

# Java

Java è il linguaggio che verrà usato principalmente. Java ha due tipi di programmi: le **applicazioni** e gli applet. L'unica differenza consiste nel fatto che le applicazioni sono concepite per essere eseguite localmente su un computer, mentre gli applet sono pensate per essere inviate via Internet ed eseguite in un computer remoto.

## Le basi

Per programmare in Java, dobbiamo iniziare con:

```
Public Class "name"{
    Public static void main (String[]args){
Import java.util.name ;
```

Public: visibilità della class

Static: metodo di classe

Import: carica una libreria da java.util

Si aprono {}, e si scrive in mezzo il codice del programma. Bisogna ricordarsi che, per ogni volta che si apre una {}, bisogna chiuderla.

## Istruzione di stampa

Esistono due istruzioni di stampa:

```
System.out.print("text");
```

Out.print = non vado a capo

Out.println = vado a capo

```
System.out.println("text"); → va a capo la linea dopo.
```

Se si mette ", allora si intende che il computer deve mostrare letteralmente quanto è scritto. Se non si includono, allora si fa riferimento a una variabile, o a un'altra parte del codice.

Una terza istruzione di stampa è il **printf**, che consente di aggiungere informazioni di formattazione per specificare aspetti come il numero di cifre da includere dopo il punto decimale.

```
System.out.printf("Prezzo stampato:%6.2f", prezzo); %6.2f
System.out.println("Prezzo con println:", prezzo);   6 = campo (numero di caratteri)
                                                       .2 = due cifre dopo il punto decimale
[OUTPUT]:
Prezzo stampato: 19.50
Prezzo stampato: 19.5
```

Il primo argomento di printf è una **specifiche di formato**. Si può specificare un campo, ovvero quanti caratteri possono essere stampati.

## Elementi lessicali

Si definiscono "elementi lessicali" le parole del linguaggio, cioè le più piccole unità cui attribuire un valore o un significato. In Java ci sono i seguenti tipi di elementi lessicali:

- parole riservate del linguaggio e caratteri speciali
  - Public
  - For
  - =
  - Class
- Stringe, sequenze di caratteri
- Numeri senza segno
- Identificatori
  - È qualcosa definito dal programmatore per identificare univocamente una classe, una variabile, un metodo, un oggetto
  - Sono sequenze di caratteri alfanumerici, la cui definizione deve sottostare a delle regole:

- Non devono iniziare con un numero
- Possono contenere solo numeri, lettere, \_ e \$
- Altri caratteri speciali non sono ammessi
- Non possono essere usate parole chiave del linguaggio
- Non possono contenere spazio
- Due variabili che hanno le stesse scope non possono avere lo stesso nome
- I nomi sono *case-sensitive*

## Il set di caratteri

Un set di caratteri è una lista di caratteri a ciascuno dei quali è associato un numero.

Il set di caratteri **ASCII** include tutti i caratteri normalmente usati su una tastiera inglese. Ciascun carattere è rappresentato con un *numero binario* che occupa *un solo byte*. Questa codifica fornisce quindi 256 caratteri (1 byte =  $2^8$ ).

Il set di caratteri **Unicode** include, oltre ai caratteri ASCII, anche i caratteri usati in lingue diverse dall'inglese. Un carattere occupa 2 byte, e pertanto la codifica fornisce 65636 caratteri (1 byte =  $2^{16}$ ). Il fatto di utilizzare la codifica Unicode invece che ASCII non ha alcun impatto per chi ha una tastiera inglese; infatti, i caratteri ASCII sono comunque una sottocategoria di Unicode.

## Le libraries

Prima di far partire una app, si possono caricare librerie aggiuntive da vari package Java. Un esempio di questo può essere la libreria *java.util.scanner*, che carica una libreria in grado di prendere l'input dalla propria tastiera.

Successivamente, bisogna predisporre il programma affinché usi questa libreria caricata. In questo caso, bisogna usare:

```
Scanner tastiera = new Scanner(System.in);
Variable = tastiera.nextInt();
```

## Compilare ed eseguire un programma Java

Per fare questo, bisogna innanzitutto aprire un terminale in grado di compilare ed eseguire programmi Java nella console. Come prima cosa, bisogna compilare il programma digitando

```
javac programname.java
```

è importante che il nome del file termini con ".java", se no il programma non è in grado di funzionare. Successivamente, per eseguire il programma Java è sufficiente digitare

```
java programname
```

## Le variabili

Una variabile può memorizzare dati, ed esistono vari tipi di variabili (data type) che possono essere usate. È una precisa locazione di memoria con un identificatore, in cui viene memorizzato un valore di un unico tipo. È particolare poiché il valore assegnato può essere modificato a run-time, da cui il nome di esso "variabile". Ci sono varie "regole" da seguire per le variabili:

- usare nomi che esistono e che descrivono bene cosa rappresentano
- iniziare il nome della variabile della lettera minuscola

Un tipo specifica l'insieme dei valori possibili e le operazioni definite per questi valori. I valori di un particolare tipo di dato sono immagazzinati in memoria nel medesimo formato.

- int = numero intero
- double = numero decimale

```
int variable;
    variable = 10;
    System.out.println(variable);
```

Type *name*; = definisco tipo e nome variabile  
*Name* = *value* = definisco il valore

## Regole per nominare le variabili

- non deve iniziare con un numero
- possono contenere soltanto numeri, lettere, \_ e \$
- non possono essere usate parole chiave del linguaggio
- Due variabili che hanno le stesse scope non possono avere lo stesso nome
- Nomi sono case-sensitive
- Usare sempre nomi estesi e che descrivano bene cosa rappresentano
- Iniziare il nome della variabile con una lettera minuscola
- Se il nome consiste di più di una parola, le parole vanno scritte una di seguito all'altra e ognuna inizia con una lettera maiuscola ad eccezione della prima
- Evitare di usare \$

## Dichiarare le variabili

Prima di poter utilizzare una variabile è necessario fornire alcune informazioni descrittive. Il compilatore ha bisogno di conoscere il nome della variabile, quanto spazio di memoria deve allocare per essa e il modo in cui codificare i dati contenuti nella variabile.

La dichiarazione della variabile indica il tipo della variabile. Poiché tipi di dato differenti vengono memorizzati nella memoria del computer in modi differenti, il computer deve conoscere a priori il tipo della variabile. Una dichiarazione è costituita dal *nome di un tipo*, seguito da un *elenco di nomi di variabili* separati da una virgola. In un programma Java, ogni variabile deve essere dichiarata prima di essere usata.

## I tipi

Un tipo determina il valore che la variabile può contenere, e le operazioni che possono essere effettuate su di essa. Java possiede due tipi di dato principali: i tipi classe e i tipi primitivi.

- **Tipo classe**
  - o Per gli oggetti di una classe
  - o Specifica il modo in cui sono memorizzati i valori del suo tipo
  - o Definisce le possibili operazioni che possono essere compiute su di essi

## I tipi primitivi

I tipi primitivi sono più semplici, e sono valori non decomponibili.

Un numero che presenta una parte decimale è detto **numero in virgola mobile**, o floating-point number. Il floating-point number può essere rappresentato da Java con *double* o *float* (meglio usare double).

Il tipo primitivo *char* viene utilizzato per rappresentare singoli caratteri, e questi caratteri devono essere racchiusi in una coppia di singoli apici 'a'.

Nome	Tipo di valore	Memoria usata	Range di valori
Byte	Intero	8 bit = 1 byte	-128, +127
Short	Intero	16 bit = 2 byte	-32768, +32767
Int	Intero	32 bit = 4 byte	-2147483648, +2147483647
Long	Intero	64 bit = 8 byte	-9223372036854775808; +9223372036854775807
Float	Floating point	32 bit = 4 byte	$10^{+38}; 10^{-45}$
Double	Floating point	64 bit = 8 byte	$10^{308}; 10^{-324}$
Char	Character	16 bit = 2 byte	Unicode/ASCII
Boolean	True or false	1 bit	True or False

## Istruzioni di assegnamento

Per assegnare il valore a una variabile, bisogna usare un'istruzione di assegnamento. Questa istruzione può essere fatta su più righe, o più semplicemente anche su una singola riga. Il simbolo `=`, quando usato in un'istruzione di assegnamento, viene chiamato **operatore di assegnamento**.

L'istruzione indica al computer di cambiare il valore memorizzato nella variabile posta a sinistra dell'operatore di assegnamento, con il valore dell'espressione posta sulla destra.

Tipo	Dichiarazione	Assegnamento
Intero	Short s; Byte b; Int x; Long l;	S = 4; B = 12; X = 46789; L = 3847191;
In virgola mobile	Float f; Double d;	F = 3.5f; D = 5.6e5;
Singolo carattere	Char c;	C = 'd';
Booleano	Boolean b;	B = true;

Una variabile che è stata dichiarata, ma alla quale non sia stato assegnato un valore mediante un'istruzione di assegnamento, è detta **non inizializzata**. È probabile che essa contenga un valore di default, ma è comunque buona norma assegnarle un valore.

## Il CAST

Il cast viene chiamato anche *compatibilità di assegnamento*. La conversione viene effettuata in modo automatico quando si assegna un valore intero a una variabile in virgola mobile, come per esempio nel caso di double variabile `Double = 7;` Un valore può essere assegnato a una variabile il cui tipo compare alla destra del tipo del valore in questo elenco:

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

Inoltre, si possono assegnare valori di tipo *char* a variabili di tipo *int* o di qualsiasi altro tipo che segue *int* nell'elenco presentato.

È possibile inoltre forzare le conversioni. Per esempio, se abbiamo

```
Double distanza = 9.0;
int punti = distanza;
```

è qualcosa di impossibile poiché non può essere assegnato a una variabile di tipo *int*, anche se il suo valore presenta tutti zeri nella parte decimale. Per assegnare un valore di tipo *double* a una variabile di tipo *int* è necessario inserire la conversione:

```
int punti = (int)distanza;
```

ovvero *variable= (tipo a cui voglio fare la conversione) nome variabile iniziale*.

La parte che viene persa da una conversione all'altra viene semplicemente scartata. Questa operazione è detta **troncamento**.

### Esempio:

```
double supposizione = 7.8;
int risposta = (int)supposizione;
```

Abbiamo assegnato inizialmente un valore *double* a una variabile chiamata *supposizione*, e poi successivamente trasformato questa variabile *double* in *int* attraverso **(int)supposizione**. Per far questo, dobbiamo creare una nuova variabile che abbia il valore trasformato di quella vecchia, in questo caso *risposta*.

Nel caso trasformassimo un char (numero) in int, ci verrà mostrato non il testo del char (che in questo caso sarebbe un numero della tastiera) ma ci verrà mostrato invece il codice UTF-16, ovvero il codice di quel carattere.

## Le costanti

In Java possiamo assegnare delle costanti, ovvero dei valori che non cambiano, non come le variabili.

Le costanti non sono necessariamente numeriche; per esempio, 'a', 'b', etc... **Le costanti con caratteri devono essere sempre tra ''**. Il loro valore non può cambiare, ma possono essere utilizzate in un'istruzione di assegnamento per cambiare il valore.

Le costanti numeriche non possono contenere virgole, e le costanti intere non possono contenere numeri decimali. Per numeri come 865000000.0, si usa la **e notation**, dove lo stesso numero si scrive come numero<sup>potenza</sup>\_de l\_10, quindi in questo caso diventa *8.65e8*.

I numeri in virgola mobile vengono memorizzati con una precisione limitata, e quindi sono quantità approssimate. Frazioni come 1/3, anche se hanno il 3 periodico, vengono memorizzate soltanto 10 cifre dopo la virgola.

## Le costanti con nome

Ci sono due modi per assegnare una costante:

- Se non ho ancora inizializzato il programma: `public static final typeofvariable Nameofvariabile = value`
- Se ho già inizializzato il programma: `final typeofvariable nameofvariabile = value`

Definendo *final*, si definisce il fatto che il valore di questa costante è quella finale, e non può essere modificata più dal programma.

## Espressioni ed operatori

La maggior parte delle espressioni fa uso di operatori. Le variabili e i numeri, ovvero gli **operandi**, possono essere combinati con questi operatori e con le parentesi per formare una espressione aritmetica.

Il risultato dell'espressione dipende dagli operandi e dal tipo di operatore. Se entrambi gli operandi sono dello stesso tipo, il risultato è di quello stesso tipo. Se uno degli operandi è un numero in virgola mobile, e il secondo è un intero per esempio, il risultato sarà in virgola mobile.

Questo perché il tipo del risultato prodotto corrisponde a uno dei tipi presenti nell'espressione, seguendo sempre l'elenco per le conversioni cast.

All'operatore di divisione occorre prestare più attenzione. Quando si combinano due operandi con l'operatore di divisione e almeno uno degli operandi è un numero in virgola mobile, il risultato corrisponde al risultato che normalmente ci si aspetta da una divisione.

Quando entrambi gli operandi sono di tipo intero, il risultato è diverso, ovvero la parte decimale è troncata.

Tutti gli operatori che si usano di solito in un'operazione vanno bene. Il quinto operatore Java è l'operatore **resto**, ovvero mostra il risultato di un'operazione, e viene chiamato con `%`.

Questo operatore è usato di più per individuare con facilità i multipli di due, tre o di qualsiasi numero. Per esempio, per compiere una certa azione solo sui numeri pari occorre sapere se un numero è dispari o pari. Un intero n è pari se `n % 2` è uguale a zero.

Gli operatori computano una qualche funzione su uno, due o tre operandi. Ne esistono di tre tipi:

- operatori **unari**: richiedono un operando (es. `++`)
  - o notazione prefissa (operatore op): l'operatore appare prima dell'operando
  - o notazione postfissa (op operatore): l'operatore appare dopo l'operando
- operatori **binari**: richiedono due operandi (es. `=`)
- operatori **ternari**: richiedono tre operazioni (uno solo: `?`)

Binari e ternari utilizzano la notazione **infissa**: l'operatore appare in mezzo agli operandi (op1 operatore op2)

Gli operatori vengono classificati come: aritmetici, aritmetici di incremento e decremento, di assegnamento, relazionali, condizionali, bit a bit

Operatore	Descrizione	Uso
+ <i>value viene salvato nella prima variabile</i>	Somma	Op1+op2
-	Sottrazione	Op1-op2
* (Value1*=value2)	Moltiplicazione	Op1*op2
/	Divisione	Op1/op2
%	Modulo (resto)	Op1%op2
-	Negazione aritmetica	-op
Pow	Potenza	Pow(op1)

Gli operatori possono venire combinati in espressioni complesse, e hanno una precedenza ben definita implicita che determina l'ordine con cui vengono valutati. Gli operatori che hanno la stessa precedenza sono valutati da sinistra a destra.

## Le regole di precedenza

Nelle operazioni tra numeri, il computer ha delle regole di precedenza. Queste regole di precedenza sono le seguenti:

- le parentesi funzionano come nelle espressioni più normali
- **Precedenza più alta**
  - o 1°: operatori unari +, -, !, ++ e -
  - o 2°: operatori aritmetici binari \*, / e %
  - o 3°: operatori aritmetici binari + e -
- Precedenza più bassa
- Le **parentesi** possono essere utilizzate per raggruppare gli elementi di una espressione aritmetica. Con l'aiuto delle parentesi, è possibile indicare al compilatore quali operazioni svolgere per prime.
- Quando le parentesi vengono omesse, il computer esegue prima le moltiplicazioni e poi le addizioni. Quando l'ordine delle operazioni non è determinato dalle parentesi, il computer eseguirà le operazioni nell'ordine specificato dalle regole di precedenza.
- Alcuni operatori hanno la stessa precedenza; in questi casi, il computer esegue le operazioni nell'ordine con cui si presentano gli operatori.
- Le parentesi facilitano la comprensibilità dell'espressione, ma troppe parentesi inutili potrebbero pregiudicare la comprensibilità dell'espressione.

## Compatibilità di assegnamento

Un valore di un tipo non può essere memorizzato in una variabile di un altro tipo, a meno che non venga convertito in un valore compatibile con il tipo di destinazione. Quando si utilizzano valori numerici, Java effettua una **conversione automatica**, per esempio quando si assegna un valore intero a una variabile in virgola mobile, per esempio.

La conversione automatica viene effettuata seguendo le regole dei tipi primitivi [byte > short > int > long > float > double]. Quindi, un valore di tipo long può essere assegnato a una variabile di tipo float o double (ovvero verso destra), ma non **verso sinistra**.

Inoltre, si possono assegnare valori di tipo char a variabili di tipo int o di qualsiasi tipo che segue int nell'elenco.

## Le conversioni di tipo

In Java è "consentito" forzare una conversione di dati. Per esempio, un valore di tipo double non può essere assegnato a una variabile di tipo int, anche se il suo valore presenta tutti zeri nella parte decimale.

Per assegnare un valore di tipo double a una variabile di tipo int, è necessario inserire la conversione (**int**) di fronte al valore o alla variabile che contiene il valore.

```
Variable_type varname = (convert) variabletoconvert;
```

↳ inserisci a quale tipo di variabile vuoi convertire

Nei casi in cui, nella conversione forzata, si verrebbe a un cambio più piccolo (es. Da double ad int si perdono i numeri dopo la virgola), la parte successiva alla virgola viene semplicemente **troncata**.

### Operatori di assegnamento ausiliari

L'operatore di assegnamento semplice = può essere preceduto da un operatore aritmetico, per esempio +, con lo scopo di svolgere un assegnamento e una modifica del valore. Per esempio, la seguente istruzione incrementa di 5 unità. Per esempio:

```
quantità += 5; è lo stesso di quantità = quantità + 5;
```

Gli **operatori di incremento e decremento** sono usati per incrementare o decrementare di una unità il valore di una variabile. L'operatore di incremento si scrive utilizzando ++, mentre per il decremento si usa un doppio --.

Questi operatori sono ereditati da C, e possono essere prefissati o post fissati alla variabile. Attenzione però che cambia il significato!

- Se viene messo prima, la prima incrementa il valore di m **prima** che avvenga l'operazione
  - o Nel caso di un ciclo con *print*, il computer **stampa già la variabile aggiornata!**
- Se viene messo dopo, viene prima eseguita un'operazione (se c'è prima) e poi il valore viene incrementato

### Operazione di input da tastiera

Per registrare un input da tastiera, Java utilizza `java.util.Scanner`, una libreria. Ci sono varie righe per inizializzare questa libreria:

- importare la definizione della classe Scanner dal package: `import java.util.Scanner;`
  - operazioni di inizializzazione: inizializzano un nuovo oggetto Scanner: `Scanner name = new Scanner(System.in);`
  - imposta una variabile per far leggere l'input da tastiera: `nomevariabile = oggetto.nextInt();`
- Questa ultima operazione, `nextInt()` legge dalla tastiera il valore *int*.

Per leggere dati di altro tipo occorre utilizzare altri metodi. Il metodo `nextDouble` si comporta come `nextInt`, con l'unica differenza che legge un tipo double anziché int.

Il metodo `next` legge invece una parola. Se noi scrivessimo:

```
String s1 = tastiera.next(); ➔ forchette
String s2 = tastiera.next(); ➔ coltelli
System.out.println(s1 + s2);
[OUTPUT]: forchette coltelli
```

In questo contesto, i caratteri di separazione sono detti **delimitatori**. Per il metodo `next`, una parola corrisponde a una qualsiasi stringa di caratteri che non contiene caratteri di spaziatura. Per leggere invece un'intera riga, occorre usare **`nextLine()`**.

La fine di una riga di input è indicata dal carattere di escape '\n', digitato nel momento in cui si preme il tasto Invio. Quando `nextLine` legge una riga di testo, trova il carattere '\n', ma non lo inserisce nella stringa restituita come risultato.

'\n'

Se si mischiano nextInt e nextLine, potrebbero crearsi problemi poiché nextInt non legge il carattere escape '\n'. Quindi, se noi chiediamo di scrivere all'utente un intero, e poi un nextLine, nextLine riporterà una stringa vuota perché **nextLine legge anche il 'n' alla riga superiore**, e non trova niente.

42 ← nextInt  
Ancora ← nextLine → \n letto da nextLine  
Ancora2

## I metodi di Scanner

A.nextInt() = restituisce un valore String che corrisponde al prossimo input da tastiera fino al primo carattere di delimitazione escluso

A.nextLine() = legge la parte rimanente della riga corrente e restituisce i caratteri letti come un valore di tipo String. Il '\n' viene letto ma scartato

A.nextInt() = legge un int da tastiera

A.nextDouble() = legge un double da tastiera

A.nextFloat() = legge un float da tastiera

A.nextLong() = legge un long da tastiera

A.nextByte() = legge un byte da tastiera

A.nextShort() = legge uno short da tastiera

A.nextBoolean() = legge il valore Boolean da tastiera. In questo caso, dovrà essere immesso true o false

## Qual è la differenza tra next e nextLine?

Il metodo next() restituisce solo ciò che precede uno spazio, non può leggere due parole separate da uno spazio; inoltre, posiziona il cursore sulla stessa riga dopo aver letto l'input. nextLine() invece legge inoltre gli spazi bianchi, e sposta automaticamente lo scanner verso il basso dopo aver restituito la riga corrente.

## Altri delimitatori di input

Il delimitatore per uno scanner può essere cambiato con la stringa "anything" usando:

```
input.useDelimiter("inserthere");
```

Dopo l'invocazione del metodo useDelimiter, la stringa immessa sarà l'unico delimitatore di input per l'oggetto.

## Le stringhe

Un valore di tipo String è una sequenza di caratteri e trattati come un unico elemento. È un tipo non primitivo, e sebbene String sia una classe, è particolare, perché si può dichiarare una variabile di questo tipo ed assegnargli un valore senza dover creare un oggetto; l'utilizzo è analogo ad un qualsiasi tipo primitivo.

```
String saluto;  
    saluto = "Hello!"  
String saluto = new String("Hello!");
```

La differenza tra String e char è che:

- char può contenere un unico carattere
- String può contenere una sequenza di caratteri

Per i char, il valore è racchiuso tra singoli apici per i char, e doppi apici per le String. Due stringhe possono essere concatenate usando l'operatore +.

Una stringa può anche contenere zero caratteri: una stringa di questo tipo viene detta **stringa vuota**, e viene rappresentata come "".

## Concatenare le stringhe

È possibile concatenare stringhe insieme, in modo da visualizzarle insieme nel terminale. Questo viene eseguito con l'operatore **+**.

Se viene posto tra due stringhe, allora stampa le due stringhe assieme; se dopo di una ci sono numeri, allora aggiunge alla stringa *il numero anteposto*, senza dover dichiarare una nuova variabile.

```
String saluto, frase;
String felice = "felice";
saluto = "Ciao ";
frase = saluto + felice + " amico mio " + 42;
[OUTPUT]: Ciao felice amico mio 42
```

## Metodi di stringhe

La classe String possiede dei metodi. La maggior parte dei metodi di String restituisce un valore. Per esempio, il metodo *length* restituisce il numero di caratteri presenti in un oggetto di tipo String.

<b>Nome.charAt(Indice)</b>	Restituisce il carattere che si trova alla posizione indice della stringa corrente. Gli indici sono numerati a partire da zero.
<b>Nome.compareTo(Altrastringa)</b>	Confronta la stringa corrente con un'altra per individuare quale viene prima in ordine lessicografico. <i>L'ordine lessicografico corrisponde all'ordine alfabetico quando entrambe le stringhe sono costituite solo da lettere.</i> <i>Valore intero negativo se la stringa corrente viene prima, zero se sono uguali, numero positivo se la stringa corrente viene dopo</i>
<b>Nome.concat(Altrastringa)</b>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente concatenati con l'altra
<b>Nome.equals(Altrastringa)</b>	Restituisce <b>true</b> Se la stringa corrente e l'altra sono uguali; altrimenti, restituisce <b>false</b>
<b>Nome.equalsIgnoreCase(Altrastringa)</b>	Come equals, ma considera uguali le lettere maiuscole e minuscole
<b>Nome.indexOf(Altrastringa)</b>	Restituisce l'indice della prima occorrenza della sottostringa Altrastringa nella stringa corrente. String frase = "Frase ciao"; Int posizione = frase.indexOf("ciao"); String frase2 = frase.substring (0, posizione) + "Facile!"
<b>Nome.lastIndexOf(Altrastringa)</b>	Restituisce l'Indice dell'ultima occorrenza della sottostringa Altrastringa all'interno della stringa corrente
<b>Nome.length()</b>	Restituisce la lunghezza della stringa corrente
<b>Nome.toLowerCase()</b>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente, ma in cui tutte le lettere maiuscole sono state sostituite con le minuscole corrispondenti
<b>Nome.toUpperCase()</b>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente, ma in cui tutte le lettere maiuscole sono state sostituite con le maiuscole corrispondenti
<b>Nome.replace(vecchio_char,nuovo_char)</b>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente, ma in cui tutte le occorrenze del vecchio carattere sono state sostituite dal nuovo

<b>Nome.substring(inizio,fine)</b>	Restituisce una nuova stringa che presenta gli stessi caratteri della sottostringa che inizia all'indice inizio fino alla fine (si può non specificare una fine) "Cane".substring(2,3) = an
<b>Nome.trim()</b>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente, ma in cui sono stati rimossi i caratteri di spaziatura in testa e in coda alla stringa
<b>Nome.replaceAll(Regex;replacement)</b>	Cerca tutte le occorrenze di un determinato carattere, e sostituisce tutte le occorrenze di quel carattere

Con le stringhe è possibile inoltre *concatenare metodi*. Per esempio, per ottenere la lunghezza della stringa dopo un ReplaceAll, si può concatenare come **String. ReplaceAll().length()**.

## Confronto tra stringhe

Per verificare se due valori di un tipo primitivo sono uguali, si utilizza l'**operatore di uguaglianza** `=`. Tuttavia, tale operatore ha un significato diverso nel caso dell'oggetto di classe String; in questo caso, occorre utilizzare il metodo `equals`, che restituisce true se due stringhe sono uguali, false se non lo sono. Nel caso delle stringhe, l'operatore `==` verifica solo se queste sono memorizzate nella stessa area di memoria.



Come altro metodo di confronto tra stringhe abbiamo `compareTo`, che confronta due stringhe per verificare il loro ordine lessografico, ovvero le lettere e i caratteri secondo la sequenza Unicode. In questo caso, il metodo restituisce:

- Un numero negativo se il primo precede il secondo
- 0 se le due stringhe sono uguali
- Un numero positivo se il primo segue il secondo

Nella tabella Unicode, tutte le lettere maiuscole precedono quelle minuscole. Quando si confrontano due stringhe che contengono caratteri minuscoli e maiuscoli, dobbiamo tenere conto che **l'ordine lessografico è diverso dall'alfabetico**, ma le lettere maiuscole e minuscole in Unicode sono ordinate tra di loro alfabeticamente.

## Caratteri di escape

Java possiede degli speciali caratteri chiamati **caratteri di escape**. Per esempio, se si volesse scrivere una stringa come "Questo "termine" è una parola", Java non sarà in grado di capire cosa compilare.

```
System.out.println("Questo \"termine\" è una parola");
```

"Questo " lo interpreta come prima stringa,  
Mentre il resto non lo reputa un'istruzione valida.

Come questo, questi sono gli altri caratteri escape:

- \"": apice doppio
- \'": apice singolo
- \\": backslash
- \\n": nuova linea, sposta l'output all'inizio della nuova riga
- \\r": carriage return. Sposta l'output all'inizio della riga corrente
- \\t": tab. Aggiunge spazi bianchi fino al nuovo punto di tabulazione

Una volta inseriti questi caratteri di escape, come per esempio "\\"Ciao\\\"", non contiene quindi otto caratteri ma sei.

# Il flusso di controllo

Con il termine "flusso di controllo" si intende l'ordine con cui vengono eseguite o valutate le diverse azioni di un programma. Vengono usati due tipi di istruzioni per regolare il flusso:

- La **selezione**, sceglie un'azione da un elenco di una o più possibili azioni
- Il **loop**, ripete un'azione più volte fino a quando non viene incontrata una qualche condizione di terminazione

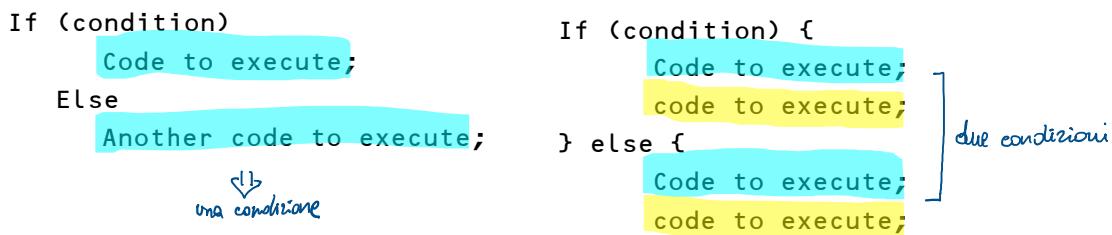
Questi due tipi di istruzioni costituiscono le strutture controllo di un programma.

## Istruzione if-else

Il significato di un'istruzione if-else è analogo a quello di un "se... allora... altrimenti..." in una frase in italiano. Quando il programma esegue un'istruzione if-else, in primo luogo controlla il risultato dell'espressione posta tra parentesi dopo la parola chiave *if*. Questa espressione deve avere un risultato che può essere o **true** o **false**, e permette di scegliere tra due rami.

- Se il risultato è **true** viene eseguita l'istruzione successiva
- Se il risultato è **false** viene eseguita l'istruzione che segue l'istruzione else

Nel caso ci deve essere una singola istruzione nell' if, allora possiamo non mettere le **parentesi graffe**.



Se dobbiamo aggiungere invece più linee di codice all'interno dell'if-else, dovremmo inserire parentesi graffe per iniziare e chiudere sia l'istruzione if e l'istruzione else.

## Le istruzioni booleane

Le espressioni booleane possono avere un valore true, o un valore false.

Un'espressione booleana non deve cominciare né terminare con le parentesi. Tuttavia, è necessario racchiudere l'espressione tra parentesi quando viene usata in un costrutto *if-else*. Per confrontare dei valori, usiamo gli **operatori di confronto**.

Notazione matematica	Nome	Notazione Java	Esempio Java
=	Uguale a	==	Saldo == 0
≠	Diverso da	!=	Entrata != tassa
>	Maggiore di	>	Spese > entrate
≥	Maggiore o uguale a	>=	Punti >= 60
<	Minore di	<	Pressione < max
≤	Minore o uguale a	<=	Spese <= entrate

Relativo a questo, è opportuno una precisazione sugli operatori di confronto e i numeri in **virgola mobile**. Dal momento che il computer può memorizzare in un'area di memoria solo un numero limitato di cifre, i numeri in virgola mobile non sono esatti dati i troncamenti ai dati, e possono diventare sempre meno accurate a ogni calcolo eseguito. Se si confrontano questi valori usando == e !=, possono differire erroneamente; è invece utile verificare se differiscono di così poco che tale differenza sia dovuta alla loro natura approssimata.

Partendo da espressioni semplici si possono costruire espressioni booleane più complesse unendo le espressioni semplici con **l'operatore logico &&**.

```
If ((pressione > min) && (pressione < max)) {
```

In questo caso, il programma verificherà entrambi la veridicità di entrambi le condizioni. Se anche una delle due risulta falsa, allora l'intero if risulterà falsa.

Per le disequazioni, si noti che non si può scrivere **min < pressione < max**; bisogna invece scrivere le diseguaglianze separatamente e connetterle con **&&**.

L'altro connettore logico per le espressioni booleane è l'*or*, espresso come **||** in Java. Le due condizioni non devono essere verificate, ma soltanto una deve essere verificata per fare in modo che l'if sia verificato.

Nome	Notazione Java	Esempio Java
<b>And logico (e)</b>	<b>&amp;&amp;</b>	(Somma > min) && (somma < max);
<b>Or logico (o)</b>	<b>  </b>	(Risposta == 'm')    (Risposta == 'a');
<b>Not logico (non)</b>	<b>!</b>	!(numero < 0)

Valore di A	Valore di B	Valore di A && B	Valore di A    B	Valore di !(A)
switchTrue	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

In modo da negare un'espressione Booleana, è opportuno farla precedere dal simbolo **!**, come per esempio **!=** (che è uguale a **≠**).

↓*basta che una sia vera!??*

## Variabili booleane

Le variabili booleane possono essere usate per rendere più comprensibili i programmi. Per esempio, in un'istruzione *if*, al posto di usare numerosi check, si può usare invece una singola variabile booleana. Questo porta vari vantaggi: il codice è più chiaro, è più efficiente perché non deve provvedere a svolgere gli stessi calcoli ogni volta, ma singolarmente verificarli quando la variabile booleana è inizializzata.

Una variabile booleana restituisce **true**, se la condizione impostata è soddisfatta, e **false** se invece non è soddisfatta.

Se la variabile booleana viene utilizzata all'interno di un ciclo *if-else*, non dobbiamo verificare se sia falsa o vera. Il codice potrebbe essere.

```
if (variabile) → verifica già se la variabile è true,  
    codice  
else  
    codice  
non c'è bisogno di ripetere con  
variabile == true
```

Anche l'espressione "variabile == true" è booleana, e sia questa forma che senza "==" possono essere usate nello stesso contesto. Questo perché se la variabile è true, entrambi le espressioni risultano vere. Se invece una risulta falsa, entrambi risultano false.

## Regole di precedenza

Java usa le stesse regole di precedenza delle espressioni anche per le variabili booleane. Se abbiamo molteplici espressioni, Java valuta ogni espressione come *true* o *false*, e il risultato dipende dall'operatore che abbiamo deciso di usare (che si vedono all'inizio di questo paragrafo).

È dunque utile usare le parentesi per decidere noi quali operazioni devono essere svolte prima. Alcuni operatori hanno la stessa precedenza, e in quel caso vengono valutati nell'ordine in cui compaiono: gli operatori binari vengono valutati *da sinistra a destra*, mentre gli unari *da destra a sinistra*.

<b>Primi</b>	Operatori unari +, -, ++, --, !
<b>Secondi</b>	Operatori aritmetici binari *, /, %
<b>Terzi</b>	Operatori aritmetici binari +, -
<b>Quarti</b>	Operatori booleani <, >, <=, >=
<b>Quinti</b>	Operatori booleani ==, !=
<b>Sesto</b>	Operatore booleano &
<b>Settimo</b>	Operatore booleano
<b>Ottavo</b>	Operatore booleano &&
<b>Nono</b>	Operatore booleano

Java, ogni volta che incontra espressioni con || e &&, molte volte non verifica tutte le condizioni che vengono poste. Se per esempio abbiamo una frase di tre condizioni

```
(Temperatura > 95) || (pioggiaCaduta > 20) || (umidita = 60)
```

Se con un codice del genere il valore della variabile è **99**, allora l'intera espressione booleana è vera indipendentemente dai valori delle altre due espressioni (se una è true, allora tutta la frase è true). Questo tipo di valutare un'espressione è detta **valutazione a corto circuito**, ma Java permette comunque di eseguirne una completa.

Quando viene effettuata una valutazione completa, tutte le espressioni vengono *sempre valutate* tutte; per usare questo metodo, è opportuno usare & per il metodo *and* e | per il metodo *or*.

## Istruzioni if-else annidate

Un'istruzione di controllo if-else può contenere qualsiasi sorta di istruzione. In particolare, si può annidare un'istruzione if-else all'interno di un'altra istruzione if-else.

```
If (saldo >= 0)
    if (tasso_interesse >= 0)
        Codice;
    else
        Codice;
else
    codice;
```

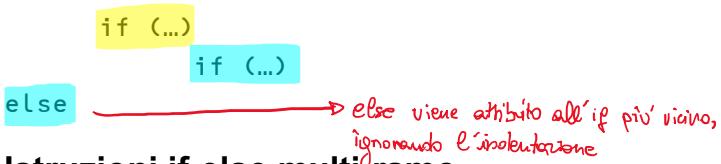
**Attenzione** con gli else però! In un blocco if-else, ciascun else è associato al più vicino if. Se noi abbiamo un codice come questo:

```
//primo
if (...) {
    if (...) {
        codice
    } else
        codice
}
//secondo
```

*l'else viene eseguito per questo if*

Nel secondo, lo else viene associato al **secondo** if, e non al primo. Questo vuol dire che il primo if non ha un else.

Nel primo invece, lo else è **al di fuori** della parentesi graffa che parte dal primo if; questo vuol dire che sono collegati



## Istruzioni if-else multi-ramo

Si possono annidare quanti blocchi if-else che si vogliono. Per esempio, per creare un if-else a quattro rami, basta creare un costrutto if-else a due rami e far sì che **ciascun ramo includa a sua volta un costrutto if-else**.

Quando viene eseguito, il computer verifica le espressioni booleane partendo dall'altra, una alla volta.

Se le casistiche risultano una sola vera in un certo istante, allora sono dette *mutuamente esclusive*. Se è vera invece più di un'espressione booleana, viene eseguita solo l'azione associata alla prima espressione booleana vera. Un costrutto if-else multi-ramo non esegue mai più di un'azione.

Se invece nessuna delle espressioni booleane risulta vera, non accade nulla, o esegue lo else posto alla fine dell'ultimo blocco if.



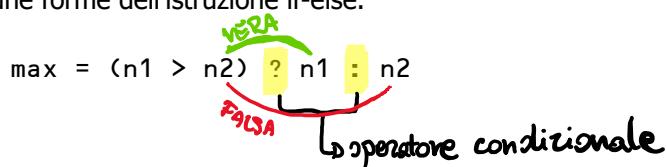
## Operatore condizionale

Al fine di essere compatibile con i vecchi stili di programmazione, Java presenta un operatore che costituisce una variante sintattica rispetto ad alcune forme dell'istruzione if-else.

```

if (n1>n2)
    max = n1;
else
    max = n2;

```



L'espressione **? :** è chiamato **operatore condizionale**. Il **?** Viene posto prima delle due espressioni, e **:** separa le due espressioni.

Se l'espressione booleana (la prima) è vera, viene restituita la seconda espressione, mentre se è falsa allora la terza.

## L'istruzione switch

Un'istruzione switch inizia con la parola chiave *switch* seguita da un'espressione di controllo. All'interno dello switch c'è un elenco di casi; ciascun caso consiste della parola chiave **case** seguita dalla stringa che l'istruzione switch deve controllare.

```

switch (variabile){ → variabile che lo switch deve controllare
    case parola:[ → il case
        code; → codice da eseguire
    break; → interrompe l'esecuzione del codice - se non ci fosse,
              dopo l'esecuzione del primo caso, il programma
              continuerebbe a eseguire anche gli altri casi
    case parola2:
        code;
}

```

```

break;
default: → nel caso nessun altro caso si verifica, il programma esegue questo
        code;
break;
} → chiusura delle switch

```

L'istruzione switch non può essere svolta su variabili di tipo double, mentre con lo switch si può gestire anche una stringa intera.

## Il metodo exit

Alle volte i programmi si trovano in situazioni che rendono insensato il proseguimento dell'esecuzione, per esempio nel caso di un inserimento di valore zero in uno spazio dove non avrebbe senso.

Successivamente, si useranno **try and catch**, ma per ora si possono usare gli if per manualmente andare a "catturare" gli errori. La differenza tra i due è che **try and catch** non necessita di un'impostazione manuale di tutti i casi in cui potrebbe andare male.

```
System.exit(0);
```

System è una classe della Java Class Library, che può essere usata anche senza istruzione di import. Il metodo exit appartiene alla classe System, e il numero **zero** è passato come argomento al metodo, che poi successivamente lo passa al sistema operativo.

La maggior parte dei sistemi operativi usa 0 per comunicare una *corretta terminazione* del programma, mentre 1 per una *terminazione non corretta* del programma.

## I cicli

I cicli permettono di ripetere una o più volte varie istruzioni. Esistono tre tipi di cicli in Java: *while-do*, *do-while* e *for*. Esistono varie parole chiave del ciclo, e una sintassi particolare:

- **Parola chiave del linguaggio:** può essere "while" o "for", ovvero la parola che identifica il tipo di ciclo da eseguire.
  - o **Tra parentesi**, dove c'è la parola chiave del linguaggio, è necessario includere un'espressione booleana che riporti vero o falso. Il ciclo si ripeterà fino a quando il valore di quest'espressione non diventa *false*.
- **Corpo:** insieme delle istruzioni che vengono tutte le volte che l'espressione booleana è vera. Una volta che l'espressione diventa falsa, il programma continua con quello che c'è dopo.
- **Iterazione:** ogni volta che viene eseguito ciò che c'è all'interno del ciclo, si chiama un'"iterazione".

Una nota importante per il ciclo è il ricordarsi di cambiare il valore dell'espressione booleana alla fine dell'iterazione. Se noi avessimo per esempio una variabile con dei numeri, è importante che alla fine del corpo questa variabile venga cambiata di valore, in modo da permettere all'espressione booleana di essere falsa, altrimenti continuerebbe ad essere vera e il ciclo continuerebbe.

È importante ricordare inoltre che **le variabili inizializzate all'interno di un ciclo sono inizializzate solo all'interno del ciclo scelto**; se quindi ne inizializziamo una, e la vogliamo usare al di fuori del ciclo in cui è inclusa, ciò non sarà possibile.

Per esempio, se abbiamo un codice di questo tipo:

```

int x = 0;
for (int i = 0; i < 10; i++) ➤ la variabile i è inizializzata
    System.out.print(i);          all'interno del ciclo for
x = x+i; ➤ pertanto, qua che c'è fuori dal for, la variabile non potrà
System.out.print(x)           essere usata

```

**Bisogna stare attenti a dove le variabili vengono inizializzate!** In questo esempio, la variabile i viene inizializzata nel for, e dunque fuori dal for non può essere usata!

variable; → se necessario, inizializzare una variabile

### Esempio

```

int conteggio = 1;
while(conteggio <=10){
    System.out.println(conteggio);
    conteggio++;
}

```

```

while (Boolean expression) {
    code
}

```

→ è importante ricordarsi  
 successivamente di introdurre  
 un'espressione nel capo al  
 fine di rendere l'espressione  
 booleana falsa

↓  
 esegue il codice fino a quando viene vera  
 assesta il valore della variabile fino a  
 rendere falsa la Boolean

## Il ciclo for

Il ciclo **for** è uno dei tre tipi di cicli in Java, e viene usato quando è presente un contatore che deve contare il numero di iterazioni. Il ciclo for è particolare poiché deve presentare un'**inizializzazione**, un **aggiornamento** e un'**espressione booleana**.

Con questo, si intende che, per configurare un for, dovremmo inizializzare una variabile, proseguire con un'espressione booleana, e al seguito inserire un'istruzione di aggiornamento, che aggiorna il valore della variabile.

```

for (initialize; Boolean expression; update) {
    code;
}

```

esegui ↑  
 valuta ↑  
 ↓  
 aggiorna

**Il ciclo for è uguale a:**

```

initialize;
while(Boolean expression) {
    code;
    update;
}

```

Si può anche non specificare un'inizializzazione e un aggiornamento all'interno del for; bisogna ricordarsi, però, di inizializzare e di scrivere un'istruzione di aggiornamento al di fuori del for e all'interno di esso, effettivamente trasformandolo in un *while*. Come sintassi, avremmo **for(; Boolean expression;)**.

## Il ciclo while

Uno dei modi per definire un ciclo in Java è tramite l'utilizzo dell'istruzione **while**, che ripete più volte l'azione definita nel corpo del ciclo finché un'espressione di controllo rimane vera. Quando questa espressione di controllo immessa tra le parentesi diventa falsa, l'istruzione termina.

```

while (conteggio <= numero) {
    System.out.println(conteggio + ", ");
    conteggio++;
}

```

→ espressione di controllo ←  
 ; false

L'ultima istruzione è importante poiché modifica il valore dell'espressione di controllo, in modo da arrivare a un certo punto alla fine del ciclo while (se non cambiasse al fine di far diventare l'espressione booleana falsa, allora il ciclo continuerebbe all'infinito).

Il corpo di un ciclo while può essere anche eseguito zero volte, nel caso il risultato della prima operazione di verifica dell'espressione booleana sia **falsa**.

## Istruzione do-while

È molto simile al ciclo while; la differenza principale consiste che il corpo di un ciclo do-while viene sempre eseguito *almeno una volta*, mentre il corpo del ciclo while potrebbe anche non essere mai eseguito.

Innanzitutto viene prima eseguito il corpo del ciclo, e successivamente *verifica l'espressione booleana di controllo*.

Se è **vera**, allora il corpo del ciclo viene eseguito di nuovo più volte, fino a quando l'espressione booleana non diventa falsa.

do {  
 corpo } ] Essendo da do prima, questa parte di codice viene eseguita.  
 ↗ ip true

`} while (Boolean expression);` → verifica dell'espressione booleana

• false

Si nota come il do sia prima della verifica dell'espressione booleana. In questo caso, quindi, anche se l'espressione booleana è falsa, il codice viene comunque eseguito una prima volta.

## Cicli annidati

Il corpo di un ciclo può contenere qualsiasi tipo di istruzione, incluso un altro ciclo. In quel caso, il programma prima verificherà il primo ciclo, ovvero quello che sta fuori, e nel caso quello sia vero, allora eseguirà successivamente tutte le istruzioni del corpo. Una volta arrivato al secondo ciclo, eseguirà quel ciclo secondo quale sia il tipo dell'altro ciclo.

## Teorema di Jacopini-Böhm

Ogni algoritmo può essere costruito utilizzando unicamente tre strutture, o *schemi di controllo*, cioè tre schemi aggregativi di istruzioni elementari o di altri algoritmi già costruiti: *sequenza*, *sezione*, *iterazione*, da applicare in modo gerarchico alla composizione di istruzioni elementari.

- **Sequenza:** normale elencazione di istruzioni perché vengano eseguite una di seguito all'altra nell'ordine in cui sono state scritte dal programmatore
- **Selezione:** scelta fra due percorsi da seguire alternativamente, che dipende da una condizione, che può essere vera o falsa.
- **Iterazione:** blocco di istruzioni che vengono ripetutamente eseguite fino a che una certa condizione cambia di stato.

## Criteri di terminazione dei cicli

I cicli possiedono quattro criteri di terminazione.

1. **Contatore:** se noi possediamo una variabile numerica, applichiamo un aggiornamento in modo da aumentare o diminuire la variabile numerica, fino a quando l'espressione booleana collegata ad essa non diventa *false*.
2. **Richiesta all'utente:** viene chiesto all'utente di immettere una stringa, o un numero, per dire al programma di fermarsi.
3. **Sentinella:** simile alla "richiesta all'utente", la sentinella continua a verificare il ciclo per la ricorrenza della sentinella, fino a quando non trova la sentinella. In quel caso, il ciclo termina.
4. **Variabile booleana:** viene inserita alla fine una variabile booleana che rende falsa la condizione del ciclo.

## I metodi

Un metodo raggruppa un insieme di istruzioni assegnando loro un nome. L'insieme di istruzioni di un metodo può essere eseguito ogni volta che è necessario semplicemente riferendosi ad esso attraverso il nome.

Esistono due tipi di metodi: metodi di istanza, e metodi di classe.

Quando si usa un metodo, si dice che si **invoca** o chiama il metodo. La sintassi di un metodo è questo:

```
public class status{  
    public static void method1(){ → intestazione del metodo → public: visibilità del metodo  
        expression; → corpo del metodo → static: tipo del metodo  
    } → void: tipo primitivo  
    Public static void main(String[]args){ → void: nome del metodo  
        method1(); → invocazione  
    } → arguments  
}
```

Esistono due tipi di metodi, ovvero quelli che restituiscono un valore (e quindi bisogna definire quale tipo di valore deve restituire), e i metodi che eseguono istruzioni senza restituire alcun valore (metodo **void**).

## Definizione dei metodi

Un metodo è definito all'interno di una **classe**; pertanto, si dice che un metodo appartiene alla classe in cui è stato definito. Viene definita con l'**intestazione**:

- **Public** viene definita **modificatore d'accesso**; con "public" indica che non ci sono particolari restrizioni sull'uso del metodo
- **Static** è un altro modificatore che regola il modo con cui il metodo può essere invocato. I **metodi di classe** hanno il modificatore "static"; mentre i metodi di istanza no
- Il **tipo di variabile** che il metodo riporta viene riportato dopo: se il metodo non restituisce alcun valore, è un metodo **void**.
  - o Se il metodo non è void, allora deve contenere almeno un'istruzione **return** al suo interno, che indica che il valore restituito dal metodo è ciò che è stato messo dopo "return", la quale può essere una qualsiasi espressione che produce un valore del tipo specificato nell'intestazione del metodo.
    - È possibile anche usare più istruzioni return all'interno del corpo di un metodo che restituisce un valore, o anche in un metodo void.
    - È possibile usare un'istruzione return in un metodo **void**; in questo caso, l'istruzione non fa altro che terminare l'esecuzione del metodo.
- Dopo il termine void, il nome del metodo è seguito da una coppia di parentesi; tra parentesi è indicata la specifica degli argomenti di cui il metodo ha bisogno per poter eseguire le istruzioni in esso definite.

Successivamente, il **corpo**, dove le istruzioni contenute nel corpo del metodo sono racchiuse tra parentesi graffe. Le variabili utilizzate per la definizione di un metodo devono essere dichiarate all'interno della definizione del metodo stesso. Queste variabili sono dette **variabili locali**.

L'**invocazione** di un metodo avviene scrivendo un'istruzione che include il nome del metodo, seguito da una coppia di parentesi, gli argomenti necessari al metodo dentro le parentesi, e da un punto e virgola. Un metodo può essere anche invocato al di fuori della classe in cui è stato definito. In questo caso, occorre far precedere il nome del metodo dal nome della classe in cui è definito seguito da un punto.

## Usare i metodi

1. Definire il metodo:
  - a. Scrivendo la sequenza di istruzioni
  - b. Assegnando alla sequenza un nome

La definizione viene fatta una sola volta e all'interno di una classe.

Definito il metodo, è possibile **invocare il metodo** usando il nome del metodo.

Quando viene invocato un metodo, vengono eseguite le istruzioni definite al suo interno. Quando tutte le istruzioni del metodo sono state eseguite, l'esecuzione viene ripristinata nella posizione in cui era stata eseguita la chiamata al metodo.

## Le variabili locali

Una variabile dichiarata all'interno di un metodo è locale di tale metodo, e si dice che la variabile è locale perché può essere usata esclusivamente all'interno del metodo che l'ha definita.

Se ciascun metodo dichiara una variabile con lo stesso nome, si avranno due diverse variabili che hanno lo stesso nome all'interno della stessa classe. Poiché il "main" è un metodo, questo vale anche per tale metodo.

## I blocchi

Un'istruzione composta è costituita da un gruppo di istruzioni Java racchiuse tra graffe; il termine **blocco** indica la stessa cosa.

Quando si dichiara una variabile all'interno di un'istruzione composta, solitamente l'istruzione composta è detta blocco. Quando si dichiara una variabile all'interno di un blocco, **la variabile è locale al blocco**, cioè ha significato soltanto all'interno del blocco. La porzione di programma in cui una variabile ha significato è detta **scope**.

È quindi possibile usare lo stesso nome per definire un'altra variabile al di fuori del blocco. Se poi si dichiara precedentemente una variabile al di fuori di un blocco, la si può usare sia all'interno sia all'esterno del blocco, e avrà lo stesso significato in entrambi i posti.

## I parametri

È necessario talvolta lasciare degli "spazi vuoti" nel metodo, e riempirli ad ogni invocazione con valori diversi, a seconda del raggio per cui si desidera calcolare l'area. Questi spazi vengono realizzati mediante i **parametri**.

```
Public class Class{  
    public static double Method1 (double value) {  
        return 2 * value * value;  
    }  
  
    public static void main(String[] args){  
        double value = tastiera.nextDouble();  
        double value2 = Method1(value);  
    }  
}
```

Annotations:

- "tipo di variabile" (yellow) points to the parameter type "double".
- "argomenti passati dal metodo" (blue) points to the argument "value" in the call "Method1(value)".
- "stesso tipo di variabile del" (yellow) points to the variable declaration "double value".
- "si possono usare variabili con lo stesso nome" (red) points to the variable declarations "double value" and "double value2".
- "tra parentesi: argomento del metodo" (red) points to the parameter name "(value)" in the method call.

La parola tra parentesi rappresenta una sorta di "sostituto temporaneo" di un valore effettivo, che sarà disponibile solo al momento dell'invocazione del metodo. Il valore effettivo è chiamato **argomento**. Il parametro formale che compare nella definizione di un metodo è una variabile locale che viene inizializzata al valore dell'argomento fornito nell'invocazione del metodo. Se poi viene usato il valore dell'argomento all'interno del metodo, questo meccanismo di assegnamento degli argomenti ai parametri formali è chiamato **chiamata per valore** (call-by-value). Ogni suo parametro viene inizializzato con il valore dell'argomento corrispondente nell'invocazione del metodo.

In molti casi, Java effettua automaticamente una **conversione di tipo**, qualora nell'invocazione di metodo venga usato un argomento il cui tipo non corrisponde a quello del parametro formale.

## Cosa succede quando si invoca un metodo

Quando si invoca un metodo, l'esecuzione passa al corpo del metodo invocato, e viene creata in memoria una struttura dati chiamata **record di attivazione**, che contiene tutte le informazioni necessarie per gestire correttamente l'esecuzione del metodo.

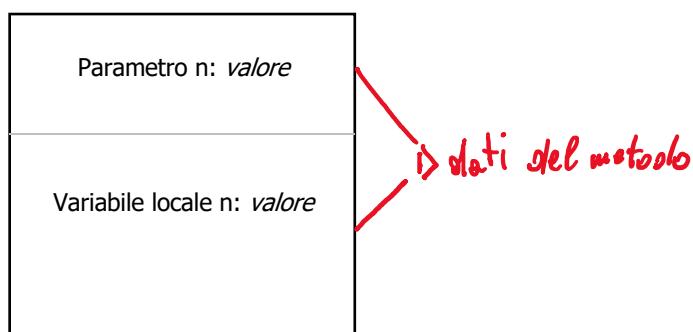
Quando un metodo termina, il controllo torna al chiamante, che deve riprendere la sua esecuzione dall'istruzione successiva alla chiamata del metodo. Se il metodo chiamato restituisce un valore, tale valore viene copiato nel campo risultato del chiamante.

## Record di attivazione

Un record di attivazione contiene dati relativi al metodo invocato e informazioni per la gestione dei metodi da esso eventualmente invocati. Le informazioni relative al metodo invocato includono **parametri formali** del metodo con gli argomenti attuali, e le **variabili locali** al metodo con i valori man mano assunti durante l'esecuzione del metodo. Le informazioni per la gestione dei metodi da esso invocati includono:

- **Indirizzo di rientro**: quale istruzione del metodo deve essere eseguita nel momento in cui il metodo invocato termina; è necessaria per gestire correttamente il rientro dei metodi.
- **Risultato**

Il record di attivazione viene quindi creato dinamicamente nel momento in cui il metodo viene chiamato e viene posto in cima a un'area di memoria denominata **stack**; rimane nello stack per tutto il tempo in cui il metodo è in esecuzione, e viene rimosso al termine dell'esecuzione.



Rientro: indirizzo

Risultato: valore

↳ informazione per la gestione di metodi invocati

Metodi che invocano altri metodi danno luogo a una sequenza di record di attivazione gestiti secondo la politica LIFO (*Last In First Out*).

## Decomposizione

Quello che si fa con un pseudocodice è quello di decomporre l'attività in più sotto-attività. Ciascuna di queste sotto-attività, poi, sono state affrontate separatamente ed è stato prodotto il codice che corrisponde a ciascuna di esse. Per completare il programma si sono, quindi, **combinare le implementazioni delle sotto-attività.**

Se una sotto-attività è di grandi dimensioni, è bene suddividerla in sotto-attività ancora più piccole, da risolvere separatamente. Queste sotto-attività potrebbero a loro volta essere composte in attività più piccole, finché le attività non diventano abbastanza piccole da poter essere progettate e implementate facilmente.

## Problemi di compilazione

Il compilatore controlla che siano state svolte tutte le operazioni necessarie, come l'inizializzazione delle variabili, o l'inclusione di un'istruzione return nella definizione di un metodo che deve restituire un valore. A volte, il compilatore chiede di effettuare una di queste operazioni, anche se il programmatore è convinto di averla effettuata o di non aver bisogno di farla.

Anche se non si riesce a individuare il problema, è bene modificare comunque il codice in modo che il suo significato risulti più chiaro al compilatore.

## Collaudare i metodi

È utile effettuare alcuni test nei metodi, usando grazie a dei **programmi driver**, ovvero dei programmi di test che vanno a testare il metodo in un caso base. Vengono chiamati così perché non fanno nulla al di fuori di esercitare, o guidare, il metodo.

Questi programmi vengono usati dal programmatore per collaudare il sistema, e possono essere molto semplici. Tutto quello che i programmi driver devono fare è fornire al metodo da collaudare una serie di argomenti, e invocare il metodo stesso. Ogni metodo definito in una classe dovrebbe essere collaudato; dovrebbe essere collaudato in un programma specifico per la verifica di quel metodo, e non di altri.

### Testing bottom-up

Uno dei modi per collaudare ciascun metodo separatamente è detto **testing bottom-up**. Se un metodo A invoca il metodo B, l'approccio bottom-up prevede che **il metodo B venga collaudato a fondo prima di collaudare il metodo A.**

Altri approcci al testing potrebbero permettere di individuare i difetti più velocemente, e con meno fatica. Alle volte, potrebbe essere necessario individuare i difetti di un metodo, prima di collaudare tutti i metodi che questo invoca. Uno **stub** è una versione semplificata di un metodo che non può esser utilizzata all'interno del programma finale, ma è sufficiente per permettere il collaudo del sistema, ed è abbastanza semplice per far sì che il programmatore sia certo che il risultato restituito da questo metodo sia corretto.

## Metodi con un numero variabile di parametri

È possibile definire all'interno della stessa classe più metodi con nome uguale, ma con lista di parametri formali diversa. Java consente di definire metodi che accettano un numero variabile di argomenti.

```
public static int text(int... arg){
```

↳ organizza in automatico l'array  
es. massimo{3,2,5,1}  
⇒ int[] arg = {3,2,5,1}

Un metodo che accetta un numero variabile di argomenti è di fatto un metodo che accetta come argomento un array, ad eccezione del fatto che il lavoro di inserire gli elementi nell'array è svolto automaticamente senza che se ne debba preoccupare il programmatore. I valori vengono **passati come argomenti**, e Java crea automaticamente l'array, e vi inserisce gli elementi.

Una specifica di parametro relativa a un numero variabile di parametri, come **int... arg**, è detta specifica **vararg**. I puntini nella specifica sono chiamati **ellissi**; si noti che l'ellissi è a tutti gli effetti parte della sintassi Java, e non un'abbreviazione utilizzata in questo libro.

Nella definizione di un metodo, si può avere una sola specifica di un numero variabile di parametri. È possibile avere, oltre a questa, anche le specifiche di un qualunque numero di parametri ordinari.

## La classe math

Nome	Descrizione	Tipo di argomento	Tipo restituito	Esempio	Valore restituito
<b>pow</b>	Potenza	Double	Double	Math.pow(2.0,3.0)	8.0
<b>abs</b>	Valore assoluto	Int, long, float, double	Stesso tipo di argomenti	Math.abs(-7)	7
<b>max</b>	Massimo	Int, long, float, double	Stesso tipo di argomenti	Math.max(5,6)	6
<b>min</b>	Minimo	Int, long, float, double	Stesso tipo di argomenti	Math.min(5,6)	5
<b>random</b>	Numero casuale	Nessuno	Double	Math.random()	Numero casuale in $\geq 0$ e $< 1$
<b>round</b>	Arrotondamento	Float o double	Int o long	Math.round(6.2)	6
<b>ceil</b>	Intero maggiore	Double	Double	Math.ceil(3.2)	4.0
<b>floor</b>	Intero inferiore	Double	Double	Math.floor(3.2)	3.0
<b>sqrt</b>	Radice quadrata	Double	Double	Sqrt(4.0)	2.0

La classe Math fornisce una serie di metodi matematici standard. Può essere scritto in due modi:

```
int risposta = Math.max(2,3);
System.out.println(risposta);
```

il valore di  
Math.max è salvata  
in una variabile

```
System.out.println(Math.max(2,3));
```

il risultato non viene memorizzato in  
una variabile, ma viene soltanto mostrato

## Gli array

Un array viene utilizzato per memorizzare collezioni di dati. Tutti i dati memorizzati nell'array devono essere dello stesso tipo.

Si supponga di voler calcolare la temperatura media registrata durante i sette giorni della settimana.

```
Scanner tastiera = new Scanner(System.in);
System.out.println("Insert 7 temperatures: ");
double somma = 0;
for (int indice = 0; indice < 7; indice++){
    double t = tastiera.nextDouble();
    somma = somma + t;
}
double media = somma/7;
```

Questo codice funziona correttamente se tutto ciò che si vuole calcolare è la media delle temperature. Si supponga di voler sapere quali temperature sono al di sopra e quali al di sotto del valore medio; per realizzare questo, è necessario leggere le sette temperature, calcolare la media e infine confrontare la media con le sette temperature inserite. Quindi, per poter confrontare ogni temperatura, è necessario memorizzare i valori delle sette temperature, e per questo si usano gli **array**.

Gli **array** sono un particolare tipo di oggetto, ma è più semplice considerarlo come una collezione di variabili dello stesso tipo.

Le variabili che hanno un'espressione intera fra parentesi quadre sono dette **variabili indicizzate**, o elementi dell'array. L'espressione tra parentesi quadre è detta **indice**, e l'indice parte da zero. Se bisogna fare riferimento a un singolo indice, bisogna inserire il numero dell'indice a cui si vuole fare riferimento tra le parentesi quadre; se invece ci si vuole riferire all'intero array, possiamo omettere le parentesi quadre.

0 X	1 Y	2 Z	3 A

È possibile creare un array tramite l'operazione **new**.

```
Tipo_base[] nome_array = new Tipo_base[dimensione];
```

L'istruzione può essere anche spezzata in due istruzioni, una per impostare l'array e uno per crearlo. È possibile dichiarare una sintassi alternativa che prevede le parentesi quadre, dopo il nome dell'array e non dopo il tipo.

Il tipo degli elementi dell'array è detto **tipo base**, il numero di elementi dell'array è detto **lunghezza**. Il valore fra parentesi quadre può essere una qualsiasi espressione che restituisca un intero; quando si crea un array, al posto di usare un numero intero, è preferibile usare una **costante intera** quando si sa esattamente il numero di elementi che l'array dovrà contenere.

Java alloca a run-time la memoria per un array. Quindi, se non si conosce la dimensione di un array durante la scrittura del codice, è possibile inserirla da tastiera durante l'esecuzione.

### Proprietà **length**

È possibile fare riferimento alle proprietà accessibili degli oggetti utilizzando una notazione che prevede il nome dell'oggetto, seguito da un punto e quindi dal nome della proprietà. La proprietà accessibile dell'array è **length**.

```
double[] array = new double[length]
```

Impostare una lunghezza di un array può essere impostato sia con un numero fisso che con una variabile, ed è meglio impostarla con una variabile/costante. Non è possibile però **impostare un valore length modificando quella proprietà**.

### Indici dell'array

L'indice del primo elemento di un array è zero. L'ultimo indice valido per un array di lunghezza n è pertanto **n-1**.

Se l'indice restituito da un'espressione non è compreso fra 0 e array.length - 1, si dice che l'array è **fuori dai limiti**; anche se il codice contiene un indice non valido, viene comunque compilato senza alcun errore, ma si avrà un errore durante l'esecuzione.

Spesso gli indici di un array escono dai limiti previsti quando un ciclo utilizzato per la scansione dell'array viene iterato troppe volte.

Si può anche partire direttamente dall'indice uno piuttosto che dall'indice zero. Questo rende il codice più leggibile, al costo di **perdere un indice**.

### Inizializzare gli array

Un array può essere inizializzato in fase di dichiarazione. Per fare ciò, basta racchiudere i valori delle variabili indicizzate fra parentesi graffe, dopo l'operatore di assegnamento. Se gli elementi di un array non vengono

Array di lunghezza 5

Numeri di elementi 3

→ → → → →

5  
elementi-1

inizializzati esplicitamente in fase di dichiarazione, vengono comunque inizializzati con **valore di default** del loro tipo base. È preferibile però eseguire esplicitamente l'inizializzazione.

Array[0]	123
Array[1]	765
Array[2]	987
Array[3]	0
Array[4]	0

Quando si verifica questa condizione, si ha a che fare con un **array parzialmente riempito**. Non è necessario che l'array sia completamente riempito; questo significa che, ad alcune variabili indicizzate non è stato assegnato un valore utile ai fini del programma. Quando l'array è solo parzialmente riempito, è necessario tenere traccia di quanto l'array sia effettivamente utilizzato, così da conoscere anche quanto ne rimane a disposizione.

Solitamente, si utilizza una variabile per contare gli elementi che vengono inseriti. Quando si accede a un array parzialmente riempito, in realtà si vuole accedere solo agli elementi significativi. Si ignora la parte rimanente quindi, e si usano due variabili **int** per tracciare a quale indice inizia e a quale finisce.

### Ciclo for-each per gli array

Il ciclo for-each, o **ciclo for potenziato**, è un tipo particolare di ciclo for specificatamente per gli array.

```
double[] a = new double [10];  
<code to fill in the array>  
for (tipo_base variabile: nome_array) ↗  
    System.out.println(element); ↗  
    non c'è bisogno di fare il ciclo for manualmente  
    questo ciclo automaticamente riunisce fino alla fine dell'array
```

Si può leggere la linea che inizia con il "for" come "*per ogni elemento in a, esegui le operazioni che seguono*". Si noti che la variabile element è dello stesso tipo degli elementi dell'array. La variabile deve essere dichiarata nel ciclo **for-each**.

Bisogna usare i due punti dopo la variabile. Quando il ciclo viene eseguito, le istruzioni corrispondenti vengono eseguite una volta per ogni elemento dell'array. L'utilizzo del ciclo for-each può rendere il codice molto più pulito e meno soggetto ad errori. Se non serve utilizzare l'indice dell'array in un ciclo for per altro scopo che per scorrere gli elementi dell'array, è preferibile un ciclo for-each.

### 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. Se abbiamo un metodo:

```
Metodo(a[i]);
```

Il fatto che questo metodo possa modificare o meno l'elemento a[i] dipende dal **tipo di base dell'array a**. Se il tipo base dell'array a è un tipo primitivo, il metodo riceve il **valore** di a[i].

Se il tipo base dell'array a è una classe, il metodo riceve un riferimento. Il metodo è quindi in grado di modificare lo stato dell'oggetto referenziato da a[i], ma non può sostituire l'oggetto con un altro.

### Array come argomenti 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:

```
public static tipo_di_ritorno nome_metodo(tipo_base[] nome_parametro)
```

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 inoltre specificare le parentesi quadre dopo il nome dell'array invece che dopo il tipo.

Quando si passa un intero array come argomento a un metodo, non devono essere usate le parentesi quadre. Si può passare un array di qualsiasi lunghezza come argomento a un metodo che accetta come parametro un array. Un metodo può modificare il valore degli elementi di un array passato come argomento.

## Argomenti del metodo main

La dichiarazione del parametro `String[] args` indica che **args è un array il cui tipo base è String**. Il metodo main quindi accetta come parametro un array di valori di tipo String.

Quando si esegue un programma il main viene invocato automaticamente, e come argomento gli viene fornito un array di stringhe di default. È però possibile fornire delle stringhe come argomento del programma, e queste stringhe diventano automaticamente elementi dell'array args, che rappresenta l'argomento di main.

Di solito, si passano argomenti a un programma quando lo si esegue dalla riga di comando.

`java ProgramTest Argument1 Argument2`

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

Per richiamare questi argomenti, si richiama come qualsiasi altro array.

## Assegnamento e uguaglianza di array

Gli operatori di assegnamento e di uguaglianza si comportano con gli array allo stesso modo in cui si comportano con gli array allo stesso modo in cui si comportano con ogni altro oggetto.

L'aspetto importante è che l'intero contenuto dell'array è memorizzato in un'unica area di memoria. In tal modo, la posizione dell'array può essere specificata con un unico indirizzo di memoria. L'operatore = però **non copia il contenuto di un array in un altro array**.

Una variabile di tipo array contiene solo l'indirizzo in cui l'array è immagazzinato in memoria. Questo indirizzo di memoria è detto **riferimento all'oggetto array in memoria**. Per tale motivo, le variabili di tipo array sono dette di tipo riferimento.

## Metodi che restituiscono array

Un metodo java può restituire un array. Per fare ciò, specifica il tipo restituito dal metodo allo stesso modo con cui si specifica un parametro di tipo array.

```
public static tipo_base[] nome_metodo(lista_parametri){  
    tipo_base[] temp = new tipo_base[dimensione_array];  
    istruzioni;  
    return temp; vengono ritornati l'intero array
```

Se si richiama l'array, non è necessario mettere le parentesi quadre.

## Ordinamento e ricerca con gli array

Si supponga di avere un array di valori. Si potrebbe avere l'esigenza di ordinare in qualche modo questi valori. Organizzare un insieme di elementi secondo un particolare ordine viene chiamato **ordinamento**. Tipicamente, gli array si ordinano in senso crescente o decrescente.

## Selection Sort

Si immagini un array a di interi, che si vuole ordinare in senso crescente. Questo significa organizzarlo in modo che:

$$a[0] \leq a[1] \leq a[2] \leq \dots \leq a[a.length-1]$$

Applicando questo algoritmo, i valori dell'array a saranno riorganizzati in modo che  $a[0]$  sia il più piccolo,  $a[1]$  il secondo più piccolo, e così via.

L'unico modo che si ha per muovere un elemento dell'array senza toccare gli altri è quello di scambiare la posizione degli elementi dell'array. Ogni algoritmo di ordinamento che scambia gli elementi è chiamato algoritmo di **ordinamento basato su scambi**.

Dopo aver trovato il valore più piccolo tra  $a[0]$ ,  $a[1]$ , ...  $a[n]$ , e averlo assegnato ad  $a[0]$ , il secondo valore più piccolo dell'array  $a$  è il valore più piccolo tra  $a[1]$ , ...,  $a[n]$ . Dopo aver assegnato tale valore ad  $a[1]$ , il terzo valore più piccolo in  $a$  è il più piccolo valore tra  $a[2]$ , ...,  $a[n]$  e così via....

```
for (indice = 0; indice < a.length - 1; indice++) {  
    // posiziona il valore corretto in a[indice]:  
    IndicedelSuccessivoPiupiccolo =  
        getIndiceDelSuccessivoPiupiccolo(indice, unArray);  
    Scambio(indice, indiceDelSuccessivoPiupiccolo, unArray);  
}
```

Si noti che gli elementi dell'array sono divisi in due segmenti:

- Uno di questi è ordinato
- L'altro non lo è

L'asserzione all'interno dell'algoritmo implica che il segmento ordinato comprende gli elementi da  $a[0]$  fino a  $a[indice]$ .

*Indice del valore più piccolo fra  
 $a[indice]$ ,  $a[indice+1]$ , ...,  $a[a.length-1]$*

*Scambia i valori in  $a[indice]$  e  
 $a[indiceDelSuccessivoPiupiccolo]$*

## Algoritmi di ordinamento

Sort è comunque l'algoritmo non più efficiente. È significativamente meno efficiente rispetto ad altri algoritmi di ordinamento ben noti. Il selection sort, però, è molto semplice. L'implementazione in codice Java di un algoritmo semplice è meno soggetta ad errori.

La classe Arrays del package java.util definisce il metodo statico **sort**. Dato unArray, un array di valori primitivi o oggetti, l'istruzione:

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

Ordina gli elementi dell'intero array in senso crescente. Se si inseriscono delle variabili in inizio e in fine, si può ordinare la sola porzione di array compresa fra l'indice inizio e fine scrivendo un int.

Se è necessario effettuare la ricerca di un elemento all'interno di un array, si può usare la **ricerca sequenziale** di un array. L'algoritmo di ricerca sequenziale è molto semplice e lineare: si guardano gli elementi dell'array dal primo all'ultimo per vedere se l'elemento richiesto è uguale a qualche elemento presente nell'array. La ricerca termina quando nell'array viene trovato l'elemento desiderato o quando viene raggiunta la fine dell'array senza aver trovato l'elemento. Se l'array è parzialmente riempito, la ricerca considera solo la porzione di array che contiene valori significativi.

## Array multidimensionali

Array che hanno esattamente due indici possono essere visualizzati su di un foglio come tabelle bidimensionali, e vengono chiamati **array bidimensionali**. Si attribuisce al primo indice la numerazione delle righe della tabella, e al secondo la numerazione delle colonne.

```
Nome_array[indice_riga][indice_colonna]
```

Gli array con più indici vengono generalmente chiamati **array multidimensionali**. In particolare, un array con n indici viene detto **array n-dimensionale**.

L'unica differenza con un array monodimensionale è l'aggiunta di una coppia di parentesi quadre, e il fatto di usare un secondo numero per specificare la seconda dimensione, corrispondente alle colonne. Si possono creare array con un numero arbitrario di indici; per aggiungere un indice basta aggiungere nella dichiarazione una nuova coppia di parentesi quadre.

Si usano due cicli for innestati. Questo corrisponde al modo comune di scandire gli elementi indicizzati di un array bidimensionale. Se si avessero tre indici, si dovrebbero utilizzare tre cicli innestati, e così via.

Anche le variabili indicizzate degli array multidimensionali sono variabili di un tipo base specificato, e possono essere utilizzate ovunque si sia consentita una variabile di quel tipo base.

Un parametro o un tipo di ritorno di un metodo può essere un array multidimensionale. La sintassi per l'intestazione del metodo è simile al caso in cui parametri o tipi restituiti siano array monodimensionali, ma bisogna usare più parentesi quadre.

```
public static tipo_di_ritorno nome_metodo(tipo_base[][]...[] nome_parametro)
```

Per un array dimensionale b, il valore di b.length è il **numero di righe**, in altre parole l'intero nella prima coppia di parentesi quadre nella dichiarazione dell'array. Il valore di b[i].length è il **numero di colonne**, cioè l'intero nella seconda coppia di parentesi quadre nella dichiarazione dell'array.

## Array irregolari

Se si effettua una dichiarazione come:

```
A = new int[3][];
```

In questo caso, si dichiara A come un array di lunghezza 3, i cui elementi possono essere array di qualsiasi lunghezza. Questo vuol dire che **le righe possono contenere un numero diverso di elementi**, ovvero che righe diverse possono avere numeri di colonne diversi.

## Ricorsione

Si può definire un metodo in termini del metodo stesso. La definizione di un metodo Java può contenere una chiamata al metodo stesso che si sta definendo, cioè **il metodo può invocare se stesso**.

Quando una parte di un algoritmo costituisce una versione ridotta dell'algoritmo completo, quest'ultimo è definito ricorsivo. Un algoritmo ricorsivo può essere implementato tramite un metodo ricorsivo, ovvero un metodo che contenga una chiamata a se stesso, detta chiamata ricorsiva.

contoAllaRovescia(3);

Per fare un metodo ricorsivo, è importante due cose:

- Avere uno o più casi nei quali il metodo esegue il compito previsto senza ricorrere ad alcuna chiamata ricorsiva. Questi casi, privi di chiamate ricorsive, sono chiamati **casi base**.
- Un caso ricorsivo, che richiama il metodo ricorsivo cambiando il parametro fornito. Ogni volta che questa chiamata viene fatta, l'esecuzione del metodo "creato" prima viene sospesa in attesa della fine dell'esecuzione del nuovo metodo con i nuovi parametri. Una volta che la chiamata ricorsiva termina, l'operazione rimasta in sospeso può riprendere e viene eseguito il resto dell'istruzione composta.
  - o Ogni volta che si effettua una nuova chiamata ricorsiva, viene sempre sospesa la "vecchia" chiamata, in attesa del termine della nuova chiamata ricorsiva

Si noti che quando viene eseguita una chiamata di un metodo ricorsivo, non accade nulla di speciale: gli argomenti vengono copiati nei parametri, e viene eseguito il codice contenuto nella definizione del metodo, così come accadrebbe per la chiamata di qualunque metodo.

Quando si incontra una chiamata ricorsiva, l'elaborazione viene temporaneamente sospesa, dato che per procedere è necessario conoscere il risultato della chiamata ricorsiva. Vengono salvate tutte le informazioni necessarie per poter riprendere l'esecuzione in seguito, e viene eseguita la chiamata ricorsiva.

Spesso, si usa un blocco if-else per determinare quale caso debba essere eseguito. Una situazione tipica è quella nella quale il metodo originale esegue un caso che prevede una chiamata ricorsiva. Quest'ultima può a sua volta eseguire un caso che richiede un'altra chiamata ricorsiva. Per un certo numero di volte, ogni chiamata ricorsiva ne genera un'altra, ma alla fine **si dovrà ricadere in uno dei casi base**.

### Ricorsione infinita

Quando si usa un metodo ricorsivo, è importante non cadere nella **ricorsione infinita**. Il compilatore Java accetterà una definizione con una ricorsione infinita. Però, se una definizione ricorsiva non garantisce che un caso base verrà sempre raggiunto, si otterrà una sequenza infinita di chiamate ricorsive, a causa della quale il programma non terminerà mai o terminerà in modo non corretto.

### Lo stack e la ricorsione

Le chiamate ricorsive sono chiamate a **metodi**, e quindi l'innovazione di un metodo comporta la creazione di un nuovo record di attivazione e il suo posizionamento **in cima allo stack**.

Grazie allo stack, il computer può tenere traccia facilmente delle chiamate ricorsive. Quando viene chiamato un metodo, viene creato un nuovo record di attivazione e i parametri formali del metodo vengono inizializzati con gli argomenti passati in ingresso al metodo.

- Il computer inizia quindi l'esecuzione del corpo della definizione del metodo
- Quando incontra una chiamata ricorsiva, interrompe l'esecuzione in corso su quel record di attivazione per determinare il risultato della chiamata ricorsiva.
  - o Prima di fare questo, salva le informazioni necessarie per poter continuare l'elaborazione rimasta in sospeso una volta determinato il risultato della chiamata ricorsiva

```
public static void contoAllaRovescia(int num){  
    if (num <= 0) {  
        System.out.println();  
    } else{  
        System.out.print(num);  
        contoAllaRovescia(num-1);  
    }  
}
```

```
public static void contoAllaRovescia(int num){  
    if (num <= 0) {  
        System.out.println();  
    } else{  
        System.out.print(num);  
        contoAllaRovescia(num-1);  
    }  
}
```

```
public static void contoAllaRovescia(int num){  
    if (num <= 0) {  
        System.out.println();  
    } else{  
        System.out.print(num);  
        contoAllaRovescia(num-1);  
    }  
}
```

```
public static void contoAllaRovescia(int num){  
    if (num <= 0) {  
        System.out.println();  
    } else{  
        System.out.print(num);  
        contoAllaRovescia(num-1);  
    }  
}
```

non viene più eseguito la chiamata ricorsiva

num > 0  
caso base raggiunto

- Queste informazioni vengono scritte su un nuovo record di attivazione che viene posto in cima alla pila
- Il computer **sostituisce** sul nuovo record di attivazione i parametri con gli argomenti passati al metodo, e inizi al'esecuzione della chiamata ricorsiva.
- Quando arriva a una nuova chiamata ricorsiva, ripete il processo di salvataggio delle informazioni sullo stack, e usa un nuovo record di attivazione per la nuova chiamata ricorsiva.

Questo procedimento prosegue finchè **qualche chiamata ricorsiva del metodo non completa la propria elaborazione senza produrre ulteriori chiamate ricorsive.**

- Il computer esamina il record di attivazione in cima alla pila, il quale contiene l'elaborazione *solo parzialmente completata* che è in attesa di quella ricorsiva appena terminata.
- Al termine di quest'ultima, il computer elimina il record di attivazione corrispondente e l'elaborazione sospesa che si trovava sotto di esso nella pila diventa quella in cima.
- Il computer **riprende l'esecuzione** dell'elaborazione sospesa che si trova in cima alla pila e così di seguito.
- Il processo continua finchè non viene completata l'elaborazione riportata sul record di attivazione alla base della pila.

C'è sempre un limite alle dimensioni dello stack. Se si verifica una lunga catena di chiamate ricorsive di un metodo, ogni chiamata ricorsiva *produrrà il salvataggio sullo stack di un altro elaborazione sospesa*. Se questa sequenza è troppo lunga, lo stack cercherà di estendersi oltre i limiti, provocando uno **stack overflow error**.

## Confronto fra metodi ricorsivi e iterativi

La versione non ricorsiva della definizione di un metodo implica solitamente un ciclo al posto della ricorsione. Un processo ripetitivo e non ricorsivo è definito **iterazione** e un metodo che implementi un processo di questo tipo è detto **metodo iterativo**.

Un metodo ricorsivo utilizza più memoria rispetto a una versione iterativa, a causa del carico aggiuntivo sul sistema che deriva dalla necessità di tenere traccia delle chiamate ricorsive e delle elaborazioni rimaste in sospeso. L'esecuzione di un metodo ricorsivo può essere più lenta di quella del corrispondente metodo iterativo.

Un metodo ricorsivo utilizza più memoria rispetto a una versione iterativa, a causa del carico aggiuntivo sul sistema che deriva dalla necessità di tenere traccia delle chiamate ricorsive e delle elaborazioni rimaste in sospeso. L'esecuzione di un metodo ricorsivo può essere più lenta di quella del corrispondente metodo iterativo.

## Metodi ricorsivi che restituiscono un valore

Un metodo ricorsivo può essere un metodo void, come visto in precedenza, o può restituire un valore. Le modalità di progettazione di un metodo che restituisce un valore sono essenzialmente le stesse valide per i metodi void.

Almeno una delle alternative comunque dovrebbe contenere una chiamata ricorsiva del metodo *che produce il valore da restituire*. Queste chiamate ricorsive devono utilizzare argomenti in qualche modo più piccoli, o risolvere versioni ridotte del compito realizzato dal metodo.

## Classi e istanze di oggetti

Una **classe** modella le proprietà comuni di un **insieme di oggetti** (concreti); questa comprende attributi e comportamenti.

Gli **oggetti** di un programma possono rappresentare oggetti del mondo reale, oppure astrazioni; una classe è la definizione di un tipo di oggetto. È come se fosse lo stampo per la costruzione di oggetti di un certo tipo ben descritto.

In una classe, è opportuno descrivere:

- *Attributi:* le proprietà degli oggetti (in Visual Basic)

- La definizione di una classe non specifica il valore degli attributi, ma semplicemente il tipo di dato
- *Comportamenti*: i metodi che gli oggetti di quella classe possono avere

Per semplici fare questa denominazione, è stato introdotto il **diagramma delle classi UML** (Universal Modeling Language), che aiuta ad identificare le classi e le relazioni fra le stesse.

Vengono definiti le seguenti proprietà:

- **Nome**: nome della classe
- **Attributi**: le proprietà dell'oggetto
- +: attributi/metodi pubblici
  - -: attributi/metodi privati
  - #: attributi/metodi protected
  - ~: attributi/metodi package (default)
  - <>: interfaccia
  - *Corsivo*: astratta
- **Operazioni**: i metodi che ciascun oggetto può avere
- **Associazione**: legame tra le classi
  - **Nome**: esprime il significato dell'associazione
  - **Ruolo**: esprime il ruolo giocato dal partner
  - **Cardinalità**: esprime il numero di istanze della classe che possono essere associate con le istanze dell'altra classe
    - \*: multi
    - **N**: esattamente n istanze
    - **N...\***: da n istanze a molte istanze
    - **\*...N**: da molte istanze a n
    - **M, N, J**: m istanze, n istanze, o j istanze
    - **0..\***: uno stesso elemento può essere prenotato in più pacchetti vacanza.
  - Vari tipi di **cardinalità**:
    - *Multipla*: esplicita più associazioni tra una coppia di classe per arricchirne la descrizione
    - *Cappio*: oggetti dello stesso tipo sono concatenati uno con l'altro.
- **Navigabilità**: (freccia) esplicita "chi vede cosa".

Ecco un esempio di un UML:

<i>Nome classe</i>	Automobile
<i>Attributi</i>	Carburante: double Velocità:double Targa: String
Metodi	Accelera(pressionePedale: double): void Decelera(pressionePedale: double): void

Tutte le istanze di un oggetto appartenente alla classe **automobile** ereditano le proprietà definite all'interno della classe

<b>Nome oggetto:</b> autoDiLuca
Carburante: 10.0
Velocità: 55.0
Targa: "DM047GH"

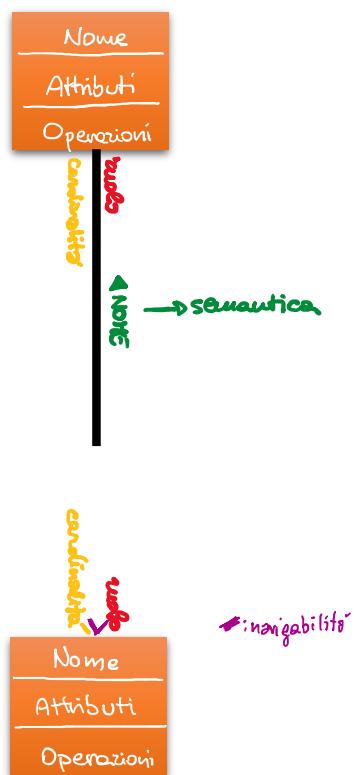
<b>Nome oggetto:</b> autoDiMarco
Carburante: 15.0
Velocità: 75.0
Targa: "LM823GA"

Una volta che all'interno delle classi vengono definite variabili di istanza di tipo classe, sussiste una **relazione associativa** tra le classi, ovvero una relazione *has-a*.

*Es. Una persona può essere caratterizzata dal fatto di possedere un'automobile → relazione associativa.*

Una relazione associativa però non contiene informazioni sufficienti per stabilire come debba essere codificata, e quindi includiamo tre altri elementi:

- **Navigabilità**: specifica la direzione di lettura dell'associazione:
  - unidirezionale, bi-direzione



- se non specificata, non fornisce alcuna informazione utile per la codifica
- **Cardinalità:** specifica quante istanze entrano in gioco in un'associazione.
- **Ruolo:** specifica la funzione che una classe svolge nei confronti della classe con cui è in associazione.
- Specifica il nome che deve essere assegnato alla variabile di istanza che realizza l'associazione.

## File delle classi e compilazione

In Java, è necessario salvare ciascuna definizione di classe in un file distinto. Un file contenente la definizione di una classe ha lo stesso nome della classe stessa, e usa come estensione .java.

È possibile compilare una classe java prima di avere un programma che la utilizzi; il bytecode generato è memorizzato all'interno di un file **.class**. Una volta che il file è stato generato, non è più necessario ricompilerlo ogni volta per usare la sua definizione.

### Variabili di istanza

Sia i dati sia i metodi vengono spesso chiamati **membri** dell'oggetto. Le **variabili di istanza**, proprio come dal nome, sono le variabili che appartengono a un dato oggetto (se si parla in Visual Basic, sono le proprietà di un certo oggetto). Infatti, **un oggetto di una classe è un elemento complesso contenente una serie di variabili di istanza**, e ciascun oggetto di una classe possiede una copia delle variabili definite all'interno della classe.

Usando il keyword **new**, è possibile creare un oggetto di un dato tipo. Il tipo viene definito all'interno di una classe, e il **nome della classe è il nome del tipo creato**.

All'interno della classe, è opportuno definire le variabili che *verranno usate poi da ognuno degli oggetti appartenenti a quella classe*. Si nota come la dichiarazione di una variabile di istanza è identica alla dichiarazione di una variabile, **senza il keyword "Static"**.

Quando viene istanziato un oggetto, i valori delle sue variabili sono automaticamente inizializzate a valori di default, che dipendono dal tipo con cui la variabile è stata dichiarata.

```
public class Class1{
    public type variable1;
    public type variable2;
}

public void method1(){
    //code to execute
}
```

**variabili di istanza**: tutti gli oggetti appartenenti a questa classe ereditano queste variabili;

ogni oggetto eredita inoltre i metodi appartenenti alla classe

```
public class Test{
    public static void main (String[]args){
        Class1 name = new Class1(); → il keyword "new" crea un nuovo oggetto appartenente alla classe, e di conseguenza eredita le variabili e i metodi
        name.variable1 = value;
        name.variable2 = value;
        name.method1; → per richiamare le variabili e/o metodi, è opportuno richiamare l'oggetto creato
    }
}
```

Ogni oggetto creato con l'operatore unario **new** eredita tutte le caratteristiche imposte nella classe, ma ogni oggetto è **per sé**, ovvero che anche se hanno le stesse variabili, hanno ognuno diversi valori.  
Un oggetto può avere al proprio interno più variabili, e l'operatore predisponde queste variabili di istanza all'interno dell'oggetto nel momento in cui lo crea.

### Metodi di istanza

È un metodo che viene **invocato su un oggetto**, e che può manipolare lo stato dell'oggetto stesso. Un metodo di istanza definito in una classe viene invocato usando un **oggetto chiamante** di quella classe.

L'invocazione viene effettuata scrivendo il nome dell'oggetto, seguito dal punto e dal nome del metodo, e infine le parentesi per eventuali parametri. L'intestazione dei metodi di istanza è la stessa dei metodi "normali", ma **non è presente il modificatore "static"**.

#### Dichiarazione metodo

```
public class Class1;  
public type methodName(parameters){  
}
```

#### Invocazione metodo

```
Class1 object = new Class1();  
object.methodName(parameters);
```

Per ovvi motivi, i metodi possono contenere istruzioni che fanno riferimento alle variabili di istanza definite nella classe.

Tutte le definizioni di metodo di istanza compaiono nella **definizione della classe alla quale appartengono**; pertanto, questi metodi possono essere usati *esclusivamente con oggetti della classe in cui il metodo è definito*.

### Variabili statiche

La dichiarazione di una variabile statica contiene la parola chiave "static", e una variabile dichiarata in questo modo è condivisa da tutti gli oggetti della sua classe.

La sua dichiarazione è quindi:

*final "non cambia valore"*

VisibilityModifier static final type name;  
*uso dimesso, non è static*

Una variabile statica può essere pubblica o privata. Analogamente alle variabili di istanza, anche le variabili statiche che non sono costanti, normalmente dovrebbero essere private e dovrebbero essere lette o modificate attraverso *metodi get e set*.

Se ci aggiungiamo la keyword **final**, allora non sarà possibile cambiare il suo valore; se invece si omette questa keyword, allora si potrà cambiare il valore di essa.

Una variabile statica potrebbe consentire comunque agli oggetti di comunicare tra di loro o di eseguire qualche azione in modo coordinato.

### Metodi statici

A volte si ha la necessità che un metodo non abbia alcuna relazione con un oggetto qualunque sia il suo tipo. Quando un metodo non ha un oggetto evidente a cui dovrebbe appartenere, si può definire il metodo come **statico**.

Il metodo è quindi membro di una classe, poiché definito all'interno di una classe, e può essere invocato senza usare alcun oggetto.

Al contrario, se un metodo viene dichiarato senza la keyword "static", allora **quel metodo appartenerà a un oggetto**.

```
VisibilityModifier static type name(arguments){
```

Per invocarlo, bisogna indicare **il nome della classe a cui appartiene**, anziché l'oggetto. Logicamente, non è possibile far riferimento in metodi del genere a una variabile di istanza. Infatti, dato che un metodo statico può essere invocato senza usare alcun oggetto, non esiste alcuna variabile di istanza alla quale ci si può riferire. Inoltre, non è possibile invocare un metodo di istanza all'interno di un metodo statico *a meno che non si abbia un oggetto da utilizzare nella chiamata del metodo di istanza*.

#### Dichiarazione metodo

```
public class Class1;  
public static type methodName(parameters){  
}
```

#### Invocazione metodo

```
Class1.methodName;
```

## Modificatori di accesso

Esistono quattro modificatori di accesso, che possono essere applicati a classi, metodi, o variabili di istanza.

- **Public:** qualsiasi altra classe può usare direttamente quella classe, metodo, o variabile di istanza attraverso il suo nome.
  - o Non è considerata una buona abitudine rendere pubbliche le variabili di istanza di una classe.
- **Private:** il suo nome non è accessibile al di fuori della definizione della sua classe, può essere usata all'interno di uno qualsiasi dei metodi definiti nella sua stessa classe. **Visibile solo dall'interno della classe stessa**
  - o Un metodo non può essere invocato al di fuori della definizione di classe, ma **può essere definito all'interno di un altro metodo appartenente alla stessa classe.**
- **Default:** è visibile dallo stesso package e dalle sottoclassi se sono nello stesso package (equivalente a "friend").
- **Protected:** visibile solo dalle classi dello stesso package e dalle sottoclassi. Può essere attribuito solo ai metodi e alle variabili interni alla classe, ma non può essere messo alla classe stessa.

	Visual Basic	Java	UML
<b>Pubblico</b>	Public	Public	+
<b>Default</b>	Friend		~
<b>Protetto</b>	Protected	Protected	#
<b>Privato</b>	Private	Private	-

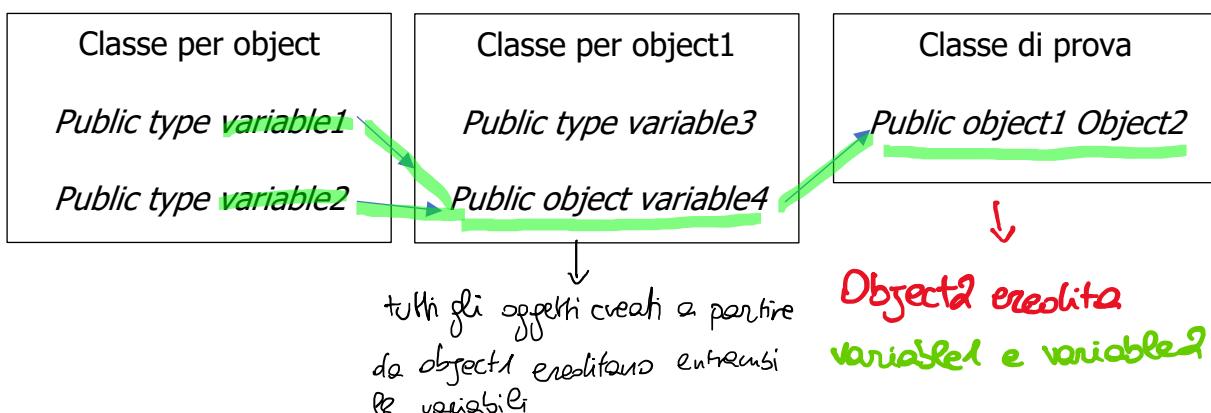
## Uso di classi e variabili

È possibile definire variabili all'interno di una classe, senza doverlo per forza inserire in un metodo. In questo caso, la dichiarazione di variabili è la stessa, solo che viene inserito comunque un **modificatore di accesso** all'inizio della dichiarazione.

```
public type name;
```

Ogni volta che verrà creato un oggetto da questa classe, usando la keyword new, l'oggetto erediterà le variabili di istanza impostate prima.

Variabili di istanza all'interno di una classe possono contenere all'interno di esse riferimenti ad altre classi. Come esempio:



Attraverso il codice, si può accedere singolarmente ai valori delle variabili di istanza, attraverso la seguente sintassi:

```
object.variable.variable.....
```

È un processo che si può ripetere, ogni volta che **referenzi un oggetto puoi riferire a una variabile sottostante.**

Usando l'esempio sopra, si può accedere a variable1 da Object 2 con: `Object2.variable4.variable1`, e poi impostargli una variabile.

Quando un oggetto viene inizializzato all'interno della memoria, viene assegnato il "valore" **null**, che è il valore di default degli oggetti. Semplicemente, vuol dire che l'oggetto non è niente.

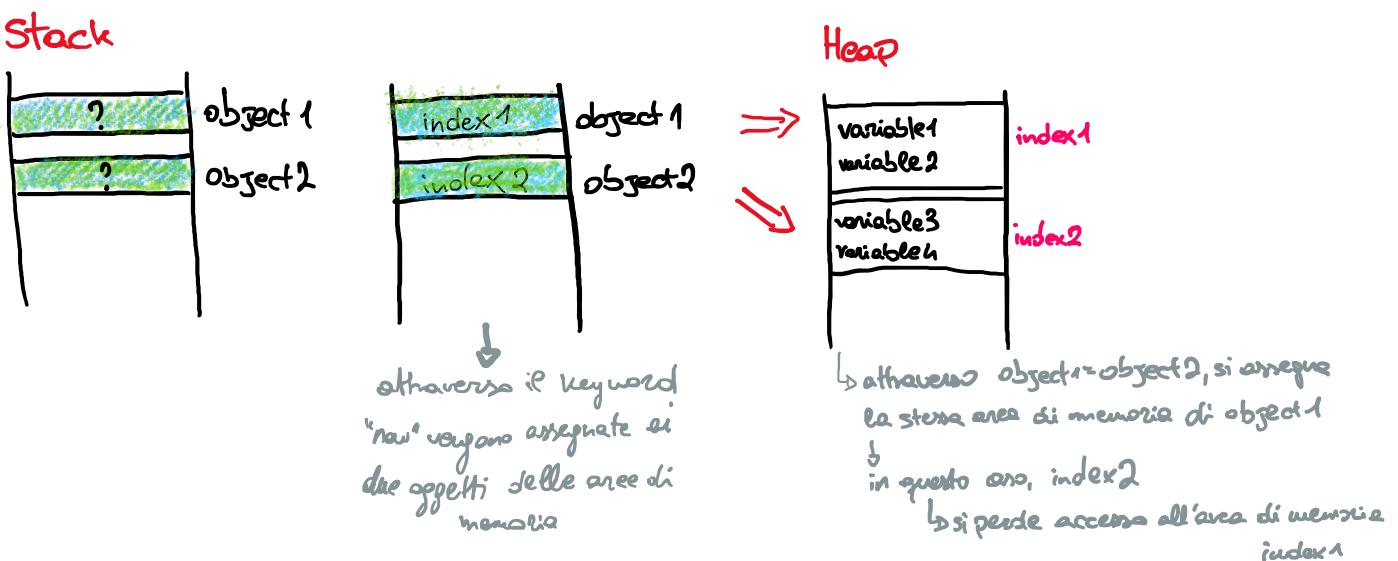
## Gestione della memoria

La gestione della memoria con gli oggetti si arricchisce, introducendo lo **heap**.

Una variabile di tipo primitivo contiene un valore del tipo specificato. Una variabile di tipo classe invece non contiene un oggetto della classe, ma **l'indirizzo dell'area di memoria in cui tale oggetto è memorizzato**. È possibile usare gli operatori = per assegnare un indirizzo di memoria a un altro oggetto; in questo modo, quando si modifica uno, allora si modificherà anche l'altro.

Generalmente, l'operatore == in questo caso verifica se due oggetti sono allo stesso indirizzo di memoria, ma **non verifica se i valori all'interno degli oggetti siano uguali**.

Sebbene un indirizzo di memoria sia un numero, una variabile di tipo classe non può essere usata come una variabile che memorizza un numero.



Anche se due oggetti possono rappresentare gli stessi valori e le stesse cose, a meno che i due oggetti facciano riferimento alla stessa area di memoria, non saranno **mai uguali**. L'operatore == controlla infatti gli indirizzi di memoria.

## Le variabili di tipo classe

Le variabili di tipo classe forniscono un nome agli oggetti. Ciascuna variabile, sia essa di tipo primitivo o di tipo classe, è implementata come un'area di memoria. Una variabile di classe **contiene l'indirizzo di memoria** dell'oggetto cui fa riferimento la variabile. L'oggetto stesso non è memorizzato nella variabile, ma in un'altra area di memoria, spesso chiamata come **riferimento all'oggetto**.

Questa è motivata: un valore primitivo necessita sempre della stessa quantità di memoria, e quindi prevede dimensioni massime per queste; questi valori non possono superare un certo tipo. Un oggetto, invece, può avere una dimensione qualsiasi, e quindi il sistema non può assegnare una quantità di memoria fissa per le variabili che fanno riferimento a oggetti.

Per questo, come già detto, alcune operazioni come = e == **si comportano in maniera differente** con le variabili di tipo classe rispetto a quelle di tipo primitivo.

## Metodi get e set

Rendere private tutte le variabili di istanza di una classe permette di avere un controllo totale su di esse. Può essere però utile poter accedere ad esse comunque.

Un **metodo get** è un metodo che permette di osservare quali sono i dati contenuti in una variabile di istanza; per convenzione, questi metodi iniziano con "get".

```
Modificator type getName(){
```

Un **metodo set** è un metodo che permette di modificare i dati memorizzati nelle variabili di istanza private; per convenzione, questi metodi iniziano con "set".

```
Modificator type setName(){
```

Definire questi metodi sembra annullare lo scopo per cui le variabili di istanza sono dichiarate private, ma in realtà definire metodi get e set amplia le possibilità di implementazione di controlli aggiuntivi, ecc.... Di solito, questi metodi sono definiti con il modificatore **Public**.

Usando questi metodi, si evita che il programmatore o l'utente vada a modificare direttamente la variabile, non solo "nascondendo" come si sia implementato una certa variabile, ma anche permettendo allo sviluppatore di **controllare come la variabile sia modificata**.

## La parola chiave “this”

Come in ogni linguaggio di programmazione, non possono esistere due variabili che hanno lo stesso nome all'interno dello stesso blocco.

Se una delle due variabili è una variabile di istanza, questo è possibile, perché le due variabili si riferiscono ad aree di memoria diverse, e quindi possono coesistere.

```
this.variablename;
```

"This" rappresenta l'oggetto che riceve l'invocazione del metodo.

### *Metodi invocanti altri metodi*

Il corpo di un metodo può contenere l'invocazione di un altro metodo. Tuttavia, se il metodo invocato si trova nella stessa classe, l'invocazione viene effettuata **senza scrivere il nome dell'oggetto**.

Se si invoca un metodo di una classe all'interno della definizione di un metodo di un'altra classe si deve, invece, includere il nome dell'oggetto e il punto.

Omettere il nome dell'oggetto che riceve l'innovazione al metodo è possibile solo se l'oggetto può essere espresso con la parola chiave "this". Se l'oggetto che deve ricevere la chiamata è diverso dall'oggetto rappresentabile con la parola chiave "this", è necessario includere il nome dell'oggetto e il punto.

## Incapsulamento

Incapsulamento significa **nascondere tutti i dettagli della definizione di una classe che non sono necessari per usare le istanze create da quella classe**.

Per usare un certo componente software, un programmatore non ha bisogno di conoscere tutti i dettagli della sua definizione. È possibile "sintetizzare" un programma solo se si scrive il software in modo che sia possibile descriverlo in poco spazio.

È una forma di *information hiding*. La definizione di una classe deve essere tale per cui un programmatore possa usarla senza conoscerne i dettagli. Deve separare la definizione di una classe in due:

- **Interfaccia:** indica ai programmatore ciò di cui hanno bisogno per usare la classe nei loro programmi. Consiste nell'intestazione dei suoi metodi pubblici e delle sue costanti pubbliche, insieme ai commenti che indicano al programmatore come usare i metodi e le costanti.
- **Implementazione:** consiste di tutti gli elementi privati della classe, principalmente le variabili di istanza private e le definizioni dei metodi pubblici e privati.

Ecco alcune linee guida:

- *Si predisponga un commento prima della definizione della classe che descriva al programmatore cosa rappresenta la classe senza descrivere come lo fa.*
- *Si dichiarino tutte le variabili di istanza della classe come private.*
- *Si forniscano metodi **get pubblici** per recuperare i dati in un oggetto, e qualunque altro metodo che possa essere utile al programmatore (che potrebbero essere per esempio **i metodi set**).*
- *Si predisponga un commento prima di ogni intestazione di metodo pubblico che specifichi chiaramente come usare il metodo.*
- *Si rendano privati i metodi ausiliari.*
- *Si scrivano commenti all'interno della classe per descrivere i dettagli implementati i.*

Quando si usa l'incapsulamento per definire una classe, si dovrebbe essere in grado di poter modificare i dettagli implementati della classe senza dover modificare alcun programma che usa la classe.

Da notare che, se c'è una variabile privata in una classe base/astratta, allora **non è visibile dalle classi derivate**.

## Documentazione automatica con javadoc

Di solito, i sistemi Java forniscono un programma chiamato **javadoc**, che genera in modo automatico la documentazione per le interfacce delle classi.

Questa documentazione indica agli altri programmatore tutto ciò che hanno bisogno di conoscere per poter usare le classi. Per generare queste quindi, è opportuno scrivere i commenti in `/** ... */`, inserendo in "..." i commenti che vogliamo inserire.

## Metodi di classe

I metodi di classe possono essere definiti all'interno di una classe, e non necessitano di una speciale sintassi.

```
Visibilità [altri modificatori] tipo nome (tipo 1, tipo 2, ...) {
```

È opportuno ricordare anche la differenza tra un metodo con static e un metodo senza. Se si fa riferimento a un metodo di classe collegato ad un oggetto, se non ha "static" allora **le variabili all'interno del metodo fanno automaticamente riferimento alle variabili dell'oggetto** da cui è stato chiamato. Se invece si inserisce "static", allora sarà obbligatorio far inserire/inserire manualmente ogni volta le istruzioni che servono.

I metodi possono restituire valori di tipo Boolean; basta specificare un valore restituito di tipo Boolean, e usare un'espressione booleana nell'istruzione di return.

## Metodo equals

Considerato il fatto che non è possibile usare le operazioni = e == per verificare se gli attributi all'interno di due oggetti sono uguali o no (N.B.: i due operatori confrontano gli indirizzi di memoria), per poter confrontare gli oggetti "come faremo noi", è opportuno definire un metodo equals.

Questo cambia a seconda della casistica, e non si può dire che ci sia una definizione generale. L'unica definizione presente è che è **un metodo che permette di verificare, confrontando gli attributi, se due o più oggetti sono uguali**.

Qui sarà opportuno dare la keyword **this**, che permette di fare riferimento ad una variabile di istanza e non a una variabile di classe. Se invece si omette, bisognerà che il metodo in cui è incluso non sia di tipo "static"; in quel caso, le variabili all'interno prendono direttamente dall'oggetto e dal metodo con cui è stato chiamato.

Se non si definisce un metodo equals per una classe, Java ne crea automaticamente uno con una definizione di default.

## Dot notation

La dot notation viene usata in Java assieme agli oggetti. Funziona nello stesso modo rispetto ad altri linguaggi, come Visual Basic.

Viene inizialmente fatto riferimento all'oggetto o classe a cui si vuole fare riferimento, seguito da un punto e poi da un **riferimento a un attributo o ad un'operazione** appartenente alla classe o all'oggetto a cui si è fatto riferimento prima.

Se un attributo è un altro oggetto, allora si può usare un'altra volta la dot notation per accedere alle proprietà e metodi di quell'oggetto all'interno di un altro oggetto.

Object1.object2.object3....

oggetti come proprietà

Si può accedere a relativi oggetti, proprietà e metodi "a catena"

## La costante null

La costante null è una costante speciale che può essere assegnata a una variabile di qualunque tipo classe. Se il compilatore richiede che una variabile di tipo classe venga inizializzata e non sono disponibili oggetti opportuni per inizializzala, allora si può usare il null. Non si può però **invocare un metodo usando una variabile inizializzata a null**; questo genera infatti l'eccezione *Null Pointer Exception*.

Null però **non è un oggetto**, è come un riferimento che non si riferisce ad alcun oggetto. Per questo quindi, se si vuole verificare che un oggetto valga null, si usano gli **operatori != e ==**.

## Costruttori

Quando si crea un oggetto di una classe usando **new**, si invoca un metodo particolare chiamato **costruttore**, che esegue in automatico le operazioni di inizializzazione necessarie.

```
Class name = new Class(parameters);  
          ↓  
le proprietà possono essere  
inizializzate automaticamente  
direttamente dall'invocazione
```

```
public className(parameters){  
    this.firstParameter = firstParameter;  
    this.secondParameter = secondParameter;  
}  
imposta le inizializzazioni
```

La prima parte dell'istruzione invocante è un riferimento a un oggetto della classe; la seconda parte crea ed inizializza un nuovo oggetto. Se non viene creato alcun metodo che specifica quali parametri devono essere presenti, **Java fornisce alle variabili di istanza un valore iniziale di default**, che però potrebbero non essere quelli desiderati.

Un costruttore può eseguire qualsiasi azione inserita nella sua definizione; infatti, hanno lo stesso compito del metodo *set*, con la differenza che i costruttori vengono invocati **esclusivamente per creare un nuovo oggetto**, mentre il metodo set è usato per cambiare lo stato di un oggetto.

Un costruttore, qualora non ci siano parametri, viene definito automaticamente da Java con i valori di default; infatti, è detto **costruttore di default**.

Un costruttore non può essere specificato come abstract.

## Classi wrapper

Un argomento di un metodo è trattato in modo diverso a seconda che l'argomento sia di un tipo primitivo o di un tipo classe. Se un metodo ha bisogno di un argomento di un tipo classe, ma si ha a disposizione un valore di tipo primitivo, è necessario **convertire il valore primitivo in un equivalente "valore" di un certo tipo classe**; per effettuare questo, possiamo usare una *classe wrapper* per ciascuno dei tipi primitivi.

Rispettivamente per i tipi *int*, *long*, *float*, *double* e *char*, abbiamo le classi wrapper ***Integer***, ***Long***, ***Float***, ***Double*** e ***Character***,

```
Integer n = new Integer(x);                                int i = n.intValue();
```

La conversione da un tipo primitivo alla sua classe corrispondente è **boxing**, mentre la conversione inversa è detta **unboxing**; entrambi possono essere svolte automaticamente da Java.

Queste classi contengono all'interno una serie di costanti e metodi statici utili; per esempio, *Integer* include *MAX\_VALUE* e *MIN\_VALUE*; entrambi sono definiti come *public static final*.

Altro metodo interessante è **parseType**, che converte una stringa nel rispettivo tipo specificato. Viene declarato come:

```
Type.parseType(laStringa.stringMethod())
```

*↳ è possibile usare metodi della Stringa  
all'interno della dichiarazione*

*Sesso tipo*

Se la stringa non è del tipo specificato, allora viene prodotto un messaggio di errore. Esiste anche il metodo **toString**, che esegue la conversione nella direzione opposta.

## Overloading

Quando si assegna lo stesso nome a due o più metodi all'interno della stessa classe si dice che si sta effettuando l'**overloading**; per compiere questa operazione, è necessario che le definizioni dei diversi metodi presentino differenze nell'elenco di parametri che ricevono. Java infatti distingue i metodi sulla base del numero di parametri che ricevono e dei tipi di parametri.

1. Se un'invocazione di metodo corrisponde alla definizione di un metodo in termini di nome, tipo del primo argomento, tipo del secondo, e così via, allora Java esegue quel metodo
2. Se non esiste alcun metodo che corrisponde, allora Java prova a eseguire automaticamente alcune delle conversioni di tipo per verificare se trova una corrispondenza.
3. Se non c'è alcuna corrispondenza nemmeno in questo caso, Java restituisce un errore durante la compilazione del programma.

Il nome di un metodo, il numero e il tipo di parametri che utilizza sono detti **firma** del metodo; nel fare l'overloading quindi, bisogna affermare che tutti i metodi di una classe devono avere **firme diverse**. Inoltre, invocazioni ambigue non sono permesse e generano un messaggio d'errore a runtime. Infatti, due metodi come:

```
Public static void method1(double n1, int n2)  
Public static void method2(int n1, double n2)
```

*existono sia int che  
double ➡ AMBIGUO*

Queste sono dichiarazioni non valide, perché Java non potrebbe decidere quale delle due funzioni deve considerare.

Non si può effettuare l'overloading di un nome di metodo fornendo due definizioni la cui **intestazione** **differisce solo per il tipo del valore restituito**.

## Privacy leak

Una classe può avere variabili di istanza di qualsiasi tipo; usare però questi tipi di variabile può introdurre un problema: le variabili di tipo classe contengono l'indirizzo di memoria in cui si trova fisicamente l'oggetto.

Se si impostano le variabili come private comunque, e non si **modificano i metodi in modo da non usare direttamente le variabili**, si potrebbe far uso di questi per modificare direttamente le variabili private, od **ottenere l'indirizzo di memoria**.

Esistono vari metodi per evitare che un privacy leak accada:

- Usare solo variabili di tipo primitivo: rappresenta una limitazione.

- Definire classi che non abbiano alcun metodo set; altra limitazione.
- Definire metodi get che restituiscono i singoli attributi delle variabili di istanza di tipo classe, al posto dell'oggetto. Attributi che sono essi stessi oggetti potrebbero essere a loro volta affetti dallo stesso problema
- Definire metodi che, una volta invocati, creano un duplicato esatto di un oggetto; questi oggetti sono detti **cloni**, che invece di restituire un oggetto referenziato da una variabile di istanza di una classe, può essere restituito un clone. Il programmatore **può modificare il clone senza alterare i dati privati**.

## Array nelle definizioni di classe

Gli array possono essere utilizzati all'interno delle definizioni di classe. Il tipo base di questi array può essere un tipo qualunque e, in particolare, può essere anche una classe.

Il tipo base di un array può essere un tipo qualunque e, in particolare, può essere anche una classe.

```
ClassType[] name = new ClassType[quantity];
```

Così come gli elementi di un array il cui tipo base è un tipo primitivo sono inizializzati a valori di default, anche gli elementi di un array il cui tipo base è un tipo classe sono inizializzati al valore di default che è la costante null.

Ovviamente, gli array di questo tipo possono essere dichiarati **dopo**, nel senso che **non dobbiamo impostare la dimensione di un array nella classe, ma quando verranno chiamati nel main**. È importante, come per il resto, usare i metodi *get* e *set* per gli array anche.

## Enumerazioni come classi

Possiamo enumerare classi direttamente usando un'istruzione del genere:

```
Enum Class {element1, element2, ...}
```

Il compilatore, in questo caso, crea la classe, e i valori elencati sono **nomi di oggetti pubblici e statici**, il cui tipo è lo stesso della classe. Tutti questi valori possono essere impiegati nel programma con un'istruzione del genere:

```
Class name = Class.element1;
```

E presenta inoltre vari metodi:

- Name.equals(class.element) —> verifica se *name* è uguale a *element*
- Name.compareTo(class.element) —> verifica se *name* precede, è uguale o segue *element* nella definizione della Class. Restituisce un intero che è negativo, zero o positivo a seconda del risultato.
- Name.ordinal() restituisce la posizione, o **valore ordinale**, di *element* nell'enumerazione.
- Name.toString() restituisce la stringa "element" – restituisce il nome dell'oggetto chiamante del metodo come una stringa.
- Name.valueOf("element") restituisce l'oggetto Name.element; la corrispondenza tra la stringa passata in input e il nome dell'oggetto nell'enumerazione deve essere esatta.

## Package

È una collezione di classi correlate a cui viene assegnato un nome. Svolge il ruolo di libreria di classi, le quali possono essere utilizzate all'interno di un qualsiasi programma. Grazie ai package, non occorre posizionare tutte queste classi nella stessa cartella del programma.

Il nome della cartella deve corrispondere al nome del package; all'interno, le classi sono disposte in file distinti, i cui nomi corrispondono al nome della classe, e ciascun file del package deve contenere la seguente riga all'inizio:

```
package package_name;
```

Ciascun file, all'interno del package, può essere usato con la **dot notation**; qualsiasi programma o classe può usare anche tutte le classi contenute in un package inserendo un'opportuna istruzione **import** all'inizio del file:

```
import package.class.class;
import package.class.*;
```

Il nome di un package non è un identificatore qualsiasi, e indica al compilatore dove trovare le classi contenute nel package; indica inoltre il **percorso della cartella contenente le classi**; per questo, ha bisogno anche di una **class path**.

Il valore della variabile **class path** indica a Java dove iniziare la propria ricerca per individuare un certo package; non è una variabile di Java, ma una variabile di sistema. Questa variabile contiene i nomi di percorsi di un certo elenco di directory, e sono dette **cartelle base del class path**. Il nome di un package specifica il percorso relativo di una cartella che contiene le classi del package.

Si noti che il nome di un package non è arbitrario, ma deve corrispondere a un elenco di cartelle contenute all'interno di una cartella base del class path.

Una classe path potrebbe essere definito in questo modo:

```
C:\programs\librerie;f:\otherprograms;.
```

Innanzitutto, con il ; possiamo definire molteplici path, con il punto possiamo aggiungere la cartella corrente al class path, ovvero **le sottocartelle del programma che si sta compilando**.

Omettere però la cartella corrente dal class path non limita solamente il numero di posti utilizzabili per trovare i package, ma potrebbe interferire con i programmi che non usano package. Se non si creano package, ma si posizionano tutte le classi nella stessa cartella, Java non sarà in grado di individuare le classi a meno che la cartella corrente non sia nel class path.

Se due programmatori differenti hanno usato lo stesso nome per una classe, ma posizionata in package diversi, l'ambiguità del nome della classe viene risolta proprio grazie al nome del package.

## Inner class

Le **inner class** sono classi definite all'interno di altre classi. Per definirla, basta includere la definizione di questa classe all'interno della classe esterna:

```
public class ClasseEsterna{
    private class InnerClass {
        //metodi
    }
}
```

La definizione della inner classe non necessariamente deve essere l'ultimo elemento della classe esterna, ma è buona prassi posizionarla così. Non deve essere inoltre privata.

Una definizione di inner class è **locale alla definizione della classe esterna**; quindi, si può riusare il nome della classe interna all'esterno della classe esterna.

I metodi delle inner class hanno accesso ai metodi e variabili di istanza delle classi esterne (*e viceversa*), **anche se sono privati**; in questa definizione, quindi, public e private sono equivalenti.

## Ereditarietà

L'ereditarietà permette di **definire una classe più generale e in seguito classi specializzate** che aggiungono nuovi dettagli alla classe generale. La classe specializzata eredita tutte le proprietà della classe generale, e il programmatore deve solo realizzare le nuove caratteristiche.

Una classe derivata è una classe definita aggiungendo variabili di istanza, e metodi a una classe esistente. La classe esistente è chiamata *classe base*, e tutte le classi specificate ereditano le variabili di istanza e i metodi pubblici della classe che estende. Le variabili di istanza, le variabili statiche e i metodi pubblici ereditati dalla classe base non sono dichiarati esplicitamente nella definizione della classe derivata, ma diventano automaticamente suoi membri.

```
Modifier class Derivative extends BaseClass
```

Date due classi, se non esiste una relazione *is-a* tra di esse, non si usa l'ereditarietà per derivare una classe dall'altra.

## Overriding

Se una classe derivata definisce un metodo che ha lo stesso nome, gli stessi parametri e anche lo stesso tipo di ritorno di un metodo della classe base, il metodo della classe derivata ridefinisce il metodo presente nella classe base.

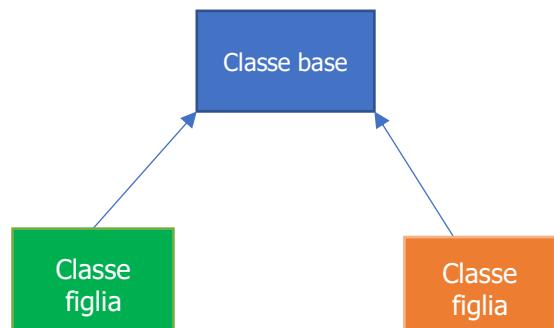
Quando si ridefinisce un metodo ereditato dalla classe base, in generale non è possibile modificare il tipo di ritorno. **Se il tipo di ritorno è una classe, il metodo ridefinito può restituire una qualsiasi delle sue classi derivate.** Il tipo di ritorno così modificato prende il nome di tipo di ritorno covariante.

È importante notare che la ridefinizione del tipo di ritorno non introduce un tipo di ritorno più restrittivo del tipo di ritorno dal metodo dichiarato nella classe base.

È possibile, inoltre, *cambiare un modificatore d'accesso da private a public, non è possibile cambiarlo. Da public a private.*

## Integrazione nell'UML

Per implementare l'ereditarietà nell'UML, dobbiamo usare frecce che partono dalla classe derivata alla classe base; queste frecce rappresentano una relazione *is-a*.



## Uso delle variabili di istanza private della classe base

Il principio in base al quale dalla definizione di un metodo di una classe derivata non sia possibile accedere a una variabile di istanza privata di una classe base potrebbe apparire errato. L'accesso alle variabili di istanza e ai metodi privati deve rispettare le regole precedentemente descritte, altrimenti la correttezza del programma potrebbe essere compromessa. Se una variabile di istanza privata di una classe fosse accessibile dalla definizione di un metodo di una classe derivata, ogniqualvolta si volesse accedere a una variabile di istanza privata basterebbe creare una classe derivata e accedervi da un metodo di questa classe. Questo vorrebbe dire rendere accessibili le variabili di istanza private definite in una classe.

## I metodi privati

I metodi privati di una classe base si comportano come le variabili di istanza private: sono del tutto indisponibili. I metodi privati di una classe base possono essere *indirettamente disponibili* da metodi nella classe derivata, per esempio se un metodo privato viene usato nella definizione di un metodo pubblico della classe base. I metodi privati, quindi, dovrebbero essere usati come **metodi di supporto** ad altri metodi, quindi il loro utilizzo dovrebbe essere limitato alla classe nella quale sono definiti.

## Modalità d'accesso protected

Un metodo dichiarato con modificatore d'accesso *protected* è accessibile per nome dalla classe alla quale appartiene, dalle classi derivate dalla classe a cui appartiene e da qualsiasi classe contenuta nello stesso package della classe alla quale appartiene. **I metodi e le variabili di istanza protected non sono invece accessibili per nome da qualsiasi altra classe che non appartenga alla casistica appena riportata.** Se ha questo modificatore "protected", allora è accessibile per nome dalla classe cui appartiene, dalle classi derivate dalla classe cui appartiene e da qualsiasi classe nello stesso package della classe cui appartiene.

## Costruttori di una classe derivata

Una classe derivata ha i suoi costruttori, e non eredita alcun costruttore dalla classe base. Nella definizione di un costruttore per la classe derivata, la prima tipica azione è di **invocare un costruttore della classe base**:

```
super(parameters)
```

Questo costruttore utilizza la parola riservata `super` come un nome di metodo per invocare un costruttore della classe base.

L'uso di questa keyword implica alcuni dettagli: `super` deve essere sempre la prima azione specificata nella definizione di un costruttore non può essere specificato più avanti nella definizione del costruttore. Se in ogni costruttore della classe derivata non si include un'invocazione esplicita al costruttore della classe base, Java includerà automaticamente un'invocazione al costruttore di default della classe base.

## Uso di metodi ridefiniti

Nella definizione di un metodo di una classe derivata, si può invocare un metodo ridefinito della classe base, facendolo precedere da `super` e un punto.

```
Super.redefined_method(parameters) //chiama il metodo della classe base
```

occupa una posizione diversa dalla classe base, e non è possibile l'ereditarietà multipla.

Se la classe C deriva dalla classe B la quale, a sua volta, deriva dalla classe A, allora un oggetto di classe C è di tipo C, ma anche di tipo B e di tipo A. Questo funziona per ogni catena di classi derivate, indipendentemente dalla sua lunghezza.

Poichè un oggetto di una classe derivata ha i tipi di tutte le classi dei suoi antenati in aggiunta al suo tipo, si può assegnare un oggetto di una classe a una variabile di ogni tipo di antenato, ma non il contrario.

## La classe object

In Java, **ogni classe deriva dalla classe Object**. Se una classe C ha come classe la classe B, quest'ultima è derivata da `Object`: quindi anche C deriva da `Object`. Pertanto, ogni oggetto di ogni classe è di tipo `Object`, così come è del tipo della sua classe e di tutte le sue classi antenate.

Anche tutte le classi che si definiscono senza utilizzare l'ereditarietà sono discendenti dalla classe `Object`.

Questa classe consente di scrivere metodi Java che hanno parametri di tipo `Object`; in questo modo sarà possibile passare come argomenti oggetti di qualsiasi tipo classe. La classe `Object` ha alcuni metodi che sono ereditati da ogni classe Java, come `equals` e `ToString`.

Un altro metodo ereditato dalla classe `Object` è il metodo `clone`. Questo metodo non ha argomenti, e

restituisce una copia dell'oggetto chiamante. Un clone è **un oggetto che ha dati identici a quelli dell'oggetto che ha invocato il metodo**. Come gli altri metodi ereditati dalla classe `Object`, il metodo `clone` deve essere ridefinito perché possa funzionare correttamente nella classe derivata.

## Polimorfismo

Il **polimorfismo** permette di modificare la definizione dei metodi nella classe derivata, e far sì che questi cambiamenti siano effettivi anche per il codice della classe base. Quindi, effettivamente, permette di assegnare più significati a uno stesso nome di metodo.

## Il binding

Il **binding** indica il processo con cui l'invocazione di un metodo viene associata a una definizione specifica del metodo. Si definisce *binding statico* l'associazione tra invocazione e definizione di metodo, che viene prodotta al momento della compilazione del codice.

Se l'associazione tra invocazione e definizione di metodo viene prodotta quando il metodo è invocato a runtime, si parla invece di *binding dinamico*, che è quello che Java usa per tutti i metodi.

Java **non utilizza il binding dinamico** per i metodi *privati*, *final* e i metodi *statici*. L'assenza di esso per i metodi statici può essere significativa quando il metodo statico viene invocato utilizzando un oggetto chiamante. Nel caso del binding statico, *la decisione su quale definizione di metodo debba essere eseguita*

*viene presa durante la compilazione.* Un metodo statico viene normalmente invocato utilizzando un nome di classe e non un oggetto chiamante. Se si invoca un metodo statico dalla definizione di un metodo non statico senza utilizzare una classe o un oggetto chiamante, l'oggetto diventa implicitamente **this**.

## 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:

```
public final void method() /class {
```

È possibile anche usare il modificatore final per le classi; con questo si indica che **non possono essere usate come classi base**.

Se un metodo è dichiarato come final, il compilatore può usare il binding statico per migliorare l'efficienza dell'applicazione.

Il tipo di una variabile determina quali metodi possono essere invocati usando la variabile, ma l'oggetto referenziato dalla variabile determina **l'esatto definizione di metodo da eseguire**. Il tipo di parametro determina quali metodi possono essere invocati utilizzando il parametro, ma l'argomento determina l'esatto definizione del metodo da eseguire.

## Upcast e downcast

Assegnare *un oggetto di una classe derivata a una variabile del tipo della classe base* è chiamato **upcast**, perché è come fare la conversione dal tipo derivato al tipo della classe base, e i diagrammi che mostrano le relazioni di ereditarietà riportano le classi base sopra le classi derivate. Non ha nessun caso insidioso.

*La conversione da una classe base a una classe derivata* viene chiamata **downcast**. Questo può presentare alcuni problemi, come per esempio in alcuni casi non aver senso; infatti, il compilatore non fa alcun tipo di controllo su questo. Per verificare se è legale effettuare un downcast, è possibile usare l'**operatore instanceof**, che controlla se un oggetto è del tipo specificato.

```
Object instanceof className
```

Il downcast funziona se l'oggetto che deve essere convertito è di quel tipo; quindi, questo controllo può essere effettuato con questo operatore.

## La classe abstract

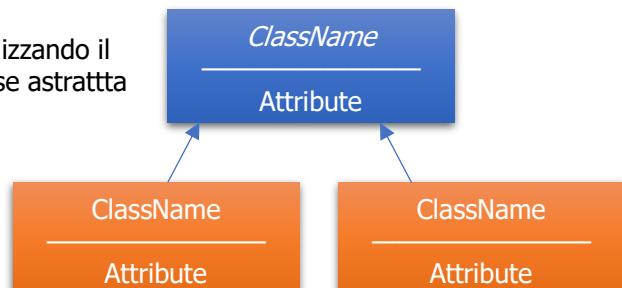
Una classe astratta è una classe che **ha alcuni metodi non completamente definiti**. Non è possibile creare nuovi oggetti utilizzando il costruttore di una classe astratta, ma è possibile utilizzare una classe astratta come classe base per la definizione di classi derivate.

Invece di fornire una definizione inventata di un metodo che si pensa di ridefinire in una classe derivata, si può dichiarare un metodo astratto:

```
public abstract void abstractMethod() {
```

Eseguire questo significa **posticipare la sua definizione al momento in cui si saprà effettivamente come definirla**. Un metodo deve essere ridefinito da ogni classe derivata dalla classe base astratta (questo vale se la classe derivata sa come definirlo). Inoltre, non si possono creare oggetti per quella classe, ma può essere soltanto come classe base per derivare altre classi. Inoltre, **è possibile inserire invocazioni di un metodo astratto all'interno di altri metodi**; grazie al binding dinamico, verrà effettivamente eseguito il metodo concreto che ha ridefinito quello astratto:

```
public abstract type name(parameter);
```



La classe è effettivamente un **tipo**. La decisione riguardo a quale definizione di metodo utilizzare dipende dalla posizione dell'oggetto nella catena di ereditarietà, e non dal tipo della variabile dell'oggetto. Nel caso in cui non trovasse la definizione del metodo invocato nella classe con cui è stato istanziato l'oggetto, Java **risale la catena di ereditarietà a partire dalla classe con cui è stato istanziato l'oggetto invocante il metodo**, finché non trova in qualche classe antenata il metodo invocato. La cercherà fino alla classe Object e, se non trovato, producerà un messaggio di errore.

Questo può essere utile per **imporre** l'implementazione di un metodo comune a tutte le sotto-classi di cui non si può dare l'implementazione nella super-classe.

## Interfacce

L'interfaccia di una classe è costituita dalle intestazioni dei metodi pubblici e dalle costanti pubbliche della classe, insieme con alcuni commenti esplicativi. Conoscendo solo l'interfaccia di una classe, un programmatore può scrivere un programma che **utilizza la classe senza sapere nulla dell'implementazione della classe stessa**; Java consente di separare l'interfaccia di una classe dalla sua implementazione, realizzando due file sorgenti distinti. Il concetto di interfaccia di classe si traduce in una interfaccia Java.

È un componente di un programma che **contiene le intestazioni di un certo numero di metodi**. Un'interfaccia può anche definire delle costanti pubbliche, e potrebbe includere i commenti che descrivono i metodi, cosicché un programmatore avrà tutte le informazioni necessarie per implementarli. Inizia con una definizione di classe:

```
public interface interfaceName{
```

Quando si scrive una classe che definisce i metodi dichiarati in un'interfaccia, si dice che **la classe implementa l'interfaccia**; questa deve definire un corpo per ogni metodo specificato nell'interfaccia. Se non definisce il corpo di tutti i metodi, deve essere dichiarata astratta. La classe potrebbe anche definire metodi non dichiarati nell'interfaccia. In generale, bisogna:

1. Includere l'espressione

```
implements interfaceName
```

all'inizio della definizione di classe. Per implementare più di un'interfaccia, è sufficiente elencare il nome di tutte le interfacce.

2. Definire ogni metodo dichiarato nell'interfaccia per creare una classe concreta

## Interfaccia come tipo

Un'interfaccia è un tipo riferimento. Così, si può scrivere un metodo che ha un parametro di un tipo d'interfaccia. Una variabile di un tipo d'interfaccia può far riferimento a un oggetto di una classe che implementa l'interfaccia.

Occorre prestare attenzione ai metodi che si invocano: se non sono presenti nell'interfaccia, il compilatore genererà un errore. Il tipo di una variabile determina i nomi dei metodi che possono essere utilizzati, ma è il **tipo dell'oggetto referenziato dalla variabile che determina quale definizioni dei metodi saranno utilizzate**. Il binding dinamico si applica alle interfacce esattamente come alle classi.

## Estendere le interfacce

È possibile definire una nuova interfaccia che **estende un'interfaccia già esistente**, utilizzando una sorta di ereditarietà. Si può creare quindi un'interfaccia formata da un insieme di metodi, ovvero quelli definiti dall'interfaccia e dalla classe.

```
public interface nomeInterface extends anotherInterface {
```

## **Eccezioni**

Per ora, le istruzioni scritte sono considerate **procedure parziali**, perché prevedono un comportamento specificato solo per uno specifico sotto-insieme di dominio. Scrivere procedure di questo tipo **compromette la robustezza** del codice, e si parte dal presupposto che il chiamante usi sempre un valore legale. Per gestire questo, abbiamo tre soluzioni:

- Usare un'istruzione per terminare è **troppo drastica**, ed è una scelta che spetta al chiamante e non al chiamato

*E.g. if (value) System.exit(-1);*

- Usare un metodo dove usiamo valori di ritorno convenzionali può **non essere fattibile**, poco informativo e condiziona il chiamante a controllare la validità del valore ritornato.

Es public boolean methodName (Parameter){}

- ### - Usare le **eccezioni**

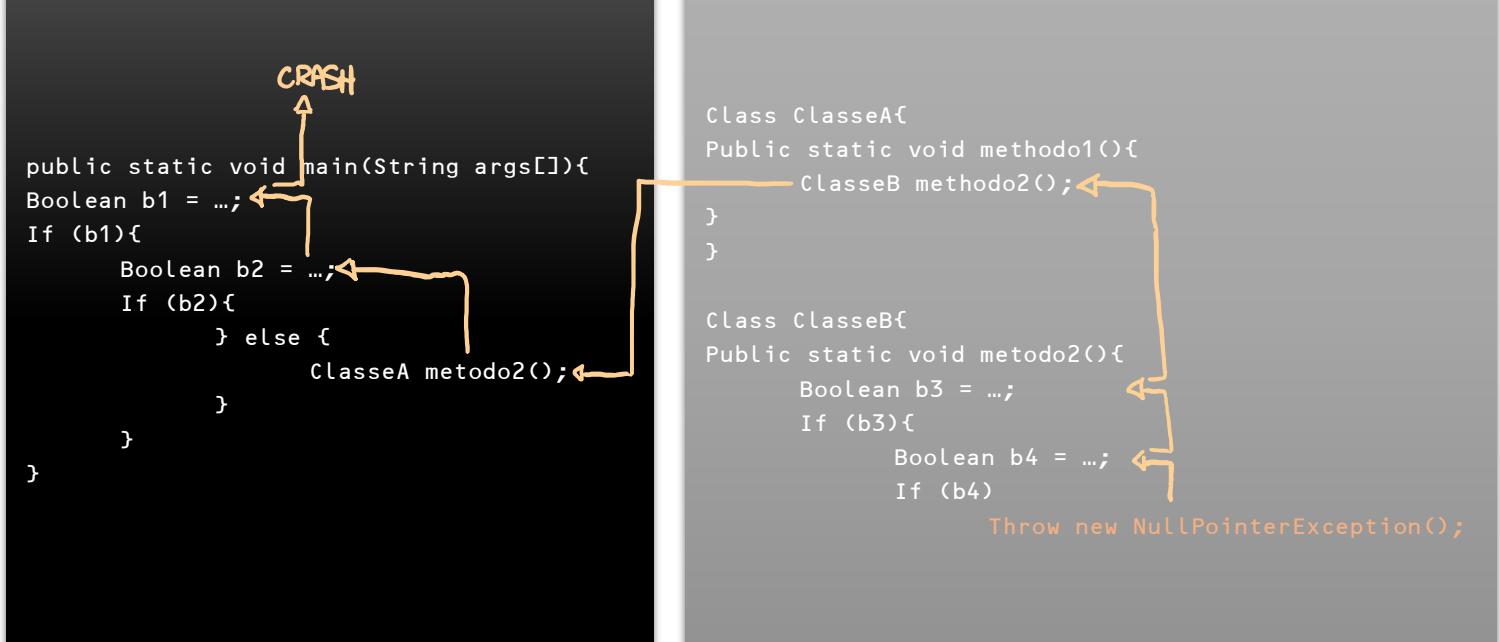
Se una soluzione finisce con un errore, viene segnato al chiamante con un'**eccezione**. Esse sono degli oggetti, e come tali hanno *un tipo e dei dati associati*. Essenzialmente, è un oggetto che segnala l'accadere di un evento anomalo durante l'esecuzione di un programma. Il processo di creazione di questo oggetto è chiamato **lancio di un'eccezione**, mentre in un'altra parte del programma si include per **gestirla**. Per poterlo invocare, abbiamo bisogno della keyword **throw**, seguito da un *reference a un oggetto eccezione* che:

- Ha metodi e dati
  - Determina il tipo dell'eccezione
  - Metodi e dati includono informazioni sul problema riscontrato

`instruction () throw new ExceptionName();`

keyword → reference ad exception → Exception: generico  
eccezioni specifiche

l'esecuzione al chiamante. Ecco qua un esempio:



## Try-catch

Per poter gestire del codice che permette di **catturare e gestire un'eccezione**, possiamo usare il costrutto **try-catch**:

```
try{
    code that can generate an exception
} catch (ExceptionType e){
    Code that manages the exception
} catch (ExceptionType e2){
    yet another code that manages another exception
} catch (Exception e){
    throw new AnotherException();
} finally {
    Code that gets ran every time
}
```

Il codice all'interno del try sarà quello da gestire; se non viene gestita un'eccezione, allora il compilatore **risalirà il programma fino a quando non trova un blocco try-catch o l'esecuzione termina.**

Si possono usare molteplici blocchi catch per **gestire più esecuzioni**.

Il tipo **exception** serve per poter gestire qualsiasi tipo di eccezione, senza dover specificare.

Il pezzo **finally** specifica un ramo che deve essere eseguito sia con un'eccezione che senza.

Possiamo anche generare un'eccezione con un tipo, per poi sollevarne una di un tipo diverso (**re-throwing**).

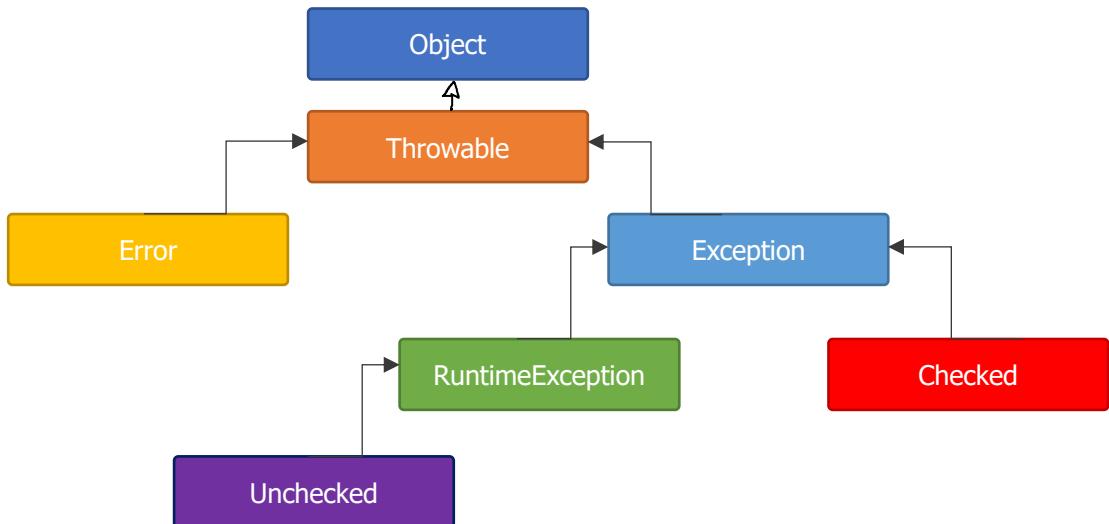
## L'oggetto exception

**L'Exception** rappresenta una qualsiasi eccezione, ed è il tipo generale. Altre eccezioni sono *sottoclassi di Exception*:

- **Object**: generico
  - o **Throwable**: rappresenta *tutti i possibili segnali di errori che un programma può evocare*
    - **Error**: errori che non dovrebbero accadere, **generati dalla JVM** e provocano sempre il crash
    - **Exception**: condizioni eccezionali che possono essere gestite dall'applicazione
      - **RuntimeException**: eccezioni generate a runtime
        - o **Unchecked exceptions**: errori di programmazione (es. *IndexOutOfBoundsException*), sono sollevate dalla JVM e possono essere gestite dal programma. Possiamo anche catturare del codice con un'eccezione unchecked, ma il **compilatore non ce lo chiede mai di fare**.
    - **Checked exceptions**: generate dalla JVM, se l'eccezione non la gestiamo all'interno del metodo che la può creare, allora *può ritornare al chiamante*. (Es. *DataFormatException*) In questo caso, deve essere **dichiarata all'interno della firma del metodo**.

```
public void method() throws ExceptionName{ ... }
try {
    method();
} catch (ExceptionName e){ ... }
```

Se vogliamo chiamare un metodo con un'eccezione, saremmo obbligati a **scrivere un blocco di codice per gestire l'errore** ogni volta che vogliamo invocarlo. Può essere anche "rimandato indietro".



Le eccezioni pre-definite sono utili per numerosi casi comuni, ma non per tutti. Possiamo quindi implementare una **classe che estende un elemento della gerarchia delle eccezioni**; questa sarà una sottoclasse, che estende la classe Exception.

#### Classe:

```

public class ExceptionName extends Exception{
    public ExceptionName(){
        Super();
    }

    Public ExceptionName(String s){
        Super(s);
    }
}
  
```

exception normale:  
messaggio di errore  
di sistema  
S: messaggio di  
errore personalizzato  
+ di sistema

#### Uso:

```

throw new ExceptionName();
throw new ExceptionName(String s);
}
  
```

#### Gestione:

```

Try {
    ...
} catch (ExceptionName e) {
}
  
```

Possiamo definire due costruttori: uno **vuoto** che permette di semplicemente generare l'exception, l'altro che **grazie all'uso di una stringa possiamo mandare un messaggio di errore personalizzato**, che poi verrà mostrato all'utente, utile per la diagnosi.

## Metodo getMessage

Ogni oggetto eccezione ha una variabile di istanza di tipo String che contiene un messaggio. Tale messaggio solitamente identifica la ragione per cui è stata generata l'eccezione.

Il valore della variabile di istanza di tipo String è *argumentString*. Il metodo *getMessage()* restituisce questa stringa. In tutte le classi di eccezioni predefinite, il metodo *getMessage* restituisce la stringa che è stata passata come argomento al costruttore. Se al costruttore non è stato passato alcun argomento, *getMessage* restituisce una stringa di default; **si dovrebbe preservare il comportamento del metodo getMessage in ogni classe di eccezione che viene definita**. Per assicurarsi che le classi di eccezioni personalizzate che vengono definite si comportino in questo modo, ci si deve preoccupare che abbiano un costruttore con un parametro di tipo stringa la cui definizione inizi con una chiamata a **super**:

```

public Exception(String message){
    super(messaggio);
    //other code
}

```

Se il costruttore della classe base gestisce correttamente il messaggio, si è sicuri che lo faranno anche tutte le classi definite in questo modo. Si dovrebbe sempre includere un costruttore di default in ogni classe di eccezione.

Sebbene sia possibile usare la classe Exception in un blocco catch, è più utile catturare eccezioni più specifiche, come IOException (nota: per IOException, è bene importare java.io). La maggior parte delle eccezioni presentate non hanno bisogno di essere importate, in quanto sono presenti nel package *java.lang*. Alcune, però, sono contenute in package differenti e necessitano di essere importate.

## Strutture dati

Una struttura dati concatenata consiste di blocchi di dati, denominati **nodi**, connessi tra loro tramite dei **collegamenti**, che possono essere visualizzati come frecce, e interpretati come passaggi a senso unico da un nodo a un altro. Il tipo più semplice di struttura dati concatenata consiste in una singola catena di nodi, chiamata **lista concatenata**.

I nodi sono realizzati come oggetti di una classe nodo, i collegamenti sono realizzati sotto forma di riferimenti, ovvero variabili di istanza del tipo della classe nodo stessa.

### Lista concatenata

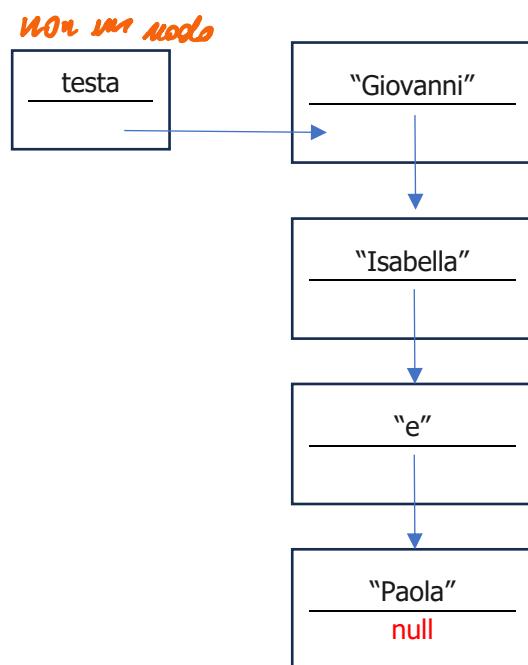
Una **lista concatenata** è una struttura dinamica che collega l'uno all'altro gli elementi di una lista. È sempre composta da oggetti noti come **nodi**; in un UML, sono rappresentati come rettangoli divisi da una linea orizzontale, con una parte i dati e nell'altra il collegamento a un altro nodo. Il collegamento *testa* non è nella lista dei nodi, poiché non è un nodo, ma un collegamento che punta al primo nodo.  
Un'implementazione della classe nodo potrebbe essere:

```

public class NodoLista {
    private String dati;
    private NodoLista collegamento;
}

```

La relazione **NodoLista** all'interno della classe **NodoLista** sembra essere circolare, e lo può essere. Ogni oggetto nodo di una lista concatenata contiene nella propria variabile di istanza *collegamento* un riferimento a un altro oggetto di questa classe.



Il codice deve essere in grado di collocarsi sul primo nodo, e poi di scoprire quando raggiunge l'ultimo nodo. La variabile contenente un riferimento al primo nodo è rappresentata da un rettangolo etichettato con **testa**, che punta al primo nodo di testa. Si indica invece la fine di una lista usando come collegamento dell'ultimo nodo un "**null**"; il programma può verificare se un nodo è l'ultimo verificando questo collegamento. Per fare sì in modo che una lista sia vuota, si assegna a *testa* il valore *null*.

## I metodi

Una lista concatenata ha una serie di metodi che possono essere utili:

```
public class ListaConcatenataDiStringhe{
    private NodoLista testa;

    public ListaConcatenataDiStringhe(){
        testa = null;
    }

    Public void mostraLista(){
        NodoLista posizione = testa;
        while (posizione != null){
            System.out.println(posizione.getDati());
            posizione = posizione.getCollegamento();
        }
    }
}
```

Analizzando di più questo metodo, la variabile *posizione* contiene un riferimento a un nodo. Inizialmente, questo contiene lo stesso riferimento della variabile di istanza *testa*, e facendo così ci si inizia a posizionare sul primo nodo. Dopo aver visualizzato i dati contenuti in un nodo, passa al successivo grazie a **getCollegamento**, che sposta la variabile *posizione* al nodo successivo. Quando la lista finisce, si interrompe.

```
Public int lunghezza(){
    Int conteggio = 0;
    NodoLista posizione = testa;
    while (posizione != null){
        conteggio++;
        posizione = posizione.getCollegamento();
    }
    Return conteggio;
}
```

Anche questo metodo usa un ciclo simile a quello precedente, ma semplicemente conta i nodi. Un procedimento che guarda tutti i nodi di una lista è detto che **attraversa la lista**.

```
Public void aggiungiNodoInTesta(String datiDaAggiungere){
    testa = new NodoLista(datiDaAggiungere, testa);
}
```

Questo metodo aggiunge un nuovo nodo all'inizio della lista concatenata, cosicchè *il nuovo nodo diventi il primo della lista*. Per collegare questo nuovo nodo al resto della lista, bisogna assegnare la variabile *collegamento* di questo nuovo nodo il riferimento del vecchio primo nodo, però tale istruzione viene automaticamente effettuata dal **costruttore**.

```
Public void eliminaNodoDiTesta(){
    If (testa != null)
        Testa = testa.getCollegamento();
    else {
        System.out.println("Si sta eliminando da una lista vuota.");
        System.exit(0);
    }
}
```

```

    }
}

```

Questo metodo rimuove il primo nodo dalla lista concatenata, e fa sì che la variabile testa faccia riferimento a quello che prima era il secondo nodo della lista concatenata.

In questo modo, il nodo non farà più parte della lista, ma visto che non è stato dato alcun comando per la distruzione del nodo in sé, persiste comunque nella **memoria del computer**. Se non ci sono altri riferimenti al nodo cancellato, Java effettua automaticamente il **garbage collection**.

```

Public NodoLista trova(String elemento){
    Boolean trovato = false;
    NodoLista posizione = testa;
    while ((posizione != null) && !trovato){
        String datiAllaPosizione = posizione.getDati();
        if (datiAllaPosizione.equals(elemento))
            trovato = true;
        else
            posizione = posizione.getCollegamento();
    }
    Return posizione;
}

```

Questo metodo innanzitutto inizializza la variabile **posizione** con il primo nodo della lista concatenata; finchè la variabile posizione non è nulla, e non ha trovato niente, allora prende i dati del nodo e li assegna a datiAllaPosizione. Se i dati alla posizione sono uguali all'elemento cercato, allora è stato trovato; altrimenti, imposta la variabile posizione al nodo successivo nella lista concatenata.

Con questi metodi dobbiamo stare attenti a un possibile **privacy leak**, poiché grazie ai metodi getCollegamento possiamo ottenere riferimenti a un oggetto privato. Inoltre, NodoLista avrebbe metodi set pubblici, il che genera decisamente una privacy leak. Il metodo getDati invece non causa una privacy leak, solo perché *la classe String non ha metodi set*, e nessuno dei metodi pubblici nella classe restituisce un riferimento a un nodo.

## Classi nodo come inner class

Sfruttando le inner class, possiamo definire le classi nodo in questa maniera.

```

public class ListaConcatenataDiStringhe {

    private NodoLista testa;
    <other methods>

    private class NodoLista { estendo inner e private, soltanto la classe esterna
        private String dati;
        private NodoLista collegamento;
ha accesso a metodi e attributi di questa classe
↓
inner ha accesso ai metodi delle external
        public NodoLista(){
            collegamento = null;
            dati = null;
        }

        public NodoLista (String valoreDati, NodoLista valoreCollegamento) {

```

```

        dati = valoreDati;
        collegamento = valoreCollegamento;
    }

    public void setDati(String nuoviDati){
        dati = nuoviDati;
    }

    public String getDati(){
        return dati;
    }

    public void setCollegamento(NodoLista nuovoCollegamento){
        collegamento = nuovoCollegamento;
    }

    public NodoLista getCollegamento(){
        return collegamento;
    }
}

```

Usando questo metodo, **si risolve la privacy leak**, poiché i metodi del nodo saranno accessibili soltanto dalla classe Lista. Inoltre, se non si intende usare una inner class altrove, dovrebbe essere resa privata, come in questo caso, ed è anche più sicuro.

Facendo così, possiamo anche semplificare i metodi get e set, *effettuando un accesso diretto alle variabili*.

## Iteratore

Un **iteratore** è una variabile che consente di visitare la lista. Per esempio, può essere usato per gli array, dove in questo caso l'iteratore sarà una variabile di tipo int.

Prendiamo come esempio:

```
for (indice = 0; indice < unArray.length; indice++)
```

In questo caso, la variabile intera *indice* è l'iteratore, e si può iterare appunto mandandolo avanti di uno. Se si crea un array di tutti gli oggetti presenti in una lista concatenata, si può iterare l'array. Questo però non è sufficiente se si vuole qualcosa che consenta anche di effettuare operazioni.

Possiamo quindi usare una qualsiasi variabile di istanza come iteratore, il chè ci permette di definire ulteriori metodi che vogliamo usare. Possiamo per esempio anche impostare un riferimento per il nodo corrente, successivo e precedente.

Se si implementa un iteratore per una lista concatenata, allora il programma ha un **modo per far riferimento a ogni nodo**; una variabile di istanza di, per esempio, *corrente*, può mantenere un riferimento a un nodo, il quale è conosciuto come **nodo all'iteratore**.

Se è definito all'interno della lista concatenata, allora è conosciuto come **iteratore interno**, mentre se è definito fuori è semplicemente conosciuto come **iteratore esterno**.

## Interfaccia Iterator

Java fornisce un'interfaccia Iterator, che è quella che usa per considerare formalmente ogni iteratore. Ha tre metodi:

- **HasNext** restituisce "vero" se l'iterazione ha un altro elemento da restituire

- **Next** restituisce il successivo elemento nell'iterazione
- **Remove** rimuove dalla collezione l'elemento restituito dall'ultima invocazione *next*.

## Varianti delle liste concatenate

### Liste concatenate doppie

Una lista concatenata ordinaria può essere percorsa in un'unica direzione, mentre una variante **doppia** ha, per ogni nodo, un collegamento al nodo successivo e al nodo precedente. Non sarà quindi più necessario usare una variabile di istanza per mantenere il nodo corrente.

Ovviamente, questa implementazione ha bisogno delle seguenti modifiche:

- Implementare due collegamenti ai nodi anziché uno
- Anche i costruttori avranno bisogno di due collegamenti anziché uno
- Per aggiungere un nuovo nodo all'inizio della lista, bisogna modificare i riferimenti a due nodi. È opportuno impostare inoltre il collegamento precedente del vecchio nodo di testa al nuovo nodo, e **assicurarsi che testa non sia null**.
- Per cancellare un nodo, bisogna ricordarsi che deve essere cancellato da entrambi i lati.

Una sintassi di una classe per implementare questa struttura dati potrebbe essere la seguente:

```
public NodoListaDoppia(String nuoviDati, NodoListaDoppia nodoPrecedente,
NodoListaDoppia nodoSuccessivo){
    dati = nuoviDati;
    successivo = nodoSuccessivo;
    precedente = nodoPrecedente;
}
```

## Pile

Una pila non è necessariamente una struttura dati concatenata, ma può essere implementata per mezzo di una lista concatenata. Una **pila** è una struttura dati nella quale gli elementi **vengono rimossi in ordine inverso rispetto a quello nel quale sono stati aggiunti**. È detta una struttura **last in/first out**. Un'implementazione potrebbe essere:

```
public class Pila {
    private NodoLista testa;

    public Pila(){
        testa = null;
    }

    public void push(String nuoviDati){
        testa = new NodoLista(nuoviDati, testa);
    }

    public String pop(){
        String dati = "";
        if (testa != null){
            dati = testa.getDati();
            testa = testa.getCollegamento();
        }
    }
}
```

```

        return dati;
    }

    public boolean vuota(){
        return (testa == null);
    }
}

```

## Code

Una coda è una struttura dati nella quale i dati sono gestiti in modo che il primo elemento inserito sia anche il primo a essere rimosso, con un sistema **first in/first out**. Una coda richiede di tenere traccia sia della testa che della fine della lista, perché qualunque azione potrebbe coinvolgere l'una o l'altra delle posizioni.

```

public class Coda {
    private NodoLista inizio;
    private NodoLista fine;

    public Coda(){
        inizio = null;
        fine = null;
    }

    public void aggiungi(String nuoviDati){
        NodoLista nuovaFine = new NodoLista(nuoviDati, null);
        if (fine != null){
            fine.setCollegamento(nuovaFine);
            fine = nuovaFine;
        } else {
            inizio = nuovaFine;
            fine = nuovaFine;
        }
    }

    public String prossimo(){
        If (inizio != null) return inizio.getDati();
        else return null;
    }

    public Boolean rimuovi(){
        if (inizio != null){
            inizio = inizio.getCollegamento();
            return true;
        } else {
            return false;
        }
    }

    public Boolean vuota() {
        return (inizio == null);
    }
}

```

```

    }

    public void cancella() {
        inizio = null;
        fine = null;
    }
}

```

## Tabelle hash

Una tabella di hash, o *mappa di hash*, è una struttura dati che immagazzina e recupera dati in memoria in maniera ufficiente. Usa una combinazione di un array, e di una lista concatenata ordinaria. Mentre la ricerca di un elemento in una lista concatenata richiede in genere un tempo proporzionale alla lunghezza della lista, una tabella di hash può eseguire la ricerca in un **numero fissato di operazioni**, indipendentemente dalla quantità di dati in essa contenuti.

Per immagazzinare un elemento in una tabella di hash si assegna una **chiave**; nota la chiave, è possibile recuperare l'elemento. La chiave individua univocamente l'elemento associato, e se un elemento non fornisce intrinsecamente una chiave univoca, si può utilizzare una *funzione di hash* per generare una.

Per cercare un elemento da una tabella di hash, per prima cosa si calcola il suo valore di hash. Successivamente, si cerca con una ricerca sequenziale l'elemento nella lista concatenata presente nella posizione dell'arrabbiata di indice pari al valore di hash ottenuto. Se l'elemento non viene trovato in questa lista concatenata, allora non è presente nella tabella di hash.

Un modo semplice per calcolare un valore di hash numerico per una stringa è quello di **sommare i codici ASCII di ogni carattere della stringa**, e calcolare il resto della divisione della somma per la lunghezza dell'array. Un esempio può essere

```

private int calcolaHash(String s) {
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
        hash += s.charAt(i);
    return hash % dimensione;
}

```

In questo esempio, si calcola inizialmente un valore non limitato, cioè la somma dei codici ASCII dei caratteri nella stringa. L'array può contenere solo un numero finito di sistemi; per ricondurre la somma a un valore minore della lunghezza dell'array, si calcola il resto della divisione della somma per la lunghezza dell'array.

Le prestazioni peggiori si avranno se tutti gli elementi inseriti vengono **associati alla stessa chiave**. Tutti gli elementi saranno immagazzinati in un'unica lista, e quindi l'operazione di ricerca richiederà un tempo proporzionale al numero di elementi nella tabella; se gli elementi da inserire hanno una distribuzione in qualche modo casuale, sarà molto improbabile che vengano associati tutti alla stessa chiave. Le prestazioni migliori si avrebbero se **ogni elemento venisse associato a una chiave diversa**, senza collisioni e l'operazione di ricerca richiederebbe sempre lo stesso tempo.

## Insiemi

Un insieme è una collezione di elementi dei quali si ignorano ordine e molteplicità. Un modo semplice di implementare un insieme consiste in una variante di una *lista concatenata*.

La classe Nodo è una **inner class**; le operazioni *nellInsieme*, *mostraInsieme* e *numeroElementi* sono virtualmente identiche alle operazioni corrispondenti per una lista concatenata. Il metodo **aggiungi** sostituisce *aggiungiNodoInTesta*, e deve essere modificato per impedire che vengano inseriti elementi duplicati.

```

public class Insieme<T>{
    private class Nodo<T>{
        private T dati;
        private Nodo<T> collegamento;

        public Nodo(){
            dati = null;
            collegamento = null;
        }

        public Nodo(T nuoviDati, Nodo<T>, valoreCollegamento) {
            dati = nuoviDati;
            collegamenti = valoreCollegamento;
        }
    }

    Private Nodo<T> testa;
    public Insieme(){
        testa = null;
    }

    public boolean aggiungi(T nuovoElemento){
        if (!nellInsieme(nuovoElemento)){
            testa = new Nodo<T>(nuovoElemento, testa);
            return true;
        }
    }
}

```

L'aggiunta di un elemento inserisce un nuovo nodo all'inizio della lista, il che richiede un numero costante di operazioni, indipendentemente dal numero di elementi nell'insieme; l'esecuzione del metodo *aggiungi* richiederà però un tempo proporzionale al numero di elementi.

```

public Insieme<T> unione(Insieme<T> altroInsieme){
    Insieme<T> insiemeUnione = new Insieme<T>();
    Nodo<T> posizione = testa;
    while (posizione != null) {
        insiemeUnione = aggiungi(posizione.dati);
        posizione = posizione.collegamento;
    }

    posizione = altroInsieme.testa;
    while (posizione != null) {
        insiemeUnione.aggiungi(posizione.dati);
        posizione = posizione.collegamento;
    }
    return insiemeUnione;
}

```

Il metodo unione su due insiemi itera sugli elementi di entrambi gli insiemi aggiungendo ognuno di essi. Se i due insiemi contengono rispettivamente n ed m elementi, si avranno in totale (n+m) chiamate. Cresce approssimativamente come  $(n+m)^2$ .

```
public Insieme<T> intersezione(Insieme<T> altroInsieme){
    Insieme<T> insiemeIntersezione = new Insieme<T>();
    Nodo<T> posizione = testa;
    while (posizione != null) {
        if (altroInsieme.nellInsieme(posizione.dati))
            insiemeIntersezione.aggiungi(posizione.dati);
        posizione = posizione.collegamento;
    }
    return insiemeIntersezione;
}
```

Un approccio diverso alla gestione degli insiemi usando tabelle di hash potrebbe consentire di ottenere un metodo *intersezione* che richieda solo un tempo proporzionale alla somma dei numeri di elementi dei due insiemi.

## Alberi

Un albero è un esempio di struttura dati concatenata di tipo più complesso. In un albero, **qualsiasi nodo può essere raggiunto partendo dal nodo radice**, lungo un percorso costituito da una sequenza di collegamenti; non sono ammessi percorsi chiusi, però. Seguendo i collegamenti, si arriverà quindi a una fine – **ogni nodo ha due collegamenti** a altri nodi, e in questo caso è chiamato **albero binario**, perché da ogni nodo partono esattamente due collegamenti. Sono possibili anche altri tipi di alberi con numeri diversi di collegamenti, ma quello binario è il tipo più comune.

La variabile di istanza **radice** ha un ruolo analogo a quello della variabile *testa* in una lista concatenata; il nodo il cui riferimento è contenuto nella variabile “radice” è detto nodo radice.

I nodi alla fine dei rami, che hanno entrambe le variabili di istanza che rappresentano i collegamenti impostati a null sono detti **nodi foglia**. Analogamente a una lista collegata, se un albero è nullo allora la radice è impostata a null.

Un albero binario ha una **struttura ricorsiva**; ogni albero contiene due sottoalberi che hanno come radici i nodi referenziati rispettivamente dalle variabili *collegamentoSinistro* e *collegamentoDestro* del nodo radice.

## Ricerca in un albero

Per ricercare in un albero, abbiamo tre ordini:

<b>Preorder</b>	<b>Inorder</b>	<b>Postorder</b>
1. Si elaborano i dati nel nodo radice	1. Si elabora il sottoalbero sinistro	1. Si elabora il sottoalbero sinistro
2. Si elabora il sottoalbero sinistro	2. Si elaborano i dati nel nodo radice	2. Si elabora il sottoalbero destro
3. Si elabora il sottoalbero destro	3. Si elabora il sottoalbero destro	3. Si elaborano i dati nel nodo radice

Esiste un'altra regola di immagazzinamento per la ricerca in alberi binari, che è così:

1. Tutti i valori nel sottoalbero sinistro sono minori del valore nel nodo radice
2. Tutti i valori nel sottoalbero destro sono maggiori o uguali al valore nel nodo radice
3. La regola si applica ricorsivamente a ognuno dei due sottoalberi

Un albero che soddisfa la regola di immagazzinamento per la ricerca in alberi binari è detto **albero di ricerca binaria**; se un albero soddisfa questa regola e se ne visualizzano gli elementi seguendo l'approccio

dell'elaborazione in-ordine, gli elementi saranno visualizzati *in ordine crescente*. In un albero che soddisfa la regola gli elementi possono essere recuperati in modo molto veloce per mezzo di un algoritmo di ricerca binaria simile a quello presentato.

Quando si effettua una ricerca in un albero che sia il più corto possibile, il metodo di ricerca nelSottoAlbero c'è, di conseguenza, anche il metodo nellAlbero, sono all'incirca tanto efficienti quanto l'algoritmo di ricerca binaria in un array ordinato. Nel caso peggiore, il tempo necessario per la ricerca **risulterà proporzionale al logaritmo** del numero di nodi nell'albero. La ricerca in un albero binario corto è molto efficiente.

Per ottenere la massima efficienza, non è necessario che l'albero sia esattamente il più corto possibile, a patto che non si discosti troppo da tale condizione. Mano a mano che l'albero diventa sempre meno corto e largo e sempre più lungo e sottile, l'efficienza si riduce sempre di più, finché nel caso estremo diventa pari a quella della ricerca in una lista concatenata con lo stesso numero di nodi dell'albero.

### Copia in profondità e superficiale

La copia in profondità di un oggetto è una copia che **non ha alcun riferimento in comune con l'oggetto originale tranne che per gli oggetti immutabili**. Ogni copia che non sia una copia profondità è definita **superficiale**. Considerando un esempio:

#### Superficiale

```
public Animale getPrimo(){  
    return primo;  
}
```

#### Profondità

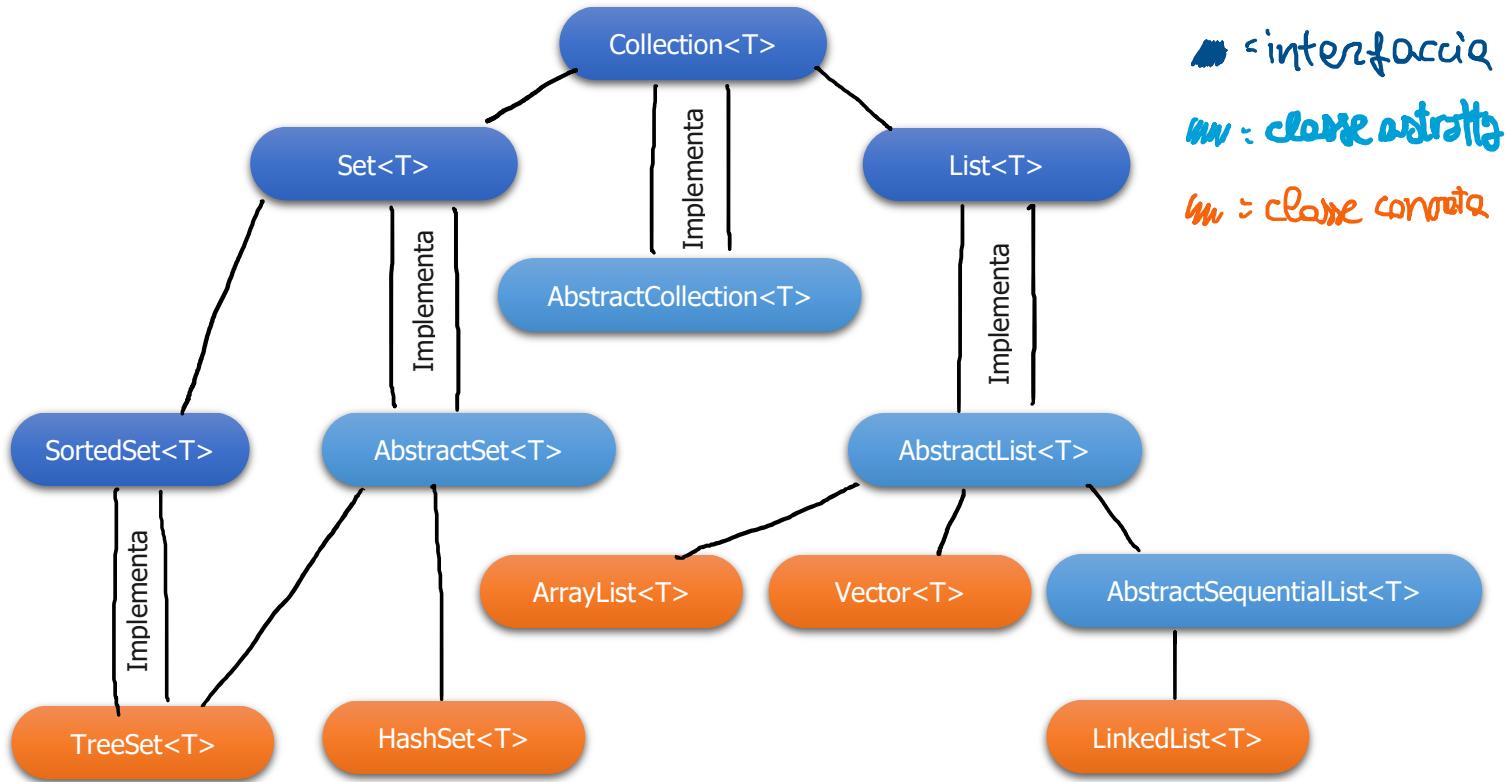
```
public Animale getPrimo(){  
    return new Animale(primo.getNome());  
}
```

↳ crea un nuovo oggetto con le stesse proprietà e metodi

Se si usa una copia in profondità, qualsiasi modifica apportata all'oggetto restituito dal metodo non si ripercuote sull'oggetto memorizzato nella variabile di istanza.

## Collezioni

Una collezione Java è una classe che contiene oggetti. La sintassi è **Collection<T>**, e questa è un'interfaccia; consente di scrivere codice che può essere applicato a tutte le collezioni Java, così che non è necessario riscrivere il codice per ogni specifico tipo di collezione. Esistono anche altre interfacce e classi astratte che sono in un senso o nell'altro ottenute dall'interfaccia Collection<T>.



Le classi e le interfacce della libreria delle collezioni Java utilizzano una modalità di specifica del tipo di parametro diversa. Tale modalità consente di specificare cose come "l'argomento deve essere un `ArrayList<T>`, ma può avere qualunque tipo base".

Usano i tipi generici ma senza specificare completamente il tipo di oggetto da sostituire al tipo di parametro. Poiché queste nuove modalità di specifica coprono un'ampia gamma di tipi di argomenti, sono note come **wildcard**; quello più semplice è `<?>`, che indica che si può usare qualunque tipo al posto del tipo di parametro. Per esempio,

```
public void metodoDiEsempio(String arg1, ArrayList<?> arg2)
```

Viene invocato passando due argomenti: il primo deve essere di tipo String, mentre il secondo può essere un `ArrayList<T>` con qualunque tipo base. Si noti che **ArrayList<?> è diverso da ArrayList<Object>**. Se la specifica di tipo è `ArrayList<?>`, allora si può passare un argomento di tipo `ArrayList<String>`; al contrario non si può passare un argomento di tipo `ArrayList<String>`, se la specifica di tipo da usare al suo posto deve essere un antenato o un discendente di qualche classe o interfaccia.

Per specificare che il tipo da sostituire al wildcard sia un antenato di qualche classe o interfaccia, si usa **super** al posto di *extends*. Per esempio, **ArrayList<? super Animale>** indica un `ArrayList<T>` il cui tipo base può essere un qualunque antenato della classe Animale.

`Collection<T>` costituisce l'interfaccia più generica della libreria Java che contiene le classi per le collezioni; descrive le operazioni di base che devono essere implementate da tutte le classi di tipo collezione. Poiché un'interfaccia è un tipo, si possono definire metodi con parametri di tipo `Collection<T>`. Il parametro può essere sostituito con un argomento che sia un'istanza di una qualunque classe di tipo collezione (*qualunque classe che implementi l'interfaccia Collection<T>*).

Altre collezioni includano:

- **AbstractSequentialList<T>**: fornisce un'implementazione efficiente degli spostamenti sequenziali lungo la lista al costo di una poco efficiente implementazione dell'accesso diretto agli elementi
- **LinkedList<T>**: classe concreta derivata da `AbstractSequentialList<T>`.
- **SortedSet<T>** e **TreeSet<T>** sono progettate per essere usate in implementazioni dell'interfaccia `Set<T>` che forniscono una versione efficiente dell'estrazione di elementi
- **ArrayList<T>** e **Vector<T>** è una `List<T>` con accesso diretto efficiente agli elementi. Se non è necessario l'accesso diretto efficiente, ma si devono poter spostare sequenzialmente gli elementi lungo la lista in modo efficiente, si può usare `LinkedList<T>`

## I metodi delle collezioni

Un metodo scritto per lavorare su un parametro di tipo `Collection<T>` funzionerà anche con tutte queste classi, e i **metodi di Collection<T> garantiscono la possibilità di usare contemporaneamente diverse classi di tipo collezione.**

<code>Boolean isEmpty();</code>	True se l'oggetto chiamante è vuoto, altrimenti false
<code>Boolean contains (Object obiettivo)</code>	True se l'oggetto chiamante contiene <b>almeno un'istanza di obiettivo</b> ; usa <code>obiettivo.equals</code> .
<code>Boolean containsAll(Collection&lt;?&gt; collezioneDiObiettivi)</code>	True se l'oggetto chiamante contiene tutti gli elementi in <code>collezioneDiObiettivi</code> . Per ogni elemento in <code>collezioneDiObiettivi</code> , il metodo usa <code>elemento.equals</code> .
<code>Boolean equals(Object altro)</code>	Sovrascrive il metodo <code>equals</code> ereditato, è per la collezione e non gli elementi della collezione
<code>Int size()</code>	Restituisce il numero di elementi contenuti nel chiamante. Se contiene più di <code>Integer.MAX_VALUE</code> , restituisce questo
<code>Iterator&lt;T&gt; iterator()</code>	Restituisce un iteratore per l'oggetto chiamante
<code>Object[] toArray()</code>	Restituisce un array contenente tutti gli elementi dell'oggetto chiamante. Se l'oggetto chiamante garantisce l'ordinamento degli elementi restituiti dal suo iteratore, questo metodo deve restituire gli elementi nello stesso ordine. <b>L'array restituito dovrebbe essere un nuovo array</b> , cosicché il chiamante non ha riferimenti a quest'ultimo.
<code>&lt;E&gt; E[] toArray(E[] a)</code>	Restituisce un array contenente tutti gli elementi dell'oggetto chiamante. L'argomento <code>a</code> è usato per <i>Specificare Il tipo di array da restituire</i> . Se <code>a</code> può contenere tutti gli elementi dell'oggetto chiamante, <code>a + usato</code> ; altrimenti, viene creato un nuovo array con lo stesso tipo di <code>a</code> . Se ha a più elementi dell'oggetto chiamante, gli altri vengono impostati su <code>null</code> .
<code>Int hashCode()</code>	Restituisce il <b>Codice hash</b> Dell'oggetto chiamante
<code>Boolean add(T elemento)</code>	Garantisce che l'oggetto chiamante contenga l'elemento specificato. True se l'oggetto chiamante è stato modificato dalla chiamata, false se non ammette elementi duplicati e contiene già elemento. False anche se l'oggetto chiamante non viene modificato.
<code>Boolean addAll(Collection&lt;? Extends T&gt; collezioneDaAggiungere)</code>	Garantisce che l'oggetto chiamante contenga tutti gli elementi in <code>collezioneDaAggiungere</code> . True se l'oggetto chiamante è stato modificato dalla chiamata, altrimenti false
<code>Boolean remove (Object elemento)</code>	Rimuove <b>Una singola istanza</b> Dell'elemento specificato dall'oggetto chiamante. True se l'oggetto chiamante conteneva l'elemento, altrimenti false
<code>Boolean removeAll(Collection&lt;?&gt; collezioneDaRimuovere)</code>	Rimuove dall'oggetto chiamante <b>Tutti gli elementi che sono contenuti anche in CollezioneDaRimuovere</b> . True se l'oggetto chiamante è stato modificato, altrimenti false.
<code>Void clear()</code>	Rimuove tutti gli elementi dall'oggetto chiamante

<code>Boolean retainAll(Collection&lt;?&gt; conservaElementi)</code>	Mantiene nell'oggetto chiamante tutti gli elementi contenuti anche in conservaElementi. Rimuove dall'oggetto chiamante tutti gli oggetti non contenuti in conservaElementi. True se l'oggetto chiamante è modificato dalla chiamata, altrimenti false.
--	--

## Set<T>

L'interfaccia Set<T> implementa Collection<T>, e **non ammette elementi ripetuti**. L'interfaccia Set<T> ha le stesse intestazioni dei metodi dell'interfaccia, ma in alcuni casi la semantica cambia.

<code>Boolean add(T elemento)</code>	Se l'elemento non è già contenuto nell'oggetto chiamante, è aggiunto e restituisce true. Altrimenti l'oggetto chiamante rimane invariato e restituisce false.
<code>Boolean addAll(Collection&lt;? Extends T&gt; collezioneDaAggiungere)</code>	Garantisce che, dopo la chiamata, l'oggetto chiamante contenga tutti gli elementi in collezioneDaAggiungere. True se l'oggetto chiamante viene modificato dalla chiamata, altrimenti false. Se collezioneDaAggiungere è un Set<T>, l'oggetto chiamante viene trasformato nella sua unione con collezioneDaAggiungere.

L'interfaccia List<T> ha più intestazioni di metodi dell'interfaccia Collection<T>, e alcuni metodi hanno una semantica diversa. Inoltre, le classi che implementano questa interfaccia **ammettono elementi ripetuti**, e **ordinano i loro elementi in una lista**.

<code>Boolean add(T elemento)</code>	Aggiunge elemento alla fine della lista di elementi dell'oggetto chiamante. Restituisce false se l'operazione È fallita.
<code>Boolean addAll(Collection&lt;? Extends T&gt; collezioneDaAggiungere)</code>	Aggiunge alla fine della lista di elementi dell'oggetto chiamante tutti gli elementi in collezioneDaAggiungere. Gli elementi vengono aggiunti nell'ordine nel quale vengono forniti da un iteratore per collezioneDaAggiungere
<code>Boolean remove(Object elemento)</code>	Elimina dalla lista di elementi dell'oggetto chiamante la prima occorrenza di elemento, se presente. Restituisce true se l'oggetto chiamante conteneva l'elemento, altrimenti false.
<code>Boolean removeAll(Collection&lt;? collezioneDaRimuovere)</code>	Rimuove dall'oggetto chiamante tutti gli elementi contenuti anche in collezioneDaRimuovere. True se l'oggetto chiamante è stato modificato, altrimenti false.
<code>Void add(int indice, T nuovoElemento)</code>	Inserisce nuovoElemento alla posizione indice nella lista di elementi dell'oggetto chiamante. L'elemento che si trovava alla posizione indice e tutti i successivi <i>vengono spostati di una posizione</i>
<code>T get(int indice)</code>	Restituisce l'oggetto alla posizione indice
<code>T set(int indice, T nuovoElemento)</code>	Imposta l'elemento alla posizione indice a nuovoElemento. Viene restituito l'elemento che si trovava originariamente in quella posizione.
<code>T remove(int indice)</code>	Rimuove l'elemento alla posizione indice dell'oggetto chiamante. Sposta gli elementi successivi a sinistra di una posizione
<code>Int indexOf(Object obiettivo)</code>	Restituisce l'indice del primo elemento uguale a obiettivo. <i>Usa il metodo Equals dell'oggetto obiettivo per verificare l'uguaglianza</i> ; -1 se non è trovato
<code>Int lastIndexOf(Object obiettivo)</code>	Restituisce l'indice dell'ultimo elemento uguale a obiettivo. <i>Usa il metodo equals dell'oggetto obiettivo per verificare l'uguaglianza</i> ; -1 se non è trovato
<code>List&lt;T&gt; subList(int daIndice, int aIndice)</code>	Restituisce una vista degli elementi alle posizioni comprese tra <b>DaIndice A AIndice</b> Dell'oggetto chiamante; l'oggetto alla posizione daIndice è incluso, mentre aIndice è escluso. La vista è composta di riferimenti all'oggetto chiamante; le modifiche alla vista modifichano potenzialmente l'oggetto chiamante, e l'oggetto restituito è di tipo List<T>.
<code>ListIterator&lt;T&gt; listIterator()</code>	Restituisce un iteratore per l'oggetto chiamante
<code>ListIterator&lt;T&gt; listIterator(int indice)</code>	Restituisce un iteratore per l'oggetto chiamante che parte da indice. Il primo elemento restituito dall' iteratore È quello alla posizione indice.

NUOVI

Esistono anche dei **metodi opzionali**; quando un'interfaccia indica un metodo come opzionale, il metodo deve comunque essere implementato quando si definisce una classe che implementa l'interfaccia. Se non si

vuole fornire un'implementazione vera del metodo, è sufficiente che quest'ultimo generi una *UnsupportedOperationException*.

Molti dei metodi di queste interfacce prevedono la generazione di eccezioni. Tutte le eccezioni sono non controllate, per cui non è necessario gestirle in un blocco *catch*, o dichiararle con clausole *throws*. Queste di solito sono intese come **strumento di debug**. Se si usa una classe collezione già esistente, possono essere interpretate come messaggi d'errore durante l'esecuzione. Se si definisce una nuova classe derivando un'altra classe, la maggior parte della generazione delle eccezioni sarà ereditata, e non ci sarà bisogno di preoccuparsene. Definendo una nuova classe che implementi una di queste interfacce, bisognerà anche **generare le eccezioni previste dall'interfaccia**.

## Classi concrete di tipo collezione

Le classi astratte `AbstractSet<T>` e `AbstractList<T>` sono fornite per semplificare l'implementazione delle interfacce, e contengono quasi esclusivamente i metodi richiesti dalle interfacce che implementano. È quindi necessario implementare non solo i metodi astratti, ma anche tutti gli altri metodi che si vogliono utilizzare.

La classe **AbstractCollection<T>** è uno scheletro di classe che implementa l'interfaccia `Collection<T>`.

### *HashSet<T>*

Se si vuole una classe che implementi l'interfaccia `Set<T>` e non sono necessari altri metodi, si può utilizzare la classe concreta `HashSet<T>`. Se si ha bisogno di avere una classe di tipo collezione che non **ammetta elementi ripetuti**, si può usare la classe `HashSet<T>`; questo è perché implementa una tabella hash. Un oggetto è memorizzato in una tabella hash, associandogli una chiave univoca. Implementa tutti i metodi di `Set<T>`, e non aggiunge altri metodi, a parte i costruttori.

<code>public HashSet()</code>	Crea un nuovo <code>HashSet</code> vuoto
<code>Public HashSet(Collection&lt;? Extends T&gt; c)</code>	Crea un nuovo insieme contenente tutti gli elementi di <code>c</code> . Se <code>c</code> è null, genera un'eccezione <code>NullPointerException</code> .
<code>Public HashSet(int dimensioneIniziale)</code>	Crea un nuovo insieme vuoto della capacità specificata. Se <code>dimensioneIniziale</code> è minore di 0, genera un'eccezione <code>IllegalArgumentException</code> .

È importante anche ridefinire i metodi **hashCode()** e **equals(Object obj)**.

- `hashCode`: Restituisce una chiave numerica che idealmente dovrebbe rappresentare un identificativo univoco per un oggetto della classe
- `Equals`: Java userà il codice hash per controllare se due oggetti sono uguali. Anche se due oggetti diversi avessero lo stesso codice hash, saranno comunque indicizzati correttamente se il metodo `equals()` indica che sono diversi. Il fatto di aver codici hash coincidenti **provocherà una collisione che riduce le prestazioni**.

Le operazioni di unione e intersezione possono essere realizzate utilizzando rispettivamente i metodi **addAll** e **retainAll**. Per stampare gli elementi in un oggetto `HashSet<T>` viene definito il metodo `stampaInsieme`.

## `ArrayList<T>` e `Vector<T>`

La differenza principale tra le due classi è che **ArrayList<T> è più recente di Vector<T>**, ed è stata creata come parte della libreria Java delle collezioni, mentre `Vector<T>` è una classe più vecchia che è stata adattata con l'aggiunta di nuovi metodi per renderla compatibile con le collezioni.

<code>public ArrayList(int capacitaIniziale)</code>	Crea un oggetto <code>ArrayList&lt;T&gt;</code> vuoto della capacitàIniziale Specificata. Quando è necessario aumentare la capacità dell' <code>ArrayList&lt;T&gt;</code> , questa viene raddoppiata.
<code>Public ArrayList()</code>	Crea un oggetto <code>ArrayList&lt;T&gt;</code> vuoto con capacità iniziale uguale a 10. Quando è necessario aumentare la capacità, viene raddoppiata.
<code>ArrayList(Collection&lt;? Extends T&gt; c)</code>	Crea un oggetto <code>ArrayList&lt;T&gt;</code> Contenente tutti gli elementi della collezione <code>c</code> , mantenendone l'ordine. Gli elementi nel nuovo oggetto <code>ArrayList&lt;T&gt;</code> avranno quindi lo stesso indice che avevano in <code>c</code> . Il nuovo oggetto è solo una <b>copia superficiale della collezione passata come argomento</b> , poiché contiene riferimenti agli elementi di <code>c</code>

<code>Vector(int capacitaIniziale)</code>	Crea un vettore vuoto della capacitaIniziale specificata. Quando è necessario aumentare la capacità, viene raddoppiata
<code>Vector()</code>	Crea un vettore vuoto con capacità iniziale uguale a 10. Se è necessario aumentare la capacità, viene raddoppiata.
<code>Vector(Collection&lt;? Extends T&gt; c)</code>	Crea un vettore contenente tutti gli elementi della collezione c, mantenendone l'ordine. Gli elementi del nuovo vettore hanno lo stesso indice che avevano in c. Il nuovo vettore è solo una <b>copia superficiale della collezione specificata</b> .

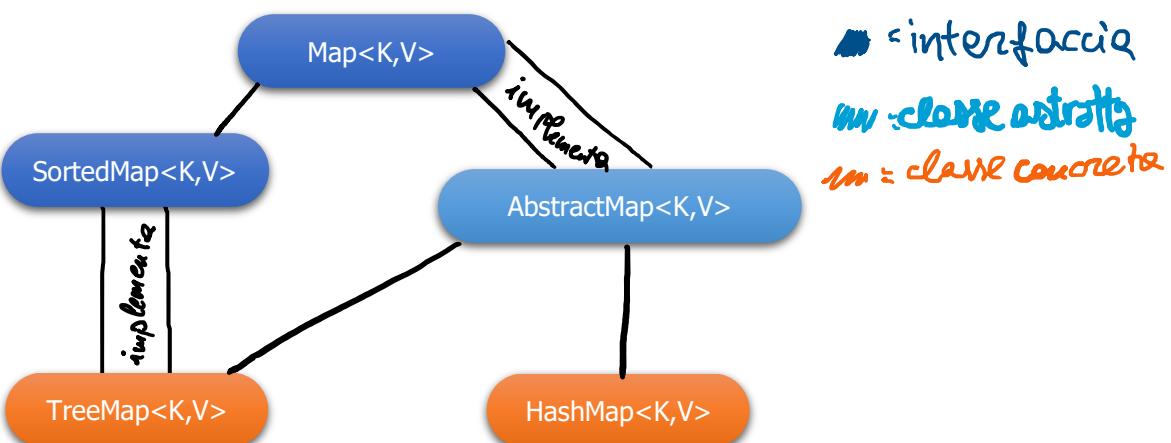
Hanno inoltre alcuni metodi di tipo array comuni ad entrambi:

<code>public T set(int indice, t nuovoElemento)</code>	Imposta a nuovoElemento l'elemento alla posizione <b>Indice</b> . Viene restituito l'elemento che si trovava precedentemente in quella posizione; facendo un'analogia con un array a, ciò corrisponde a impostare l'elemento a[indice] al valore nuovoElemento; l'indice deve essere $\geq 0$ , e $<$ delle dimensioni della lista.
<code>T get(int indice)</code>	Restituisce l'elemento alla posizione specificata; corrisponde a restituire a[indice] dato un array a. Indice $\geq 0$ , $<$ delle dimensioni correnti dell'oggetto chiamante
<code>Boolean Add(T nuovoElemento)</code>	Inserisce nuovoElemento alla fine della lista di elementi dell'oggetto chiamante, incrementando di 1 la dimensione della lista. Se necessario viene aumentata la capacità dell'oggetto chiamante. True se l'aggiunta è stata completata.
<code>Boolean Add(int indice, T nuovoElemento)</code>	Inserisce nuovoElemento alla posizione specificata dell'oggetto chiamante e incrementa di 1 la dimensione di quest'ultimo. Gli elementi con indice maggiore o uguale a indice vengono spostati in avanti di una posizione.
<code>Boolean addAll(Collection&lt;? Extends T&gt; c)</code>	Aggiunge tutti gli elementi contenuti in c dopo gli elementi dell'oggetto chiamante secondo l'ordine dettato da un iteratore di c. Il comportamento di questo metodo non è garantito se la collezione c coincide con l'oggetto chiamante o con qualunque collezione che include direttamente o indirettamente l'oggetto chiamante.
<code>Boolean addAll(int indice, Collection&lt;? Extends T&gt; c)</code>	Inserisce nell'oggetto chiamante tutti gli elementi contenuti in c, cominciando dalla posizione indice. Gli elementi vengono inseriti nell'ordine determinato da un iteratore di c. Gli elementi che si trovavano originariamente nella posizione indice e in quelle successive vengono spostati a indici più grandi.
<code>T remove(int indice)</code>	Elimina l'elemento alla posizione specificata e lo restituisce. La dimensione dell'oggetto chiamante viene ridotta di un'unità, ma la sua capacità è invariata. L'indice di ognuno degli elementi dell'oggetto chiamante con indice maggiore di indice viene decrementato di un'unità.
<code>Boolean remove(Object elemento)</code>	Rimuove dall'oggetto chiamante la prima occorrenza di elemento. Se elemento era contenuto nell'oggetto, l'indice di ognuno degli elementi successivi viene decrementato di 1. True se l'elemento è stato trovato, altrimenti false. Se l'elemento è stato rimosso, la dimensione dell'oggetto chiamante viene ridotta di 1, ma la capacità rimane invariata.
<code>Boolean isEmpty()</code>	True se l'oggetto chiamante è vuoto, altrimenti false.
<code>Boolean contains(Object obiettivo)</code>	True se obiettivo è un elemento dell'oggetto chiamante, altrimenti false. Per verificare l'uguaglianza, usa il metodo equals di obiettivo.
<code>Int indexOf(Object obiettivo)</code>	Restituisce l'indice del primo elemento uguale a obiettivo. Per verificare l'uguaglianza, usa il metodo equals dell'oggetto obiettivo. Se non viene trovato, restituisce -1
<code>Int lastIndexOf(Object obiettivo)</code>	Restituisce l'indice dell'ultimo elemento uguale a obiettivo. Per verificare l'uguaglianza, usa il metodo equals dell'oggetto obiettivo. Se non viene trovato, restituisce -1
<code>Iterator&lt;T&gt; iterator()</code>	Restituisce un iteratore per l'oggetto chiamante
<code>ListIterator&lt;T&gt; listIterator()</code>	Restituisce un listIterator<T> per l'oggetto chiamante
<code>ListIterator&lt;T&gt; listIterator(int indice)</code>	Restituisce un iteratore di lista per l'oggetto chiamante che parte dalla posizione indice. Il primo elemento restituito dall' iteratore è quello alla posizione indice.
<code>Object[] toArray()</code>	Restituisce un array contenente tutti gli elementi dell'oggetto chiamante nelle stesse posizioni
<code>&lt;E&gt; E[] toArray(E[] a)</code>	E può essere qualunque tipo di oggetto, non è necessario che coincida con il tipo T della collezione. Restituisce un array contenente tutti gli elementi dell'oggetto chiamante nelle stesse posizioni. A serve per specificare il tipo di array da restituire, il tipo dell'array restituito è lo stesso di a. Se a può contenere la collezione, viene usato per contenere gli elementi dell'array restituito, altrimenti viene creato un nuovo array dello stesso tipo di a. Se a ha più elementi dell'oggetto chiamante, gli elementi in più sono impostati a null.

<code>Int size()</code>	Restituisce il numero di elementi contenuti nell'oggetto chiamante
<code>Int capacity()</code>	Restituisce l'attuale capacità dell'oggetto chiamante
<code>Void ensureCapacity(int nuovaCapacita)</code>	Incrementa la capacità dell'oggetto chiamante per garantire che possa contenere un numero di elementi pari almeno a nuovaCapacita.
<code>Void trimToSize()</code>	Riduce la capacità dell'oggetto chiamante in modo che coincida con la sua dimensione attuale
<code>Object clone()</code>	Restituisce una copia superficiale dell'oggetto chiamante.

## Mappe

La libreria java delle mappe è simile a quella delle collezioni, a parte il fatto che lavora su **collezioni di coppie ordinate**. Gli oggetti della libreria delle mappe possono implementare funzioni e relazioni matematiche, e quindi possono essere utilizzati per costruire classi per la gestione di basi di dati. Ogni oggetto all'interno **ha una chiave K con un valore associato V**, e viene scritto `Map<K,V>`, `AbstractMap<K,V>`, `HashMap<K,V>`. L'interfaccia `Map<K,V>` specifica le operazioni di base che tutte le classi mappa dovrebbero implementare.



Ha i seguenti costruttori e metodi:

<code>Map&lt;K,V&gt;()</code>	Costruttore vuoto
<code>Map&lt;K,V&gt;(parameter)</code>	Creare un oggetto con gli stessi elementi di quello passato come argomento
<code>Boolean isEmpty()</code>	True se l'oggetto chiamante è vuoto, altrimenti false
<code>Boolean containsValue(Object valore)</code>	True se l'oggetto chiamante contiene almeno una chiave associata a un'istanza di valore
<code>Boolean containsKey(Object chiave)</code>	True se l'oggetto chiamante contiene chiave come una delle sue chiavi
<code>Boolean equals(Object altra)</code>	Metodo equals per la mappa in se e non gli elementi all'interno della mappa
<code>Int size()</code>	Restituisce il numero di coppie (chiave, valore) nell'oggetto chiamante
<code>Int hashCode()</code>	Restituisce il codice hash dell'oggetto Chiamante
<code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code>	Restituisce una vista di tipo Set costituita dalle coppie (chiave, valore) della mappa. Le modifiche alla mappa sono riflesse nell'insieme e viceversa
<code>Collection&lt;V&gt; values()</code>	Restituisce una vista di tipo Collection contenente tutti i valori nella mappa. Le modifiche alla mappa sono riflesse nella collezione e viceversa
<code>V get(Object chiave)</code>	Restituisce il valore al quale l'oggetto chiamante associa chiave. Se chiave non è nella mappa, restituisce null (è possibile anche associare a una chiave il valore null)
<code>V put(K chiave, V valore)</code>	Associa chiave a valore nella mappa. Se chiave era già associata a un valore, viene sostituito il vecchio valore; altrimenti restituisce null

PutAll(Map<? Extends K, ? Extends V> mappaDaAggiungere)	Aggiunge alla mappa chiamante tutte le associazioni di mappaDaAggiungere.
---	---