


JavaBook

Programma del corso Java 01

Java SE Programmer

Mauro Bogliaccino

Caratteristiche principali di Java

Java è un linguaggio di programmazione di alto livello e orientato agli oggetti, concepito da un team guidato da James Gosling a partire dal 1991 e ufficialmente rilasciato dalla Sun Microsystems nel 1995. L'obiettivo principale era sviluppare un linguaggio semplice e familiare. Le sue caratteristiche distintive includono:

1. **Orientato agli oggetti:**
    - Incorpora concetti come ereditarietà e polimorfismo.
    - Utilizza il collegamento dinamico.
    - Non supporta l'ereditarietà multipla e l'overload degli operatori.
2. **Gestione automatica della memoria:**
    - La memoria è allocata e deallocata automaticamente dal sistema, riducendo il rischio di errori di gestione della memoria da parte degli sviluppatori.
3. **Portabilità:**
    - Capacità di eseguire programmi su piattaforme diverse senza richiedere modifiche o ricompilazioni.

Caratteristiche di Java

Gli obiettivi primari nella creazione del linguaggio sono i seguenti:

1. **Orientato agli oggetti, semplice e familiare:**
    - Basato su C, ma sviluppato da zero per essere estremamente semplice, senza l'uso di puntatori, macro o registri.
    - Scopo di semplificare la programmazione e ridurre gli errori.
2. **Robusto e sicuro:**
    - Supporta la sicurezza attraverso il concetto di sandboxing.
    - Verifica del bytecode, restrizioni nell'accesso alla rete e assegnazione dei tipi restrittiva contribuiscono alla robustezza e sicurezza.

- La gestione della memoria è affidata al sistema.

3. **Indipendente dalla piattaforma:**
   - Utilizza la macchina virtuale Java (JVM) per garantire l'indipendenza dalla piattaforma.
   - Il linguaggio macchina è espresso tramite bytecode.
4. **Ad alte prestazioni:**
   - Risulta più efficiente di altri linguaggi interpretati grazie all'utilizzo della JVM.
5. **Interpretato, multi-thread, distribuito e dinamico:**
   - Il bytecode deve essere interpretato, offrendo vantaggi in termini di compattezza, efficienza e sicurezza del codice.
   - Supporta la programmazione multi-thread, consentendo l'esecuzione simultanea di diverse attività.
   - Progettato per esecuzioni remote e distribuite, con un sistema dinamico che carica classi quando sono richieste.

In sintesi, Java è un linguaggio versatile e potente che ha raggiunto un'elevata popolarità grazie alla sua facilità d'uso, robustezza e portabilità su diverse piattaforme

Versioni Java

Version	Release date	End of Free Public Updates	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	December 2030	N/A
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018		
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A

Version	Release date	End of Free Public Updates	Extended Support Until
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA
Java SE 18	March 22, 2022		
Java SE 19	September 20, 2022		

Java Language Changes

Espressioni

Le espressioni in programmazione sono combinazioni di valori, operatori e chiamate di funzioni che possono essere valutate per produrre un risultato. Le espressioni possono rappresentare calcoli aritmetici, valutazioni booleane, concatenazioni di stringhe e altro ancora. Le espressioni sono fondamentali per la creazione di logica e la manipolazione dei dati all'interno di un programma. Ecco alcuni esempi di espressioni:

1. **Espressioni Aritmetiche:**

```
int risultato = 2 + 3 * (5 - 1);
```

   In questo esempio, l'espressione aritmetica viene valutata secondo le regole di precedenza degli operatori.

2. **Espressioni Booleane:**

```
boolean condizione = (x > 5) && (y <= 10);
```

   Qui, l'espressione booleana verifica se entrambe le condizioni sono vere.

3. **Espressioni di Concatenazione di Stringhe:**

```
let nomeCompleto = nome + " " + cognome;
```

   Questa espressione concatena le variabili `nome` e `cognome` in una stringa più lunga.

4. **Espressioni di Assegnamento:**

```
x = y + 5;
```

In questo caso, l'espressione assegna a `x` il valore di `y` più 5.

5. **Chiamate di Funzioni:**

```
double risultatoFunzione = Math.sqrt(25);
```

L'espressione chiama la funzione `sqrt` della classe `Math` per calcolare la radice quadrata di 25.

6. **Espressioni Ternarie:**

```
let risultato = (x > 0) ? "Positivo" : "Negativo";
```

Questa espressione ternaria restituisce "Positivo" se `x` è maggiore di 0 e "Negativo" altrimenti.

7. **Espressioni di Array e Oggetti:**

```
valore = array[indice];
```

Questa espressione ottiene il valore di un elemento specifico in un array.

Le espressioni possono essere più complesse con l'uso di parentesi per determinare l'ordine di valutazione. Possono anche coinvolgere variabili, costanti e altri costrutti del linguaggio di programmazione. In generale, le espressioni sono fondamentali per scrivere codice che esegue calcoli e prende decisioni in base alle condizioni e ai dati presenti nel programma

per esempio nel programma

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

        double area = 5*10/2;
        System.out.println (area);
    }
}
```

Il programma risolve l'espressione aritmetica `5*10/2`, memorizza in `area` il risultato e lo stampa a video

Espressioni aritmetiche e precedenza

singoli "letterali"

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

operatori aritmetici

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

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

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

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

operazione di divisione

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

esempi

Domande:

1. Gli operatori possono essere utilizzati nella costruzione di espressioni, che calcolano valori.
    - Risposta: Gli operatori possono essere utilizzati nella costruzione di espressioni, che calcolano valori.
2. Le espressioni sono i componenti principali delle istruzioni.
    - Risposta: Le espressioni sono i componenti principali delle istruzioni.
3. Le istruzioni possono essere raggruppate in blocchi.
    - Risposta: Le istruzioni possono essere raggruppate in blocchi.
4. Il seguente frammento di codice è un esempio di un'espressione composta.

```
1 * 2 * 3
```

- Risposta: Vero, è un'espressione composta che calcola il prodotto di 1, 2 e 3.

5. Le istruzioni sono approssimativamente equivalenti alle frasi nei linguaggi naturali, ma invece di terminare con un punto, un'istruzione termina con un punto e virgola.
    - Risposta: Vero, le istruzioni terminano con un punto e virgola.
6. Un blocco è un gruppo di zero o più istruzioni tra parentesi graffe bilanciate e può essere utilizzato ovunque sia consentita una singola istruzione.
    - Risposta: Vero, un blocco è un gruppo di istruzioni racchiuso tra parentesi graffe e può essere utilizzato come una singola istruzione.

Esercizi:

Identifica i seguenti tipi di espressioni-istruzioni:

1. `aValue = 8933.234;` // istruzione di assegnazione
2. `aValue++;` // istruzione di incremento
3. `System.out.println("Hello World!");` // istruzione di invocazione del metodo
4. `Bicycle myBike = new Bicycle();` // istruzione di creazione dell'oggetto

Operatori nei linguaggi di programmazione

Negli linguaggi di programmazione, gli operatori sono simboli speciali o parole chiave che eseguono operazioni su uno o più operandi. Gli operandi sono i valori o le variabili su cui l'operatore agisce. Gli operatori sono fondamentali per eseguire operazioni aritmetiche, logiche, di confronto e altre azioni specifiche all'interno di un programma. Ecco una breve definizione di alcuni tipi comuni di operatori:

1. **Operatori Aritmetici:**
    - Eseguono operazioni matematiche come l'addizione, la sottrazione, la moltiplicazione e la divisione.
    - Esempi: `+` (addizione), `-` (sottrazione), `*` (moltiplicazione), `/` (divisione).
2. **Operatori di Confronto o Relazionali:**
    - Confrontano due valori e restituiscono un valore booleano che indica se la relazione è vera o falsa.
    - Esempi: `==` (uguale a), `!=` (diverso da), `<` (minore di), `>` (maggiore di), `<=` (minore o uguale a), `>=` (maggiore o uguale a).
3. **Operatori Logici:**
    - Eseguono operazioni logiche su valori booleani. Solitamente utilizzati in strutture di controllo decisionale.
    - Esempi: `&&` (AND logico), `||` (OR logico), `!` (NOT logico).
4. **Operatori di Assegnamento:**
    - Assegnano un valore a una variabile.
    - Esempio: `=` (assegnamento), `+=` (assegnamento con somma), `-=` (assegnamento con sottrazione), `*=` (assegnamento con moltiplicazione), `/=` (assegnamento con divisione).

5. **Operatori di Incremento e Decremento:**
    - Modificano il valore di una variabile incrementandolo o decrementandolo di una certa quantità.
    - Esempi: `++` (incremento), `--` (decremento).
6. **Operatori Bitwise:**
    - Eseguono operazioni bit a bit su numeri interi.
    - Esempi: `&` (AND bit a bit), `|` (OR bit a bit), `^` (XOR bit a bit), `~` (NOT bit a bit), `<<` (shift a sinistra), `>>` (shift a destra).
7. **Operatori Ternari:**
    - Sono operatori condizionali che valutano una condizione e restituiscono un valore in base al risultato della condizione.
    - Esempio: `condizione ? valore_se_vero : valore_se_falso`.

Gli operatori sono essenziali per manipolare dati e controllare il flusso di esecuzione all'interno di un programma, consentendo la creazione di logica complessa e la gestione di variabili e valori.

Operatori in Java

Operatori aritmetici

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

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

Operatori di assegnazione

Operatore	Significato

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

Operatori relazionali

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

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

Operatori per Booleani

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

Operatore	Significato
&&	AND logico
||	OR logico
!	NOT
^	exclusive OR

Attenzione:

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

Operatori su reference

Per i riferimenti/reference, sono definiti:

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

Operazioni matematiche complesse

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

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

Esempio

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

Comparazione del tipo di dato: Type Comparison Operator

- `instanceof` - Verifica se un certo oggetto è istanza di un certo Tipo di dato
- p.es: `if (a instanceof Automobile) //fai qualcosa`

Caratteri speciali

Literal	Represents
`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Backslash
`\"`	Double quote
`\ddd`	Octal character
`\xdd`	Hexadecimal character
`\udddd`	Unicode character

- raccolta esempi
- altri esempi

Domande:

1. Considera il seguente frammento di codice.

```
arrayOfInts[j] > arrayOfInts[j+1]
```

Quali operatori contiene il codice?

- Risposta: Contiene l'operatore di confronto ">", che confronta se l'elemento nell'indice j è maggiore dell'elemento nell'indice j+1 nell'array.

2. Considera il seguente frammento di codice.

```
int i = 10;
int n = i++%5;
```

- Quali sono i valori di i e n dopo l'esecuzione del codice?
  - Risposta: Dopo l'esecuzione, i è 11 e n è 0.
- Quali sono i valori finali di i e n se, invece di utilizzare l'operatore di incremento postfix (i++), si utilizza la versione prefix (++i))?
  - Risposta: Con l'operatore di incremento prefix, i diventa 11 e n diventa 1.

3. Per invertire il valore di un booleano, quale operatore utilizzeresti?
   - Risposta: L'operatore di negazione logica "!".
4. Quale operatore viene utilizzato per confrontare due valori, = o == ?
   - Risposta: L'operatore di confronto è "==".
5. Spiega il seguente frammento di codice: `result = someCondition ? value1 : value2;`
   - Risposta: Questo è un operatore ternario condizionale. Se someCondition è vera, result verrà assegnato a value1; altrimenti, verrà assegnato a value2.

Esercizi:

1. Modifica il programma seguente per utilizzare gli assegnamenti composti:

```
class ArithmeticDemo {
     public static void main (String[] args){
          int result = 1 + 2; // result è ora 3
          System.out.println(result);

          result = result - 1; // result è ora 2
          System.out.println(result);

          result = result * 2; // result è ora 4
          System.out.println(result);
```

```
            result = result / 2; // result è ora 2
            System.out.println(result);

            result = result + 8; // result è ora 10
            result = result % 7; // result è ora 3
            System.out.println(result);
        }
}
```

- Risposta: Modificare gli operatori di assegnamento nelle espressioni con gli operatori composti (+=, -=, *=, /=, %=).

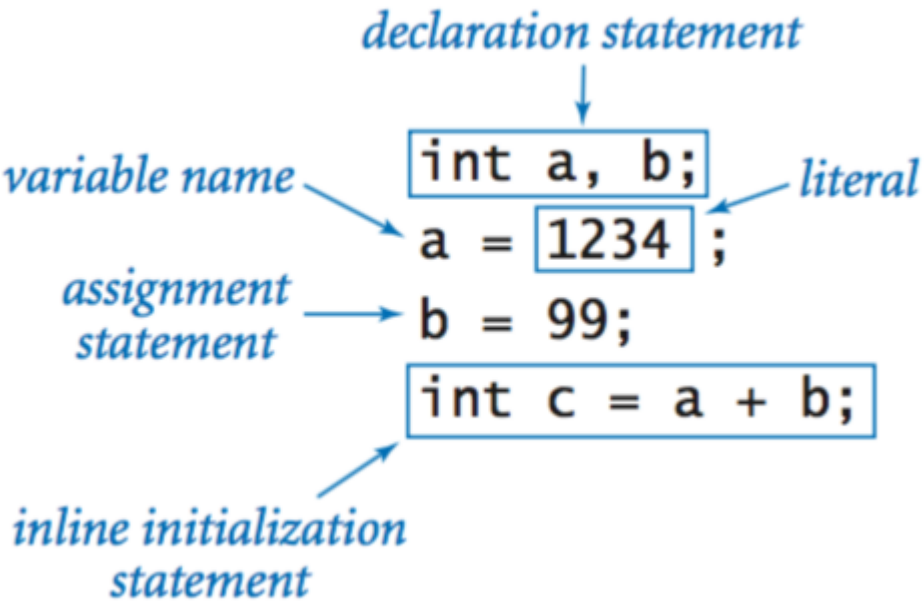
2. Nel programma seguente, spiega perché il valore "6" viene stampato due volte di seguito:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        System.out.println(i);    // "4"
        ++i;
        System.out.println(i);    // "5"
        System.out.println(++i);  // "6"
        System.out.println(i++);  // "6"
        System.out.println(i);    // "7"
    }
}
```

- Risposta: La linea `System.out.println(i++);` stampa "6" perché utilizza l'incremento postfix, quindi stampa il valore di i prima di incrementarlo. Successivamente, quando viene stampato nuovamente, i è stato incrementato a 7.

Le variabili e le costanti

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



esempio

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

        int base , altezza ; //dichiarazione di variabili locali
        double area ;

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

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

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

- Si capisce meglio quali siano la base e l'altezza del triangolo
- Si capisce meglio che cosa calcola il programma
- Evita la necessità di un commento per spiegare cosa contiene

Dichiarazione

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

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

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

Assegnazione

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

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

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

Dichiarazione + Assegnazione

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

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

Ad esempio:

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

uso della variabile locale

```
//1) dichiarazione
int mioNumero;
//2) inizializzazione
mioNumero = 100;
```

```
//3) uso della variabile locale
System.out.println(mioNumero);
```

NB: una variabile **locale** deve **SEMPRE** essere **inizializzata**, prima di poter essere utilizzata

Scope: ambito di visibilità delle variabili

```
class Nascoste {
  static int x,y; //Def. var. globali

  static void f() {
    int x;
    x = 1;  // Locale
    y = 1;  // Globale
    System.out.println(x);
    System.out.println(y);
  }

  public static void main (String[] args) {
    x = 0; // Globale
    y = 0; // Globale
    f();
    System.out.println(x);
    System.out.println(y);
  }
}
```

NB: una variabile **locale** deve **SEMPRE** essere **inizializzata**, prima di poter essere utilizzata

Costanti

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

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

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

L'attributo final

Definisce un dato elemento come non più modificabile

- Applicato a variabile la trasforma in costante
- Applicato a un metodo
  - Ne impedisce l'overriding in classi derivate
  - Ne rende possibile l'inlining (binding statico - più efficiente)
- Applicato a una classe
  - Impedisce di derivare da essa altre classi (la classe deve essere una foglia dell'albero di ereditarietà)

Input dall'utente

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

Domande:

1. Il termine "variabile di istanza" è un altro nome per ___.
   - Risposta: Variabile membro.
2. Il termine "variabile di classe" è un altro nome per ___.
   - Risposta: Variabile statica.
3. Una variabile locale memorizza uno stato temporaneo; è dichiarata all'interno di un ___.
   - Risposta: Metodo.
4. Una variabile dichiarata tra la parentesi di apertura e chiusura della firma di un metodo è chiamata ___.
   - Risposta: Parametro.
5. Quali sono gli otto tipi di dati primitivi supportati dal linguaggio di programmazione Java?
   - Risposta:
     - byte
     - short
     - int
     - long
     - float
     - double
     - char

        - boolean

6. Le stringhe di caratteri sono rappresentate dalla classe ___.
    - Risposta: Stringa.
7. Un ___ è un oggetto contenitore che tiene un numero fisso di valori di un singolo tipo.
    - Risposta: Array.

Esercizi:

1. Crea un piccolo programma che definisce alcuni campi. Prova a creare alcuni nomi di campi illegali e verifica quale tipo di errore il compilatore produce. Usa le regole e le convenzioni di denominazione come guida.

```
public class EsempioNomiCampi {
    // Nomi di campi legali
    int nomeCampoValido;
    String altroNomeCampoValido;

    // Nomi di campi illegali (decommenta per vedere gli errori del
compilatore)
    // int 123NomeCampoIllegale;  // Errore: gli identificatori devono
iniziare con una lettera o un underscore
    // double nome Campo Illegale;  // Errore: gli spazi non sono
consentiti negli identificatori
    // float $nomeCampoIllegale;  // Errore: '$' non è un carattere valido
per un identificatore
}
```

2. Nel programma creato nell'Esercizio 1, prova a lasciare i campi non inizializzati e stampa i loro valori. Prova la stessa cosa con una variabile locale e verifica quale tipo di errori del compilatore puoi produrre. Conoscere gli errori comuni del compilatore renderà più facile riconoscere bug nel tuo codice.

```
public class EsempioInizializzazioneVariabili {
    // Campi
    int campoNonInizializzato;  // Il compilatore fornirà un valore
predefinito (0 per int)

    public void stampaValori() {
        // Variabile locale
        int variabileLocaleNonInizializzata;

        // Decommentare la riga successiva causerà un errore del
compilatore
        // System.out.println(variabileLocaleNonInizializzata);  // Errore:
la variabile potrebbe non essere stata inizializzata
```

```
        System.out.println("Campo Non Inizializzato: " +
campoNonInizializzato);
    }
}
```

Questi esercizi dimostrano l'importanza di seguire le convenzioni di denominazione e di inizializzare le variabili prima di utilizzarle per evitare errori di compilazione.

Tipi di dato primitivi

I tipi di dato primitivi in Java rappresentano i dati più elementari e fondamentali che possono essere utilizzati per dichiarare variabili e memorizzare valori. Essi non sono oggetti e non hanno metodi. Ecco una lista dei tipi di dato primitivi in Java:

1. **byte:**
    - Dimensione: 8 bit
    - Intervallo: -128 a 127
2. **short:**
    - Dimensione: 16 bit
    - Intervallo: -32,768 a 32,767
3. **int:**
    - Dimensione: 32 bit
    - Intervallo: -2,147,483,648 a 2,147,483,647
4. **long:**
    - Dimensione: 64 bit
    - Intervallo: -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
5. **float:**
    - Dimensione: 32 bit
    - Precisione: Circa 7 cifre decimali
6. **double:**
    - Dimensione: 64 bit
    - Precisione: Circa 15 cifre decimali
7. **char:**
    - Dimensione: 16 bit
    - Rappresenta un singolo carattere Unicode
8. **boolean:**
    - Dimensione: Non specificata (spesso implementato come 1 bit)

- Può assumere solo i valori `true` o `false`

Questi tipi di dato primitivi sono essenziali per gestire valori numerici, caratteri e informazioni booleane in maniera efficiente. Quando si utilizzano questi tipi di dato, non si fa riferimento a oggetti come avviene con i tipi di dato non primitivi. Ad esempio, un `int` è un tipo di dato primitivo, mentre un oggetto di tipo `Integer` è un tipo di dato non primitivo. I tipi di dato primitivi in Java sono fondamentali per la programmazione e vengono spesso utilizzati nella gestione di variabili e operazioni matematiche di base.

Perché utilizzare tipi primitivi

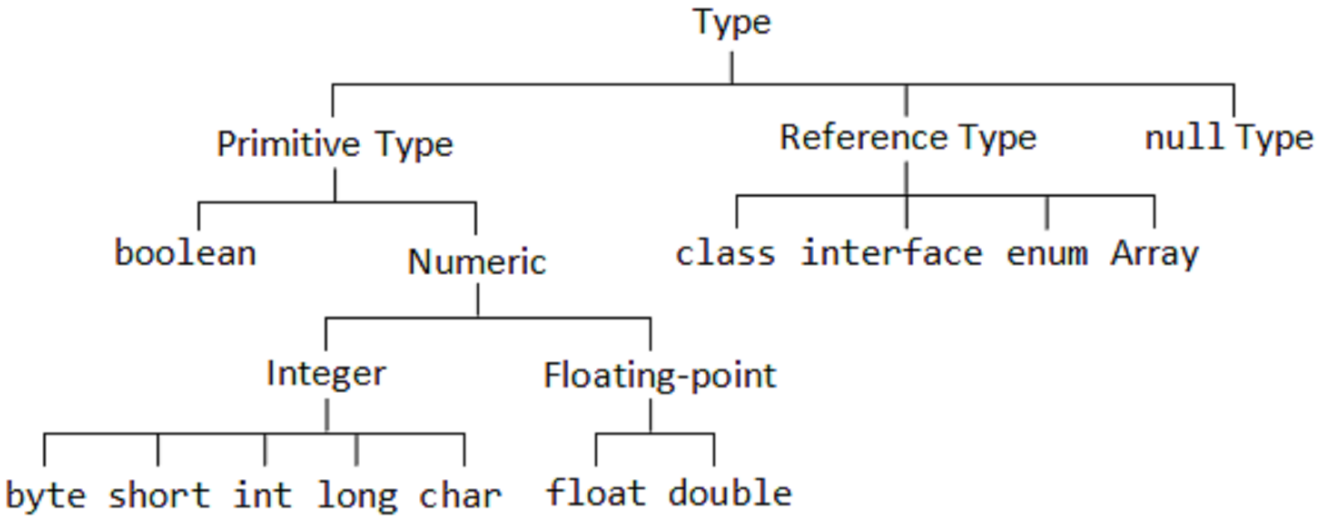
Java utilizza tipi di dati primitivi per garantire un'efficienza di esecuzione ottimale e una gestione della memoria più diretta. Ecco alcune ragioni principali per le quali Java utilizza tipi di dati primitivi:

1. **Efficienza di Memoria:**
   - I tipi di dati primitivi sono più efficienti in termini di utilizzo della memoria rispetto agli oggetti. Sono piccoli e occupano uno spazio fisso in memoria, indipendentemente dalla dimensione dell'intervallo del valore che possono rappresentare.
2. **Velocità di Esecuzione:**
   - Le operazioni su tipi di dati primitivi sono generalmente più veloci rispetto alle operazioni su oggetti. Ciò è dovuto al fatto che i tipi di dati primitivi sono direttamente supportati dalle istruzioni di basso livello del processore.
3. **Semplicità e Leggibilità del Codice:**
   - L'uso di tipi di dati primitivi rende il codice più semplice e leggibile. La sintassi per dichiarare e manipolare variabili di tipo primitivo è più concisa rispetto a quella degli oggetti.
4. **Supporto Diretto dal Linguaggio:**
   - I tipi di dati primitivi sono supportati direttamente dal linguaggio Java, senza la necessità di dover utilizzare oggetti wrapper (come `Integer`, `Double`, etc.) per rappresentare valori primitivi.
5. **Facilità di Comparazione:**
   - I tipi di dati primitivi possono essere facilmente comparati utilizzando operatori di confronto come `==` e `<`, senza la necessità di sovrascrivere il metodo `equals()`.
6. **Compatibilità con Altri Linguaggi:**
   - Java è progettato per essere interoperabile con altri linguaggi di programmazione. L'uso di tipi di dati primitivi facilita l'interazione con linguaggi che trattano dati in modo simile (ad esempio, C, C++).
7. **Supporto per Operazioni Matematiche:**
   - I tipi di dati primitivi supportano operazioni matematiche dirette, come addizione, sottrazione, moltiplicazione e divisione, senza la necessità di conversioni complesse o overhead di gestione degli oggetti.

8. **Gestione di Grandi Quantità di Dati:**
    - In applicazioni che trattano grandi quantità di dati, l'uso di tipi di dati primitivi contribuisce a ridurre la pressione sulla gestione della memoria e aumenta le prestazioni complessive del programma.

In sintesi, l'uso di tipi di dati primitivi in Java è guidato dalla necessità di ottenere prestazioni efficienti, gestire la memoria in modo diretto e semplificare la scrittura del codice. Tuttavia, Java fornisce anche classi wrapper per tipi primitivi quando è necessario trattare i dati come oggetti, fornendo una maggiore flessibilità in contesti specifici.

Tipi di dato in Java



I mattoncini fondamentali

Tabelle riassuntive: tipi di dato primitivi

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

I caratteri sono considerati interi

I tipi numerici, i char

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

- `'a' '%' '\n'` (char)
- `'\123'` (\ introduce codice ASCII)

Tipo boolean

- `true`
- `false`

Esempi

```
//tipi interi
byte b = 254;//8 bit
short s = 10;//16 bit
int i = 15;//32 bit
long l = 10000000000001;//64 bit

//tipi reali
float f = 26.012f;//32 bit
double d = 123.567;//64 bit

//booleans
boolean fatto = true;
boolean daFare = false;

//char - rappresentabili anche come interi
char ch = 'a';
char ch2 = ';';
```

uso della variabile locale

```
//1) dichiarazione
int mioNumero;
//2) inizializzazione
mioNumero = 100;
//3) uso della variabile locale
System.out.println(mioNumero);
```

NB: una variabile **locale** deve **SEMPRE** essere **inizializzata**, prima di poter essere utilizzata

Esempio tipi primitivi

```
//dichiarazione e inizializzazione contemporanea
byte mioByte = 127;
System.out.println(mioByte);

short mioShort = 851;
```

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

long mioLong = 34093L;
System.out.println(mioLong);

double mioDouble = 3.14159732;
System.out.println(mioDouble);

float mioFloat = 324.4f;
System.out.println(mioFloat);

char mioChar = 'y';
System.out.println(mioChar);

boolean mioBoolean = true;
System.out.println(mioBoolean);
```

Tipi primitivi: range di valori ammissibili

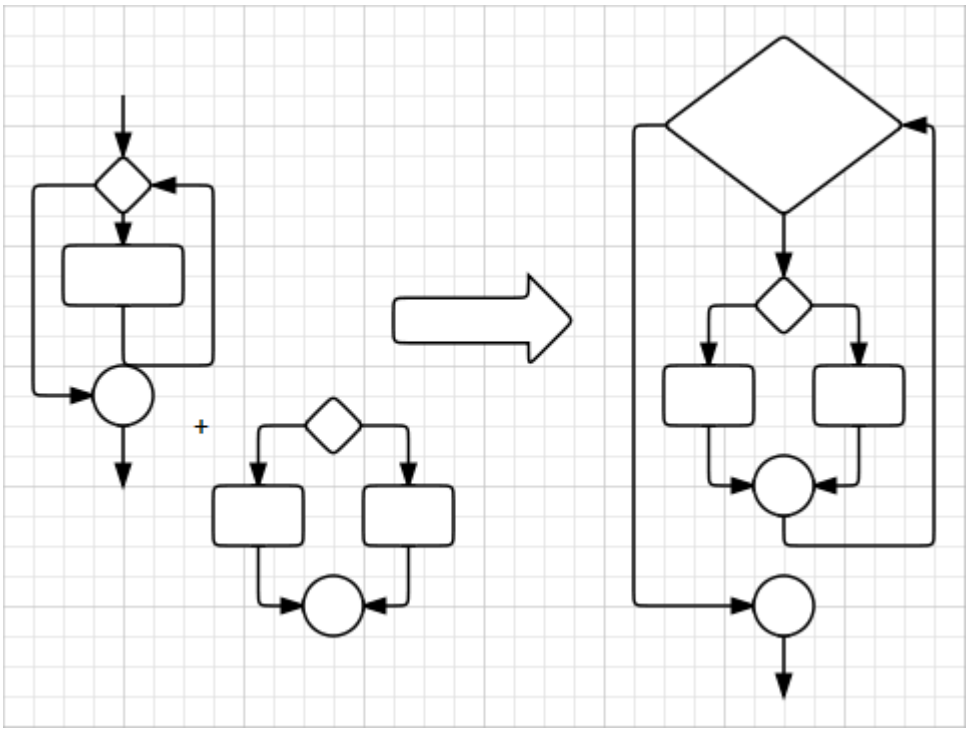
Data Type	Bits	Minimum	Maximum
byte	8	-128	127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-9.22337E+18	9.22337E+18
float	32	See the docs	
double	64	See the docs	

- raccolta esempi
- altri esempi sui tipi primitivi
- Everything you'll ever need to work with Java primitive types!

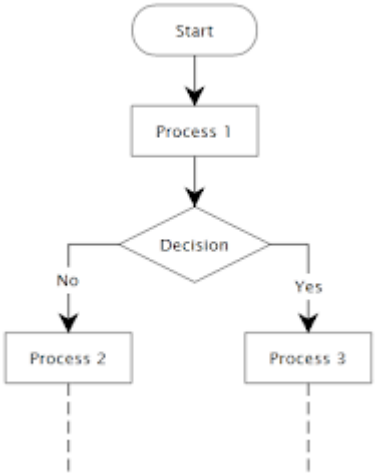
Il controllo del flusso

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

- IF – ELSE
- WHILE e DO-WHILE
- FOR e FOREACH
- SWITCH - CASE



Selezione, scelta condizionale



if statements

- E' un'istruzione condizionale, permette cioè di eseguire un blocco di istruzioni solo se si verifica una determinata condizione.

```
if (condition) {

    //statements;

}
```

else if [opzionale]

- Indichiamo anche cosa fare se non si supera la condizione

```
else if (condition2) {
    //statements;
}
```

else [opzionale]

```
else {
//statements;
}
```

Switch Statements

Serve per gestire in maniera più ordinata varie condizioni, è un modo più elegante in alcune situazioni.

```
switch (espressione) {
    case valore1:
    //statements;
    break;
    ...
    case valoren:
    //statements;
    break;
    default:
    //statements;
}
```

Cicli definiti - for

Se il numero di iterazioni è **prevedibile**.

```
for (partenza, fine, incremento)
blocco di istruzioni da ripetere
```

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

//statements;

}
```

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

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

Cicli indefiniti - while

si ripete il ciclo fintanto che una condizione è verificata (è vera)

- la condizione booleana `true/false`
- determina la continuazione del programma
- ed esegue l'elenco delle operazioni del blocco
- Da usare se il numero di iterazioni **non è noto** all'inizio del ciclo.

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

//Esempio: quando il numero di iterazioni dipende da valori in input
while(true) {
    x = Integer.parseInt(JOptionPane.showInputDialog("Immetti numero
positivo"));
    if (x > 0) break;
}
```

do-while

- Quando voglio eseguire almeno una volta l'istruzione, anche se la condizione è impostata su `false`
- Si verifica la condizione dopo il primo ciclo

```
do {

//statements;

} while (condition);
```

Cicli con interruzione: break

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

- Se stiamo eseguendo un ciclo, possiamo utilizzare la parola `break` per interromperlo in qualsiasi momento.
- Si interrompe quindi il ciclo e si riprende l'esecuzione delle istruzioni al di fuori di esso.

Esempio: verifica se un array contiene o meno numeri negativi

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

Cicli con salto: continue

Invece terminare completamente il ciclo ed uscire fuori, `continue` interrompe solo l'iterazione corrente e passa alla successiva.

```
   {
      int a;

      for ( a = 1; a <= 10 ; a++ )            // Run a = 1, 2, ..., 10
      {
         if ( a == 4 )
         {
            continue;               // Skip printing 4...
         }

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

```
        }
    }
```

Cicli con condizione

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

Esempio: stampa tutti i numeri pari fino a 100

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

Cicli con accumulatore

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

Esempio: somma i primi 100 numeri interi.

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

Esempio: data una stringa s, ottieni la stringa rovesciata

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

Cicli misti

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

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

Cicli annidati

Se un ciclo appare nel corpo di un altro ciclo.

Esempio: stampa quadrato di asterischi di lato n

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

codice esempi d'uso

- raccolta esempi
- altri esempi
- 01_if-elseif-else
- 02_switch
- 03_while
- 04_for
- 05_foreach
- 06_labels
- giochi
- programmi

Risposte alle Domande:

1. La più basilare istruzione di flusso di controllo supportata dal linguaggio di programmazione Java è l'istruzione if-else.
    - Risposta: Vero, l'istruzione if-else è l'istruzione di flusso di controllo più basilare.
2. L'istruzione switch consente qualsiasi numero di percorsi di esecuzione possibili.
    - Risposta: Vero, l'istruzione switch permette percorsi di esecuzione multipli.
3. L'istruzione do-while è simile all'istruzione while, ma valuta la sua espressione in fondo al ciclo.
    - Risposta: Vero, l'istruzione do-while valuta la sua espressione alla fine del ciclo.
4. **Domanda:** Come si scrive un ciclo infinito utilizzando l'istruzione for?

- **Risposta:**

```
for ( ; ; ) {

}
```

5. **Domanda:** Come si scrive un ciclo infinito utilizzando l'istruzione while?

   - **Risposta:**

```
while (true) {

}
```

Array

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

Il riferimento ad array

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

Cosa sono gli array?

Nel linguaggio di programmazione Java, gli array sono oggetti (§4.3.1), vengono creati dinamicamente e possono essere assegnati a variabili di tipo Object (§4.3.2). Tutti i metodi della classe Object possono essere invocati su un array. Java docs

In Java gli array sono Oggetti

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

Dimensione dell'array

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

Creazione di un Array

L'operatore new crea un array:

- Con costante numerica

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

- Con costante simbolica

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

- Con valore definito a run-time

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

riempire l'array utilizzando un inizializzatore

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

```
int[] numeriPrimi = {2,3,5,7,11,13};
//...
// La virgola finale e' facoltativa
int [] numeriPari = {0, 2, 4, 6, 8, 10,};
```

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

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

In Java viene fatto il bounds checking

- Maggior sicurezza
- Accesso più lento

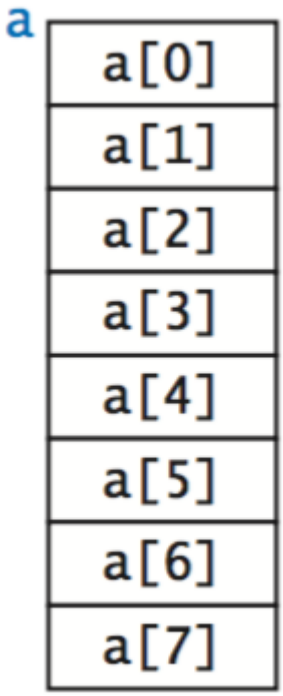
Array Mono-dimensionali (vettori)

Dichiarazione di un riferimento a un array

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

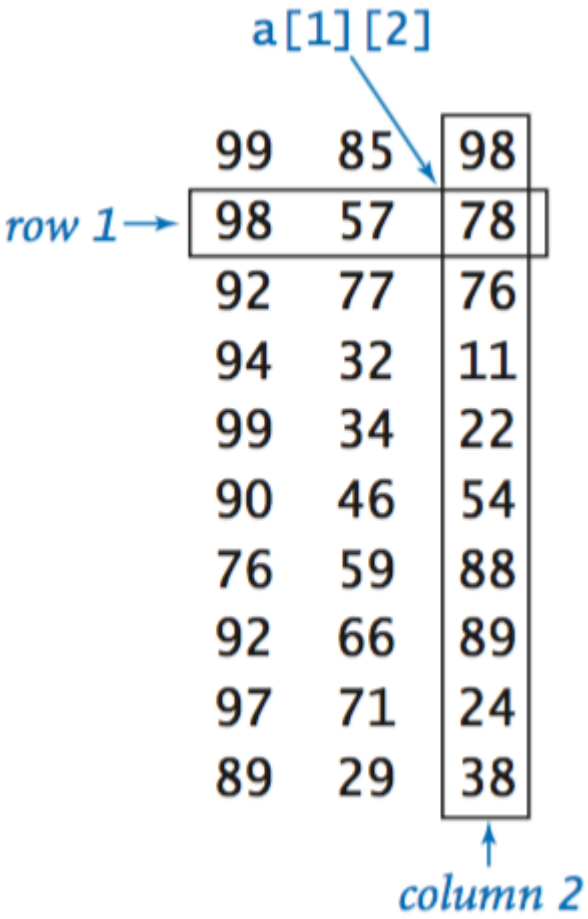
La dichiarazione di un array non assegna alcuno spazio

`voti == null`



Array Multi-dimensionali

- **Array contenenti riferimenti ad altri array**
- Sintatticamente sono estensioni degli array a una dimensione
- matrice: array di array, regolare, irregolare, rettangolare...



- **Le righe non sono memorizzate in posizioni adiacenti**
- Possono essere spostate facilmente

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

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

Esempi di Array

Array Monodimensionali

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

Array Multidimensionali

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

Array irregolari Sono possibili righe di lunghezza diverse**

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

Array di oggetti

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

L'inizializzazione va completata con quella dei singoli elementi

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

Algoritmi per lavorare con gli array: la classe `java.util.Arrays`

- **Il pacchetto java.util contiene metodi statici di utilità per gli array**
- Copia di un valore in tutti gli (o alcuni) elementi di un array
    - `Arrays.fill (<array>, <value>);`
    - `Arrays.fill (<array>, <from>, <to>,<value>);`
- Copia di array
    - `System.arraycopy (<arraySrc>, <offsetSrc>,<arrayDst>, <offsetDst>, <#elements>);`
- Confronta due array
    - `Arrays.equals (<array1>, <array2>);`

- Ordina un array (di oggetti che implementino l'interfaccia Comparable)
    - `Arrays.sort (<array>);`
- Ricerca binaria (o dicotomica)
    - `Arrays.binarySearch (<array>);`

altre risorse

- raccolta esempi
- altri esempi
- esempi ed esercizi su array

Stringhe e Caratteri

Caratteristiche principali

Classi disponibili

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

Definizione

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

- Oppure

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

- Solo per il tipo String vale l'inizializzazione

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

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

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

Esempio

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

altri esempi sulle stringhe

Concatenare stringhe

- Operatore concat
    - `myString1.concat(myString2)`
    - `String s2 = "Ciao".concat(" a tutti").concat("!");`
    - `String s2 = "Ciao".concat(" a tutti".concat("!"));`
- Utile per definire stringhe che occupano più di una riga
- Operatore `+ "questa stringa" + "e` formata da tre" `+` "stringhe"
- La concatenazione funziona anche con altri tipi, che vengono automaticamente convertiti in stringhe
- `System.out.println ("pi Greco = " + 3.14);`

NB: La classe String crea nuovi oggetti ogni volta che concateni con +, meglio usare la classe StringBuilder...

Carattere i-esimo

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

Lunghezza stringa

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

Confronta stringa con altra stringa

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

Confronta con altra stringa facendone la differenza

- `int compareTo(String str)`
- `myString.compareTo("stringa")` ritorna un valore >=< 0

Trasforma int in String

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

Restituisce indice prima occorrenza di c

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

Altri metodi

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

compareTo

```
int compareTo(String other)
```

Esegue una comparazione lessicale. Ritorna un intero:

- `< 0` se la stringa corrente è minore della stringa other
- `= 0` se le due stringhe sono identiche
- `> 0` se la stringa corrente è maggiore di other

indexOf

```
int indexOf(int ch)
```

Restituisce l'indice del carattere specificato

lastIndexOf

```

```

```
int lastIndexOf(int ch)
```

E' come indexOf() ma viene restituito l'indice dell'ultima occorrenza trovata

length

```
int length()
```

Restituisce il numero di caratteri di cui è costituita la stringa corrente

replace

```
String replace(char oldChar, char newChar)
```

Restituisce una nuova stringa, dove tutte le occorrenze di oldChar sono rimpiazzate con newChar

substring

```
String substring(int startIndex)
```

Restituisce una sottostringa della stringa corrente, composta dai caratteri che partono dall'indice startIndex alla fine

substring

```
String substring(int startIndex, int number)
```

Restituisce una sottostringa della stringa corrente, composta dal numero number di caratteri che partono dall'indice startIndex

toLowerCase

```
String toLowerCase()
```

Restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri minuscoli

toUpperCase

```
String toUpperCase()
```

Restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri maiuscoli

codice esempi d'uso

- raccolta esempi
- altri esempi
- stringbuilder

Casting e promotion

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

expression	*expression type*	*expression value*
(1 + 2 + 3 + 4) / 4.0	double	2.5
Math.sqrt(4)	double	2.0
"1234" + 99	String	"123499"
11 * 0.25	double	2.75
(int) 11 * 0.25	double	2.75
11 * (int) 0.25	int	0
(int) (11 * 0.25)	int	2
(int) 2.71828	int	2
Math.round(2.71828)	long	3
(int) Math.round(2.71828)	int	3
Integer.parseInt("1234")	int	1234

casting dei tipi reference (oggetti)

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

esempio promotion

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

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

esempio promotion II

```
  int valore1 = 56;
  int valore2 = valore1;

  System.out.println(valore2);

  //promozione
  long valoreLong1 = valore1;
  System.out.println(valoreLong1);
 }
```

esempi type casting

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

System.out.println(  Integer.toBynaryString(b)  ); // 101
System.out.println(  Integer.toBynaryString(261)  ); // 100000101
```

```
int a = (int) 1936.27;

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

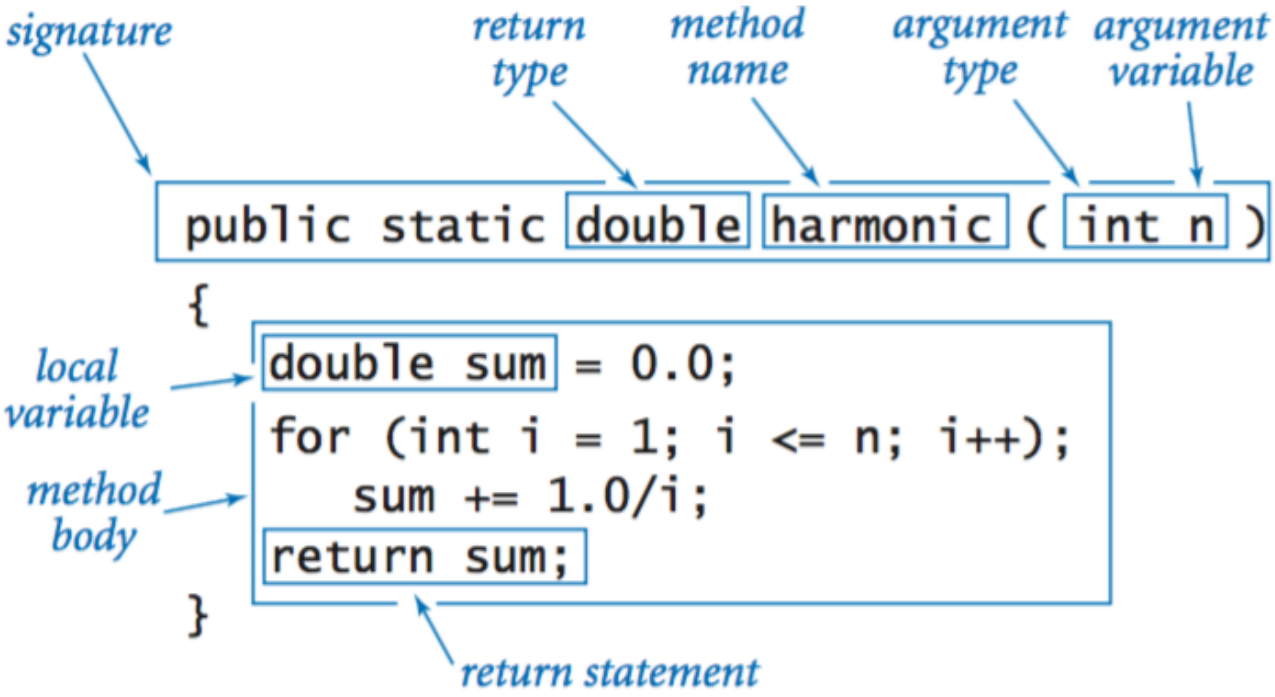
con il tipo boolean non si può fare il typecasting

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

esempi

metodo

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



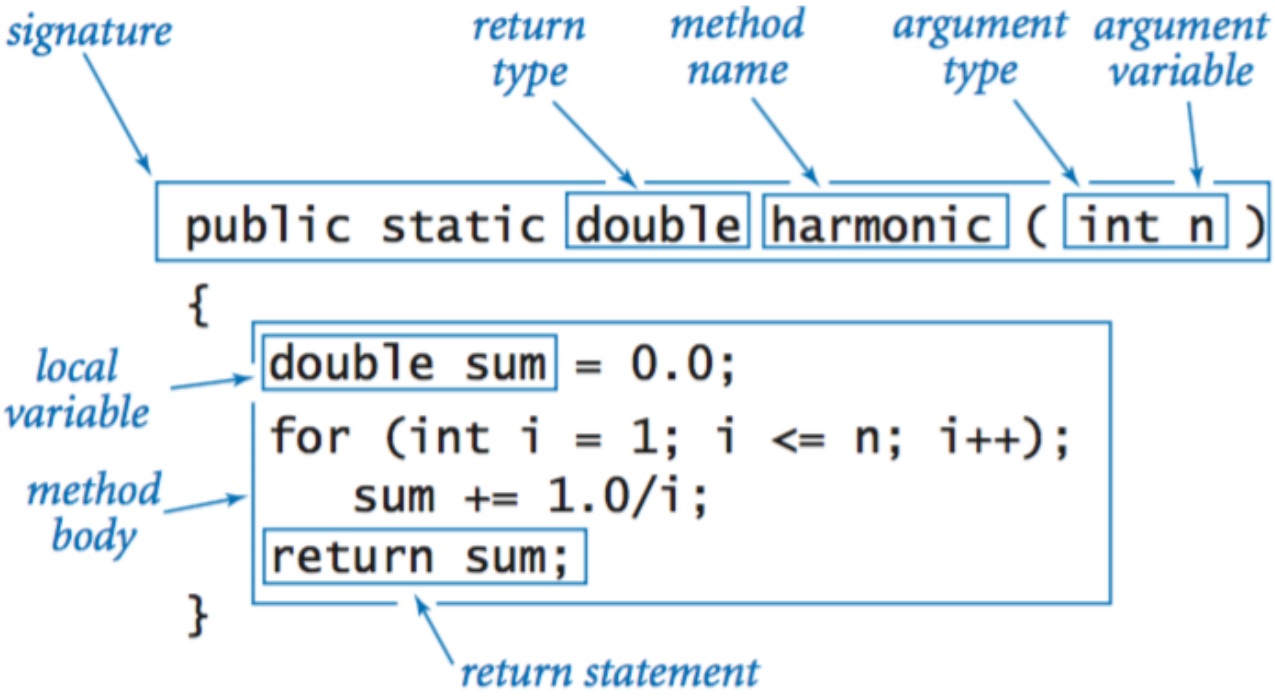
Componenti dei metodi

Più in generale, le dichiarazioni di metodo hanno **sei** componenti (alcuni sono opzionali), nell'ordine:

- **Modificatori**, come public, private e altri che imparerai in seguito.
- Il **tipo restituito**: il tipo di dati del valore restituito dal metodo o void se il metodo non restituisce un valore.
- Il **nome del metodo**: le regole per i nomi dei campi si applicano anche ai nomi dei metodi, ma la convenzione è leggermente diversa.
- L'**elenco di parametri** tra parentesi: un elenco delimitato da virgole di parametri di input, preceduti dai rispettivi tipi di dati, racchiusi tra parentesi (). Se non sono presenti parametri, è necessario utilizzare parentesi vuote.
- Un **elenco di eccezioni**, opzionali, da discutere in seguito.
- Il **corpo del metodo**, racchiuso tra parentesi graffe: il codice del metodo, inclusa la dichiarazione delle variabili locali, va qui.

Argomenti attuali e formali

- Ogni volta che si **invoca** un metodo si deve specificare una lista di argomenti attuali
- Gli argomenti attuali e formali sono in corrispondenza posizionale
- Gli argomenti attuali possono essere delle variabili o delle espressioni
- Gli argomenti attuali devono rispettare il tipo attribuito agli **argomenti formali**



Overloading dei metodi

- E' possibile definire **metodi con lo stesso nome** ma liste degli argomenti diverse, cioè varianti diverse dello stesso metodo
- Si definisce *signature*, o **firma** del metodo l'insieme di nome ed argomenti: p.es `raddoppia(String s);`,`raddoppia(int i);`
- La diversità delle liste riguarda il **numero**, **tipo** e **ordine** di argomenti formali, **non** il loro nome
- A seconda degli argomenti passati verrà selezionato ed eseguito il metodo appropriato (se esiste)
- **Non** è ammesso **overloading** sul **tipo** ritornato: metodi con nome e lista degli argomenti uguali ma tipo ritornato diverso non vengono distinti e danno luogo ad errori di compilazione

Metodi ausiliari (static)

- Il modificatore `static` permette di creare **metodi statici**: quelli dichiarati `static`
- I metodi static sono richiamabili attraverso nome della classe
- p.es: `Math.sqrt()`
- Se sono anche `private` e aiutano a separare la logica dal metodo `main`, in caso di interfacce testuali (che girano nella `console`), vengono definiti anche `metodi ausiliari`
- Nella programmazione ad oggetti bisogna farne un uso estremamente limitato!

```
public class ProvaMetodiStatic
{
    public static void main(String[] args) {
        metodoUno();
        metodoUno();
        metodoDue();
    }
```

```
    public static void metodoUno() {
        System.out.metodoln("Hello World");
    }

    public static void metodoDue() {
        metodoUno();
        metodoUno();
    }
}
```

quando e perché usare i metodi (ausiliari)

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

Metodi non static

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

Invocazione di metodi non static

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

Argomenti correlati

- I metodi costruttori
- I metodi getters e setters

Metodi predicativi

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

codice esempi d'uso

- raccolta esempi
- Esempi sui metodi ausiliari

Classi Java

- Una classe è uno dei concetti fondamentali di OOP.
- Una classe è un modello o un progetto per la creazione di oggetti.
- Una classe non consuma memoria.
- Una classe può essere istanziata più volte.
- Una classe fa una, e solo una, cosa.

Una classe è uno dei concetti fondamentali di OOP. Una classe è un insieme di istruzioni necessarie per costruire un tipo specifico di oggetto. Possiamo pensare a una classe come a un modello, un progetto o una ricetta che ci dice come creare oggetti di quella classe.

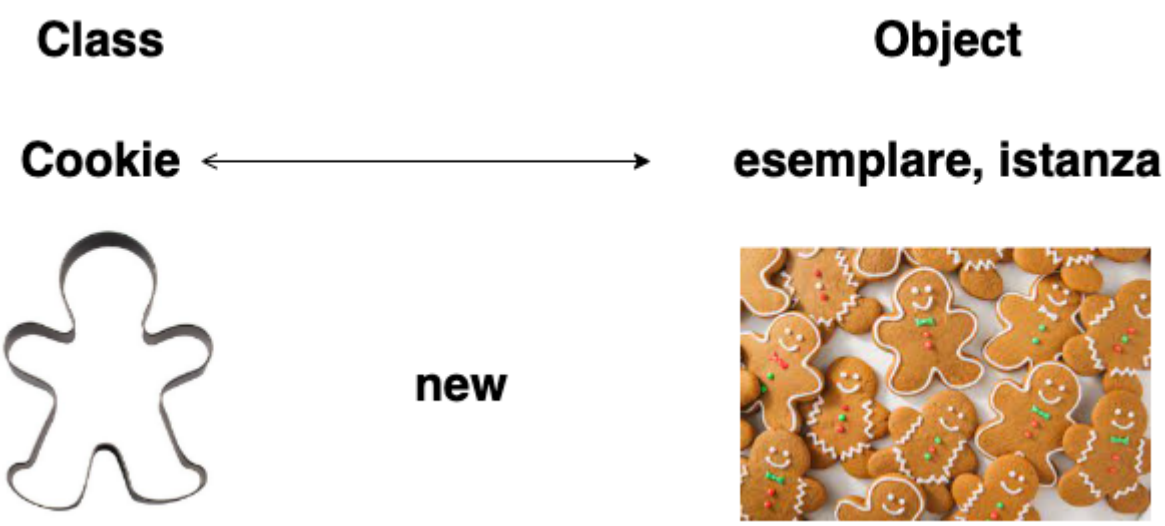
La creazione di un oggetto di quella classe è un processo chiamato istanziazione e di solito viene eseguito tramite la parola chiave `new`.

Possiamo istanziare tutti gli oggetti che vogliamo. Una definizione di classe non consuma memoria salvata come file sul disco rigido. Una delle migliori pratiche che una classe dovrebbe seguire è il principio di responsabilità singola (SRP): una classe dovrebbe essere progettata e scritta per fare una, e solo una, cosa.

Java è un linguaggio orientato agli oggetti

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

La classe è lo 'stampo' per gli oggetti



Le classi definiscono

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

Possono essere definite

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

La "gestione" di una classe avviene mediante

- Definizione della classe
- Instanziazione di Oggetti della classe

Struttura di una classe

```
package model;

public class Persona {

    //proprietà private - vedi incapsulamento
    private String nome;
    private String cognome;
    private int eta;

    //metodo costruttore
    public Persona (String nome, String cognome, int eta)  {
    this.nome = nome;
    this.cognome = cognome;
    this.eta = eta;
    }

    //per gestire le proprietà vedi metodi getters and setters
    //metodi...

    @Override
```

```
    public String toString () {
    return this.nome + " " + this.cognome + " " +  this.eta;
    }
}
```

Le classi in Java

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

Classi e documentazione

- Java è dotato di una libreria di classi "pronte all'uso" che coprono molte esigenze
- Usare classi già definite da altri è la norma (principio DRY)
- La libreria Java standard è accompagnata da documentazione che illustra lo scopo e l'utilizzo di ciascuna classe presente
- Dalla versione 9 di Java la libreria è stata divisa in moduli
- Documentazione Java 8
- Documentazione Java 9
- Documentazione Java 11
- Documentazione Java 14
- Documentazione Java 17

esempi classi