Università di Trieste

Laurea in ingegneria elettronica e informatica

Enrico Piccin - Corso di Fondamenti di informatica - Prof. Sylvio Barbon Junior

Anno Accademico 2021/2022 - 1 Marzo 2022

Indice

Listings

1 Marzo 2022

1 Introduzione a Java

Java è un linguaggio di programmazione rigido e fortemente tupizzato, ovvero esige la dichiarazione di tutte le vairabili, con il loro tipo e con una loro inizializzazione. La **struttura formale** del linguaggio, pertanto, è predomincante ed è ciò che caratterizza Java rispetto ad altri linguaggi di programmazione, come *Python* o *PHP*.
Java, inoltre, è un linguaggio che opera al di sopra del sistema operativo, eliminando l'interdipendenza con il substrato software che permette l'esecuzione dei programmi: in altre parole, cambiando il sistema operativo non cambia il **codice sorgente**, ma solamente la piattaforma sulla quale esso viene eseguito; si dice, in tale caso, che **Java è un linguaggio portabile**.
Tuttavia, se da un lato tale impostazione costituisce un vantaggio, dall'altro rappresenta anche un limite: a parità di programma da eseguire, un programma scritto in Java sarà più lento, nell'esecuzione, di un programma scritto in C, per esempio, il quale è fortemente dipendente dall'ambiente software di esecuzione e permette di controllarne la gestione della memoria e dell'indirizzamento al fine di ottenere un'ottimizzazione massima. Questo in quanto Java è un **linguaggio interpretato**, che necessita prima di una **pseucodifica** per essere seguito su una **macchina virtuale**, mentre C è un *linguaggio compilato direttamente* sfruttrando la struttura del sistema operativo nativo.

1.1 Editor

La realizzazione di un programma in Java parte dalla scrittura del *codice sorgente* mediante un **editor**, anche chiamate **IDE** (dall'inglese *Integrated Development Environment*), ossia un programma di utilità che consente di inserire e memorizzare un testo e di modificarlo successivamente per effettuare aggiunte o correzioni.

1.2 Compilazione

Il codice sorgente scritto in Java che, proprio per questo, presenta estensione **.java**, al fine di poter essere eseguito dalla macchina, deve essere **compilato** da un **compilatore** (che nel caso di Java è il **javac**, o **JDK**, il quale permette di trasformare delle istruzione elaborate con un linguaggio *user-oriented* (di **alto livello**) in un linguaggio *machine-oriented* (di **basso livello**), ovverosia più vicino alla macchina, tale per cui la macchina lo possa comprendere ed eseguire. Nel caso di Java, la "macchina" che andrà ad eseguire le istruzioni così tradotte è una **macchina virtuale**, mentre nel caso del linguaggio C, la compilazione serve ad ottimizzare il programma finale, grazie alla forte interdipendenza tra linguaggio di programmazione e hardware sottostante.
Il risultato della compilazione di un codice sorgente .java è il **codice compilato** (o **bytecode**), con estensione **.class**: esso non deve essere più compilato e può essere eseguito su ogni piattaforma hardware-software.

1.3 Lettura e interpretazione

Il codice compilato (o bytecode), viene letto dal **loader**, ossia dalla macchina virtuale **java JRE**, la quale effettua la verifica del codice compilato (**bytecode verifier**) e ne interpreta le istruzioni.

1.4 Ambienti

Il linguaggio Java, per quello che si è detto, è un **linguaggio portabile** (in quanto può essere eseguito in diversi ambienti hardware e software) e un **linguaggio ad oggetti puro**, in quanto si basa sul **paradigma di programmazione orientato agli oggetti** (OOL, Object Oriented Language). Due, inolre, sono delle tecnologie che permettono una maggiore gestione dei programmi:

1. **Garbage Collector**: tool di gestione della memoria, per motivi di sicurezza e di semplicità di programmazione, pulendo ed elimando locazioni di memoria inutilizzate;

2. **Multithreading**: tecnologia che consente la gestione dell'esecuzione contemporanea di più task all'interno della stessa applicazione. Il processore, tramite la tecnologia **time sharing** permette di gestire lo scheduling dei processi in modo tale da permettere un'esecuzione quanto più simultanea delle diverse task.

La tecnologia Java, infine, si avvale di tre importanti tool per la sua esecuzione:

1. **JDK** (Java Development Kit): un set completo di **API** (librerie di funzioni precostituite che consentono di eseguire operazioni automaticamente) e una serie di tool per lo sviluppo di applicazioni Java;
2. **JSE** (Java Standard Edition): pacchhetto base per lo sviluppo di normali applicazioni Java (che possono andare dagli **Applets** a **Desktop applications** con *Swing* o *Java FX* per l'interfaccia grafica). Esistono anche **JEE** (Java Enterprise Edition) per lo sviluppo di applicazioni server di grandi aziende e **JME** (Java Mobile Edition) per lo sviluppo di applicazioni mobile.
3. **JRE** (Java Runtime Environment), ambiente per l'esecuzione delle applicazioni Java tramite **JVM**, o Java Virtual Machine.

1.5 Linguaggi di programmazione

I linguaggi di programmazione si possono suddividere in due grandi categorie:

1. linguaggi di **basso livello** (**linguaggi macchina** e Assembly), che sono, quindi, linguaggi che il calcolatore è in grado di comprendere;
2. **linguaggi ad alto livello** o **linguaggi evoluti**, quindi più vicini al programmatore e che consentono l'uso di **nomi simbolici** che corrispondono a più set di istruzioni linugaggio macchina.

1.6 Programmi traduttori: compilatore e interprete

Di seguito si espone la definzione di **compilatore**:

COMPILATORE

Il **compilatore** è un programma che traduce un programma sorgente scritto in linguaggio ad alto livello in un programma oggetto in linguaggio macchina.
Esso, inoltre, svolge tre importanti funzioni di analisi:

1. **analisi lessicale**: consente di individuare le parole chiave e riservate del linguaggio di programmazione, ccosì come del vocabolario e permette di costruire la tabella dei simboli;
2. **analisi sintattica**: ossia il controllo della correttezza delle istruzioni al fine di verificare se essse sono scritte rispettando la sintassi del linguaggio Java, cioé le regole della grammatica;
3. **analisi semantica**: ovvero il controllo della validità del significato degli elementi in base al contesto (per esempio la presenza delle istruzioni dichiarative necessarie).

Pertanto il compilatore individua e segnala tutti gli **errori formali**: se vi sono errori, la traduzione non può avvenire. Se non ci sono errori, il compilatore passa alla fase di sintesi durante la quale viene generato il codice oggetto, di solito in formato rilocabile.
Sarà poi compito del **linker** trasformare il programma oggetto in programma eseguibile (con estensione .class), il quale verrà poi eseguito dalla macchina virtuale JVM grazie all'**interprete** che traduce le istruzioni del linguaggio ad alto livello in linguaggio macchina, una per una, al momento dell'esecuzione.

Durante il **runtime** vengono anche rilevati degli errori di esecuzione, dovuti non alla scorrettezza formale del codice sorgente, ma ad un uso scorretto del programma.
L'impiego di un compilatore o di un inteprete presenta una serie di vantaggi e di svantaggi:

Compilatore	**Interprete**
VANTAGGI	
maggior velocità di esecuzione	semplicità di messa a punto dei programmi poiché si lavora in ambiente interattivo
risparmio di memoria	maggior portabilità dei programmi
rilevazione di tutti gli errori formali	
consente la segretezza del programma sorgente, in quanto dopo essere stato compilato non è possibile risalire al codice di partenza	
non è necessario codice aggiuntivo	
SVANTAGGI	
tempi di creazione del programma più lunghi, per la ricerche di librerie e funzioni specifiche	maggiore occupazione di memoria
minore portabilità	l'esecuzione risulta più lenta
	non dà garanzia di correttezza sintattica
	non consente la segretezza del codice sorgente

Figura 1: Vantaggi e svantaggi del compilatore e dell'interprete

Osservazione: Si osservi che Java è **parzialmente compilato** e **interpretato**.

1.7 Soluzione di problemi con il calcolatore

In informatica, un **problema** è una qualsiasi situazione alla quale è necessario trovare una soluzione. Per la risoluzione di un problema è necessario analizzare diversi elementi

1. la situazione iniziale (quali sono i dati del problema): i dati che riguardano il problema sono tutte le informazioni disponibili all'inizio e quelle che si desiderano ottenere come soluzione del problema;
2. che cosa si desidera ottenere (risultati): le azioni che possono essere eseguite nel processo risolutivo sono le operazioni che il computer è in grado di eseguire; l'insieme delle istruzione costituisce l'**algoritmo**;
3. le risorse a disposizione, variabili a seconda dell'hardware sottostante.

1.8 Algoritmo e programma

Di seguito si espone la definizione di **algoritmo**:

ALGORITMO

Un algoritmo è un procedimento che permette di ottenere dei risultati (dati in uscita o di **output**) partendo da alcuni dati iniziali (dati di ingresso o di **input**).
L'algoritmo deve essere

1. **generale**: non deve risolvere un singolo caso particolare, ma tutta una serie di problemi dello stesso tipo;
2. **deterministico**: i risultati devono essere sempre gli stessi, partendo dagli stessi dati iniziali, cioè l'esito dell'esecuzione dell'agoritmo non deve dipendere da aleatorietà o effetti casuali.

Il **programma**, invece, si ottiene codificando l'algoritmo in un **linguaggio di programmazione**, cioè scrivendo le istruzioni dell'algoritmo secondo la sintassi del linguaggio scelto.

4 Marzo 2022

2 Pratica di programmazione in Java

Si consideri il seguente esempio di codice Java, ovvero il codice sorgente:

```
1  public class Java1
2  {
3    public static void main(String args[]) {
4      System.out.println("Ciao mondo!");
5    }
6  }
```

Listing 1: Esempio di codice sorgente in Java

Da osservare che il nome del file **Java1.java** deve essere lo stesso della classe principale. All'interno del codice vi sono diverse **parole riservate** che vengono opportunamente segnalate con una colorazione differente: essendo riservate, esse non possono essere utilizzate dal programmatore per il proprio codice sorgente, ma solamente per la definizione di codice strutturale.
La parola riservata *class* è la base del paradigma di programazione orientato agli oggetti. Il **metodo principale** *main* viene così chiamato perché è il metodo centrale del programma; inoltre la sua stringa di definizione serve per indicare che il proramma può richiedere memoria per essere eseguito.
La **keyword** *void* indica che tale metodo non ritorna nessun valore dopo la sua esecuzione, mentre *static* significa che al metodo viene assegnato un **indirizzo di memoria fisso** (statico), per cui è possibile richiamare il programma per la sua esecuzione, il quale verrà eseguito in una parte di memoria statica. La **keyword** *public* sta ad indicare che tale funzione è visibile globalmente e può essere richiamata, quindi eseguita, anche da altri programmi.
Le parentesi graffe servono per racchiudere al loro interno uno **scope**, ovverosia un'ambiente limitato, in cui le dichiarazioni e le funzioni in esso presenti sono valide solamente all'interno dello scope stesso e non possono essere viste esternamente.
La riga 5 rappresenta il vero e proprio programma e, richiamando un metodo (*println*) presente all'interno di una sub-libreria (*out*) di una libreria del sistema operativo (*System*), permette di scrivere a schermo la **stringa** "Ciao mondo!" (si potrebbe anche stmpare un numero).
Dopo aver elaborato il proprio codice sorgente, si procede alla compilazione dello stesso: ogni possibile errore di sintassi o di struttura del programma viene segnalato dal compilatore stesso.
Per la compilazione è sufficiente scrivere

```
1  javac Java1.java
```

Listing 2: Esempio di compilazione in Java

Se la compilazione va a buon fine si ottiene il codice compilato **Java1.class** e per la sua esecuzione è sufficiente scrivere

```
1  java Java1
```

Listing 3: Esempio di esecuzione in Java

2.1 Computabilità

Di seguito si fornisce la definizione di **computabilità**:

COMPUTABILITÀ

Un problema si dice **computabile** se la sua soluzione può essere descritta mediante un algoritmo. Per contro, i problemi che non sono risolvibili con un procedimento algoritmico si dicono **non computabili**.

2.2 Paradigma di programmazione

Al fine di permettere una corretta computabilità del proprio codice è necessario prescieglire il paradigma di programmazione più adatto alla risoluzione del problema. Di seguito si fornisce la definizione di **paradigma di programmazione**:

PARADIGMA DI PROGRAMMAZIONE

Un **paradigma di programmazione** è un modello che permette di descrivere astrattamente l'algoritmo (cioè il metodo di soluzione di un problema).

La descrizione dei dati e delle azioni per risolvere il problema dipende dal **paradigma di programmazione** utilizzato e per linguaggi che usano lo stesso paradigma, almeno parzialmente dal linguaggio scelto: infatti potrebbero essere disponibili per esempio tipi o strutture di dati diverse. I principali paradigmi di programmazione sono:

- il paradigma **dichiarativo**, il quale descrive **come** deve essere risolto un problema;
- il paradigma **procedurale**, il quale descrive **che cosa** si vuole ottenere;

2.2.1 Paradigma dichiarativo

La programmazione con il **paradigma dichiarativo** si può suddividere in:

- **programmazione logica**: descrive i fatti e le relazioni tra i fatti e permette di ricavarne delle conseguenze, quale [PROLOG];
- **programmazione funzionale**: il programma è costituito da un insieme di funzioni che elaborano liste di simboli, quale [LISP];
- **linguaggi di markup**: usano dei codici (*tag*) per stabilire il ruolo degli elementi, quale [HTML];
- **linguaggi di interrogazione di database**, quale [SQL].

2.2.2 Paradigma procedurale

La programmazione con il **paradigma procedurale** si può suddividere in:

- **programmazione imperativa**: si basa su esplicite richieste fatte all'esecutore del programma; la soluzione del problema è descritta come una sequenza di azioni che producono dei cambiamenti di stato nell'ambiente (come la modifica del valore delle variabili); esempi di tali linguaggi sono Assembler, pascal, C, Cobol;
- **programmazione orientata agli oggetti**: parte dal concetto di oggetto che descrive proprietà e azioni che un oggetto può compiere; la soluzione del problema è descritta dalle interazioni tra gli oggetti; esempi di tali linguaggi sono Java, C++, C#, Python.

2.3 Linguaggio imperativo

Il vocabolario usato da ciascun linguaggio comprende delle parole riservate e delle parole che possono essere definite dal programmatore seguendo alcune semplici regole.

- Le parole **riservate** permettono di definire i dati e di scrivere le istruzioni;
- Le parole **definite** dal programmatore (o **identificatori**) sono in genere i nomi delle variabili e i nomi delle procedure da richiamare.

Esempio: Si consideri il seguente esempio di codice Java che permette di realizzare una somma tra due numeri costanti $(3 + 4)$:

```
public class Somma
{
  public static void main(String args[]) {
    int num1, num2, somma; // Dichiarazione delle variabili senza inzializzazione

    num1=3; // Assegnazione del valore 3 alla variabile num1
    num2=4; // Assegnazione del valore 4 alla variabile num2

    somma=num1+num2;

    System.out.println(num1 + "+" + num2 + "=" + Integer.toString(somma));
  }
}
```

Listing 4: Somma di due numeri costanti in Java

Pertanto, la sintassi di dichiarazione di una variabile è la seguente

```
[modificatore] tipo_di_dato nome_della_variabile [=inizializzazione];
```

Listing 5: Dichiarazione di una variabile in Java

Per *[modificatore]* è da intendersi una specifica della visibilità della variabile (protected, private o public).
Il *tipo di dato* da specificare è essenziale, in quanto in base ad esso si ha una differente gestione della memoria.
Anche il *nome della variabile* è fondamentale, in quanto esso deve essere **significativo** e deve aiutare il prgorammatore nella memorizzazione delle variabili dichiarate e da impiegare. È molto utile in questo l'utilizzo della notazione **camelCase**, che prevede di indicare la prima letterea di una parola in maiuscolo e il restante in minuscolo: se si considera una *variabile*, essa deve essere scritta con iniziale minuscola e poi impiegando la notazione camelCase, così come per una *funzione*; se si consiera una *costante*, essa vine scritta tutta in maisucolo; se si considera una *classe*, essa avrà l'iniziale maiucola: questo anche perché il linguaggio Java è **case-sensitive**.
Le variabili, naturalmente, devono essere interpretate come delle regioni di memoria, limitate e non immutabili, in cui vengono memorizzate informazioni di diversa natura. È possibile, naturalmente, riservare al proprio programma diverse regioni di memoria, con un limite di dimensione, ovviamente, e specificando in maniera esplicita la tipologia di informazione da memorizzare. Ovviamente esiste una sostanziale differenza tra dichiarazione di una variabile e innizializzazione della stessa: infatti, è possibile dichiarare una variabile senza bisogno di inizializzarla.
Da notare, tuttavia, che la dichiarazione deve sempre essere preceduta dalla specifica della tipologia di dato; ogni assegnazione (che rappresenta un aggiornamento del contenuto informativo della variabile), invece, eccezion fatta per l'inizializzazione, non deve mai essere preceduta dalla specifica della tipologia del dato.
Naturalmente, una variabile viene utilizzata per contenere delle informazioni, le quali possono essere recuperate semplicemente specificando il nome della variabiele di interesse all'interno del programma.

Osservazione: Naturalmente le stringhe esplicite, all'interno del progrmma, devono essere sempre rcchiuse dalle dalle virgolette, mentre affinché una variabile venga letta e interpretata come tale deve essere specificata con solo il suio nome e non tra le virgolette.
La concatenazione delle stringhe, nel linguaggio Java, avviene tramite il simbolo di addizione "+".
Naturalmente, per commentare parti di codice si utilizza una sequenza di caratteri racchiusa tra "/*" e "*/" (**commento tradizionale**), oppure tra "//" e un terminatore di linea (commento di fine linea). Naturalmente, le parti di codice che vengono contrassegnate come commento, all'interno del programma verranno totalmente ignorate e non compilate dal compilatore stesso.
La conversione da stringa a intero avviene grazie al metodo **toString()** della libreria **Integer**, il quale prende come parametro proprio la stringa da convertire. In questo caso, la libreria impiegata è stata elaborata da un'altro svilupatore, il quale l'ha messa a disposizione del linguaggio Java e fancendo sì che essa possa essere richiamata senza bisogno di codice aggiuntivo.

2.4 Errori in Java

Un esempio molto comune di **errore** in Java è quello **di sintassi**, che recita: "**cannot find symbol**": tale errore si verifica quando viene utilizzato all'interno del codice una variabile non dichiarata o un comando non previsto dal liguaggio Java stesso; Il messaggio di errore "; expected" indica che un'istruzione non è stata terminata correttamente dal punto-vrigola, il quale è un elemento fondameentale nella programmazione per segnalare a Java che un'istruzione è stata terminata.
Un **errore**, invece, **di esecuzione**, a differenza di un errore di programmazione o sintassi come quelli precedenti, come una **Exception**, è un errore che il programmatore commette nel definire le operazioni del programma (come dividere zero per zero o aprire un file inesistente, o accedere a delle posizioni di un array fuori range, per esempio) e, quindi, non genera un messaggio di errore da parte del compilatore in fase di compilazione, ma è un errore che si palesa durante l'esecuzione (si parla, in questo caso, di **Runtime Exception**).
Un **errore logico**, invece, è un errore che il programmatore commette nel definire la logica della programmazione: pertanto la sintassi è corretta e le operzioni che il programma deve svolgere sono corrette; tuttavia ciò che il programma deve fare non è corretto in termini logici. Questo, ovviamente, è il più difficile da individuare: è il cosiddetto **bug**;

7 Marzo 2022

3 I dati in Java

L'algoritmo non è soltanto un **insieme di istruzioni**; una componente fondamentale dell'algoritmo è costituita dai **dati** e una fase essenziale della stesura dell'algoritmo è la scelta dei dati da utilizzare. Java è un linguaggio **strongly typed**, ovvero di **tipicizzazione stretta**. I dati, in particolare, sono di due tipi

- dati di **input**: l'utente deve fornire un input al programma per l'esecuzione
- dati di **output**: il risultato dell'elaborazione, cioé i dati che il programma restituisce all'utente

mentre i **tipi di dati** sono due categorie

1. Primitivi
   (a) Integral
      i. Primitivi
      ii. boolean
      iii. byte
      iv. short
      v. int
      vi. long
      vii. char
      viii. float
   (b) Floating
      i. float
      ii. double
   (c) enumerate
2. Derivati
   (a) array
   (b) class
   (c) interface

I tipi di dati primitivi maggiormente utilizzati sono

Tipo Primitivo	Descrizione
boolean	valori che possono essere *true* o *false*
char	caratteri di 16 bit (UNICODE)
byte	interi di 8 bit con segno
short	interi di 16 bit con segno
int	interi di 32 bit con segno
long	interi di 64 bit con segno
float	reali di 32 in virgola mobile
double	reali di 64 bit

Da notare che una variabile di tipo primitivo può essere utilizzata direttamente (senza la clausola **new**).

3.1 Booleano

Il tipo **boolean** (booleano) ha come insieme di valori le due costanti simboliche *true* e *false*.
Con elementi di questo tipo si possono effettuare delle operazioni logiche, le quali producono come risultato un valore booleano. Esse sono:

1. "|", ossia barra verticale o "pipe", che corrisponde ad un OR logico (disgiunzione);
2. "^", ossia un apice, che corrisponde ad un OR-ESCLUSIVO logico o XOR;
3. "&", ossia la "E" commerciale, che corrisponde ad un AND logico (congiunzione);
4. "!", ossia il punto esclamativo, che corrisponde ad un NOT logico (negazione);
5. "==", ossia un "doppio uguale", che corrisponde ad un'uguaglianza;
6. "!=", che corrisponde al "diverso";

Si consideri l'esempio seguente:

```
1  public class Comparazione
2  {
3    public static void main(String args[]) {
4      int x = 5;
5
6      System.out.println(x + "== 4" + (x==4));
7    }
8  }
```

Listing 6: Comparazione tra due numeri costanti in Java

3.2 Intero

Un tipo **intero** ha per valori tutti i numeri interi compresi fra -2^{n-1} e $+2^{n-1}-1$, dove n è il numero di bit usati per la sua rappresentazione.
Con elementi di questo tipo **int** si possono eseguire operazioni elementari, che sono:

1. “+”, somma;
2. “−”, sottrazione;
3. “∗”, moltiplicazione;
4. “/”, quoziente della divisione;
5. “%”, resto della divisione;
6. “++”, incremento di una unità;
7. “−−”, decremento di una unità;

È possibile anche eseguire il **confronto** tra dati di tipo intero, quali

1. “==”, uguale;
2. “! =”, diverso;
3. “>”, maggiore;
4. “>=”, maggiore o uguale;
5. “<”, minore;
6. “<=”, minore o uguale;

ancora una volta, il risultato di tale comparazione è un valore booleano. I numeri interi possono esssere convertiti anche in caratteri, seguendo la codifica della tabella ASCII, come mostrato di seguito:

```
public class ConversioneASCII
{
  public static void main(String args[]) {
    int x = 44;
    int y = 65;

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

Listing 7: Conversione da intero a carattere in Java

Con gli **interi** è possibile effettuare **operazioni bit a bit**, che richiedono, in ogni caso, delle conoscenze sulle rappresentazione dei numeri interi e vengono utilizzate raramente: si impiegano quando si necessita di lavorare ad un basso livello:

Operazioni bit a bit	
	complemento (unario) bit a bit
&	AND bit a bit
|	OR bit a bit
^	OR ESCLUSIVO bit a bit
<<	traslazione a sinistra della distanza specificata inserendo bit uguali a 0
>>	traslazione a destra della distanza specificata replicando il segno (bit più significativo)
>>>	traslazione a destra della distanza specificata inserendo bit uguali a 0

3.3 Floating

Con i dati di tipo "floating" possono essere rappresentati i tipi reali:

1. $+, -$, ossia più unario e meno unario
2. $++, --$, incremento e decremento
3. $+, -$, somma e sottrazione
4. $*, /$, moltiplicazione e divisione reale
5. % resto $\boldsymbol{r}$ della divisione reale fra n e d, ossia

$$\boldsymbol{r} = n - d * q$$

Si consideri, a tal proposito, il seguente esempio:

```
1  public class RealNumbers
2  {
3    public static void main(String args[]) {
4      x1 = 2.8f;
5      x2 = 45.890d;
6      float y = 17f;
7      double z = 3d;
8      double r = y / z;
9
10     System.out.println("y / z = " + r);
11   }
12 }
```

Listing 8: Operazioni reali in Java

In generale non è necessario specificare sempre "f" o "d" per *float* o *double*: principalmente si sta specificando quanta memoria utilizzare.

3.4 Definizione di variabili

Le variabili di un tipo primitivo si definiscono come nei seguenti esempi:

```
1  public class DichiarazioneVariabili
2  {
3    public static void main(String args[]) {
4      int n;
5      doubled d = 10.0;
6      boolean b, bb = true;
7      charc c = '\t';
8    }
9  }
```

Listing 9: Dichiarazione di variabili in Java

I dati che descrivono un problema si possono suddividere in **costanti** e **variabili**:

- i dati variabili possono cambiare valore;
- i dati costanti sono dati che hanno un valore predefinito, che non cambia. Una costante è una variabile *final* con valore iniziale specificato, come mostrato di seguito. Generalmente il nome della variabile viene specificato in maiuscolo ed è importante che il valore non cambi, specialmente quando ad uno stesso codice sorgente lavorano più programmatori.

```
1  public class DichiarazioneCostanti
2  {
3    public static void main(String args[]) {
4      final int CNST_1 = 10;
5      final int CNST_2 = CNST_1*3;
6    }
7  }
```

Listing 10: Dichiarazione di costanti in Java

3.5 Espressioni

Un'espressione è composta da **operandi** e **operatori**, e viene calcolata eseguendo in sequenza una serie di operazioni.
Per ogni operatore, vengono prima valutati gli operandi e quindi applicato l'operatore stesso. Alcuni operatori richiedono esplicitamente che un operando sia una variabile, mentre l'utilizzo dei delimitatori "parentesi tonde" individua sottoespressioni che vengono considerate operandi e che quindi vengono calcolate per prime.

```
1  public class OrdineOperazioni
2  {
3    public static void main(String args[]) {
4      double x;
5
6      double y1 = x / 3 * 2;
7      double y2 = (x / 3) * 2;
8    }
9  }
```

Listing 11: Ordine delle operazioni in Java

Osservazione: Gli operatori / e $*$ hanno la stessa precedenza, per cui l'ordine di esecuzione viene determinato dall'associatività: gli operatori aritmetici binari sono associativi a sinistra, e l'espressione $x/3*2$ viene calcolata come se fosse scritta $(x/3)*2$.

3.6 Operatore condizionale

L'operatore condizionale ha la forma:

```
1  public class OperatoreCondizionale
2  {
3    public static void main(String args[]) {
4      op1 ? op2 : op3;
5    }
6  }
```

Listing 12: Operatore condizionale in Java

Tale operatore richiede che il primo operando *op1* sia di tipo booleano: se questo vale *true* il risultato è il valore dell'operando *op2*, altrimenti il valore dell'operando *op3*.

3.7 Incremento e decremento

Gli operatori (unari) "++" e "−−" richiedono come operando una variabile:

- il valore aggiornato della variabile se prefissi;
- il valore originario della variabile se postfissi.

Quando tali operatori sono prefissi hanno la priorità nell'operazione, mentre se sono postfissi, prima viene eseguita l'operazione e poi dopo l'incremento/decremento, come mostrato di seguito

```
1  public class Incremento
2  {
3    public static void main(String args[]) {
4      int a = 5;
5      System.out.println("Il valore di a e' uguale a" + a++);
6      a = 5;
7      System.out.println("Il valore di a e' uguale a" + ++a);
8    }
9  }
```

Listing 13: Incremento in Java

8 Marzo 2022

3.8 Altri operatori

Di seguito si espongono ulteriori operatori utili alla programazione in Java

Operatore	Descrizione
[]	Definisce un array, come **char**[]
.	Riferisce a un membro di un oggetto
(tipo)	Converte, ossia effettua un **casting** dela variabile alla quale quale è affiancato in base al *tipo* specificato, se il tipo di dato da convertire è compatibile con tale conversione (che non altera il contenuto informativo della variabile, ma solo la sua interpretazione, come numero o carattere, per esempio). La conversione deve rispettare una specifica gerarchia e può essere eseguita in modo diretto solamente tra i tipi di dato primitivo. Per i dati derivati è necessario impiegare metodoi opportuni di classi specifiche. Un programma interno alla macchina virtuale, che prende il nome di **Garbage-Collector**, opera all'interno dell'**heap** di memoria dinamica per cercare di ottimizzare lo spazio (come quando si istanza un oggetto senza poi disfarsene al termine del suo utilizzo).
new	Crea un nuovo oggetto e lo alloca in una parte dinamica della memoria dell'elaboratore, chiamata **heap**, più libera e non controllata dal sistema operativo, che differisce dall'area di memoria più contenuta, ma più veloce, riservata alle variabili primitive, chiamata **stack**, maggiormanete controllata dal sistema operativo.
instanceof	restituisce *true* se *op*1 è una istanza di *op*2, come nel caso di *op*1 *instanceof op*2

3.9 Casting - Conversione di tipo

Una conversione di tipo consente:

- nel caso di un **operatore aritmetico binario**, di trasformare il tipo di un operando nel tipo dell'altro operando;
- nel caso dell'**operatore di assegnazione**, di trasformare il tipo del risultato di una espressione nel tipo della variabile a cui il nuovo valore deve essere assegnato;
- nell'ambito dei tipi "integral" e "floating", rispettivamente, verso il tipo che **usa un maggior numero di bit per la rappresentazione**, con nessuna perdita di informazione.

3.10 Classe Math

La classe **Math** fa parte del package **java.lang**, contiene come **membri statici** (per cui non c'è bisogno di istanziare un nuovo oggetto Math con **new** per accedervi) alcune *costanti* e diverse *funzioni matematiche* di uso comune, come il valore assoluto, la radice n-esima, operazioni trigonometriche, arrotondamento, etc.

3.11 Classe String

Una **stringa** è una sequenza di caratteri, che può anche non avere alcun carattere (**stringa vuota**). Una variabile appartenente a questo tipo, detta variabile stringa, rappresenta il riferimento di un oggetto stringa che memorizza una stringa costante.

```
1  public class ClasseStringa
2  {
3    public static void main(String args[]) {
4      // Inizializzazione della stringa primaStringa con il parametro "Ciao mondo!"
5      String primaStringa = new String("Ciao mondo!");
6    }
7  }
```

Listing 14: Inizializzazione di una stringa in Java

All'interno della inizializzazione è presente il metodo costruttore **new String()** che permette di istanziare un oggetto della classe String specificata, allocandovi uno spazio di memoria nell'heap di memoria.
La classe **String** è una classe le cui istanze rappresentano dellle stringhe di caratteri alfanumerici. Tutte le stringhe nei programmi Java, come "abc", vengono implementate come istanze della classe *String*.

3.12 Classe Integer

La classe **Integer** consente di creare un oggetto classe a partire da un valore del tipo primitivo **int**: pertanto la classe *Integer* è è un **wrapper**: le classi wrapper sono predefinite nel linguaggio e servono a racchiudere in oggetti i valori di un tipo primitivo al fine di ottimizzarne i metodi e la manipolazione, come mostrato di seguito:

```
1  public class ClasseInteger
2  {
3    public static void main(String args[]) {
4      // Istanza di un Integer senza new, ma direttamente attraverso il metodo
5      // della classe Integer
6      Integer numero = Integer.parseInt("300");
7      System.out.println(numero * 2);
8    }
9  }
```

Listing 15: Classe Integer in Java

Altri esempi di classi wrapper sono

1. Boolean
2. Byte
3. Character
4. Integer
5. Long
6. Float
7. Double

3.13 Classe Console

All'interno della classe **Console** sono presenti le funzioni atte ad effettuare una emulazione dell'ambiente di linea di comando al fine di effettuare **letture da tastiera e scritture su video** di valori appartenenti ai principali tipi di dato primitivo (tipi boolean, char, int, double) e al tipo stringa (String).
Tale classe è estremamemente utile per ottimizzare il codice, facilitando l'interazione con l'utente, e renderlo versatile, ossia capace di adattarsi a diversi tipi di input e output che possono derivare dell'esecuzione del programma.
I programmi Java, infatti, possono utilizzare classi predefinite, ma anche e soprattuto facenti parte di un **package** esterno, tipicamente dei package che costituiscono le **Java API**, attraverso la loro importazione, come nel caso della libreria **java.io.Console** mostrata di seguito:

```
1  import java.io.Console;
2
3  public class ClasseConsole
4  {
5    public static void main(String args[]) {
6      Console console = System.console(); // Inizializzazione oggetto Console
7
8      System.out.println("Inserisci il primo numero:");
9      String input = console.readLine();  // Lettura dato da tastiera
10     int pNum = Integer.parseInt(input); // Conversione in intero del dato letto
11     System.out.println("Il primo numero e': " + pNum);
12
13     System.out.println("Inserisci il secondo numero:");
14     input = console.readLine();         // Lettura dato da tastiera
15     int sNum = Integer.parseInt(input); // Conversione in intero del dato letto
16     System.out.println("Il primo numero e': " + sNum);
17
18     System.out.println("La somma tra i due numeri e': " + (pNum+sNum));
19   }
20 }
```

Listing 16: Classe Console in Java

14 Marzo 2022

In Java esistono 3 principali tipi di input

- System.in (java.lang.System);
- String[] args, tramite il terminale;
- Console (java.io.Console);
- Scanner (java.util.Scanner);

3.14 Classe Scanner

La **lettura formattata** di valori di un tipo primitivo (escluso il tipo byte) o di tipo stringa, si effettuano comunemente utilizzando stream appartenenti alla classe **Scanner**.
La classe Scanner presenta diverse funzionalità:

- Lettura della **password** senza fare eco ai caratteri inseriti.
- I metodi di lettura **sono sincronizzati**.
- È possibile utilizzare la **sintassi della stringa di formato**.
- Non funziona in un ambiente non interattivo (come in un IDE).

A differenza della classe Console, la quale

- presenta metodi convenienti per analizzare le primitive (nextInt(), nextFloat(), ...).
- presenta metodi di lettura **non sincronizzati**.

Per impiegare la classe Scanner è necessario procedere all'importazione della libreria **java.util.Scanner**:

```
1  import java.io.Console;
2  import java.util.Scanner;
3
4  public class ClasseScanner
5  {
6    public static void main(String args[]) {
7      Console console = System.console();     // Inizializzazione oggetto Console
8      Scanner sc = new Scanner(con.reader()); // Inizializzazione oggetto Scanner
9
10     con.printf("**************************\n");
11     con.printf("Un altro programma! \n");
12     con.printf("**************************\n");
13
14     con.printf("Qual e' il tuo nome? \n >>>");
15     String nome = sc.nextLine();                 // Lettura dato da tastiera
16
17     con.printf("Ciao " + nome + "! Quanti anni hai? \n >>>");
18     int anni = Integer.parseInt(sc.nextLine());  // Lettura dato da tastiera
19
20     con.printf("Ok, grazie dell'informazione!");
21   }
22 }
```

Listing 17: Classe Scanner in Java

In cui è noto che **Integer** è un **wrapper**, ossia una classe che ingloba il tipo di dato primitivo **int** e ne amplia l'utilizzo, grazie a nuovi metodi complessi e l'utilizzo di memoria dinamica.
È possibile, tramite la classe Scanner, impiegare metodi per analizzare le primitive (nextInt(), nextFloat(), ...), come mostrato di seguito:

```java
import java.io.Console;
import java.util.Scanner;

public class IMC
{
  public static void main(String args[]) {
    Console console = System.console();     // Inizializzazione oggetto Console
    Scanner sc = new Scanner(con.reader()); // Inizializzazione oggetto Scanner

    con.printf("*****************************\n");
    con.printf("Calcolo - Indice Massa Corporea (IMC)");
    con.printf("*****************************\n");

    con.printf("Altezza: \n >>>");
    double altezza = sc.nextDouble();// Lettura dato da tastiera

    con.printf("Peso: \n >>>");
    int altezza = sc.nextInt();      // Lettura dato da tastiera

    con.printf("IMC = %.2f", (peso / (altezza*altezza)));
  }
}
```

Listing 18: Calcolo Indice Massa Corporea in Java

In cui "%.2f" specifica il formato con due cifre decimali.

3.15 Classi e Oggetti

I **tipi primitivi** prevedono la definizione di **variabili** di un dato tipo; com'è noto, una variabile di un tipo primitivo ha un nome e un valore e il tempo di vita di una tale variabile **dipende dal luogo in cui è definita** (per esempio, le variabili definite nel corpo di una funzione sono automatiche).
Per esempio, un oggetto **Stringa** viene creato (e allocato in memoria dinamica **heap**) per mezzo dell'operatore *new* seguito da un **costruttore**.
Il recupero della **memoria dinamica** non più utilizzata viene effettuata dalla piattaforma **Java** per mezzo di una specifica routine (chiamata **Garbage Collector**), che periodicamente libera quelle zone di memoria della macchina virtuale occupate da oggetti per i quali non esiste più alcuna variabile/oggetto che li riferisce.
Pertanto, mentre per le variabili di tipo primitivo è il sistema a gestirne l'allocazione, per gli oggetti, istanze di opportune classi, è il programmatore a dover fare un buon uso della memoria, coadiuvato dal programma **Garbage Collector**.

4 Istruzioni in Java

Di seguito si espongono le principali istruzioni in linguaggio Java, che permettono di strutturare in maniera più complessa il programma:

- Istruzioni codizionali
- Istruzioni ripetitive
- Istruzioni di salto

4.1 Istruzioni condizionali

Si consideri il seguente codice sorgente Java:

```
1  import java.io.Console;
2  import java.util.Scanner;
3
4  public class IMC2
5  {
6    public static void main(String args[]) {
7      Console console = System.console();      // Inizializzazione oggetto Console
8      Scanner sc = new Scanner(con.reader()); // Inizializzazione oggetto Scanner
9
10     con.printf("*****************************\n");
11     con.printf("Calcolo - Indice Massa Corporea (IMC) (V2)");
12     con.printf("*****************************\n");
13
14     con.printf("Altezza: \n >>>");
15     double altezza = sc.nextDouble(); // Lettura dato da tastiera
16
17     con.printf("Peso: \n >>>");
18     int altezza = sc.nextInt();       // Lettura dato da tastiera
19
20     float imc = (peso / (float)(altezza*altezza));
21     con.printf("Il tuo IMC e' %.2f, questo significa che sei ", IMC);
22
23     if(imc<16)
24       con.printf("sottopeso grave\n");
25     else if(imc<18.8)
26       con.printf("sottopeso\n");
27     else if(imc<25)
28       con.printf("normale\n");
29     else if(imc<30)
30       con.printf("sovrappeso\n");
31     else
32       con.printf("obeso\n");
33   }
34 }
```

Listing 19: Esempio dell'istruzione condizionale if in Java

Come si può notare, le **istruzioni condizionali** comprendono l'istruzione **if** e l'istruzione **switch**. La **condizione** è un'espressione che deve produrre un risultato di tipo **booleano** (notare che non è possibile nessuna conversione di tipo fra intero e booleano, né implicita né indicata esplicitamente dal programmatore). Da notare che la parte **else** può anche mancare, per cui se la condizione è vera, viene eseguita l'istruzione che segue **if**, altrimenti quella che segue **else** (se presente).

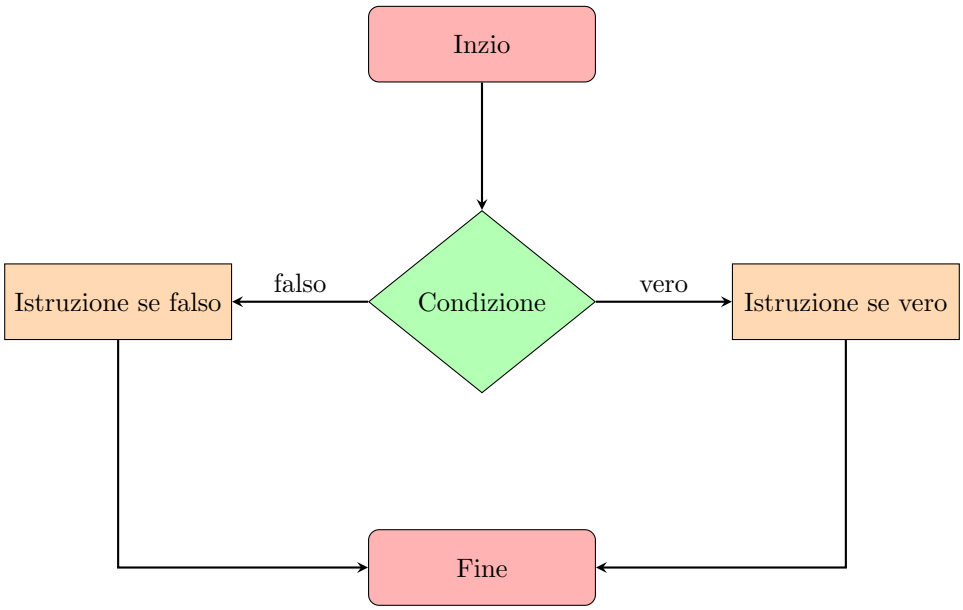

Figura 2: Diagramma di flusso con istruzione condizionale

Naturalmente, è possibile anche concatenare fra di loro diverse istruzioni condizionali, come mostrato di seguito:

```
public class Radici
{
  public static void main(String args[]) {
    double a,b,c,delta;
    a = Float.parseFloat(args[0]);
    b = Float.parseFloat(args[1]);
    c = Float.parseFloat(args[2]);

    if(a==0 && b==0)
      System.out.println("Equazione degenere");
    else if(a==0) {
      System.out.println("Equazione di primo grado");
      System.out.println(-c/b);
    } else {
      delta = b*b - 4*a*c;
      if(delta<0)
        System.out.println("Determinante negativo!");
      else {
        delta = Math.sqrt(delta);
        System.out.println("Equazione di secondo grado");
        System.out.println((-b+delta)/(2a));
        System.out.println((-b-delta)/(2a));
      }
    }
  }
}
```

Listing 20: Calcolo delle radici in Java

In cui, naturalmente, se non vengono poste delle partesi graffe per delmitare lo *scope* dell'istruzione condizionale significa implicitamente che lo *scope* è costituito da una sola riga di codice.

Usando l'istruzione **switch**, la condizione deve essere un'espressione di tipo *byte*, *char*, *short* o *int*, ed ogni espressione costante (*constant-expression*), valutata durante la compilazione.
La sequenza di istruzioni contenute nell'alternativa selezionata viene quindi eseguita con la seguente logica:

- l'ultima istruzione della sequenza deve produrre la terminazione dell'istruzione **switch**.

- tale terminazione può essere ottenuta con un'istruzione **break**. Se tale istruzione viene a mancare, verranno eseguiti tutti i case sottostanti, fino a che non viene incontrato un **break**.

Si consideri l'esempio seguente:

```
1  public class NumeroPari
2  {
3    public static void main(String args[]) {
4      int numero = Integer.parseInt(args[0]);
5      switch(numero%2){
6        case 0:
7          System.out.println("Il numero " + numero + " e' pari");
8        break;
9
10       default:
11         System.out.println("Il numero " + numero + " e' dispari");
12       break;
13     }
14   }
15 }
```

Listing 21: Esempio dell'istruzione condizionale switch in Java

Osservazione: Si osservi che è anche possibile ripetere uno stesso **case** più volte, ma verrà sempre eseguito il primo per ordine di precedenza.
Inoltre si possono raggruppare anche più **case** insieme, come mostrato di seguito:

```
1  public class NumeroPari
2  {
3    public static void main(String args[]) {
4      int numero = Integer.parseInt(args[0]);
5      switch(numero){
6        case 0: case 2: case 4: case 6: case 8: case 10:
7          System.out.println("Il numero " + numero + " e' pari");
8        break;
9
10       case 1: case 3: case 5:
11       System.exit(1);
12
13       default:
14         System.out.println("Il numero " + numero + " e' dispari");
15       break;
16     }
17   }
18 }
```

Listing 22: Esempio di multiple-case switch in Java

Osservazione: Si osservi che l'istruzione **System.exit(1)** permette di terminare l'esecuzione del programma corrente (esattamente come l'istruzione **return**).

15 Marzo 2022

4.2 Istruzioni ripetitive

Le **istruzioni ripetitive** comprendono l'istruzione *while*, l'istruzione *do-while* e l'istruzione *for*:

- **While**: La **condizione viene valutata come prima istruzione** ed è un'espressione che deve produrre un risultato di tipo booleano. Se la condizione è vera, viene eseguita l'istruzione che costituisce il corpo del costrutto e l'istruzione while viene ripetuta, altrimenti l'istruzione while termina.
- **Do-While**: La **condizione viene valutata dopo** che è stata eseguita l'istruzione che costituisce il corpo del costrutto, e, se vera, l'istruzione do-while viene ripetuta.
- **For**: L'inizializzazione è un'espressione di assegnamento che inizializza una variabile di controllo (eventualmente con definizione di tale variabile), e l'aggiornamento (o passo) è un'espressione che dà un nuovo valore alla variabile di controllo stessa (la incrementa o la decrementa, generalmente).

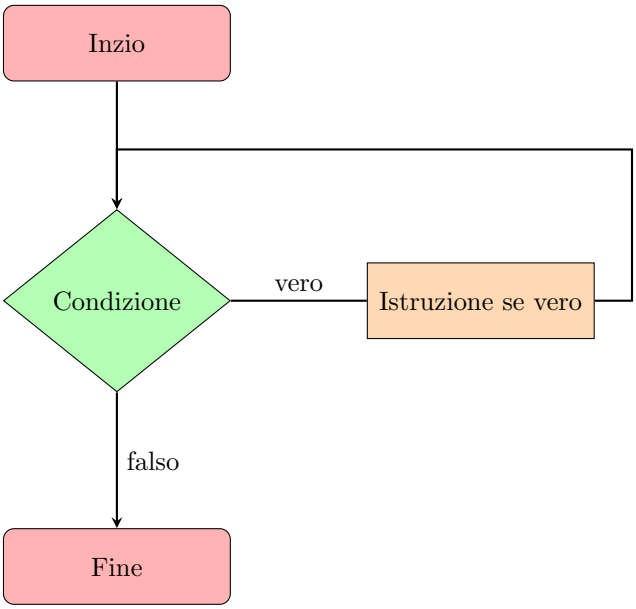

Figura 3: Diagramma di flusso con istruzione ripetitiva

Esempio: Si espone di seguito un programma che mostra un primo utilizzo dell'istruzione ripetitiva **while**:

```
1  public class MCD
2  {
3    public static void main(String args[]) {
4      int alfa,beta,mcd;
5      alfa = Integer.parseInt(args[0]);
6      beta = Integer.parseInt(args[1]);
7
8      while(alfa != beta) {
9        System.out.println(":alfa"+alfa);
10       System.out.println(":beta"+beta);
11       if(alfa > beta)
12         alfa -= beta;
13       else
14         beta -= alfa;
15     }
16     mcd = alfa;
17     System.out.println("*************** MCD:"+mcd);
18   }
19 }
```

Listing 23: Istruzione ripetitiva while in Java

Esempio: Si espone di seguito un programma che mostra un primo utilizzo dell'istruzione ripetitiva **do-while**:

```
import java.io.Console;
import java.util.Scanner;

public class InputPari
{
  public static void main(String args[]) {
    int c;

    Console con = System.console();
    Scanner sc = new Scanner(con.reader());

    do{
      con.printf("Immettere un numero pari:");
      c = Integer.parseInt(sc.next());
      System.out.println(":beta"+beta);
    }while(c%2!=0);
    mcd = alfa;
    con.printf("Grazie!\n");
  }
}
```

Listing 24: Istruzione ripetitiva do-while in Java

Esempio: Si espone di seguito un programma che mostra un primo utilizzo dell'istruzione ripetitiva **do-while**:

```
public class Fatt
{
  public static void main(String args[]) {
    int num, fattoriale=1;
    num = Integer.parseInt(args[0]);

    for(int i=1;i<=num;i++){
      fattoriale*=i;
      System.out.println(fattoriale);
    }
    System.out.println(fattoriale);
  }
}
```

Listing 25: Istruzione ripetitiva for in Java

4.3 Istruzioni di salto

Un'**istruzione di salto** cambia il flusso di ripetizioni di esecuzioni dei codice. Le istruzioni di salto sono le seguenti:

- Istruzione **break**: Se l'identificatore viene inserito, esso produce la terminazione del costrutto **switch**, **while**, **do-while** o **for** in cui compare;
- Istruzione **continue**: Se l'identificatore viene inserito, esso produce la terminazione della iterazione corrente del ciclo **while**, **do-while** o **for** in cui compare, con esecuzione della iterazione successiva;
- Istruzione **return**: Essa produce la terminazione della funzione in cui compare.

Esempio: Si espone di seguito un programma che mostra un primo utilizzo dell'istruzione di salto **break**:

```
public class EsempioBreak
{
  public static void main(String args[]) {
    for(int i=1;i<=10;i++){
      System.out.println("i:"+i);
      if(i==5) break;
    }
    System.out.println("Fine!");
  }
}
```

Listing 26: Istruzione di salto break in Java

Esempio: Si espone di seguito un programma che mostra un primo utilizzo dell'istruzione di salto **continue**:

```
public class EsempioContinue
{
  public static void main(String args[]) {
    for(int i=1;i<=10;i++){
      if(i==5) continue;
      System.out.println("i:"+i);
    }
    System.out.println("Fine!");
  }
}
```

Listing 27: Istruzione di salto continue in Java

Esempio: Si espone di seguito un programma che mostra un primo utilizzo dell'istruzione di salto **return**:

```
public class EsempioReturn
{
  public static void main(String args[]) {
    for(int i=1;i<=10;i++){
      System.out.println("i:"+i);
      if(i==5) return;
    }
    System.out.println("Fine!");
  }
}
```

Listing 28: Istruzione di salto return in Java

21 Marzo 2022

5 Array

Com'è noto, in Java esistono **tipi di dato primitivi**, i quali prevedono la definizione di variabili di un dato tipo, ove per variabile di tipo primitivo è da intendersi una locazione di memoria nello **stack** avente un **nome** e un **valore**; il **tempo di vita** di una variabile dipende dal luogo (*scope*) in cui è definita.
I **tipi derivati** (enumerazioni, array, stringhe, ecc) prevedono la definizione del tipo, la definizione di variabili di quel tipo e la creazione di oggetti di quel tipo: un tipo derivato è costituito da più componenti (anche una sola) raggruppate in un'unica entità; una variabile di un tipo derivato ha le stesse caratteristiche di una variabile di un tipo primitivo, con la differenza che il suo valore è un indirizzo, quello di un oggetto di quel tipo. Un oggetto di un tipo derivato ha un tempo di vita dinamico: viene creato esplicitamente quando serve (per mezzo dell'operatore new), allocato in una zona di memoria, detta, appunto, **memoria dinamica** (o **heap**).

5.1 Enumerazione

I tipi **enumerazione** vengono usati per rappresentare un numero limitato di valori associati a informazioni non numeriche, per cui gli enumeratori sono identificatori di costanti e gli oggetti riferiti dagli enumeratori vengono creati implicitamente nel momento in cui viene definito il tipo.

```
1  public class Semaforo
2  {
3    public static void main(String args[]) {
4      enum SemaforoEnum {verde,giallo,rosso};
5      SemaforoEnum attuale = SemaforoEnum.verde;
6      System.out.println("Il valore attuale e': "+attuale);
7      for(SemaforoEnum i : SemaforoEnum.values())
8        System.out.println(i);
9    }
10 }
```

Listing 29: Enumerazione in Java

5.2 Array monodimensionale

Il tipo di dato **aray** rappresenta un **aggregato** di elementi di uno **stesso tipo**: ogni elemento si accede mediante un indice che ne individua la posizione (un numero intero positivo n di elementi, il cui indice va da 0 a $n-1$); gli elementi di un array vengono selezionati per mezzo dell'operatore di indicizzazione "[]".

```
1  public class EsempioArray
2  {
3    public static void main(String args[]) {
4      int[] numArray1 = new int[5]; // Dichiarazione e inizializzazione dell'array
5      int[] numArray2[];            // Dichiarazione dell'array
6      int numArray3 = {1,2,3,4,5};  // Dichiarazione e inizializzazione dell'array
7
8      numArray1[1] = 300;           // Modifica di un valore dell'array
9      numArray2 = new int[100];     // Allocazione dell'array
10
11     System.out.println(numArray1[1]);
12     System.out.println(numArray2[99]);
13     System.out.println(numArray3[0]);
14   }
15 }
```

Listing 30: Esempio di array in Java

L'**operatore di assegnamento** (=) può essere applicato anche a variabili array dello stesso tipo: vengono però coinvolti i riferimenti, e non gli oggetti array, come mostrato di seguito:

```java
public class EsempioArray2
{
  public static void main(String args[]) {
    int[] numArray1 = {1,2,3,4}; // Dichiarazione e inizializzazione dell'array
    int[] numArray2[];           // Dichiarazione dell'array

    numArray2 = numArray1;
    System.out.println("numArray1:"+numArray1[0]);
    System.out.println("numArray2:"+numArray2[0]);

    numArray2[0] = 256;
    System.out.println("numArray1:"+numArray1[0]);
    System.out.println("numArray2:"+numArray2[0]);
  }
}
```

Listing 31: Operazioni di assegnamento tra array in Java

In cui *numArray1* e *numArray2* puntano alla stessa locazione di memoria: ogni modifica che riguarda uno dei due array viene riflesso anche nell'altro.
Per avere una copia dell'array e del suo contenuto e non solo dell'indirizzo di memoria bisogna impiegare il metodo **System.arraycopy(arr1,pStartOrig,arr2,pStartDest,len)**, in cui

- *arr1*: è l'array di origine;
- *pStartOrig*: è la posizione iniziale nell'array di origine;
- *arr2*: è l'array di destinazione;
- *pStartDest*: è la posizione iniziale nei dati di destinazione;
- *len*: è il numero di elementi dell'array da copiare.

5.3 Array multidimensionale

Un oggetto array (**array primario**) può avere come elementi altri oggetti array (**array secondari**): in questo caso negli elementi dell'array primario sono memorizzati i riferimenti degli array secondari (e non direttamente gli array secondari), come mostrato di seguito:

```java
public class EsempioArray2D
{
  public static void main(String args[]) {
    int[][] numArray2D = {{1,2,3,4}, {5,6}, {7}}; // Array bidimensionale

    for(int i=0;i<numArray2D.length;i++){
      for(int j=0;j<numArray2D[i].length;j++)
        System.out.println("["+i+"]["+j+"]:"+numArray2D[i][j]);
    }
  }
}
```

Listing 32: Array multidimensionale in Java

Osservazione: Per ricercare se un dato elemento è presente all'interno di un array, ed eventualmente conoscerne la posizione, occorre precisare se l'array è o meno ordinato: se l'array non è ordinato, non si può fare altro che una ricerca completa, come esposto di seguito:

```java
public class RicercaArray2D
{
  public static void main(String args[]) {
    int[][] numArray2D = new int[3][4]; // Inizializzazione array a 0

    numArray2D[1][3]=1;
    for(int i=0;i<numArray2D.length;i++){
      for(int j=0;j<numArray2D[i].length;j++)
        if(numArray2D[i][j]!=0)
          System.out.println("Found numArray2D["+i+"]["+j+"]:"+numArray2D[i][j]);
    }
  }
}
```

Listing 33: Array multidimensionale in Java

6 Stringa

Una stringa è una *sequenza di caratteri*, che può anche non avere alcun carattere (stringa vuota); in Java esiste come classe il tipo String (package java.lang): una variabile appartenente a questo tipo, detta variabile stringa, rappresenta il riferimento di un oggetto stringa.
Un oggetto stringa viene creato (e allocato in memoria dinamica) per mezzo dell'operatore **new** seguito da un costruttore:

```
public class DichiarazioneStringa
{
  public static void main(String args[]) {
    String str1 = "Prima stringa";
    String str2 = new String();
    str2 = "Seconda stringa";
    String str3 = new String("Terza stringa");

    System.out.println(str1);
    System.out.println(str2);
    System.out.println(str2);
  }
}
```

Listing 34: Esempio dichiarazione stringa in Java

6.1 Concatenazione

La concatenazione di stringhe avviene tramite l'operatore +:

```
public class DichiarazioneStringa
{
  public static void main(String args[]) {
    String str1 = new String("ABC");
    String str2 = new String("DEF");
    String conc = str1+str2;
    System.out.println(conc);

    conc+=100+"";
    System.out.println(conc);
  }
}
```

Listing 35: Dichiarazione di una stringa in Java

6.2 Lunghezza

La lunghezza di una stringa è determinata dal valore restituito dalla funzione **length()**, la quale è più propriamente un metodo della classe **String**:

```
public class LunghezzaStringa
{
  public static void main(String args[]) {
    String str1 = new String("Ciao mondo!");
    System.out.println("La lunghezza della stringa str1 e'"+str1.length);
  }
}
```

Listing 36: Lunghezza di una stringa in Java

Anche gli array presentano una propria lunghezza, la quale, però, non cosituisce un metodo di una classe precisa, ma rappresenta un attributo dell'array stesso, per cui si può richiamare come **arr.length**.

6.3 Uguaglianza

Per confrontare le stringhe memorizzate in due oggetti stringa si utilizza la funzione **equals()**;

```
1  public class ConfrontaStringhe
2  {
3    public static void main(String args[]) {
4      String str1 = "Italia";
5      String str2 = "Italia";
6      String str3 = "Brasile";
7
8      System.out.println(str1==str2);
9      System.out.println(str1.equals(str2));
10
11     System.out.println(str1==str3);
12     System.out.println(str1.equals(str3));
13   }
14 }
```

Listing 37: Uguaglianza fra stringhe in Java

Osservazione: La ragione della differenza tra == e *equals* è dovuta al fatto che == confronta l'indirizzo di memoria di due stringhe che è, ovviamente, differente, mentre *equals* confronta il contenuto memorizzato all'interno delle stringhe che può essere anche uguale.

6.4 Selezione di un carattere

Per conoscere il carattere di una stringa ad una specifica posizione bisogna impiegare la funzione **charAt()**:

```
1  public class CarattereStringa
2  {
3    public static void main(String args[]) {
4      String str1 = "123456789";
5      System.out.println("Il sesto carattere di "+str1+" e': " + str1.charAt(6));
6    }
7  }
```

Listing 38: Uguaglianza fra stringhe in Java

6.5 Confronto

Le stringhe memorizzate in due oggetti stringa possono essere confrontate fra loro (lessicograficamente) per mezzo della funzione compareTo().
Il risultato del confronto è

- negativo, se la prima stringa è più corta della seconda (e il valore di ritorno specifica il numero di caratteri di differenza);
- positivo, se la prima stringa è più lunga della seconda (e il valore di ritorno specifica il numero di caratteri di differenza);
- zero, se la prima stringa è uguale alla seconda.

```java
public class ConfrontoStringhe
{
  public static void main(String args[]) {
    String strP = "abc";
    String strM = "abcd";
    String strL = "abcdef";

    System.out.println(strP.compareTo(strM));
    System.out.println(strP.compareTo(strL));
    System.out.println(strL.compareTo(strM));
    System.out.println(strL.compareTo(strP));
    System.out.println(strP.compareTo("123"));
  }
}
```

Listing 39: Confronto fra stringhe in Java

Osservazione: Si osservi che, generalmente, per deteminare il risultato del confronto tra stringhe, si converte ciascun carattere secondo la tabella ASCII e si esegue la differenza fra i valori ottenuti: il risultato è proprio l'esito di tale operazione di differenza, come nell'ultimo caso sopra descritto, in cui il risultato della comparazione è 48 in quanto la differenza tra ciascun carattere alfanumerico è 48, secondo la tabella ASCII.

22 Marzo 2022

7 Funzioni in Java

Com'è noto, la classe principale di un programma è la funzione *main()*; tuttavia, è possibile creare delle nuove funzioni che saranno richiamata dalla funzione *main()* o da altre classi.
Nella definizione di una funzione occorre indicare:

- il modificatore di visibilità;
- il tipo di istanziazione (static o meno);
- il tipo di ritorno;
- il nome;
- eventuali parametri in ingresso;
- le istruzioni da eseguire

Osservazione: Si osservi che metodo *Java main()* è sempre statico, in modo tale che il compilatore possa richiamarlo senza la creazione di un oggetto o prima della creazione di un oggetto della classe.

7.1 Struttura

Il corpo di una funzione (essendo un blocco) non può contenere altre definizioni di funzione, ma solo definizioni di:

- variabili;
- classi.

Una funzione può restituire come valore un risultato, oppure non avere nessun risultato (in tale caso verrà specificato il tipo di ritorno tramite la parola chiave **void**); il corpo, inoltre, deve prevedere almeno una istruzione **return**, con la specifica dell'espressione il cui valore rappresenta il risultato stesso.

7.2 Visibilità

Le funzioni, come le stesse variabili, possono presentare differenti visibilità:

- **public**: sono accessibili sia dall'interno che dall'esterno della classe;
- **protected**: possono essere utilizzate all'interno della classe stessa e all'interno delle classi derivate;
- **private**: possono essere utilizzate soltanto all'interno della classe stessa.
- **default** (ossia senza specificazione): sono accessibili solo da tutte le classi nel medesimo package.

Si ricordi che lo *scope* di una variabile rappresenta l'area di codice nella quale un identificatore resta associato ad un indirizzo di memoria: ogni blocco (cioè ogni gruppo di linee di codice racchiuso tra parentesi graffe) definisce uno *scope* e ogni variabile locale ha come scope l'area di codice che inizia dalla definizione della variabile stessa e termina con il blocco corrente.
In Java, a tal proposito, si distinguono quattro tipi di variabili:

- **variabili locali**: valide solamente all'interno di un metodo, di un ciclo, etc.;
- **variabili di istanza o globali**: variabili dichiarate all'esterno di ogni metodo e quindi visibili da ciascuna funzione;

- **variabili di classe**: variabili **static** di una classe che possono essere impiegate senza istanziare un oggetto della classe stessa; tali variabili sono condivise da tutte le istanze della classe, in quanto per le variabili statiche esiste **un'unica locazione di memoria**;
- **variabili static + final**: sono costanti di classe; infatti tramite il modificatore **final** è possibile specificare che una variabile viene inizializzata una sola volta;

7.3 Overloading

Si osservi che funzioni differenti possono avere lo **stesso identificatore** (*sovrapposizione* o *overloading*): esse devono, però, avere una differente firma (*signature*):

- parametri formali differenti in numero;
- e/o tipo;
- e/o ordine.

Si consideri l'esempio seguente:

```
1  public class Overloading
2  {
3    public static final double PI = 3.141592;
4
5    public static void main(String args[]) {
6      double raggio = Double.parseDouble(args[0]);
7
8      System.out.println("Il volume della sfera e': "+vSfera(raggio));
9      System.out.println("Il volume della sfera e': "+vSfera(args[0]);
10   }
11
12   public static double vSfera(double raggio){
13     return (4/3 * PI * raggio * raggio * raggio);
14   }
15
16   public static double vSfera(int raggio){
17     return (4/3 * PI * raggio * raggio * raggio);
18   }
19
20   public static double volumeSphera(String str){
21     double raggio = Double.parseDouble(str);
22     return (4/3 * PI * raggio * raggio * raggio);
23   }
24
25 }
```

Listing 40: Overloading in Java

7.4 Ricorsione

Una funzione può invocare non solo un'altra funzione, ma anche se stessa: in questo caso si dice che la funzione è **ricorsiva** (ovvero si ha una nuova istanza della medesima funzione mentre l'istanza attuale non è ancora conclusa). Un classico esempio è quello del calcolo del fattoriale di un numero intero positivo n, che può essere espresso ricorsivamente nel seguente modo:

```java
public class Fattoriale
{
  public static void main(String args[]) {
    int n = Integer.parseInt(args[0]);

    System.out.println("Il fattoriale di "+n+" e': "+fact(n));
  }

  public int fact(int n){
    if(n==0) return 1;
    return n*fact(n-1);
  }
}
```

Listing 41: Funzione ricorsiva in Java

4 Aprile 2022

8 Classi e Oggetti

Una classe è un **modello** che **descrive** una certa categoria di oggetti. Essa comprende un certo numero di membri:

- variabili membro o campi dati;
- funzioni membro o metodi;
- costruttori (metodo chiamato per la creazione degli oggetti);

come viene mostrato nell'esempio seguente:

```
class MiaClasse   // Dichiarazione della classe
{
  int n;          // Dichiarazione della variabile

  MiaClasse(){    // Dichiarazione del costruttore
    /* ... */
  }

  int fai(int a){ // Dichiarazione del metodo
    /* .. */
  }
}
```

Listing 42: Esempio di classe Java

Una variabile di un tipo classe (brevemente chiamata **variabile classe**) è un riferimento di oggetto di quella classe.
Un oggetto di un tipo classe (brevemente chiamato **oggetto classe**) è un'istanza della classe (oggetto istanza): viene ottenuto per mezzo dell'operatore **new**:

```
MiaClasse varA;                  // Variabile di tipo classe non istanziata
varA = new MiaClasse();          // Oggetto di tipo classe istanziato
MiaClasse varB = new MiaClasse(); // Oggetto di tipo classe istanziato
```

Listing 43: Esempio variabile e oggetto di classe Java

8.1 Visibilità

L'**identificatore** della classe e quello dei membri della classe sono sempre visibili all'interno della classe; tuttavia, un membro di una classe può avere uno dei modificatori seguenti:

- **modificatore public**: membro visibile alle altre classi;
- **modificatore private**: membro **non** visibile alle altre classi.

8.2 Distruttori

In Java **non esistono distruttori** (come per esempio in C++), per cui non avviene (come per la creazione) una distruzione esplicita degli oggetti ad opera del programmatore; il recupero della memoria dinamica non più utilizzata viene effettuata dalla piattaforma Java per mezzo di una specifica routine (il **Garbage Collector**).

```java
public class UsaConto
{
  public static void main(String args[]) {
    ContoCorrente cc = new ContoCorrente(1000);

    cc.versa(700);

    if(cc.saldo>200)
      cc.preleva(200);
    if(cc.saldo>900)
      cc.preleva(900);

    System.out.println("Il saldo finale e': " + cc.saldo);
  }
}

public class ContoCorrente
{
  public double saldo;

  public ContoCorrente(double saldoIniziale){
    this.saldo=saldoIniziale;
  }

  public void versa(double somma){
    this.saldo*=somma;
  }

  public void preleva(double somma){
    this.saldo-=somma;
  }
}
```

Listing 44: Esempio di multiclasse in Java

5 Aprile 2022

Nella programmazione orientata agli oggetti, lo stato non è più uno stato unico globale (come nel caso della programmazione imperativa): esso, infatti, è composto da **tutti gli stati interni di tutti gli oggetti attivi**.
Naturalmente, la programmazione orientata agli oggetti **semplifica la realizzazione di programmi complessi**, in quanto

1. si identificano le varie **entità** da rappresentare tramite **classi** e **oggetti**;
2. zi specificano gli **attributi** e i **metodi** di ogni classe accessibili dalle altre classi (ossia l'interfaccia pubblica della classe);
3. si implementano le varie classi separatamente, concentrandosi su unaper volta;
4. Le varie classi possono essere implementate da programmatori diversi, indipendentemente dallo scopo del progetto;
5. la manutenzione e gli aggiornamenti dei programmi è più agevole e sistematica.

8.3 Incapsulamento

L'**incapsulamento** consiste nella **protezione dell'implementazione**; infatti, in base a questa logica

- **ogni oggetto è indipendente** e comunica con gli altri attraverso lo **scambio di messaggi**;
- il grande vantaggio è dato dalla **flessibilità** e dalla sistematicità di costruzione di un oggetto e del suo successivo riutilizzo.

Nella programmazione orientata agli oggetti, solitamente, si pone in stretta relazione un'informazione con il comportamento specifico, che agisce (tramite opportuni **metodi**) su tale informazione. L'incapsulamento è proprio legato al concetto di "impacchettare" in un oggetto i dati e le azioni che sono riconducibili ad un singolo componente; in questo modo, quando si vuole evitare che gli attributi di una classe siano modificabili dall'esterno della classe, è opportuno impostare il modificatore di visibilità della classe da public a **private** (ossia privato); in questo modo, non potendo più accedere a tali attributi direttamente, sarà necessario impostare degli opportuni metodi per impostare (**set**) il valore dell'attributo e per ottenerne (**get**) il valore, come mostrato di seguito:

```
1  public class UsaConto
2  {
3    public static void main(String args[]) {
4      ContoCorrente cc = new ContoCorrente(1000);
5
6      cc.versa(700);
7
8      if(cc.getSaldo()>200)
9        cc.preleva(200);
10     if(cc.getSaldo()>900)
11       cc.preleva(900);
12
13     System.out.println("Il saldo finale e': " + cc.getSaldo());
14   }
15 }
16
17 public class ContoCorrente
18 {
19   private double saldo; // Attributo private e non accessibile dall'esterno
20
21   public ContoCorrente(double saldoIniziale){
22     this.saldo=saldoIniziale;
23   }
24
25   public void versa(double somma){
26     this.saldo*=somma;
27   }
```

```
28
29    public void preleva(double somma){
30      this.saldo-=somma;
31    }
32
33    public void setSaldo(double saldo){
34      // Utilizzo della parola riservata this per refernziare l'attributo della
      classe
35      this.saldo=saldo;
36    }
37
38    public double getSaldo(){
39      // Utilizzo della parola riservata this per refernziare l'attributo della
      classe
40      return this.saldo;
41    }
42  }
```

Listing 45: Esempio di incapsulamento in Java

Esempio: Si consideri un altro esempio di incapsulamento:

```
1   public class AvviaMacchina
2   {
3     public static void main(String args[]) {
4       new AvviaMacchina();
5     }
6
7     AvviaMacchina(){
8       Macchina m500 = new Macchina("500");
9       Macchina mPunto = new Macchina("Punto");
10      Macchina mFerrari = new Macchina("Ferrari",340);
11
12      Macchina[] garage = {m500,mPunto,mFerrari};
13
14      float lim=120.0f;
15      for(int vel=0;vel<lim;vel++){
16        for(int m=0;m<garage.length;m++){
17          garage[m].piuVel();
18        }
19      }
20
21      for(int m=0;m<garage.length;m++){
22        System.out.println("La macchina "+garage[m].getName()+" viaggia a "+garage[
      m].velAt()+"km/h");
23      }
24    }
25  }
26
27  public class Macchina
28  {
29    private String nome;
30    private double velMax, velAt;
31
32    public Macchina(String nome){
33      this.nome=nome;
34      this.velMax=200;
35      this.velAt=0;
36    }
37
38    public Macchina(String nome, double velMax){
39      this.nome=nome;
40      this.velMax=velMax;
41      this.velAt=0;
42    }
43
44    public String getNome(){
45      return this.nome;
46    }
47
48    public void setNome(String nome){
49      this.nome=nome;
50    }
```

```
51
52      public void piuVel(){
53        this.velAt++;
54      }
55
56      public void menoVel(){
57        this.velAt--;
58      }
59
60      public double getVelAt(){
61        return this.velAt;
62      }
63    }
```

Listing 46: Esempio esteso di incapsulamento in Java

11 Aprile 2022

8.4 Ereditarietà

L'ereditarietà permette essenzialmente di definire delle classi a partire da altre già definite, per mezzo di una opprtuna **gerarchia**;

8.5 Polimorfismo

Il polimorfismopermette di scrivere un client che può servirsi di oggetti di classi diverse;