparator comp): Crea un oggetto TreeMap vuoto in cui gli elementi sono ordinati secondo la specifica data dal comparatore.

TreeMap(Map m): Crea una TreeMap con le voci della mappa che verranno ordinate usando l'ordine naturale delle chiavi.

TreeMap(SortedMap sm): Crea una TreeMap con le voci della mappa ordinata data che verranno archiviate nello stesso ordine della mappa ordinata data.

La TreeMapclasse fornisce vari metodi che ci consentono di eseguire operazioni sulla mappa.

1 Inserimento

- **V put(K k,V v)** - inserisce la mappatura chiave/valore specificata (voce) nella mappa
- **void putAll(Map<K,V> m)** - inserisce tutte le voci dalla mappa specificata a questa mappa
- **V putIfAbsent(K, k, V v)** - inserisce la mappatura chiave/valore specificata nella mappa se la chiave specificata non è presente nella mappa

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {
        // Creare la mappa
        TreeMap<String, Integer> numeri = new TreeMap<>();

        // Uso put()
       numeri.put("Quatto", 4);
       numeri.put("Due", 2);
        // uso putIfAbsent()
        numeri.putIfAbsent("Sei", 6);
        System.out.println("La mappa numeri: " + numeri);

        //Creare TreeMap da numeri
        TreeMap<String, Integer> numeri1 = new TreeMap<>();
        numeri.put("Uno", 1);

        // uso di  putAll()
       numeri1.putAll(numeri);
        System.out.println("la Mappa numeri1: " + numeri1);
    }
}
```

output:

```
La mappa numeri: {Due=2, Quatto=4, Sei=6}
la Mappa numeri1: {Due=2, Quatto=4, Sei=6, Uno=1}
```

2 Accesso agli elementi

Set<Entry<k,V>> entrySet(): Restituisce un oggetto di tipo Set contenente le coppie chiave-valore presenti in questo oggetto di tipo Map.

Set<K> keySet() : Restituisce un insieme di tutte le chiavi di una mappa ad albero

Collection<V> values(): Restituisce una collection di tutti i valori

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {
        // Creare la mappa
        TreeMap<String, Integer> numeri = new TreeMap<>();

        // Uso put()
        numeri.put("Uno", 1);
       numeri.put("Quatto", 4);
       numeri.put("Due", 2);
        // uso putIfAbsent()
        numeri.putIfAbsent("Sei", 6);
    // Uso entrySet()
        System.out.println("Key/Value mappings: " +
numeri.entrySet());

        // uso keySet()
        System.out.println("Keys: " + numeri.keySet());

        // uso values()
        System.out.println("Values: " + numeri.values());
    }
}
```

Output:

```
Key/Value mappings: [Due=2, Quatto=4, Sei=6, Uno=1]
Keys: [Due, Quatto, Sei, Uno]
Values: [2, 4, 6, 1]
```

3 Rimuovere gli elementi

remove(key) - restituisce e rimuove la voce associata alla chiave specificata da una TreeMap

remove(key, value) - rimuove la voce dalla mappa solo se la chiave specificata è associata al valore specificato e restituisce un valore booleano

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {
        // Creare la mappa
        TreeMap<String, Integer> numeri = new TreeMap<>();

        // Uso put()
        numeri.put("Uno", 1);
       numeri.put("Quatto", 4);
       numeri.put("Due", 2);
        // uso putIfAbsent()
        numeri.putIfAbsent("Sei", 6);
        System.out.println("La mappa: " + numeri);
   // remove method
        int value = numeri.remove("Due");
        System.out.println("Rimosso value: " + value);

        // remove method con due parametri
        boolean result = numeri.remove("Quattro", 3);
        System.out.println("è stato rimosso {Quattro=3} ? " +
result);

        System.out.println("La nuova mappa: " + numeri);
    }
}
```

Output:

```
La mappa: {Due=2, Quatto=4, Sei=6, Uno=1}
Rimosso value: 2
è stato rimosso {Quattro=3} ? false
La nuova mappa: {Quatto=4, Sei=6, Uno=1}
```

4 Sostituisci gli elementi di TreeMap

replace(key, value) - sostituisce il valore mappato da quello specificato chiave con il nuovo valore

replace(key, old, new) - sostituisce il vecchio valore con il nuovo valore solo se il vecchio valore è già associato alla chiave specificata

replaceAll(function) - sostituisce ogni valore della mappa con il risultato di quello specificato funzione

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {
        // Creare la mappa
        TreeMap<String, Integer> numeri = new TreeMap<>();

        // Uso put()
        numeri.put("Uno", 1);
       numeri.put("Quatto", 4);
       numeri.put("Due", 2);
       numeri.put("Tre", 3);

        // uso putIfAbsent()
        numeri.putIfAbsent("Sei", 6);
        System.out.println("La mappa: " + numeri);

        // Uso di replace()
        numeri.replace("Tre",11);
        numeri.replace("Uno", 1, 15);
        System.out.println("Nuova mappa " + numeri);

        // Uso replaceAll()
        numeri.replaceAll((key, valore) -> valore%2);
        System.out.println("TreeMap dopo replaceAll: " + numeri);
    }

}
```

Output:

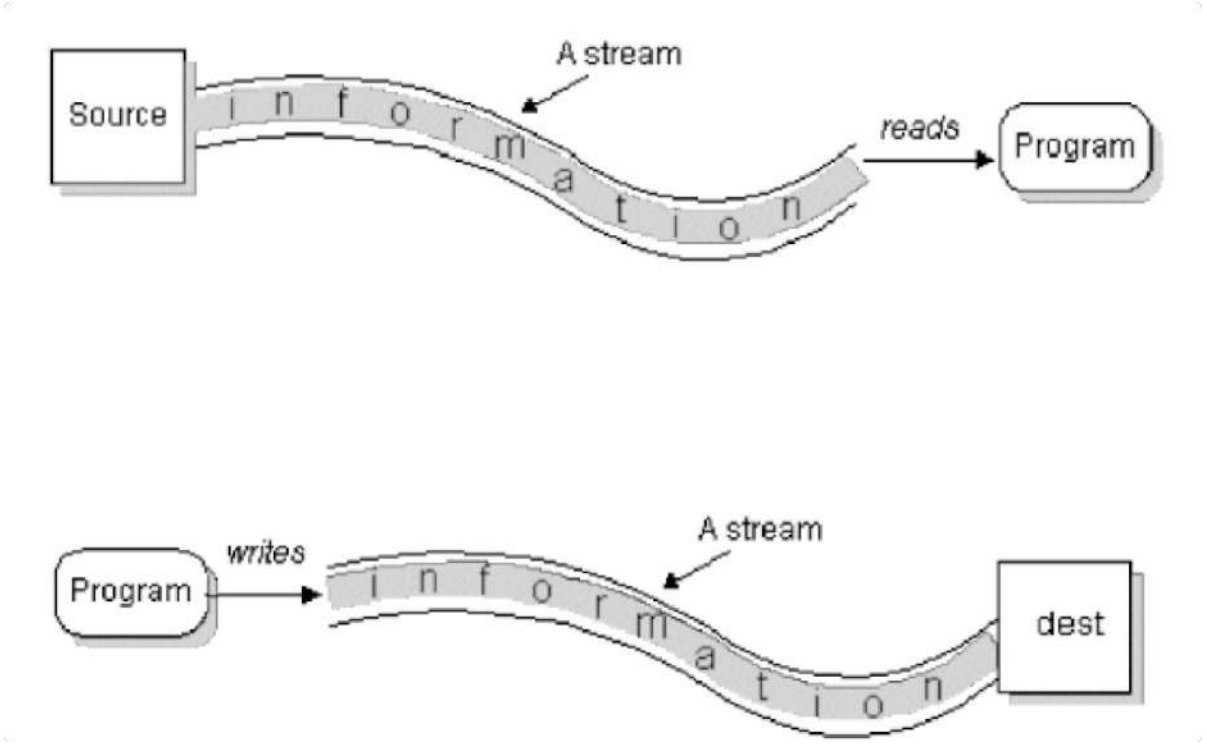
```
La mappa: {Due=2, Quatto=4, Sei=6, Tre=3, Uno=1}
Nuova mappa {Due=2, Quatto=4, Sei=6, Tre=11, Uno=15}
TreeMap dopo replaceAll: {Due=0, Quatto=0, Sei=0, Tre=1, Uno=1}
```

Poiché la classe TreeMap implementa NavigableMap, fornisce vari metodi per navigare sugli elementi della mappa.

Iinput-output da File

I programmi hanno bisogno di utilizzare informazioni lette da fonti esterne, o inviare informazioni a destinazioni esterne (file, dischi, reti, memorie o altri programmi).

In Java, l'I/0 è gestito in termini di flussi di dati. Un flusso di dati (generalmente indicato con il termine inglese stream) può essere costituito da caratteri, numeri o generici byte. Se i dati fluiscono nel programma, lo stream è detto stream di input. Se, al contrario, i dati fluiscono dal programma, lo stream è detto stream di output.



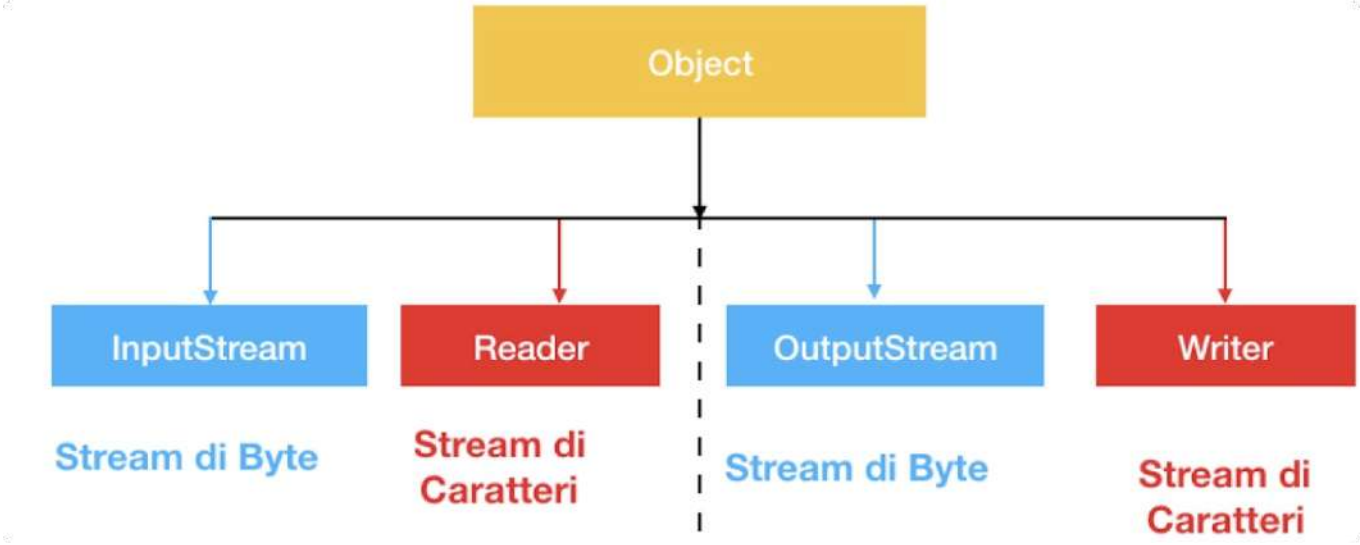
Gli stream sono realizzati come istanze di alcune classi. Gli oggetti di tipo **Scanner,** utilizzati per leggere dati da tastiera, sono degli stream di input. L'oggetto **System.out** è un esempio di stream di output.

Per prelevare informazioni da una fonte esterna (un file, una rete etc…), un programma deve aprire uno stream su essa e leggerne le informazioni in maniera sequenziale. Allo stesso modo un programma può inviare ad una destinazione esterna aprendo uno stream su essa e scrivendo le informazioni sequenzialmente.

Lettura	Scrittura
Apri lo stream *while* (ci sono ancora dati) leggi dato chiudi lo stream	Apri lo stream *while* (ci sono ancora dati) scrivi dato chiudi lo stream

Il **package java.io** contiene una collezione di classi che supportano tali algoritmi di I/O. Le classi di tipo stream sono divise in due gerarchie separate (anche se simili) in base al tipo di informazione che devono trasportare:

1. **1.classi basate sui byte**
2. **2.classi basate sui caratteri**



Il flusso di byte viene utilizzato per leggere e scrivere un singolo byte (8 bit) di dati.

Tutte le classi di flusso di byte derivano da classi astratte di base chiamate InputStreame OutputStream.

Il flusso di caratteri viene utilizzato per leggere e scrivere un singolo carattere di dati.

Tutte le classi del flusso di caratteri derivano da classi astratte di base Reader e Writer.

In questo capitolo verranno visti stream che consentiranno di collegare un programma a dei file, anziché a tastiera e schermo. Quindi prima di analizzare le classi di I/O vediamo come si gestisce un File in java.

Introduzione alla gestione dei file

Per lavorare con i file Java mette a disposizione diverse classi che consentono di:

1. creare file e directory

2. cercare file e directory

3. rinominare file e directory

4. cancellar e file e directory

5, scrive e all'interno di un file

6. leggere il testo contenuto in un file

Le classi principali si trovano nel package java.io e sono:

per la gestione :

- java.io.File

per la lettura :

- java.io.FileReader
- java.io.BufferedReader

per la scrittura :

- java.io.FileWriter
- java.io.BufferedWriter
- java.io.PrintWriter

La Classe File:

Questa classe permette di creare, eliminare, rinominare e cercare un file o una directory (e sottodirectory).

I costruttori della classe File sono i seguenti:

- File(String pathname)
- File(String dir, String subpath)
- File(File dir, String subpath)

Di seguito qualche esempio:

- File dir = new File("/usr" "local"): //istanzia di una directory e un file su un sistema Unix
- File file = new File(dir, "Abc.java");//istanzia di una directory e un file su un sistema Windows
- File dir2 = new File("C:directory"):

• File file2 = new File(dir2. "Abc.iava"):

È possibile utilizzare il separatore per il path dei file per i vari sistemi operativi in maniera dipendente, ma anche utilizzare come separatore "/" pure su sistemi Windows. La scelta migliore è utilizzare la costante statica della classe File (dipendente dal sistema operativo):

File.pathSeparator

che vale:

"'\\" per Windows "/" e Unix.

Per esempio:

```
File file = new File("'." + File.pathSeparator + "abc.iava")
```

Istanziare un file non significa però crearlo fisicamente sul file system; per farlo è necessario utilizzare gli stream.

I metodi principali sono:

Metodo	Descrizione
exists()	Restituisce il valore *true* se il file esiste, altrimenti *false*.
isFile()	Restituisce il valore *true* se il file è un file, altrimenti *false*.
isDirectory()	Restituisce il valore *true* se il file è una directory, altrimenti *false*.
getName()	Restituisce una stringa con il nome del file o della directory.
getParent()	Restituisce una stringa con il nome della directory padre del file.
length()	Restituisce la dimensione del file in byte.
lastModified()	Restituisce il timestamp dell'ultima modifica al file.
canRead()	Restituisce il valore *true* se il file può essere letto, altrimenti *false*.
canWrite()	Restituisce il valore *true* se il file può essere scritto, altrimenti *false*.

```
import java.io.*;
import java.util.Scanner;
public class file{

public static void main(String arg[]){
    System.out.println("\n\n\n Immetti il nome del file
cercato:");
    Scanner tastiera=new Scanner(System.in);
String nome=tastiera.nextLine() ;
File f=new File(nome); // (4)
if (!f.exists()) System.out.println("NON ESISTE il file
"+nome); // (5)
else {
long dim=f.length(); // (6)
System.out.print("Il file: "+nome+" e' lungo:"+dim+"
byte.");
if (f.canWrite()) // (7)
System.out.println(" E' di lettura e scrittura.");
else System.out.println(" E' di sola lettura.");
}
}
}
```

L'oggetto di tipo File può essere utilizzato anche per riferirsi alle directory. Per esempio:

File directory = new File("c: \\windows");

Con le directory, oltre ai metodi elencati precedentemente, si possono utilizzare anche i seguenti:

Metodo	Descrizione
list()	Restituisce un array di *String* con l'elenco dei file e delle sottodirectory contenuti nella directory.
listFile()	Restituisce un array di *File* con l'elenco dei file e delle sottodirectory contenuti nella directory.
mkdir()	Crea la directory il cui nome è stato indicato nel costruttore della classe *File*.

Ottenere l'elenco di directory

Per visualizzare il contenuto di una directory si può utilizzare l'oggetto File in due modi diversi:

1. chiamando il metodo list() senza argomenti per ottenere il contenuto completo dell'oggetto File.

i

```
mport java.io.*;
public class directory{
public static void main(String arg[]){
File f=new File(".");
File a[]=f.listFiles();
for (int i=0; i<a.length;i++){
  String s=a[i].toString();
  System.out.println(a[i].getName());
}
}
}
```

2) usare list(FilenameFIter a) che accetta il parametro java.io.FilenameFilter e consente di filtrare i file e le cartelle per visualizzare un elenco limitato, per esempio dei soli file con estensione class.

i

```
mport java.util.regex.*;
import java.io.*;
import java.util.*;
public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if (args.length == 0) {
            list = path.list();
        } else {
            list = path.list(new DirFilter(args[0]));
        }
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for (String dirItem : list) {
            System.out.println(dirItem);
}
}
}
class DirFilter implements FilenameFilter {
    private Pattern pattern;
    public DirFilter(String regex) {
        pattern = Pattern.compile(regex);
    }
    public boolean accept(File dir, String name) {
        return pattern.matcher(name).find();
    }
}
```

I metodi per modificare i file:

Metodo	Descrizione
delete()	Cancella il file.
setReadOnly()	Imposta l'attributo di sola lettura al file.
renameTo(File)	Rinomina il file con il nome indicato nell'oggetto *File* passato come parametro.

Istanziare un file non significa però crearlo fisicamente sul file system; per farlo è necessario utilizzare gli stream.

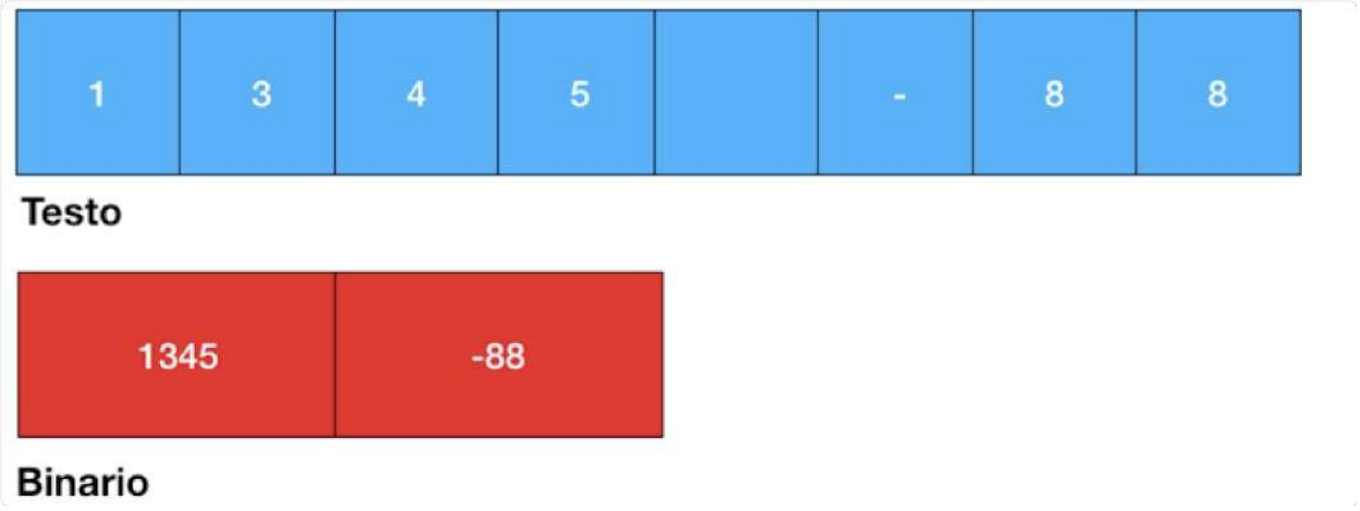
File di testo e file binari

I file trattati da Java possono essere classificati in due categorie:

• file di testo

• binario

Ognuno dei due tipi di file ha i propri stream e metodi per elaborarli. Il tipo di file, determina quali classi debbano essere utilizzate per l'input e per l'output. Il vantaggio principale dei file di testo è che è possibile crearli, visualizzarli e modificarli utilizzando un editor di testi. Per un file binario, le operazioni di lettura e scrittura devono generalmente essere eseguite da un programma apposito. In un file di testo ogni carattere è rappresentato per mezzo di uno o due bvte. a seconda che il sistema utilizzi la codifica ASCII o Unicode. Quando un programma scrive un valore in un file di testo, il numero di caratteri che vengono scritti è lo stesso che si avrebbe scrivendo lo stesso valore su schermo per mezzo del metodo System.out. println. Per esempio, la scrittura in un file di testo del numero 1345 comporta la scrittura di quattro caratteri nel file, come mostrato nella Figura.



I file binari immagazzinano tutti i valori dello stesso tipo primitivo nello stesso formato. Ogni valore è quindi salvato come sequenza dello stesso numero di byte. Per esempio, i valori di tipo int occupano ognuno quattro byte. Un programma Java interpreta questi byte in modo molto simile a quanto fa con i dati nella memoria principale. E per questo motivo che la gestione dei file binari è molto efficiente.

File di testo

Vediamo le operazioni di I/O sui file di testo.

Le 4 classi base astratte forniscono dei metodi generici per leggere e scrivere "flussi" di dati dall'input e verso l'output.

Le classi derivate si dividono in due categorie, specializzate in due sensi:

1. classi (dette sorgenti) che, senza aggiungere funzionalità, specializzano le classi astratte rispetto alla sorgente/destinazione destinazione dei flussi.

o Per l'input. il flusso può diventare un file o un buffer:

o Per l'output, il flusso può diventare un file o un buffer;

2. classi (dette di filtraggio) che, non preoccupandosi della sorgente/destinazione dei flussi, specializzano e aumentano le funzionalità delle classi astratte per fare in modo di poter leggere/scrivere non più soltanto stram di byte o di caratteri, ma dati strutturati, quali:

- i tipi primitivi di Java;

- interi oggetti;

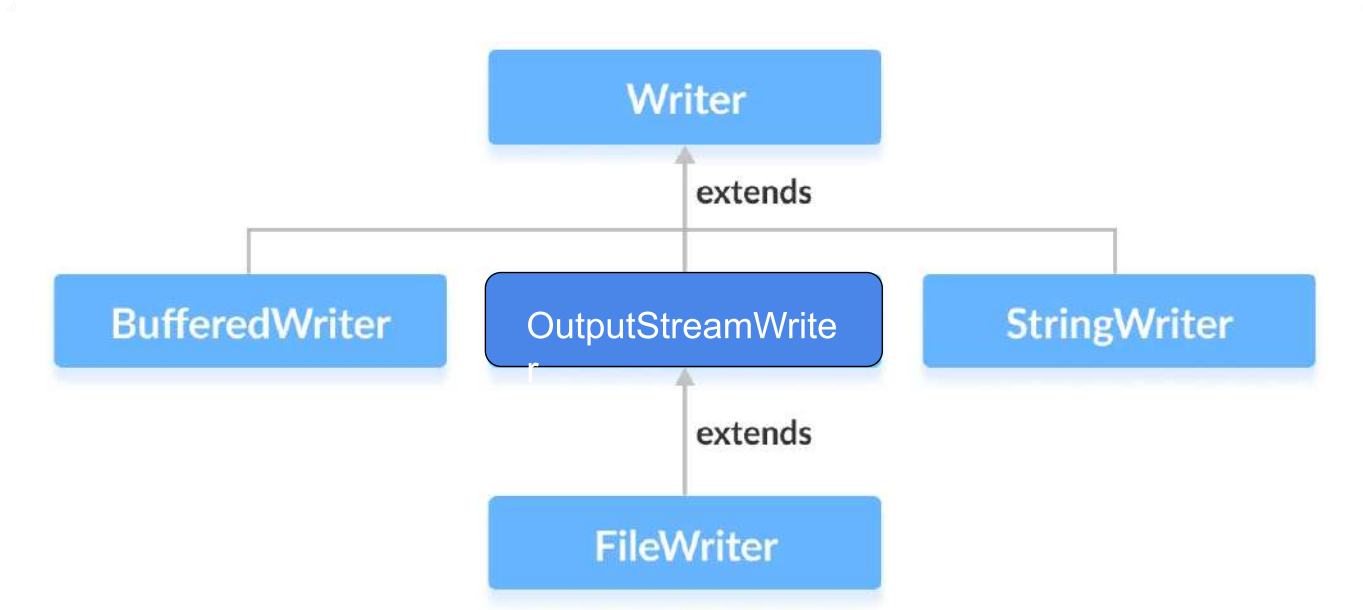
inoltre, ci sono classi che specializzano le funzionalità permettendo forme complesse di filtraggio ed elaborazione dei dati.

Per usare l'I/O, si tratta di "comporre" le classi concrete che ci interessano in modo da avere un oggetto che presenti tutte le funzionalità richieste. Si utilizza il metodo dell'incapsulamento:

- si crea un oggetto da una classe sorgente, per definire il flusso specifico dei dati;

- si crea un oggetto da una classe di filtraggio concreta del secondo tipo, e si passa al costruttore l'oggetto stream prima creato (come per incapsularlo dentro)
- si possono poi eseguire ulteriori incapsulamenti della classe di filtraggio in un'altra classe di filtraggio al fine di ottenere, tramite successivi incapsulamenti, tutte le funzionalità previste.

Scrittura di un File di Testo



java.io.FileWriter

La classe FileWriter permette di scrivere i caratteri in un file di testo.

Costruttori

public **FileWriter(String fileName):** Crea un oggetto FileWriter a cui viene assegnato un nome file se il file esiste viene eliminato.

public **FileWriter(String fileName, boolean append)**: Crea un oggetto FileWriter a cui viene assegnato un nome file con un valore booleano che indica se è true, i dati verranno scritti alla fine del file anziché all'inizio.

public **FileWriter(File file):** Crea un oggetto FileWriter dato un oggetto File se il file esiste viene eliminato.

public **FileWriter(File file, boolean append):** Crea un oggetto FileWriter dato un oggetto File. Se il secondo argomento è true, i byte verranno scritti alla fine del file anziché all'inizio.

I costruttori generano una IOException -se il file indicato esiste ma è una directory anziché un file normale, non esiste e non può essere creato o non può essere aperto per nessun altro motivo

Metodi

public **Writer append(char c):** inserisce il carattere specificato nello stream corrente

public **void flush():** forza a scrivere tutti i dati presenti nello stream alla destinazione corrispondente

public **void close()** - chiude lo stream

public **void write(int a):** Scrive un singolo carattere. Il carattere da scrivere è contenuto nei 16 bit di ordine inferiore del valore intero dato; i 16 bit di ordine superiore vengono ignorati.

public **void write(char c[])** //scrive un arrayd i caratteri

public **void write(String str):** scrive una stringa

public **void write(char c[],int offset, int length)** //scrive lenght caratteri di c[] iniziando da offeset

public **void write(string str, int offset, int length)**

Lanciano una IndexOutOfBoundsException- Se offset è negativo, o length è negativo, o offset+length è negativo o maggiore della lunghezza della stringa o array dato.

Tutti i metodi lanciano una IOException - Se si verifica un errore di I/O

Per aprire un file di testo, tipicamente si crea un oggetto di tipo FileWriter

FileWriter f= new FileWriter ("prova.txt"): crea uno stream f per scrivere sul file prova.txt.

L'istruzione crea uno stream f per scrivere sul file prova.txt. Rappresentando graficamente gli stream si ottiene la seguente figura:

Se il file prova.txt esiste viene sovrascritto e viene cancellato tutto il suo contenuto.

Se si vuole aprire un file esistente, per accodare dei valori, si deve sostituire la creazione del FileWriter nel seguente modo:

FileWriter file = new FileWriter("prova.txt", true): Il valore del secondo parametro indica la condizione di accodamento (append), se true il file viene aperto per aggiungere i dati in coda a quelli preesistenti. Altrimenti se false, il file viene aperto in scrittura e ciò comporta la cancellazione di un eventuale archivio preesistente.

Esempio:

```
import java.io.FileWriter;
import java.io.Writer;
```

```java
public class Main {
public static void main(String args[]) {
String data = "Ciao Mondo";
File f = new File("prova.txt");
  try {
            // Crea Writer usando FileWriter
      Writer output = new FileWriter(f);
// Writes string to the file
       output.write(data);
       output.write("\n");//a capo
        output.write(49);// scrive il carattere 1
       output.append('0');//aggiunge il carattere 0
            // chiusura di writer
            output.close();
        }

        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output nel file prova.txt

Lo stream precedente si può migliorare incapsulano lo stream tramite le classi:

BufferedWriter: Che è simile alla classe FileWriter ma permette di scrivere i caratteri nel file in blocchi. I caratteri vengono memorizzati temporaneamente in un buffer temporaneo. Periodicamente i caratteri vengono letti dal buffer e scritti fisicamente sul file, quindi, le prestazioni migliorano notevolmente.

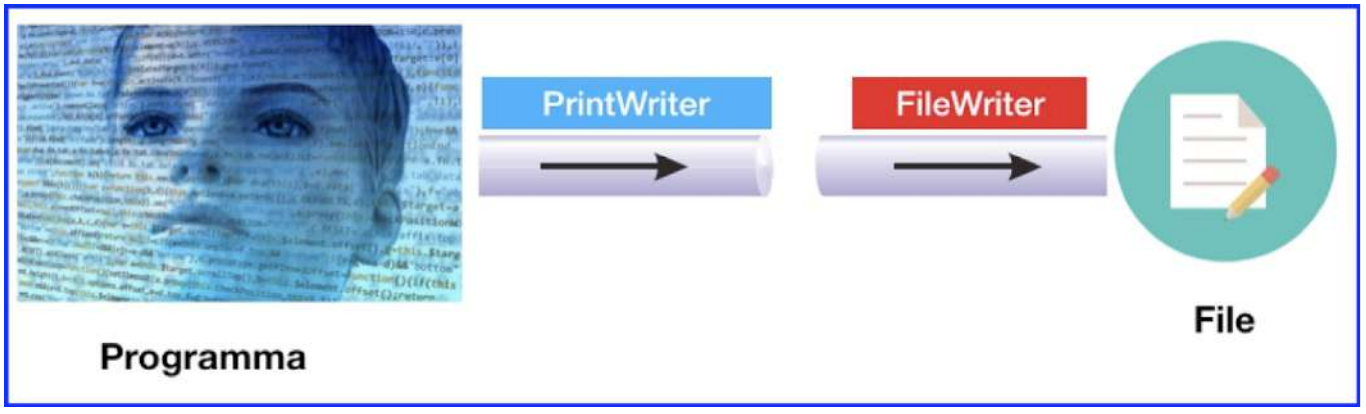
PrintWriter è simile alla classe BufferedWriter ma permette di scrivere nel file stringhe formattate.

In questo caso per aprire un file di testo, si crea un oggetto di tipo FileWriter

FileWriter f= new FileWriter ("agenda.txt"):// crea uno stream f per scrivere sul file agenda.txt.

A sua volta, FileWriter è incapsulato in un oggetto di tipo PrintWriter. Quindi:

PrintWriter fOUT = new PrintWriter(f)://si scrive su questo



Della classe PrintWriter si possono utilizzare i metodi:

print(). printIn0) e printf() per scrivere caratteri in un file aperto con successo.

Esempio:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class Main {
 public static void main(String[] args) {
    File f = new File("agenda.txt");
        try {
         FileWriter out=new FileWriter(f);
         PrintWriter output=new PrintWriter(out);
            output.println("Rossi Mario 03456722122");
```

```
            output.println("Bianchi Marita 03224567");
            output.println("Fantozzi Ugo 0654346744");
            output.println("Silvani Anna 05466824567");
            output.flush();
            output.close();
} catch (IOException ex) {
  System.out.println("errore nella scrittura del file");
        }
    }
}
```

Output:

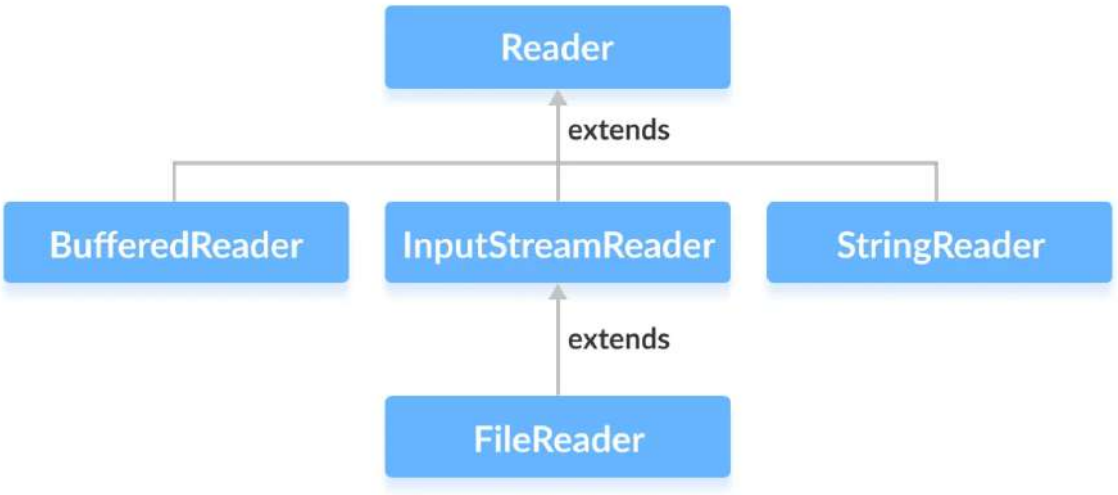
Inoltre questa classe contiene il costruttore:

PrintWriter(File file) che permette di scrivere su un file senza incapsulare un'altro stream.

Lettura di un file di testo

La classe Reader del pacchetto java.io è una superclasse astratta che rappresenta un flusso di caratteri.

Poiché Reader è una classe astratta, non è utile di per sé. Tuttavia, le sue sottoclassi possono essere utilizzate per leggere i dati.



La classe FileReader

L'apertura di un file di testo per le operazioni d'input, viene eseguita con la dichiarazione dell'oggetto FileReader.

Costruttori

FileReader(File file): Crea un nuovo FileReader , dato il File da cui leggere.

FileReader(FileDescriptor fd): Crea un nuovo FileReader , data la FileDescriptor da cui leggere.

FileReader(String fileName): Crea un nuovo FileReader , dato il nome del file da cui leggere.

I costruttori lanciano un eccezione FileNotFoundException - se il file indicato non esiste, è una directory piuttosto che un file normale, o per qualche altro motivo non può essere aperto per la lettura.

Metodi di lettura

boolean ready() - verifica se lo stream è pronto per essere letto

int read(char[] array) - legge i caratteri dal flusso e li memorizza nell'array specificato

int read(char[] array, int start, int length) - legge il numero di caratteri pari a lunghezza dal flusso e archivia nell'array specificato a partire da inizio

void mark(int a) - segna la posizione nel flusso fino a cui sono stati letti i dati

void reset() - riporta il controllo al punto del flusso in cui è impostato il contrassegno

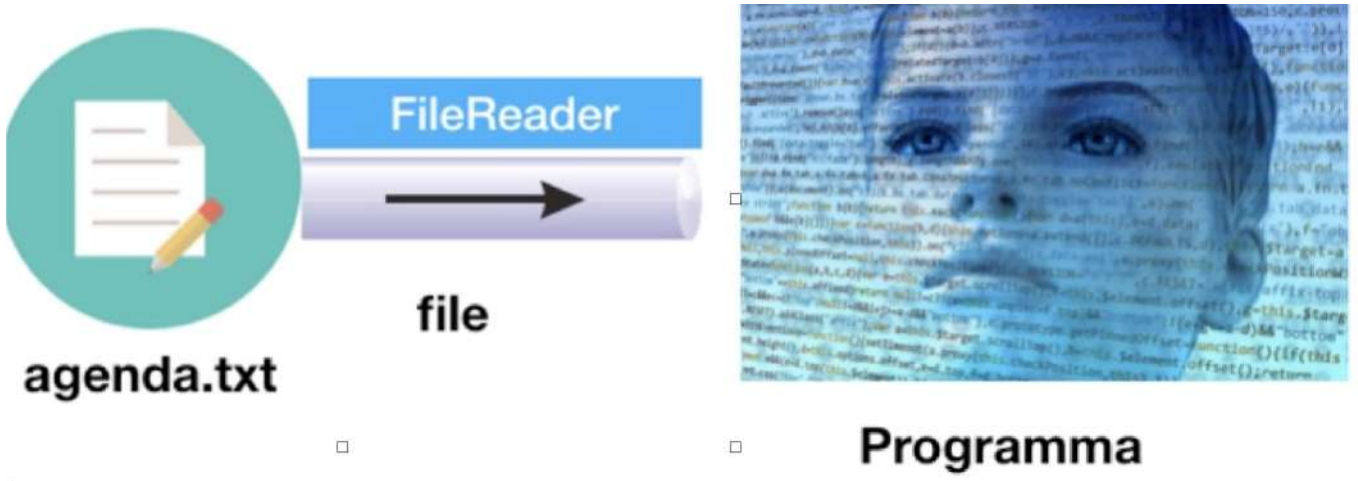
long skip(long n) - elimina il numero di caratteri specificato dallo stream

I metodi lanciano una IOException - se si verifica un errore di I/O

L'istruzione:

Reader file = new FileReader("agenda.txt");

crea un flusso dal file verso il programma.



Esempio:

```
public class Main {
public static void main(String[] args) {
  File f = new File("agenda.txt");
    try {
       char[] a = new char[48];
       Reader input = new FileReader(f);
       input.read(a);
       System.out.print(a);
       char car = ' ';
       int i;
       while ((i = input.read()) != -1) {
            System.out.print((char) i);
            }
```

```
      } catch (IOException ex) {
       System.out.println("errore di I/O");
       }
    }
}
```

Output:

```
Rossi Mario 03456722122
Bianchi Marita 03224567
Fantozzi Ugo 0654346744
Silvani Anna 05466824567
```

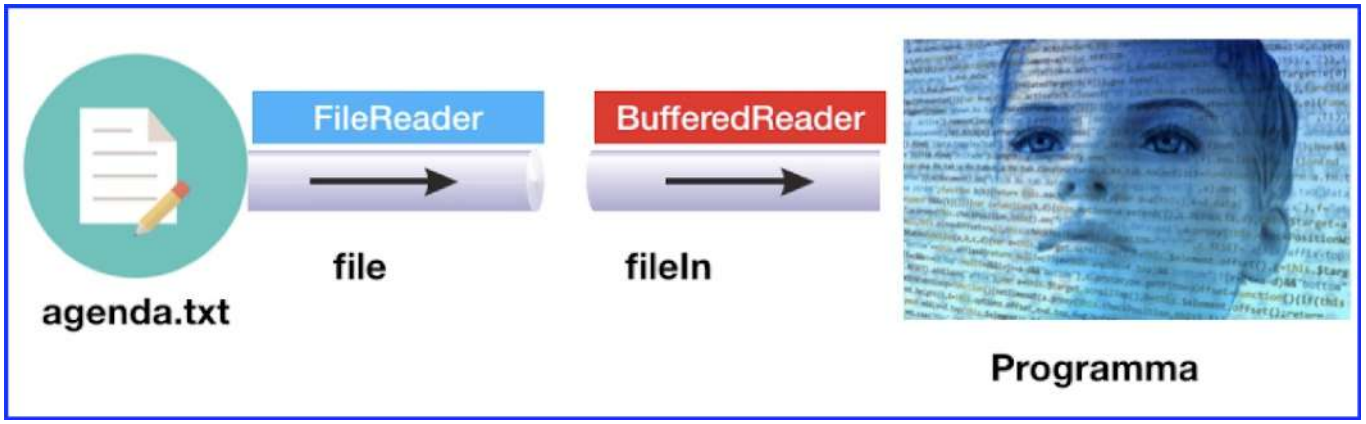
Per migliorare le prestazioni e agevolare la lettura di Stringhe si decorala lo stream FileReadere tramite la classe BufferedReader.

FileReader file = new FileInputStream("agenda.txt");

BufferedReader fileIn = new BufferedReader(file);

La prima riga crea uno stream file per leggere dal file agenda.txt. Le operazioni di lettura non vengono fatte direttamente su questo stream, ma sul secondo stream creato dalla classe BufferedReader, che contiene i metodi per la lettura dei dati memorizzati.

Rappresentando graficamente gli stream si ottiene la seguente figura:



Per leggere le informazioni contenute in un file di testo si usano due metodi della classe BufferedReader:

o read(): legge un singolo carattere, ma come valore di ritorno restituisce un intero (-1 quando viene raggiunta la fine del file). Per ottenere il carattere letto si deve effettuare il casting: char c = (char) fileln.readO;

o readLine(): legge una riga di testo e come valore di ritorno restituisce una stringa. Se è stata raggiunta la fine del file il metodo restituisce un valore null.

La chiusura di uno stream, sia di input che di output, viene fatta richiamando il metodo close nel seguente modo:

f.close();

```
public class Main {

    public static void main(String[] args) {

  File f = new File("agenda.txt");
  try {
       FileReader in = new FileReader(f);
     BufferedReader input=new BufferedReader(in);
       String s=null;
     while ((s = input.readLine()) != null) {
            System.out.println(s);
            }

        } catch (IOException ex) {
         System.out.println("errore di I/O");
        }
    }

}
```

Codice di Esempio di un agenda

Un altra possibilità per leggere in un file di testo è usare la classe Scanner passando come argomento il file da leggere.

Scanner leggi=new Scanner(new File("agenda.txt"));

Nuovo I/O – Lettura di testo

Per compiere operazioni sui file la classe principale di riferimento si chiama Files localizzata nel package java.nio.file. Questa classe offre un ricco set di metodi statici (oltre 50 escludendo quelli ereditati da Object) per leggere, scrivere e manipolare file e directory.

Metodo	Descrizione
public static byte[] readAllBytes(Path path)	legge tutti i byte del file indicato dal parametro path e li ritorna in un array di tipo byte.
public static List<String> readAllLines(Path path, Charset cs)	legge tutte le righe di testo dal file indicato dal parametro path e le ritorna come un oggetto di tipo List. I byte sono decodificati nei corrispondenti caratteri rispetto al charset indicato dal parametro cs.
public static Path write(Path path, byte[] bytes, OpenOption... options)	scrive i byte indicati dal parametro byte nel file indicato dal parametro path. È possibile passare come ultimo parametro una serie di oggetti che implementano l'interfaccia OpenOption (come l'enumerazione StandardOpenOption), che consentono di indicare parametri per l'apertura o la creazione della risorsa (CREATE, READ, WRITE,APPEND e così via).
public static Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)	scrive la sequenza di righe indicate dal parametro lines nel file indicato dal parametro path, codificando i caratteri in byte secondo il charset del parametro cs. È
public static BufferedReader newBufferedReader(Path path, Charset cs)	apre per la lettura il file indicato dal parametro path decodificandone i byte nei relativi caratteri come indicato dal parametro cs. Ritorna un oggetto di tipo BuffereReader che può essere utilizzato per la lettura efficiente del file corrispondente.
public static BufferedWriter newBufferedWriter(Path path, Charset cs, OpenOption... options)	apre o crea per la scrittura il file indicato dal parametro path codificandone i caratteri nei bytecome indicato dal parametro cs e utilizzando il parametro options per fornire una serie di parametri per l'apertura o la creazione della risorsa. Ritorna un oggetto di tipo BufferedWriter con cui scrivere in modo efficiente nel relativo file.

Il metodo readAllLines() che utilizza la codifica dei caratteri predefinita è stato introdotto in jdk1.8 permette di leggere un file di testo in un'unica istruzione.

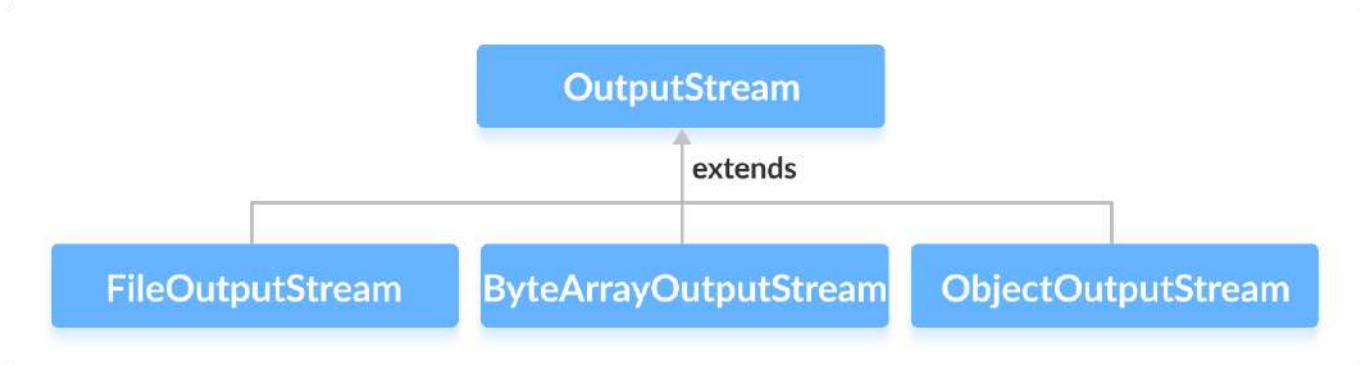
```
public class Main {
 public static void main(String[] args) {
File f=new File("agenda.txt");
```

```
try {
List<String> lista=  Files.readAllLines(f.toPath());
      for(String s: lista)
                System.out.println(s);


        } catch (IOException ex) {
              System.out.println("Errore i/o");
        }
    }

}
```

Codice agenda nio

File binario

Per scrivere su un file binario si fa riferimento alla classe **OutputStream** del pacchetto java.io che è una super classe astratta che rappresenta un flusso di output di byte.



Poiché OutputStreamè una classe astratta, non è utile di per sé. Tuttavia, le sue sottoclassi possono essere utilizzate per scrivere dati.

La classe:

FileOutputStream

permette di scrivere in un file di byte.

Costruttori

FileOutputStream (File f): Crea uno stream di output per scrivere nel file rappresentato dall'oggetto File specificato.

FileOutputStream (String nome): Crea uno stream di output per scrivere nel file col nome specificato.

FileOutputStream (FileDescriptor fd): Crea uno stream di output per scrivere nel file specificato.

FileOutputStream (File f, boolean append): Crea uno stream di output per scrivere nel file rappresentato dall'oggetto File specificato.

FileOutputStream (String nome, boolean append): Crea uno stream di output per scrivere nel file rcol nome specificato.

append- se true, i byte verranno scritti alla fine del file anziché all'inizio se è false il file esistente viene eliminato

I costruttori lanciano una FileNotFoundException - se il file esiste ma è una directory anziché un file normale, non esiste ma non può essere creato o non può essere aperto per nessun altro motivo.

Metodi

void close() Chiude lo Stream. IOException - se si verifica un errore di I/O

void write(byte[] b) Scrive l'array di byte specificata. IOException - se si verifica un errore di I/O. IOException - se si verifica un errore di I/O

void write(byte[] b, int off, int len) Scrive i len byte dall'array di byte specificato a partire dall'offset off.

void write(int b) Scrive il byte specificato. IOException - se si verifica un errore di I/O

L'apertura di un file strutturato per le operazioni di output, viene eseguita con le dichiarazioni dei seguenti oggetti:

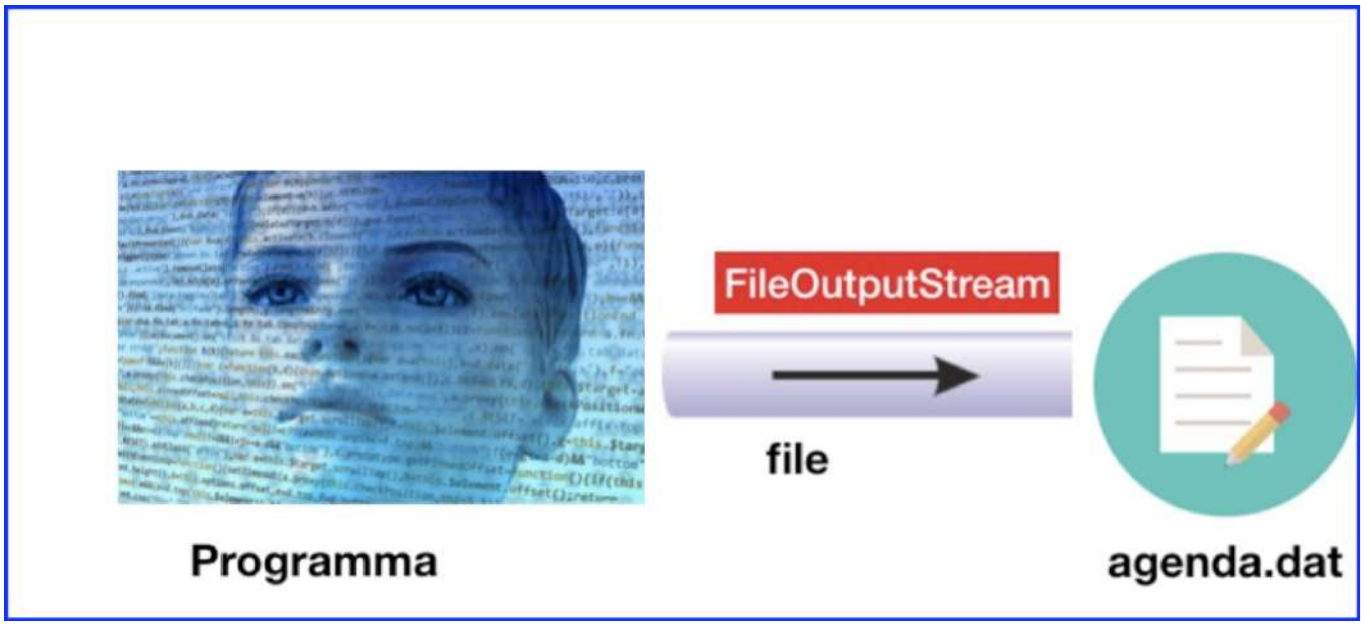
FileOutputStream file = new FileOutputStream ("agenda.dat");

comandi, se il file esiste viene sovrascritto e viene cancellato tutto il suo contenuto. Se si vuole aprire un file esistente, per accodare dei valori, si deve sostituire la creazione del FileOutputStream nel seguente modo:

FileOutputStream f = new FileOutputStream("agenda.dat", true);

Il valore del secondo parametro indica la condizione di accodamento (append):

- true il file viene aperto per aggiungere i dati in coda a quelli preesistenti.
- false (o non presente): il file viene aperto in scrittura e ciò comporta la cancellazione di un eventuale archivio preesistente.




```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;

public class Main {

    public static void main(String[] args) {
```

```
        String data = "ciao mondo";

 try {
  OutputStream out = new FileOutputStream("prova.dat");
 // Convert la stringa in un array di bytes
    byte[] dataBytes = data.getBytes();
//scrittura nello  stream
    out.write(dataBytes);
//scrittura del carattere 1
     out.write(49);
// chiusura dello  stream
      out.close();
    } catch (IOException ex) {
            System.out.println("Errore i/o");
        }
    }
}
```

Per migliorare le prestazioni e permettere di scrivere interi oggetti e dati primitivi si incapsula l'oggetto FileOutputStream usando la classe:

ObjectOutputStream

Costruttore

public ObjectOutputStream (OutputStream out) Crea un ObjectOutputStream che scrive nell'OutputStream specificato. Lancia:

- IOException - se si verifica un errore di I/O durante la scrittura
- SecurityException - se la sottoclasse non attendibile sovrascrive illegalmente i metodi sensibili alla sicurezza
- NullPointerException- se lo out è null

Si possono utilizzare diversi metodi per la scrittura. Ogni metodo memorizza su file un particolare tipo di dato e assume la forma

writeDato

.Al posto di Dato si sostituisce il tipo di dato che si vuole memorizzare. Per esempio:

fileOut.writeInt(25400);

fileOut.writeDouble(12.36);

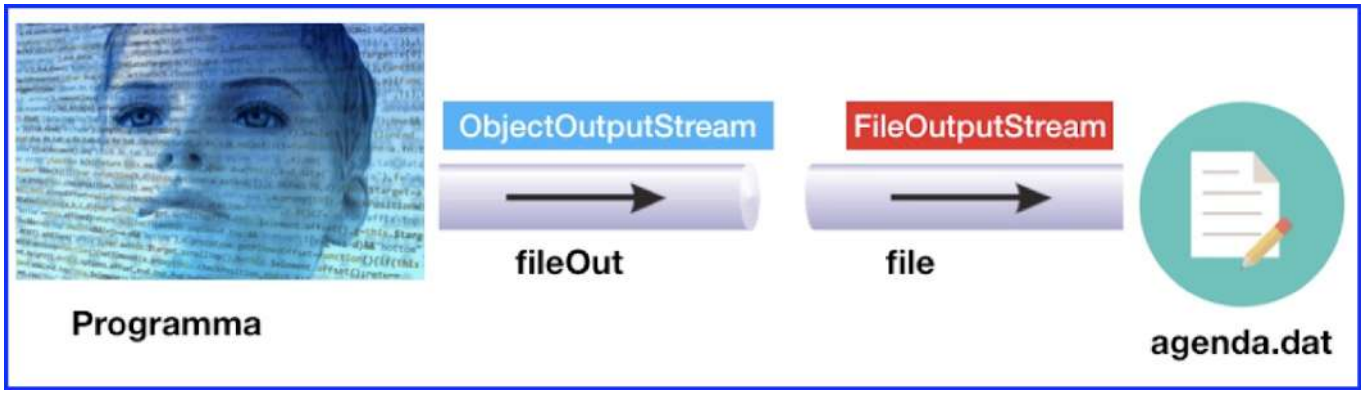
Prima di chiudere il file è importante eseguire **il metodo flush,** che serve per scrivere su disco tutti i dati che sono attualmente contenuti nel buffer dello stream. Il metodo di scrittura più importante è

writeObject

Con questo metodo è possibile salvare su di un file anche gli oggetti memorizzando tutti i suoi attributi non statici. Una classe, le cui istanze si vogliono rendere persistenti, deve implementare l'interfaccia Serializable.

FileOutputStream file = new FileOutputStream ("elenco.dat");

ObjectOutputStream fileOut = new ObjectOutputStream(file);




```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;

public class Main {
```

```
    public static void main(String[] args) {
        String data = "ciao mondo";

        try {
            OutputStream f = new
FileOutputStream("prova.dat");
            ObjectOutputStream out = new
ObjectOutputStream(f);

            //scritturadi una stringa
            out.writeUTF(data);
            //scrittura di un intero
            out.writeInt(49);
             //scrittura di un Double
            out.writeDouble(49.54);
            out.flush();
            // chiusura dello  stream
            out.close();
        } catch (IOException ex) {
            System.out.println("Errore i/o");
        }
    }
}
```

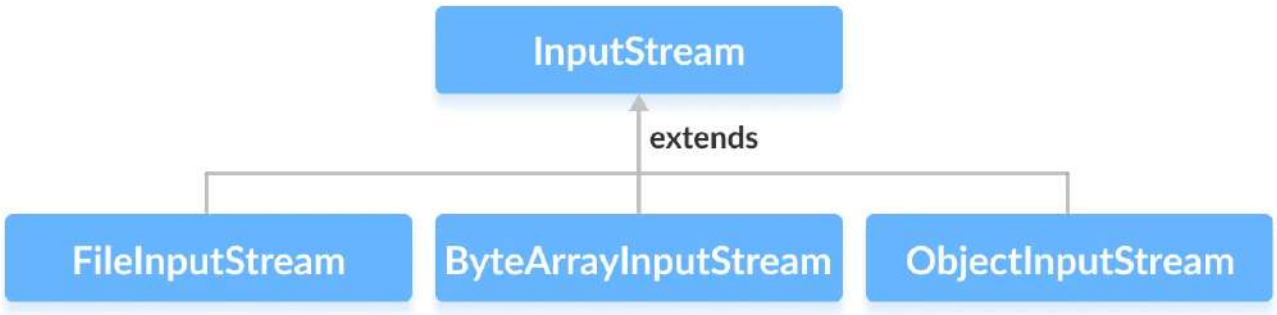
Input File Binario

Per aprire uno stream di byte in input si fa uso della classe

InputStream

del pacchetto java.io.

Poiché InputStream è una classe astratta, non è utile di per sé. Tuttavia, le sue sottoclassi possono essere utilizzate per leggere i dati.



Per leggere da un file si utilizza la classe

FileInputStream

Costruttori

FileInputStream(File file) Crea uno Stream sul file effettivo, indicato dall'oggetto File passato come parametro.

FileInputStream(FileDescriptor fdObj) Crea uno Stream utilizzando il descrittore di file fdOb.

FileInputStream(String nome) Crea uno Stream sul file dal nome.

Viene lanciata una FileNotFoundException - se il file non esiste, è una directory piuttosto che un file normale, o per qualche altro motivo non può essere aperto in lettura.

Metodi

int available() Restituisce una stima del numero di byte rimanenti che possono essere letti (o ignorati) nello stream di input senza essere bloccati dalla chiamata successiva di un metodo.

void close() Chiude lo stream.

int read() Legge un byte di dati dallo stream. Ritorna: il byte di dati o -1se viene raggiunta la fine del file.

int read(byte[] b) Legge un'array di byte dallo stream. Ritorna: il numero totale di byte letti nel buffer, o -1se non ci sono più dati perché è stata raggiunta la fine del file

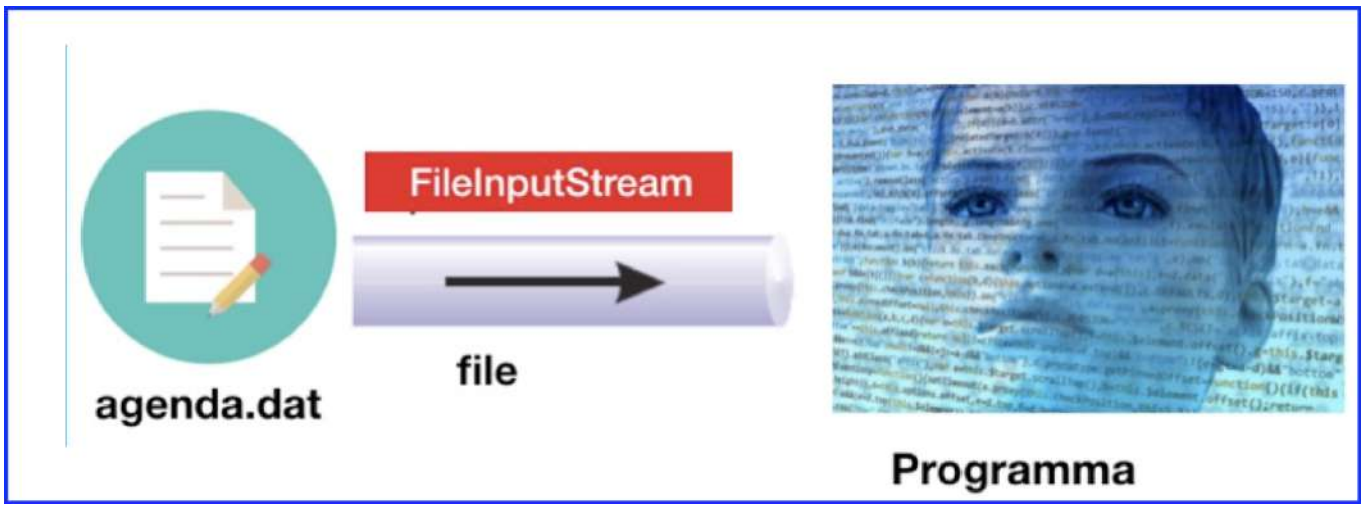
int read(byte[] b, int offset, int len) Legge fino a len byte di dati partendo dall'offset. Ritorna: il numero totale di byte letti nel buffer, o -1se non ci sono più dati perché è stata raggiunta la fine del file

long skip(long n) Salta ed elimina n byte di dati dallo stream. Ritorna:il numero effettivo di byte ignorati.

Viene Lanciata una IOException - se si verifica un errore di I/O.

L'apertura di un file strutturato per le operazioni di input, viene eseguita con le dichiarazioni dell'oggetto:

FileInputStream in = new FileInputStream ("elenco.dat");




```
public class Main {
 public static void main(String[] args) {
try {
  InputStream input = new FileInputStream("prova.dat");
      // Read byte from the input stream
     int i;
      while((i=input.read())!=-1)
      System.out.print((char)i);
```

```
        // Close the input stream
        input.close();
    } catch (IOException ex) {
            System.out.println("Errore i/o");
        }
    }
}
```

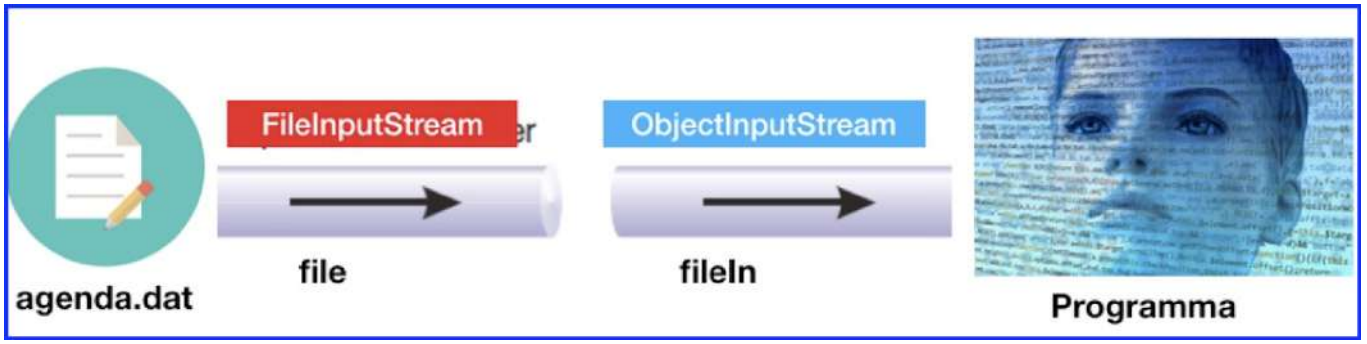
Questa classe può essere usata insieme alla classe FileOutputStream per fare delle copie di un file.

Per migliorare le prestazioni e la semplicità del codice si incapsula l'oggetto FileInputStream tramite la classe

ObjectInputStream

FileInputStream file = new FileInputStream ("elenco.dat");

ObjectInputStream fileIn = new ObjectInputStream(file);



Per leggere le informazioni contenute in un file strutturato, si usano i metodi della classe ObjectInputStream. Questi metodi assumono la forma

readDato

dove al posto di Dato si sostituisce il tipo di dato che si vuole leggere. Per evitare di ottenere inconsistenza, i dati devono essere letti nello stesso ordine in cui sono stati salvati.

Tra i metodi di lettura c'è il metodo readObject che consente di recuperare un oggetto precedentemente salvato.

Questo metodo restituisce un oggetto di classe Object che attraverso il casting può essere riportato alla sua classe originaria. La lettura di un oggetto comporta la creazione di una nuova istanza della classe, in cui a ogni attributo viene assegnato il valore letto dal file.

Se non si conosce quanti sono i dati contenuti nel file, si può usare un ciclo infinito per continuare a leggere finché viene generata l'eccezione EOFException. Questa eccezione segnala che si è raggiunta la fine del file e non ci sono più dati da leggere; si può quindi interrompere.

```
public class Main {

    public static void main(String[] args) {
 try {
 InputStream in = new FileInputStream("prova.dat");
ObjectInputStream input= new ObjectInputStream(in);
      String s=input.readUTF();
      System.out.println(s);
       int i=input.readInt();
      System.out.println(i);
      double d=input.readDouble();
      System.out.println(d);

      // Close the input stream
      input.close();
    } catch (IOException ex) {
            System.out.println("Errore i/o");
        }
    }
}
```

Output:

```
ciao mondo
```

49

49.54

Codice agenda

Streami e file ad accesso diretto

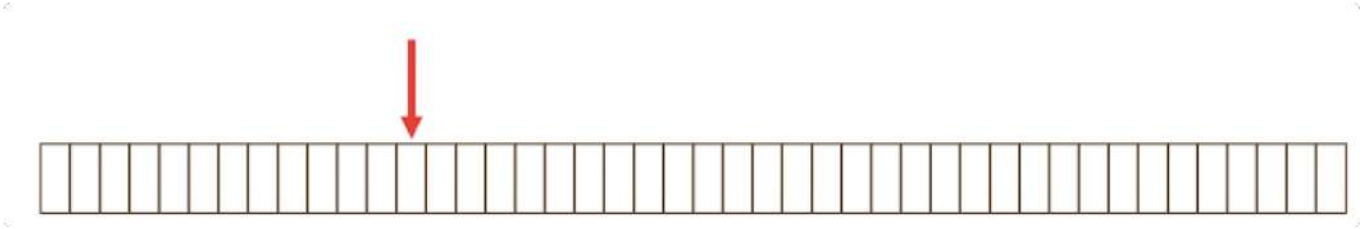
I file ad accesso diretto sono modellati da oggetti della classe

RandomAccessFile

In tali file è possibile scrivere o leggere dati in corrispondenza di specifiche posizioni del supporto. La classe implementa i metodi delle interfacce

DataInput e DataOutput

Il file viene visto come sequenza di byte, con un indice (file pointer) che identifica la posizione per la successiva operazione di I/O. Dopo una operazione di I/O, la posizione del file pointer viene aggiornata.



Costruttori

o RandomAccessFile (String name. String mode)

o RandomAccessFile (File file, String mode)

Il parametro mode stabilisce se il file va aperto in sola lettura, nel caso il suo valore sia "r", oppure sia in lettura/scrittura qualora il valore sia "rw"

Metodi

long getFilePointer(): restituisce il valore corrente del puntatore al file, valutato conteggiando il numero di byte che lo separano dall'inizio dello stream

-void seek (long pos): assegna al puntatore il valore pos, indicante la posizione del byte in corrispondenza al quale effettuare la successiva operazione di lettura o di scrittura.

void closel)

int skipBvtes(int n) :Salta n bvtes di input

Metodi in lettura

int read()//Legge un byte dal file

int read(byte b[]) //Legge b.lenght byte da questo file e li mette in un array di byte

int read(byte b[], int off, int len) //Legge len byte dal file e li mette in un array iniziando da off

boolean readBoolean()

byte readByte()

char readChar() //Legge un unicode carattere

double readDouble()

float readFloat()

int readInt()

String readLine() //Legge la prossima linea di testo dal file

short readShort() // Legge un 16-bit con segno numero

Metodi di scrittura

void writeBoolean (boolean v)

void writeByte(int v)

void writeBytes (String s) //scrive la stringa in un file come sequenza di bytes

void writeChar (int v)

void writeDouble(double v)

void writeFloat(float v)

void writeInt(int v)

void writeLong (long v)

void writeshort(int v)

```
import java.io.*;
import java.util.Scanner;
public class Main{
```

```
public static void main(String arg[])throws IOException{
FileOutputStream outF = new FileOutputStream ("FileDiProva");
 //scriviamo l'alfabeto sul file
for (char ch = 'a'; ch <= 'z'; ch++)
         outF.write (ch);
      outF.close();

      // Lettura file ad accesso diretto

RandomAccessFile inpF = new RandomAccessFile ("FileDiProva", "r");
      long j=inpF.length()-1;
 System.out.println("inserisci da quale byte si vuole leggere da 2
a "+j);
      Scanner tastiera= new Scanner(System.in);
      int i=tastiera.nextInt();
      for (int k = i; k >=0; k--) {
         inpF.seek (k);
         char ch = (char) inpF.readByte();
         System.out.print (ch);
      }
```

```
        inpF.close();
    }
}
```

Prova il Codice

La gestione delle date

Quando si scrive un programma capita di utilizzare le date e gli orari. Alcuni esempi di utilizzo delle date sono:

- data di nascita
- data di una fattura
- data di un ordine di acquisto o di vendita
- data di un evento

Quando si lavora con le date bisogna considerare diversi fattori, tra cui:

1. Il formato della data (dd/mm/yyyy o yyyy/mm/dd...
2. Il fuso orario

Le classi java per la gestione delle date sono:

1. java.sql.Timestamp
2. java.util.Date
3. java.util.Calendar
4. java.util.GregorianCalendar
5. java.text.SimpleDateFormat
6. java.time

La prima classe usata per trattare le date era la classe:

Classe Date

java.util.Date

Questa classe rappresenta l'intervallo di tempo espresso in millisecondi che va dal 1 gennaio 1970 al momento di creazione dell'oggetto Date. L'istante temporale viene calcolato in base al default time della JVM.

Con la versione di java 1.2 è stato introdotta la classe:

java.util.Calendar

ma la situazione non è migliorata di molto in quanto anche questa classe ha gli stessi problemi di Date e cioè:

o E' mutabile, mentre una data dovrebbe essere auspicabilmente immutabile

o Entrambe rappresentano intervalli di tempo

o I mesi partono da 0. Generando confusione

Le classi di utilità per formattare le date si possono usare solo con Date e non con Calendar. Non sono thread-safe e quindi inadatte, senza gli adeguati accorgimenti, in applicazioni concorrenti.

In Java 8 è disponibile una nuova API iava.time che risolve molte limitazioni presenti nelle precedenti versioni di Java.

Tuttavia queste classi sono comunque importanti visto che tutto il codice Java scritto sino all'avvento di Java 8 ne ha fatto uso.

I costruttori di un oggetto Date:

Date(): Crea un oggetto di tipo date e lo inizializza in modo che rappresenti il momento in cui è stato assegnato misurato al millisecondo più vicino.

Date(long date): Crea un oggetto di tipo date e lo inizializza in modo che rappresenti il numero specificato di millisecondi dal tempo base standard noto come "l'epoca", ovvero 1 gennaio 1970, 00:00:00 GMT.

```
import java.util.Date;
public class Main {
  public static void main(String[] args) {
    Date dataA = new Date();
    System.out.println(dataA);
  }
}
```

Questo codice crea un oggetto che contiene la data attuale e la stampa.

Metodi

`boolean`	`after(Date when)` Verifica se questa data è successiva alla data specificata.
`boolean`	`before(Date when)` Verifica se questa data è precedente alla data specificata.
`Object`	`clone()` Restituisce una copia di questo oggetto.
`int`	`compareTo(Date anotherDate)` Confronta due date per l'ordine.
`boolean`	`equals(Object obj)` Confronta due date per l'uguaglianza.
`static Date`	`from(Instant instant)` Ottiene un'istanza di **`Date`** da un oggetto **`Instant`**.

La maggior parte dei metodi e dei costruttori della classe java.util.Date sono deprecati, questo significa che si deve evitare di utilizzarli nello sviluppo di nuovi programmi.

Per questo motivo per operare correttamente con un oggetto di classe Date, bisogna farlo utilizzando altre classi, quali:

1. DateFormat
2. SimpleDateFormat.
3. Calendar

Se si preferisce memorizzare le date come oggetti di tipo Date, si riesce a evitare l'utilizzo di metodi deprecati facendo la conversione tra gli oggetti Date e Calendar, quando di devono manipolare le date.

La classe DateFormat

Questa classe è utile per ottenere la conversione di un oggetto di tipo Date in una stringa di testo utilizzando diversi stili di visualizzazione, attraverso l'uso di quattro constanti definite nella stessa classe. Le costanti degli stili di visualizzazione di una data sono:

STILE	ESEMPIO
DateFormat.SHORT	04/12/21
DateFormat.MEDIUM	4-dic-2021
DateFormat.LONG	4 dicembre 2021
DateFormat.FULL	sabato 4 dicembre 2021

Alcuni metodi utili:

DateFormat **getDateInstance(int stile, Locale unLocale)** – metodo statico che istanzia un oggetto di classe DateFormat secondo un fissato stile di visualizzazione della data; richiede due parametri: il primo imposta lo stile (vedi tabella precedente), il secondo imposta la localizzazione (vedi esempi seguenti).

String f**ormat(Date data)** – converte un oggetto di classe Date, in una data sotto forma di una stringa (String).

Date **parse(String data)** – converte una data fornita come String, in un oggetto di classe Date. Se la stringa non può essere convertita lancia un'eccezione di **classe ParseException**. Affinché però nella conversione questo metodo possa segnalare in maniera rigorosa tutti gli errori, è necessario impostare opportunamente il metodo setLenient().

void **setLenient(boolean element)** – imposta il tipo di controlli da effettuare durante la conversione di una stringa in un oggetto di classe Date e quindi se segnalare o meno alcuni errori di conversione. Richiede un parametro booleano da impostare su

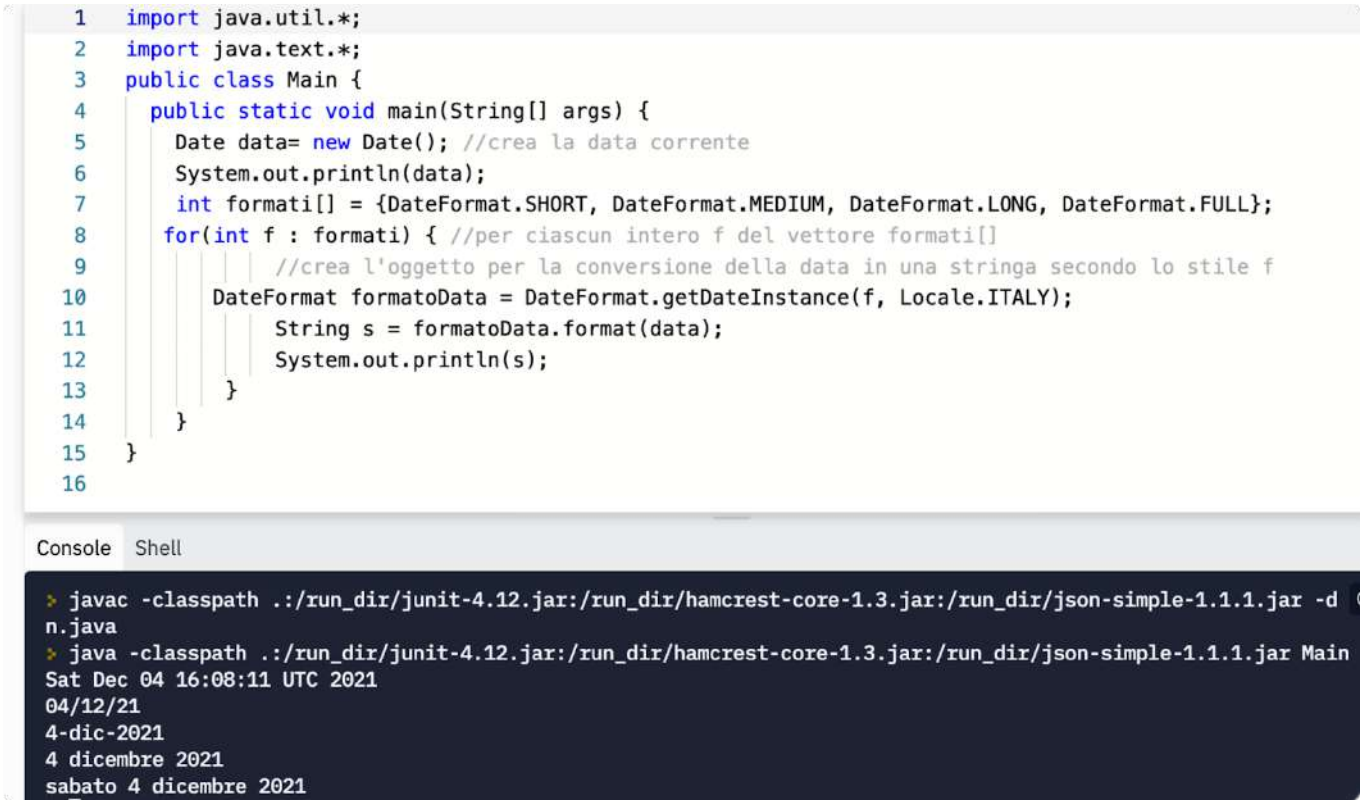
false se si vuole che il metodo parse() non sia clemente, ma al contrario rigoroso, nel calcolo e quindi conversione della data (es. non accetterà una data del tipo 31/02/2021, che per default invece viene accettata e interpretata come 03/03/2021.

Il seguente spezzone di codice istanzia un oggetto *Date* con la data corrente e la visualizza a video nel formato SHORT:

```
Date d = new Date();
DateFormat formatoData;
formatoData = DateFormat.getDateInstance(DateFormat.SHORT,
Locale.ITALY);
String s = formatoData.format(d);
System.out.println(s);
```

Il codice seguente stampa la data odierna senza formattazione e con le diverse formattazioni dovute ai diversi stili.

```
import java.util.*;
import java.text.DateFormat;
public class Main {
  public static void main(String[] args){
    Date data= new Date();//crea la data corrente

    System.out.println(data);//senza formattazione
int formati[] = {DateFormat.SHORT,DateFormat.MEDIUM,
DateFormat.LONG, DateFormat.FULL};
 for(int f : formati) {
 DateFormat formatoData = DateFormat.getDateInstance(f,
Locale.ITALY);
   String s = formatoData.format(data);//effettua la
conversione di date in String
   System.out.println(s);
        }
  }
}
```



```
1  import java.util.*;
2  import java.text.*;
3  public class Main {
4    public static void main(String[] args) {
5      Date data= new Date(); //crea la data corrente
6      System.out.println(data);
7      int formati[] = {DateFormat.SHORT, DateFormat.MEDIUM, DateFormat.LONG, DateFormat.FULL};
8     for(int f : formati) { //per ciascun intero f del vettore formati[]
9              //crea l'oggetto per la conversione della data in una stringa secondo lo stile f
10         DateFormat formatoData = DateFormat.getDateInstance(f, Locale.ITALY);
11             String s = formatoData.format(data);
12             System.out.println(s);
13          }
14     }
15 }
16
```

Console Shell

```
javac -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1.3.jar:/run_dir/json-simple-1.1.1.jar -d
n.java
java -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1.3.jar:/run_dir/json-simple-1.1.1.jar Main
Sat Dec 04 16:08:11 UTC 2021
04/12/21
4-dic-2021
4 dicembre 2021
sabato 4 dicembre 2021
```

Il codice che segue utilizza il metodo parse(String data) per trasformare una stringa in un oggetto Date.

```
import java.util.*;
import java.text.DateFormat;
import java.text.ParseException;
public class Main {
   public static void main(String []args){
        String s;
        Date d = null;
        //si procura la data sotto forma di una stringa
nel formato SHORT
        System.out.println("Inserisci la data
[gg/mm/yyyy]: ");
        Scanner in = new Scanner(System.in);
        s = in.nextLine();
        //converte la stringa della data in un oggetto di
```

```
classe Date
        try{
            DateFormat formatoData =
DateFormat.getDateInstance(DateFormat.SHORT,
Locale.ITALY);
            //imposta che i calcoli di conversione della
data siano rigorosi
            formatoData.setLenient(false);
            d = formatoData.parse(s);
        } catch (ParseException e) {
            System.out.println("Formato data non
valido.");
        }
        //visualizza la data non formattata
        System.out.println(d);
    }
}
```

La classe SimpleDateFormat

La classe SimpleDateFormat è una classe derivata dalla classe DateFormat, quindi, eredita i metodi pubblici della superclasse, ai quali si aggiungono i propri. La classe SimpleDateFormat consente di definire dei pattern personalizzati per l'output.

Costruttori

SimpleDateFormat () formato della data per le impostazioni locali predefinite.

SimpleDateFormat (String pattern) formato della data secondo il pattern.

SimpleDateFormat (String pattern, DateFormatSymbols formatSymbols) formato della

data usando il modello e i simboli di formato data indicati.

SimpleDateFormat (String pattern, Locale locale) formato della data specificando i simboli del formato per le impostazioni locali specificate.

```
String pattern = "dd-MM-yyyy":
SimpleDateFormat formato;
formato = new SimpleDateFormat (pattern);
String data = formato.format (new Date ());
System.out.println(data);
```

È possibile analizzare una stringa e trasformala in un oggetto Date utilizzando il metodo

parse() dell'oggetto SimpleDateFormat.

Esempio:

```
String pattern = "dd-MM-yyyy";
SimpleDateFormat formato = new SimpleDateFormat
(pattern);
Date data = SimpleDateFormat.parse("09-03-1963");
System.out.println(data);
```

Fuso orario

Gli esempi mostrati utilizzano il fuso orario predefinito del sistema.

E possibile impostare il fuso orario utilizzando il metodo:

seTimeZone()

di un oggetto **SimpleDateFormat**.

Il metodo setTimeZone () accetta come parametro un'istanza java.util.Timezone. Ecco un esempio che mostra come impostare il fuso orario:

SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy HH: mm: ssZ");

formato.setTimeZone(TimeZone.getTimeZone("Europe/Rome"));

```
import java.util.*;
import java.text.*;
public class Main {
 public static void main(String[] args) {
  Date data= new Date();
  SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy
HH:mm:ssz");
  df.setTimeZone (TimeZone.getTimeZone("Europe/Rome"));
  System.out.println("Fuso orario di roma:"+
df.format(data));
  df.setTimeZone
(TimeZone.getTimeZone("Europe/London"));
 System.out.println("Fuso orario di Londra:"+
df.format(data));

  }
}
```

Output:

Fuso orario di roma:05-12-2021 10:51:31CET
Fuso orario di Londra:05-12-2021 09:51:31GMT

Calendar e GregorianCalendar

La classe Calendar è una classe astratta che definisce tutti i metodi per la gestione e manipolazione delle date.

Ad esempio, può:

1. Aggiungi un mese o un giorno alla data corrente
2. Controllare se l'anno è bisestile;
3. Restituire singoli componenti della data (ad esempio, estrarre il numero del mese da una data intera)

Un altro importante potenziamento della classe Calendar è la costante Calendar.ERA, con cui è possibile indicare una data antecedente all'era volgare (BC - prima di Cristo) o all'era volgare (AD - Anno Domini).

La classe Calendar non può essere istanziata perché è una classe astratta, quindi per il suo utilizzo è necessario fare riferimento a una sua implementazione. Per ottenere un'istanza della classe Calendar si può:

1. Utilizzare il metodo statico **getInstance()** della classe Calendar che restituisce un'istanza di Calendar in base all'ora corrente nel fuso orario predefinito con le impostazioni internazionali predefinite.
2. Utilizzare la classe java.util.GregorianCalendar che è un'implementazione della classe Calendar. new GregorianCalendar(); inizializza il calendario con la data e l'ora correnti nel fuso orario con le impostazioni internazionali del sistema operativo:

Esempio:

```
Calendar data = Calendar.getInstance();

Calendar data2 = new GregorianCalendar();
```

Si inizializza l'oggetto Calendar con la data e l'ora predefinite in base alle impostazioni internazionali del sistema operativo.

È possibile anche specificare una combinazione di data, ora locale e fuso orario per far questo la classe astratta fornisce dei metodi per la conversione della data tra uno

specifico istante temporale e una serie di campi del calendario come: MONTH, YEAR. HOUR, ecc. E la classe GregorianCalendar mette a disposizione vari costruttori.

Metodi di calendar

Calendar.getInstance0): Calendar.getInstance (TimeZone zona)

Calendar.getInstance (Locale aLocale)

Calendar.getInstance (TimeZone zona, Locale aLocale)

Costruttori di GregorianCalendar

new GregorianCalendar(2018, 6, 27, 16, 16, 47); si specifcano l'anno, il mese, il giorno, l'ora di inizio, il minuto e il secondo per il fuso orario predefinito con le impostazioni internazionali predefinite.

new GregorianCalendar(T imeZone.getTimeZone("GMT+5:30")); Si passa il fuso orario come parametro per creare un calendario in questo fuso orario con le impostazioni internazionali predefinite.

new GregorianCalendar(new Locale("en", "IN")); Si passano le impostazioni internazionali come parametro per creare un calendario in questo locale con il fuso orario predefinito.

new GregorianCalendar(TimeZone.getTimeZone("GMT+5:30"), new Locale("en", "IN")); Si passano sia il fuso orario che le impostazioni locali come parametri.

Metodi

METODO	DESCRIZIONE
abstract void add (int field, int amount)	Viene utilizzato per aggiungere o sottrarre la quantità di tempo specificata al campo calendario specificato, in base alle regole del calendario.
int get (int field)	Viene utilizzato per restituire il valore del campo calendario specificato.
abstract int getMaximum (int field)	Viene utilizzato per restituire il valore massimo per il campo del calendario specificato di questa istanza di Calendar.
abstract int getMinimum (campo int)	Viene utilizzato per restituire il valore minimo per il campo del calendario specificato di questa istanza di Calendar.
Data getTime ()	Viene utilizzato per restituire un oggetto Date che rappresenta il valore temporale di questo calendario.

```
import  java.util.Calendar;
public class  Main {
    public static void  main (String [] args) {
    Calendar calendar = Calendar.getInstance ();
    System.out.println ( "La data corrente è:"  + calendar.getTime ());
    calendar.add (Calendar.DATE, - 15 );
    System.out.println ( "15 giorni fa:"  + calendar.getTime ());
    calendar = Calendar.getInstance ();
    calendar.add (Calendar.MONTH,4);
    System.out.println ( "tra 4 mesi:"  + calendar.getTime ());
    calendar = Calendar.getInstance ();
    calendar.add (Calendar.YEAR,  2 );
    System.out.println ( "tra 2 anni:"  + calendar.getTime ());
    }
}
```

```
a version "1.8.0_31"
a(TM) SE Runtime Environment (build 1.8.0_31-b13)
a HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
data corrente è:Tue Jan 15 10:17:31 UTC 2019
giorni fa:Mon Dec 31 10:17:31 UTC 2018
 4 mesi:Wed May 15 10:17:31 UTC 2019
 2 anni:Fri Jan 15 10:17:31 UTC 2021
```

Codice

- Il campo MONTH della classe Calendar non va da 1 a 12 ma da 0 a 11, dove 0 è gennaio e 11 dicembre.
- Il giorno della settimana va da 1 a 7 , ma domenica, e non il lunedì è il primo giorno della settimana.
    - 1 = domenica, 2 = lunedì, ... 7 = sabato.

Per ottenere informazioni **sull'ANNO, MESE, GIORNO e ORA** si utilizza il metodo:

get()

passandogli come paramentro le costanti statiche definite nella classe (YEAR, MONTH, DAY, ..)

```
int anno = dataAttuale.get(GregorianCalendar.YEAR);
int mese = dataAttuale.get(GregorianCalendar.MONTH) + 1;//i mesi partono da 0
int giorno = dataAttuale.get(GregorianCalendar.DATE);
int ore = dataAttuale.get(GregorianCalendar.HOUR);
int minuti = dataAttuale.get(GregorianCalendar.MINUTE);
int secondi = dataAttuale.get(GregorianCalendar.SECOND);
```

La classe GregorianCalendar implementa dei metodi per effettuare confronti ed operazioni con le date. Di seguito sono riportati a cuni esempi.

Confronto di due date

```
GregorianCalendar data1 = new GregorianCalendar(2008, 11, 18);
GregorianCalendar data2 = new GregorianCalendar(2007, 11, 10);
if (data1.before(data2) ) {
System.out.println("data 1 precede data 2"):
}else if ( data1.after(data2) ) f
System.out.println("data2 precede data 1"):
felse{
System.out.println("Le date sono uguali");
```

I campi del calendario possono essere modificati usando i metodi:

add (), roll () e set ().

Il metodo **add ()**: ci consente di aggiungere tempo al calendario in un'unità specificata in base al set di regole interne del calendario, L'esecuzione del metodo adì () impone un ricalcolo immediato dei millisecondi del calendario e di tutti i campi.

Il metodo **roll()**: aggiunge una quantità al campo del calendario specificato senza modificare i campi più grandi.

il metodo **set ():** permette di impostare direttamente un campo del calendario su un valore specificato Il valore temporale del calendario in millisecondi non viene ricalcolato fino a quando non viene effettuata la chiamata successiva a get (), getTime (), add () o roll ().

Pertanto, più chiamate a set () non attivano calcoli superflui.

```java
import  java.util.Calendar;
import  java.util.GregorianCalendar;
public class  Main {
    public static void  main (String [] args) {
    Calendar calendar = new GregorianCalendar(2021, Calendar.JANUARY , 25);
    calendar.set(Calendar.HOUR, 19);
    calendar.set(Calendar.MINUTE, 42);
    calendar.set(Calendar.SECOND, 12);
   System.out.println("data scelta:"+calendar.getTime());

    calendar.add(Calendar.MONTH, -2); // due mesi fa
    System.out.println("due mesi prima:"+calendar.getTime());

    }
}
```

```
onsole  Shell
javac -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1.3.jar:/run_dir/json-simple-1.1.1.jar -d . Main.java
java -classpath .:/run_dir/junit-4.12.jar:/run_dir/hamcrest-core-1.3.jar:/run_dir/json-simple-1.1.1.jar Main
data scelta:Mon Jan 25 19:42:12 UTC 2021
due mesi prima:Wed Nov 25 19:42:12 UTC 2020
```

Codice

Il metodo add() non ha solo causato la modifica del mese: anche l'anno è cambiato dal 2021 al 2020

output:

data scelta: Mon Jan 25 19:42:12 UTC 2021
due mesi prima con il metodo roll()):Thu Nov 25 19:42:12 UTC 2021

```
1  import  java.util.Calendar;
2  import  java.util.GregorianCalendar;
3  public class  Main {
4      public static void  main (String [] args) {
5      Calendar calendar = new GregorianCalendar(2021, Calendar.JANUARY , 25);
6      calendar.set(Calendar.HOUR, 19);
7      calendar.set(Calendar.MINUTE, 42);
8      calendar.set(Calendar.SECOND, 12);
9    System.out.println("data scelta:"+calendar.getTime());
10
11     calendar.roll(Calendar.MONTH, -2); // due mesi fa
12     System.out.println("due mesi prima con il metodo roll)=:"+calendar.getTime());
13
14     }
15 }
```

Codice

Un altro aspetto interessante di questa classe è lavorare con le ere. Per creare una data "BC", si utilizza il campo Calendar.ERA.

Per creare la data di nascita di Giulio Cesare 15 marzo 44 a.C.

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
    public static void main(String[] args){
       GregorianCalendar cesare = new GregorianCalendar(44,
Calendar.MARCH, 15);
   cesare.set(Calendar.ERA, GregorianCalendar.BC);
   DateFormat df = new SimpleDateFormat("dd MMM, yyy GG");
   System.out.println(df.format(cesare.getTime()));
     }
}
```

Abbiamo usato la classe SimpleDateFormat per stampare la data in un formato più facile da capire (le lettere "GG" indicano che vogliamo che venga visualizzata l'era).