ELEMENTI DI BASE DEL LINGUAGGIO

Introduzione

Il Java è un linguaggio di programmazione orientato agli oggetti (OO: Object Oriented), studiato e realizzato dall'azienda statunitense Sun Microsystem (spesso semplicemente Sun) negli anni '90 dello scorso secolo.
Il Java è un linguaggio object oriented (OOL: Object Oriented Language) di tipo "puro", perché è stato sviluppato appositamente per implementare (realizzare) in un progetto software tutte le caratteristiche della programmazione orientata agli oggetti (OOP: Object Oriented Programming). Spesso per "Java" non si intende soltanto il linguaggio di programmazione, ma anche la tecnologia fondata sull'insieme delle tecniche alla cui base vi è il linguaggio Java.
Per comprendere meglio le caratteristiche del Java, analizziamo le motivazioni e il quadro storico-scientifico che ne hanno determinato lo sviluppo.

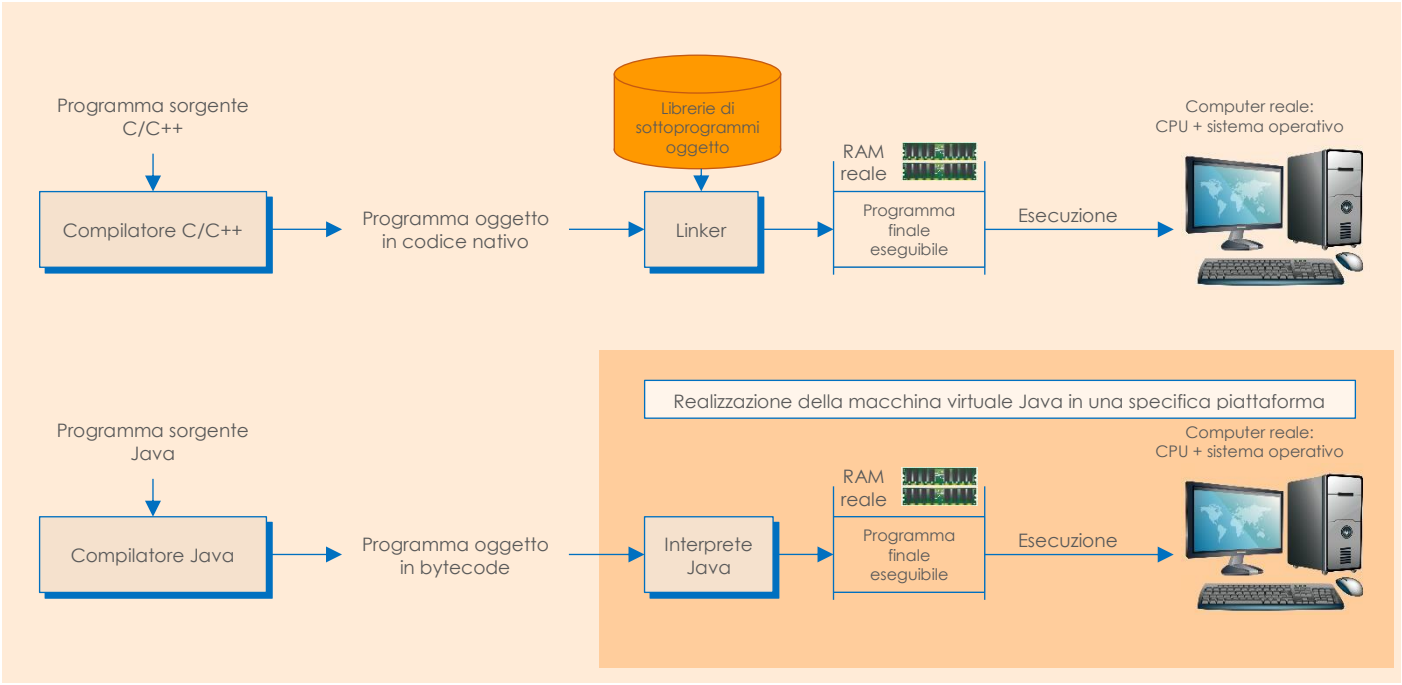
Breve storia del linguaggio

Le origini del Java derivano da quelle del linguaggio di programmazione C++ che discende a sua volta dal C. Il C è un linguaggio di programmazione sviluppato nel 1972 da Dennis Ritchie, ricercatore nei laboratori Bell dell'azienda AT&T negli Stati Uniti, con l'obiettivo di utilizzarlo per completare il codice sorgente del sistema operativo UNIX. Il C segue il paradigma (modello) procedurale con alcune funzioni per il controllo dell'hardware, tipiche dei sistemi operativi. Il C++ nasce ufficialmente nel 1983 da un'idea di Bjarne Stroustrup, ricercatore presso gli stessi laboratori di Ritchie. Stroustrup, dovendo realizzare software molto complessi, completo il linguaggio aggiungendogli i costrutti tipici dell'OOP. Il C++ è quindi un OOL ibrido perché permette ai programmatori di realizzare progetti software seguendo il paradigma procedurale e/o quello orientato agli oggetti. Il C++ è quindi una versione del linguaggio C incrementata dai costrutti dell'OOP, come sottolinea il carattere speciale ++ (due + scritti in sequenza, spesso letti in inglese *plus plus*) che in C (così come in Java) rappresenta un operatore speciale la cui funzione è quella di incrementare di una unità il valore di una variabile.
Il linguaggio Java nasce ufficialmente il 23 maggio del 1995 in una presentazione organizzata da John Gage, direttore della divisione informatica della Sun Microsystem, e Marc Andreessen, vice presidente della Netscape. Java fu presentato come una nuova tecnologia (insieme di tecniche) software che stava per essere incorporata in Netscape Navigator, che in passato rappresentava uno dei più diffusi programmi per la navigazione (browser) nel World Wide Web (o semplicemente Web) della rete mondiale di computer Internet.
Lo sviluppo di Java aveva però avuto inizio nel dicembre del 1991 presso i laboratori della Sun da un progetto di ricerca denominato *the Green project* (letteralmente "il progetto verde"). Il gruppo di ricercatori del *Green project* aveva avuto l'incarico di studiare un nuovo linguaggio, adatto per controllare apparecchiature elettroniche di largo consumo (in particolare per la televisione via cavo negli Stati Uniti) e computer "tascabili", indicati con la sigla PDA (Personal Digital Assistants), in grado di funzionare senza fili (wireless). Il *Green project* scarto l'impiego del C, perché si sarebbe dovuta sviluppare una versione diversa del codice del programma per ogni tipo di hardware e software di base. I ricercatori partirono quindi dalle basi del C++, cercando però di modificarlo per renderlo, da un lato, il più semplice possibile e, dall'altro, adatto alla creazione di progetti, anche complessi, in grado di essere eseguiti su macchine differenti. Il Java eredita comunque dal C e dal C++ la caratteristica di essere un linguaggio nato da programmatori per altri programmatori con l'obiettivo primario di rendere più semplice l'attività di codifica, in particolare di programmi complessi. Il gruppo *Green project* sviluppo quindi la prima versione del linguaggio, denominata Oak (letteralmente: quercia), con la caratteristica di essere indipendente dalla piattaforma o multi-piattaforma (cross platform); un programma poteva quindi essere eseguito su una qualsiasi macchina reale in modo indipendente dall'hardware (famiglia di microprocessori) e dal sistema operativo.
Nel seguito, seguendo la terminologia della Sun, per **piattaforma** intenderemo un elaboratore individuato da un particolare sistema operativo e dall'hardware, il cui funzionamento è determinato essenzialmente dal tipo di linguaggio macchina proprio di una specifica famiglia di microprocessori.
Terminata l'esperienza del Green project, il nome del linguaggio fu modificato nel 1995 in Java, che nello slang USA significa caffè (da cui deriva il logo ufficiale del linguaggio della Sun). Nei primi anni '90, l'attenzione internazionale era però focalizzata sulla diffusione del World Wide Web in Internet, per cui i ricercatori della Sun capirono che il nuovo linguaggio, indipendente dalla piattaforma, sarebbe stato ideale anche per il Web.
Rimane nella storia del Java il 1995, quando James Gosling presentò la prima pagina per il Web interattiva in una conferenza internazionale a Monterey al centro della Silicon-Valley in California. La pagina includeva l'animazione di una molecola, realizzata con un'applicazione Java, che ruotava su se stessa quando Gosling muoveva il puntatore del mouse sullo schermo del computer. Le pagine per il Web, che fino a allora includevano esclusivamente testo e immagini fisse, potevano essere rese dinamiche e interattive inserendo dei programmi Java di piccole dimensioni (limitato numero di byte) e indipendenti dalla piattaforma (hardware del computer e browser), denominati applet.

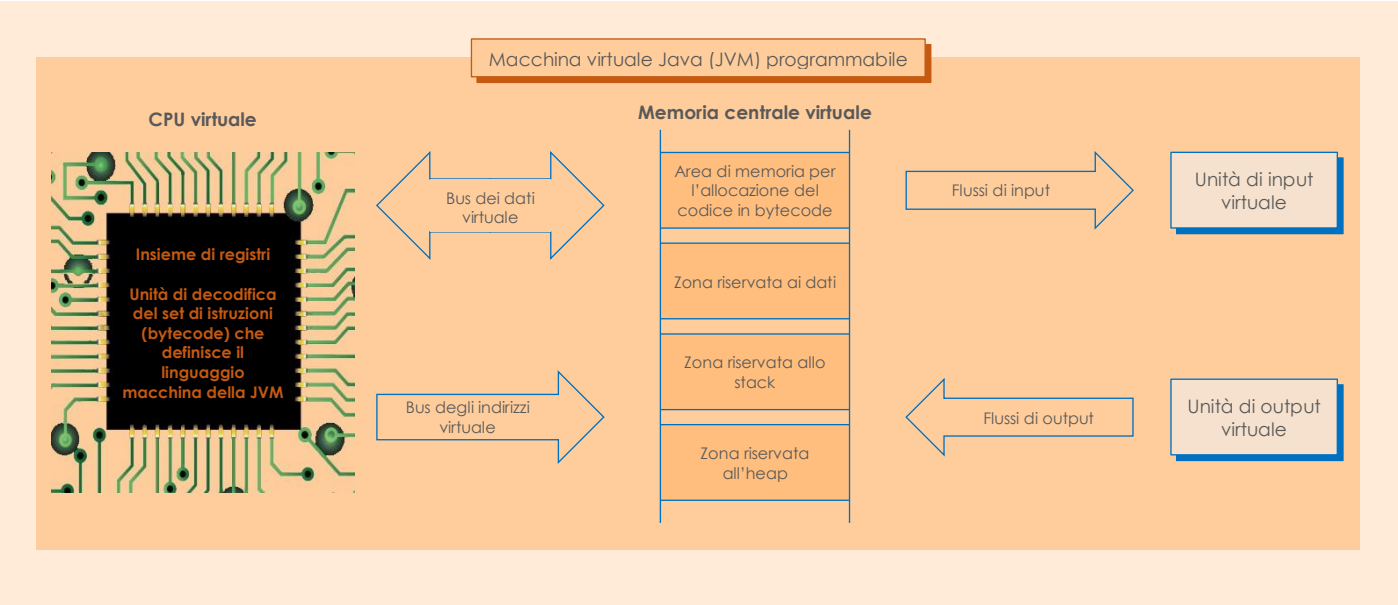
Dopo il successo del lancio del Java nel 1995, la Sun mise a disposizione, in uno dei suoi siti Web in Internet, la possibilità di effettuare gratuitamente il download dell'ambiente per lo sviluppo di programmi e applet Java, negli hard disk locali di chiunque fosse interessato a utilizzare questa nuova tecnologia.

Bytecode e la macchina virtuale Java

Il compilatore Java traduce automaticamente un codice sorgente in un nuovo programma formato da una sequenza di bytecode (letteralmente "codice a byte"). Un bytecode è un'istruzione eseguibile da una macchina virtuale Java (JVM: Java Virtual Machine) programmabile.
Nel caso di molti linguaggi, tra cui il C/C++, il compilatore traduce un programma sorgente in un codice oggetto scritto con il linguaggio macchina specifico di una famiglia di CPU. Possiamo pensare al codice oggetto come a un sottoprogramma che, per formare il programma finale eseguibile, deve essere collegato (*linking*) con altri sottoprogrammi (forniti dal sistema operativo, dal compilatore e/o dallo stesso programmatore), mediante uno strumento software denominato linker, disponibile negli IDE per la programmazione. Il programma eseguibile finale può quindi essere eseguito da una CPU reale, dopo essere stato trasferito (allocato) nella memoria centrale (in genere una RAM). Un programma eseguibile scritto con il linguaggio macchina di una famiglia di CPU, è anche denominato codice nativo (*native code*).
Nel caso del Java, il compilatore traduce un programma sorgente in un codice "oggetto" scritto con istruzioni bytecode "eseguibili" mediante una macchina virtuale programmabile (la JVM), che quindi non esiste nella realtà. Il compito di trasformare il bytecode nel codice nativo e di eseguirlo in un computer reale è invece affidato all'interprete Java specifico di ogni tipo di piattaforma. La figura seguente pone a confronto il processo di compilazione di programmi sorgente C/C++ e Java, con la successiva esecuzione del codice oggetto in un computer reale (caso C/C++) e la realizzazione della macchina virtuale Java in una specifica piattaforma (caso Java).



L'azione di un interprete Java è diversa da quella dei software interpreti di altri linguaggi; quest'ultimi traducono in linguaggio macchina direttamente le istruzioni ad alto livello del programma sorgente, mentre un interprete Java traduce in linguaggio macchina dei comandi che sono già una trasformazione del programma sorgente in una forma più vicina alla macchina, anche se virtuale. Una JVM è un processore programmabile, svincolato da qualsiasi tecnologia hardware e software, che la Sun definisce in modo standard mediante un'architettura ideale i cui elementi sono descritti nella figura seguente.



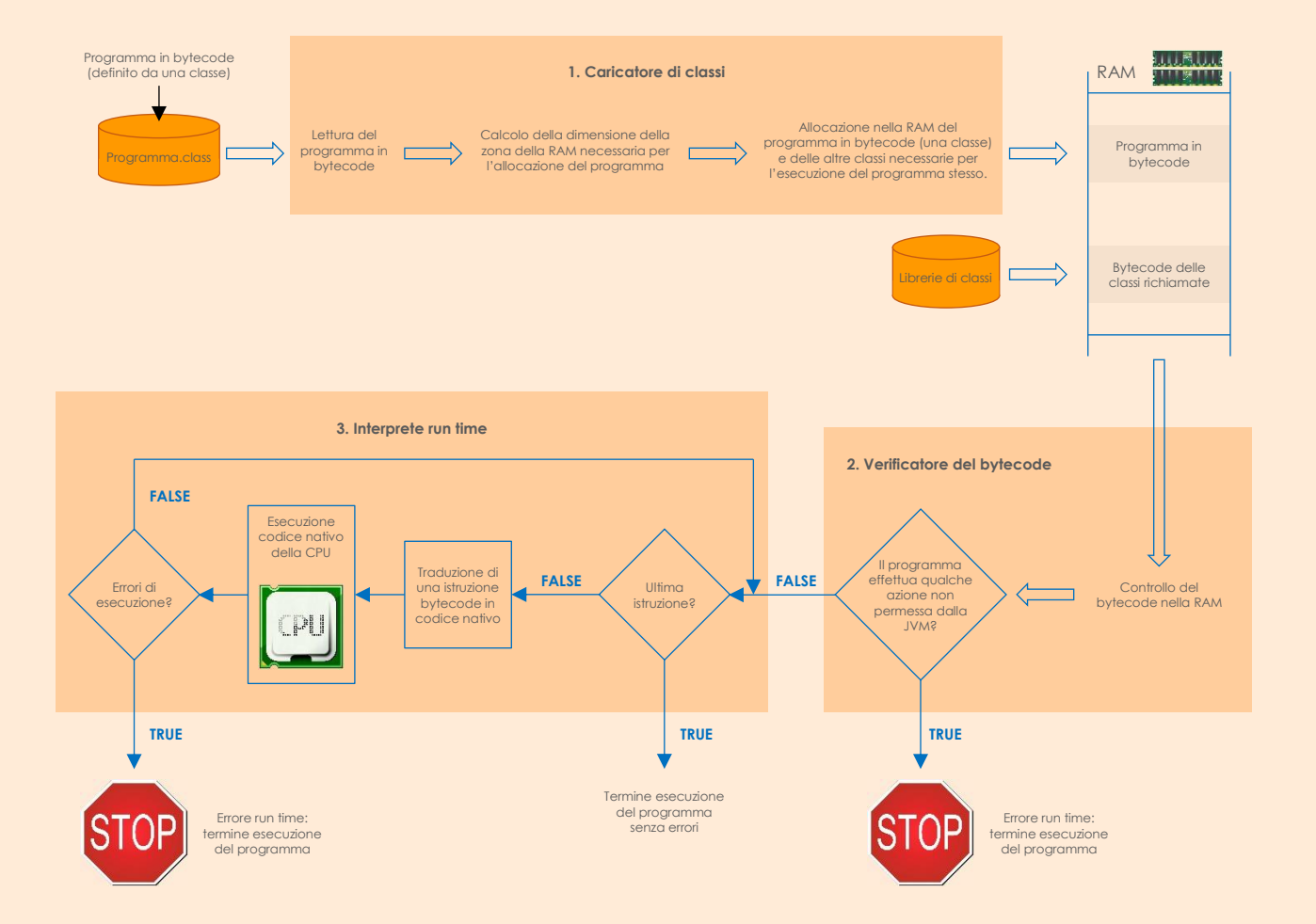
Una macchina virtuale Java è quindi realizzata in ogni specifica piattaforma mediante un differente interprete del linguaggio che esegue bytecode per un computer reale. Un programma in bytecode è indipendente dalla piattaforma, dovendo essere "eseguito" dalla medesima JVM definita dalla Sun per qualsiasi computer. Per essere sempre disponibile, il codice sorgente Java deve essere memorizzato in un file (denominato file sorgente) di estensione *.java*. Il compilatore Java accetta quindi in ingresso un file *.java* e produce un nuovo file di estensione *.class* con comandi in bytecode. Un file *.java* e portabile su qualsiasi computer per cui produrrà sempre, dopo la sua compilazione, il medesimo file *.class* con il bytecode. Al contrario, un compilatore Java è un programma eseguibile, per cui il suo codice nativo sarà differente e specifico di ogni singola piattaforma, anche se qualsiasi traduttore Java implementerà sempre i medesimi algoritmi per tradurre i codici sorgente in quelli in bytecode per la JVM.

Esecuzione del bytecode

L'esecuzione di un programma in bytecode nell'ambiente run time Java avviene, nell'ordine, in tre fasi, schematizzate nella tabella e nella figura che seguono. Ognuna delle tre fasi per l'esecuzione del bytecode è realizzata, in modo automatico e "invisibile" per gli operatori, mediante uno strumento software inserito nello stesso JRE.

Fase di esecuzione di un programma	Descrizione
1. Caricamento del programma in bytecode, eseguito dal **caricatore di classi** (*class loader*)	Il JRE legge il programma in bytecode (memorizzato in un file *.class* su una memoria di massa) per verificare quali altri elementi software (classi) sono necessari per l'esecuzione del programma e quindi determinare lo spazio totale necessario nella memoria centrale (RAM) del computer. Il JRE richiama quindi il class loader che trasferisce il codice del programma da eseguire (a sua volta una classe) e quello delle classi necessarie (sempre in bytecode) in una zona della RAM. In questa fase, il caricatore di classi riserva una zona della memoria centrale per l'esecuzione del programma, impedendo che altre applicazioni possano modificare il bytecode. Questa azione garantisce la sicurezza del codice, impedendo l'attacco di programmi non autorizzati, quali a esempio i virus.
2. Controllo del codice, effettuato dal **verificatore del bytecode** (*bytecode verifier*)	Il verificatore del bytecode analizza il codice nella RAM, pronto per essere eseguito, per controllare che il programma non effettui alcuna delle "azioni" proibite definite nelle specifiche della macchina virtuale Java. Questa fase è diversa nelle applicazioni rispetto alle applet, dove le regole di sicurezza sono molto più restrittive. Nel caso di violazione di una delle regole di sicurezza della JVM, l'esecuzione del programma si blocca con un errore run time.

3. Esecuzione del bytecode mediante l'**interprete run time** (*run time interpreter*)	L'interprete run time legge, in sequenza, un'istruzione bytecode dopo l'altra nella RAM, la traduce in linguaggio macchina e la esegue nella piattaforma reale. Gli errori run time in questa fase dovrebbero essere limitati a quelli commessi dagli utenti, perché gli altri errori sono già stati eliminati dal verificatore del bytecode

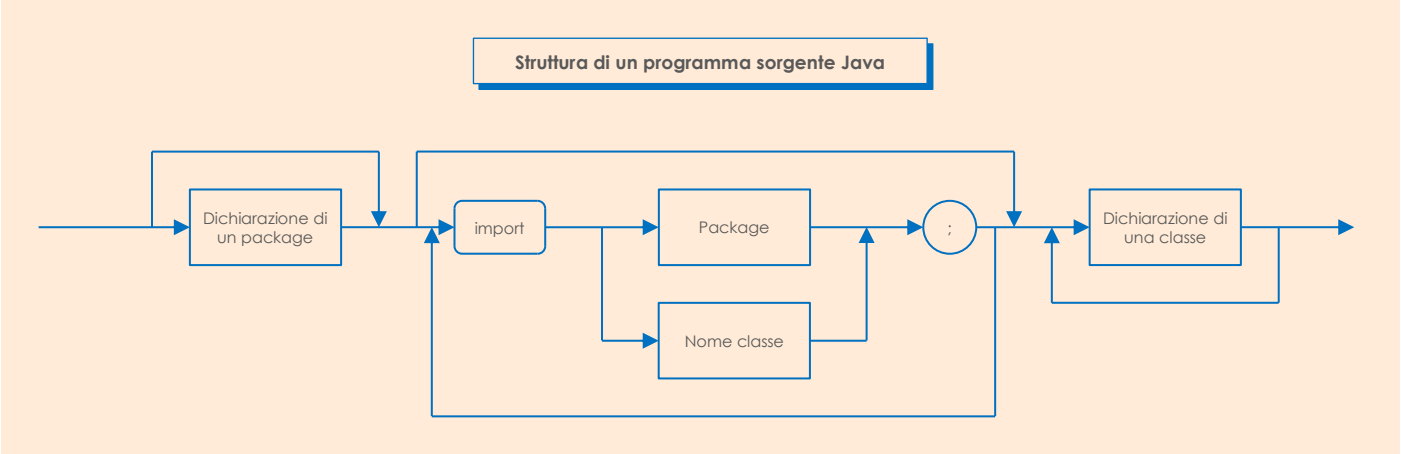


L'interpretazione di un programma in bytecode è più lenta dell'esecuzione di un programma compilato in codice nativo. I programmatori della Sun hanno però ridotto i tempi per l'interpretazione del bytecode con la distribuzione delle nuove versioni dell'JDK/JRE agendo sui seguenti fattori.

- Miglioramento di alcuni costrutti del linguaggio con l'obiettivo di rendere più semplice il bytecode generato e quindi facile da tradurre nel codice macchina degli interpreti run time.
- Gestione automatica della memoria da parte della JVM.
- Ottimizzazione del codice nativo con cui sono scritte le diverse versioni del JRE.

Struttura di un programma sorgente Java

La struttura di un programma sorgente Java è descritta nel seguente diagramma sintattico.



Il grafo precedente suggerisce come un programma Java sia formato da un insieme di una oppure più classi. Al nostro livello di studio, possiamo pensare a una classe come a un "contenitore" software che definisce gli attributi (proprietà o caratteristiche) e l'elaborazione (o comportamento) del programma che vogliamo realizzare. Gli attributi sono definiti mediante variabili, mentre l'elaborazione è realizzata dichiarando dei sottoprogrammi, denominati metodi, come rappresentato nel seguente schema (nel gergo dei programmatori "scheletro") di codice sorgente che descriva l'organizzazione di una generica classe Java.

```
public class NomeClasse [extends NomeSuperclasse]optional [implements NomeInterfaccia]optional {
        [<dichiarazione attributo;>repeat]optional
        [<dichiarazione metodo;>repeat]optional
}
```

La parola chiave *public* indica che la classe è pubblica, per cui può essere utilizzata da qualsiasi classe esterna. La parola chiave *extends* permette di estendere gli attributi e il comportamento della classe in fase di progettazione, aggiungendo le variabili e l'elaborazione di un'altra classe, denominata superclasse. Questo meccanismo, tipico dell'OOP, che studieremo in seguito, è denominato ereditarietà. La parola chiave *implements* dichiara invece che una classe deve realizzare il comportamento di un'altra classe speciale, denominata interfaccia.
Il linguaggio Java distingue le lettere maiuscole da quelle minuscole per cui la parola chiave *class* e differente da *CLASS* oppure *Class*, che quindi non saranno riconosciute dal compilatore. Nel programma sorgente le istruzioni sono separate da un punto e virgola e ogni comando può essere scritto con un formato libero in più linee. Per migliorare l'ordine e la chiarezza del codice sorgente, è però preferibile scrivere il programma in modo ordinato con una istruzione per ogni linea.
Per utilizzare una classe in un programma, si deve definire una copia (o istanza) dei suoi attributi, che nella programmazione orientata agli oggetti viene denominata oggetto. Un'istanza, o oggetto di una classe, permette anche di richiamare i metodi introdotti dal programmatore durante la dichiarazione della classe.
Il comando *import*, con cui può iniziare un programma Java, segnala al compilatore la classe o l'insieme delle classi da includere. Il codice in bytecode delle classi sarà poi effettivamente incluso nel programma dal JRE al momento dell'esecuzione del programma. Una delle caratteristiche principali del Java (e dell'OOP in generale) è la riusabilità dei programmi realizzati. Il riuso del software avviene mediante le classi che, una volta definite dai programmatori, possono essere riutilizzate in altri programmi semplicemente richiamando le loro funzionalità mediante il comando *import*. La stessa tecnologia Java è definita mediante un insieme di classi di base, che formano il *Java Foundation Classes* (JFC). Le classi "fondamenta" del Java sono memorizzate nell'JDK, nella sottocartelle contenute nei percorsi *C:\Programmi\java\jre-x.x.x\lib* e *C:\Programmi\java\jdk-x\lib*.
Nella tecnologia Java le classi sono organizzate in gruppi di classi (in bytecode), denominati package, ognuno dei quali è dedicato alla soluzione di una specifica categoria di problemi. Un gruppo di classi di un package può essere considerato dai programmatori come una libreria di classi. La tabella seguente descrive le funzioni dei package principali della JFC, che impareremo a studiare durante lo studio del linguaggio.

Package della JFC	Contiene l'insieme delle classi che definiscono …
java.lang	… le funzionalità elementari del linguaggio quali, a esempio, la definizione dei tipi di dato per le variabili (numeri interi, reali, caratteri ecc.) e delle operazioni matematiche più frequenti.
java.util	… alcuni comandi utili per i programmatori per la definizione delle date e delle ore, la gestione degli array o la creazione di modelli dei dati complessi.
java.io	… le operazioni di input/output principali.
java.applet	… il comportamento delle applet di Java 1.
java.awt, javax.swing,	… i comandi principali per costruire l'interfaccia utente grafica di applicazioni e delle applet Java 2
java.net	… alcuni comandi per stabilire il collegamento con altri calcolatori inseriti in una rete di computer.

Per includere una specifica classe contenuta in un package si usa il comando *import* con la forma
```
import nome_package.nomeClasse;
```
mentre per includere tutte le classi di una libreria si deve modificare la sintassi precedente nel modo seguente:
```
import nome_package.*;
```
Tra i package fa eccezione quello *java.lang*, che è automaticamente inserito dal compilatore Java. L'istruzione:
```
import java.lang.*;
```
inserita prima della dichiarazione di una nuova classe è quindi ridondante, perché è aggiunta automaticamente dal compilatore prima di iniziare la traduzione del codice sorgente.
Un comando del tipo *import nome_package.** rallenta l'attività del compilatore Java che, prima di iniziare la traduzione, deve caricare nella memoria centrale (RAM) tutte le classi di un package. In ogni caso, il compilatore utilizzerà soltanto le classi che sono richiamate nel codice sorgente del programma. L'alternativa all'inclusione di interi package è quella di inserire un lungo elenco di comandi *import* seguiti dai nomi delle classi specifiche.
I programmatori Java preferiscono l'istruzione *import nome_package.** per aumentare la chiarezza del codice, accettando un intervallo di tempo aggiuntivo per la compilazione, che nei computer con buone prestazioni (CPU e RAM) è spesso trascurabile.

Applicazioni stand-alone console

Un'applicazione stand-alone console, o semplicemente applicazione console, è eseguita con un'interfaccia di tipo testuale in cui le operazioni di input avvengono mediante la tastiera mentre l'output è visualizzato in forma di testo sullo schermo dell'unità video.
Nei sistemi operativi basati su un'interfaccia utente grafica, le operazioni sulla console (tastiera più unità video) sono effettuate in una finestra in cui compare soltanto testo.
La struttura di un'applicazione console è descritta mediante il seguente scheletro generale di codice sorgente.

```
public class NomeClasse [extends NomeSuperclasse]optional [implements NomeInterfaccia]optional {
        [<dichiarazione attributo;>repeat]optional
        [<dichiarazione metodo;>repeat]optional
        public static void main(String args[]) {
                <istruzione;>repeat
        }
}
```

In una applicazione console, il metodo speciale *main()*, che deve sempre essere presente, dichiara il punto in cui inizia l'esecuzione del programma. All'interno di *main()*, il programmatore deve quindi inserire la sequenza di istruzioni che realizza un algoritmo progettato, richiamando altri metodi della stessa classe e/o elementi (oggetti) di altre classi. Il programma termina dopo l'esecuzione dell'ultima istruzione scritta prima della parentesi graffa chiusa di *main()*. La sequenza di istruzioni inserita tra le parentesi graffe, aperta e chiusa, di un metodo, definisce il suo corpo.
La parola chiave *static* dichiara *main()* come un metodo statico (o di classe), ovvero una elaborazione che può essere richiamata anche senza creare nuovi oggetti della classe in cui il metodo stesso è definito. Come approfondiremo in seguito, i metodi statici sono richiamati scrivendo il loro nome, separato da un punto, dopo quello della classe a cui appartengono, con una sintassi generale del tipo seguente:

NomeClasse.nomeMetodoStatico()

Istruzioni di input/output di base per la console

Nel seguito, per realizzare le prime applicazioni console nel linguaggio Java, impiegheremo le seguenti forme semplificate per i metodi per l'input e per l'output, che (al nostro livello di studio) possiamo considerare come dei comandi di input/output.

Istruzioni di output

```
System.out.print(costanteStringa |or variabile |or oggetto);
System.out.println(costanteStringa |or variabile |or oggetto);
```

I metodi precedenti visualizzano (oppure stampano) sullo schermo dell'unità video una *costanteStringa*, il contenuto di una variabile o le proprietà di un oggetto (convertite in una stringa). Una stringa è un testo formato da una lista di caratteri, che nel caso delle costanti devono essere racchiusi tra una coppia di simboli '" (virgolette). Il metodo *println()*, oltre a stampare sul video, introduce anche un ritorno a capo (carattere speciale di nuova linea). Nelle istruzioni di output è possibile impiegare il simbolo + che permette di aggiungere altri valori alla fine del testo che lo precede, con una forma modificata (ad esempio con *println()*) del tipo seguente:

```
System.out.println(costanteStringa1 + variabile + costanteStringa2 + oggetto);
```

Istruzioni di input

```
InputStreamReader flusso = new InputStreamReader(System.in);
BufferedReader tastiera = new BufferedReader(flusso);
variabileStringa = tastiera.readLine();
```

L'operazione di input avviene definendo un flusso di caratteri (basato su una memoria tampone o buffer) creato mediante un oggetto (di nome *tastiera*) della classe *BufferedReader*. L'oggetto *tastiera* proviene a sua volta dalla trasformazione di un flusso senza buffer (di nome *flusso*) della classe *InputStreamReader*. Il metodo *readLine()* della classe *BufferedReader*, applicato a un oggetto *tastiera* con il comando *tastiera.readLine()*, raccoglie tutti i dati digitati dall'utente in una linea della console e li assegna a una variabile di tipo stringa (*variabileStringa*). Le classi per le operazioni di input appartengono al package *java.io*. Per utilizzare le istruzioni di input precedenti si deve inserire, dopo il nome del metodo in cui sono inserite, il comando *throws IOException*, necessario per la gestione degli errori in fase di esecuzione (run time).

Studieremo in seguito come l'input dei dati in un'applicazione console può anche essere realizzato "raccogliendo" i dati digitati nella riga di comando che avvia il programma mediante l'interprete.

La prima applicazione console

Utilizzando i concetti precedenti possiamo scrivere la nostra prima applicazione console significativa in Java mediante la classe *PrimaConsole*.

```
import java.lang.*;

public class PrimaApplicazioneConsole {
	public static void main(String args[]) {
		System.out.print("\n\n\n\n\n\t Entriamo nel mondo delle applicazioni console del Java!
\n\n\n\n\n");
	}
}
```

Il programma precedente, eseguito nell'ambiente run time Java, produce il seguente output sull'unità video della console.

"*Entriamo nel mondo delle applicazioni console del Java!*"

Nel codice abbiamo incluso anche la riga facoltativa *import java.lang.*;*, per introdurre l'impiego di *import* in un programma.

Applicazioni stand-alone con interfaccia utente grafica (GUI: Graphical User Interface)

Un'applicazione stand-alone con interfaccia utente grafica, o semplicemente applicazione con GUI, è eseguita in un ambiente in cui l'interazione con l'operatore è basata sull'impiego di una o più finestre e di eventi associati al mouse e/o alla tastiera.

La struttura del codice sorgente di un'applicazione stand-alone con GUI è identica a quella di un programma console, con la sostanziale differenza che nel metodo *main()* si devono inserire le istruzioni per la creazione e la gestione dell'interfaccia grafica per l'utente.

Creazione di una GUI

La GUI di una applicazione è basata su una finestra, che può essere creata mediante un oggetto appartenente alla classe JFrame. Una finestra è aperta, per default, con lo stile del sistema operativo nativo in cui è eseguita l'applicazione. Per definire un nuovo oggetto della classe JFrame si impiega la seguente sintassi generale:

JFrame *nomeOggetto* = **new** JFrame();

Dopo aver definito un nuovo frame è necessario impostarne l'aspetto richiamando i metodi della classe con una sintassi semplificata del tipo seguente:

nomeOggetto.nomeMetodo();

La tabella successiva presenta alcuni dei metodi principali che possiamo richiamare per personalizzare l'aspetto della finestra di una applicazione.

Come fare per ...	Si impiega il metodo ...
... modificare il titolo di una finestra.	JFrame.setTitle(*stringaTitolo*);
... definire la dimensione di una finestra impostata come *larghezzaxaltezza* in pixel. Per default, una finestra appare nell'angolo in alto a sinistra dello schermo.	JFrame.setSize(*larghezza*, *altezza*);
... definire, oltre alla *larghezza* e l'*altezza*, la posizione *x* e *y* della finestra sullo schermo.	JFrame.setBounds(*x*, *y*, *larghezza*, *altezza*);
... terminare l'applicazione, quando la finestra viene chiusa con le tecniche predefinite dal sistema operativo.	JFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
... visualizzare oppure nascondere una finestra.	JFrame.setVisible(true); JFrame.setVisible(false);

Istruzioni di input/output di base per la GUI

Uno dei modi per effettuare operazioni di input/output, dopo aver visualizzato la finestra dell'applicazione, è quello di aprire una o più finestre di dialogo figlie del frame principale. La tabella che segue presenta due metodi statici della classe JOptionPane che possono essere richiamati per gestire finestre di dialogo per l'I/O.

Come fare per visualizzare ...	Si impiega il metodo statico ...
... una finestra di dialogo di messaggio per mostrare un'informazione e/o il risultato di una elaborazione a un utente.	JOptionPane.showMessageDialog(**null**, *stringaMessaggio*);
... un dialogo di input che presenta un messaggio e memorizza la risposta digitata dall'operatore in una variabile stringa.	JOptionPane.showInputDialog(**null**, *messaggio*);

Nella tabella precedente, *null* rappresenta una costante del Java impiegata per inizializzare un oggetto di una classe. Ognuno dei dialoghi precedenti è modale, ovvero non consente all'operatore di passare all'applicazione che lo ha generato, finché il dialogo rimane aperto.

La prima applicazione con GUI

Utilizzando i concetti precedenti possiamo scrivere la nostra prima applicazione con GUI in Java mediante una classe di nome *PrimaAppGUI*.

```
import javax.swing.*;

public class PrimaApplicazioneGUI {
	public static void main(String args[]) {
		JFrame finestra = new JFrame();
		finestra.setTitle("Prima applicazione con GUI (Graphical User Interface)");
		finestra.setBounds(0, 0, 1900, 1000);
		finestra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		finestra.setVisible(true);
		JOptionPane.showMessageDialog(null, "Entriamo nel mondo delle applicazioni con GUI del
Java!");
	}
}
```

Il programma precedente, mostra un frame di dimensioni 1900 x 1000. Dopo l'apertura della finestra dell'applicazione, appare nella GUI un dialogo modale con il seguente messaggio impostato nel programma.

"Entriamo nel mondo delle applicazioni con GUI del Java!"

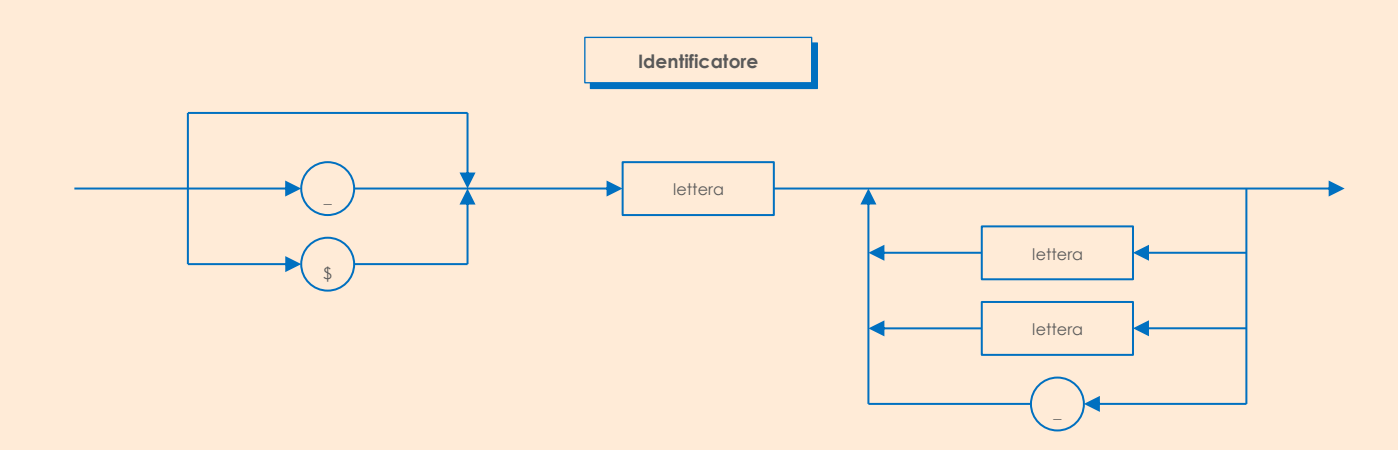
La finestra dell'applicazione si comporta come qualsiasi altra finestra del sistema operativo, per cui può essere ridimensionata e chiusa. La chiusura del frame provoca, grazie all'impostazione *JFrame.EXIT_ON_CLOSE*, il termine della stessa applicazione.

L'alfabeto e le regole lessicali del linguaggio

L'alfabeto del Java è formato da tutti i caratteri rappresentati nel codice alfanumerico Unicode. La tecnica Unicode associa a ogni simbolo una sequenza di 16 bit (o 4 cifre esadecimali), estendendo l'insieme dei dati rappresentati a 65536 (2^{16}). Il vantaggio dell'Unicode, rispetto all'ASCII, è quello di poter codificare i caratteri di tutte le lingue del mondo, tra cui il greco, l'arabo e i linguaggi orientali basati sugli ideogrammi. I programmatori greci e giapponesi potranno quindi scrivere codici sorgente Java usando i propri alfabeti formati, rispettivamente, dai simboli α, β, γ ... e da ideogrammi.
I primi 256 caratteri del codice Unicode (da 0 a 255) coincidono con quelli del codice ASCII esteso a 8 bit, standardizzato come ISO-Latin-1. Se quindi utilizziamo caratteri della lingua italiana e di quella inglese, i simboli principali dell'alfabeto Java sono: le lettere maiuscole e minuscole dell'alfabeto inglese (*a*...*z*, *A*...*Z*), le cifre decimali (*0*...*9*) e i caratteri speciali (le lettere accentate, lo spazio, i simboli di punteggiatura, le parentesi, i segni matematici ecc.).
Le lettere minuscole e maiuscole sono simboli diversi dell'alfabeto del Java, per cui le lettere *c* e *C* sono due caratteri differenti. Anche le lettere accentate quali *à*, *è*, *é*, *ì*, *ò* e *ù*, sono caratteri validi del linguaggio.
Partendo dall'alfabeto, le regole lessicali permettono al programmatore di definire le parole valide all'interno di un programma sorgente. L'insieme di tutte le parole che possiamo costruire a partire dall'alfabeto, applicando le regole lessicali, formerà il dizionario (o vocabolario) del linguaggio. Le parole del Java (token o elementi lessicali) possono essere identificatori (creati dal programmatore), parole chiave, letterali, operatori e simboli separatori.

Identificatori

Un identificatore (*identifier*) è un nome creato dal programmatore che individua in modo univoco un elemento di un progetto software quale, ad esempio, una variabile, un'etichetta, un attributo, un metodo o una classe.
Per costruire una parola che sia un identificatore Java valido, si deve seguire il grafo sintattico seguente.



Un nome valido deve sempre iniziare con una lettera, una sottolineatura _ (*underscore*) o un simbolo del dollaro $ e può contenere un numero di simboli illimitato, anche se alcuni IDE di programmazione possono imporre dei vincoli sulla lunghezza massima degli identificatori. I nomi che iniziano con i simboli _ o $ sono usati raramente, e soltanto in alcune piattaforme, per rappresentare variabili riconosciute dai sistemi operativi.
Il Java è *case sensitive* (distingue le lettere maiuscole da quelle minuscole), per cui gli identificatori: *areaPoligono*, *AREAPOLIGONO* e *ArEaPoLiGoNo*, rappresentano tre nomi distinti validi. Il linguaggio accetta i caratteri Unicode per costruire i propri nomi, per cui l'identificator *velocitàIniziale*, che contiene la lettera accentata *à*, è un identificatore valido.
I programmatori Java seguono però alcune regole informali, tra cui ricordiamo le seguenti.

- ✓ I nomi delle classi iniziano con una lettera maiuscola.
- ✓ Gli identificatori di variabili, attributi e metodi, hanno la prima lettera minuscola.

- ✓ Se un nome è formato da un insieme di parole con lettere minuscole, i singoli termini iniziano con una lettera maiuscola oppure sono separati da una sottolineatura.
- ✓ Gli identificatori devono essere mnemonici (ricordare la funzione per cui sono usati), per aumentare la documentazione del codice sorgente.

In un programma per il calcolo dell'orbita di un satellite, gli identificatori *velocitàSatellite* oppure *velocità_satellite* sono più adatti rispetto al nome *prova*, per rappresentare la velocità del satellite intorno alla Terra. Il nome della classe del programma potrà essere *Satellite* e non *satellite*.

Parole chiave

In un linguaggio artificiale, una parola chiave (*keyword*) è un identificatore riservato che indica al compilatore un'azione, una dichiarazione o un'istruzione eseguibile, che il programmatore vuole realizzare nel programma sorgente. La tabella seguente presenta tutte le *keyword* attualmente utilizzate dal Java 2.

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

Le keyword Java usano solo caratteri minuscoli e non possono essere utilizzate per rappresentare nomi validi creati dal programmatore. Per rappresentare un elemento predefinito in un programma possiamo usare l'identificatore *defaulValue*, ma non *default* che è una keyword del linguaggio.

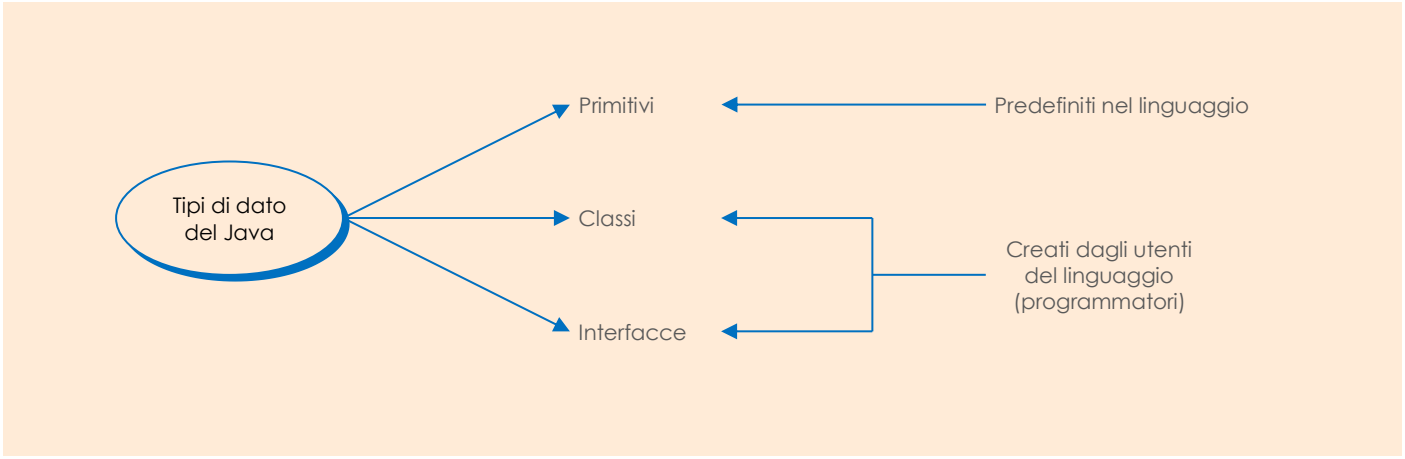
Letterali, operatori, simboli separatori e commenti

Un letterale (*literal*) è un dato costante (codificato) introdotto da un programmatore in un codice sorgente che rimane inalterato durante la traduzione nel programma eseguibile in linguaggio macchina.
Esempi di letterali sono il numero intero -542, il numero reale 3,14159 e le costanti booleane *true* e *false*. L'unica trasformazione subìta da un letterale, nel passaggio dal programma sorgente a quello eseguibile, è la sua conversione da un formato comprensibile all'uomo in quello binario. Esamineremo i letterali, durante lo studio dei tipi di dato primitivi del Java.
Il Java dispone di numerosi operatori impiegati, ad esempio, per definire operazioni quali quelle algebriche (ad esempio + – * /) oppure di confronto (ad esempio < e >). Gli operatori Java possono essere rappresentati da un unico simbolo, da due caratteri (a esempio == e !=) oppure da tre caratteri (ad esempio >>>), che il compilatore riconosce come un'unica parola.
I simboli separatori sono impiegati nel codice sorgente per distinguere tra loro le istruzioni e/o evidenziare parti diverse del costrutto di un linguaggio. I simboli separatori sono: il punto e virgola (;), la virgola (,), il punto(.), le parentesi (graffe {}, tonde () e quadre []) e i caratteri di spaziatura (a esempio uno spazio).
Un commento è un'annotazione inserita dal programmatore in un codice sorgente che viene ignorata dal compilatore durante la fase di traduzione nel codice in bytecode. Un commento multilinea inizia con i simboli /*, prosegue con le frasi del commento vero e proprio e termina con */. Un commento a linea singola inizia con la coppia di simboli // e deve terminare alla fine della stessa linea. I commenti di documentazione sono stati introdotti per permettere la generazione automatica di informazioni su un programma in formato HTML. Le informazioni sono suddivise in un insieme di pagine Web con la stessa organizzazione che i programmatori della Sun usano per documentare le classi JFC nell'SDK. Tale commento inizia con i simboli /**, prosegue con alcune frasi e termina con */. La generazione automatica delle pagine Web avviene mediante il generatore di documentazione javadoc.exe

Le strutture di dati del Java

Ogni linguaggio di programmazione dispone di un insieme di strutture di dati che il programmatore impiega per tradurre (in modo manuale) in un programma sorgente il modello dei dati astratto del suo problema. Una struttura di dati si presenta sotto forma di una oppure più frasi (costrutto) che seguono le regole grammaticali del linguaggio.
Le strutture di dati sono modelli per i dati concreti che il compilatore e l'interprete Java traducono nelle strutture interne del programma eseguibile poste nella memoria dell'elaboratore. Nel seguito, per memoria intenderemo quella della JVM (idealmente illimitata) oppure quella centrale (in genere la RAM limitata),

riservandoci di trattare le memorie di massa (ad esempio l'hard disk) quando studieremo le istruzioni per la gestione dell'I/O su file.
L'insieme delle strutture di dati definisce il modello dei dati di un linguaggio, che in Java è basato sui concetti di: variabile, oggetto e tipo di dato. A loro volta, i tipi di dato sono suddivisi in tre categorie con caratteristiche differenti, come schematizzato nella figura successiva.



Nel seguito, considereremo come tipi di dato creati dai programmatori soltanto le classi, essendo un'interfaccia una classe con caratteristiche particolari.

Variabili

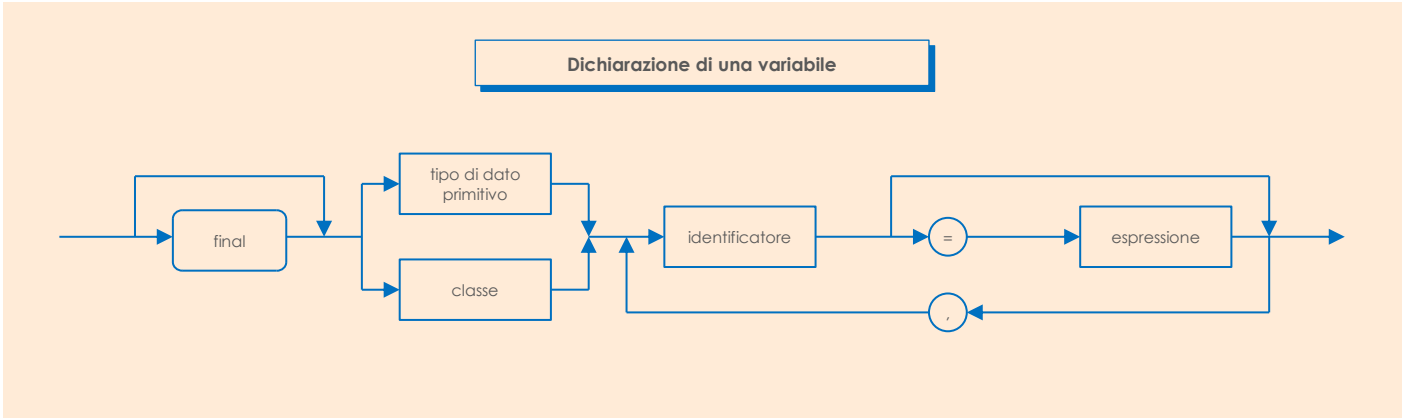
Una variabile (*variable*) è un contenitore per dati elaborati da un programma, al quale il programmatore può associare un identificatore del linguaggio. Dobbiamo però distinguere gli aspetti logici, a livello del codice sorgente, da quelli fisici del computer. A livello macchina, le variabili individuano una serie di celle (o locazioni) della memoria in cui saranno allocati i dati da elaborare.
A livello del codice sorgente, il programmatore può:

- ✓ associare alle locazioni di memoria un nome simbolico (che soddisfa le regole lessicali del linguaggio);
- ✓ modificare i dati contenuti nella zona di memoria.

A differenza di una variabile, una costante (*constant*) individua una zona della memoria il cui contenuto non può essere modificato durante l'esecuzione del programma.

Dichiarazione di una variabile

Una dichiarazione è un'istruzione non eseguibile che definisce per la JVM alcune impostazioni che non comportano la generazione di bytecode.
La dichiarazione di una variabile è basata sul seguente diagramma sintattico.



Seguendo il diagramma sintattico precedente, gli unici elementi obbligatori per introdurre una nuova variabile sono il suo tipo di dato (tipo di dato primitivo o classe) e il suo identificatore (o nome).
La dichiarazione di una variabile specifica la linea del programma sorgente in cui il compilatore (per la JVM) e l'interprete (in esecuzione) dovranno:

1. riservare nella memoria lo spazio necessario (operazione di allocazione);
2. associare alla zona di memoria creata un identificatore ed eventualmente anche un valore iniziale.

Una dichiarazione specifica anche l'ambito (scope) o regione (spazio) di visibilità, ovvero la parte del programma in cui una variabile può essere utilizzata nelle istruzioni. Finché non affronteremo come costruire nuove classi personalizzate dal programmatore, dichiareremo variabili soltanto all'interno dei metodi. Questo tipo di variabili la cui regione di visibilità (o semplicemente visibilità) è limitata esclusivamente al corpo di un metodo sono denominate variabili locali. All'interno del corpo di un metodo, i nomi delle variabili non possono essere duplicati.
In Java è possibile inizializzare una variabile nel punto della sua dichiarazione assegnandogli un valore iniziale che può anche essere il risultato fornito da una espressione.
A esempio, la seguente applicazione console:

```
public class AreaRettangolo {
	public static void main(String args[]) {
		double base = 7.8, altezza = 6.2;
		double area = base*altezza;
		System.out.print("\n\n\n\n\n\t L'area del rettangolo di base " + base + " e altezza " +
altezza + " vale " + area + "\n\n\n\n\n");
	}
}
```

permette di calcolare e stampare a video l'area di un rettangolo. Il calcolo dell'area è eseguito in modo dinamico durante la dichiarazione della variabile reale area:

```
double area = base*altezza;
```

che è inizializzata mediante l'espressione *base*altezza*, in cui il simbolo * rappresenta l'operazione di moltiplicazione.
In alternativa, dopo aver dichiarato un nuovo identificatore in un programma sorgente, la variabile può essere inizializzata impiegando una istruzione di assegnazione, che in Java ha il seguente formato generale:

nomeVariabile = espressione;

Il simbolo = rappresenta l'operatore di assegnazione che ordina al compilatore di:

1. calcolare il risultato dell'espressione scritta a destra dell'uguale;
2. assegnare il risultato alla variabile posta a sinistra di =.

Dichiarazione di una costante

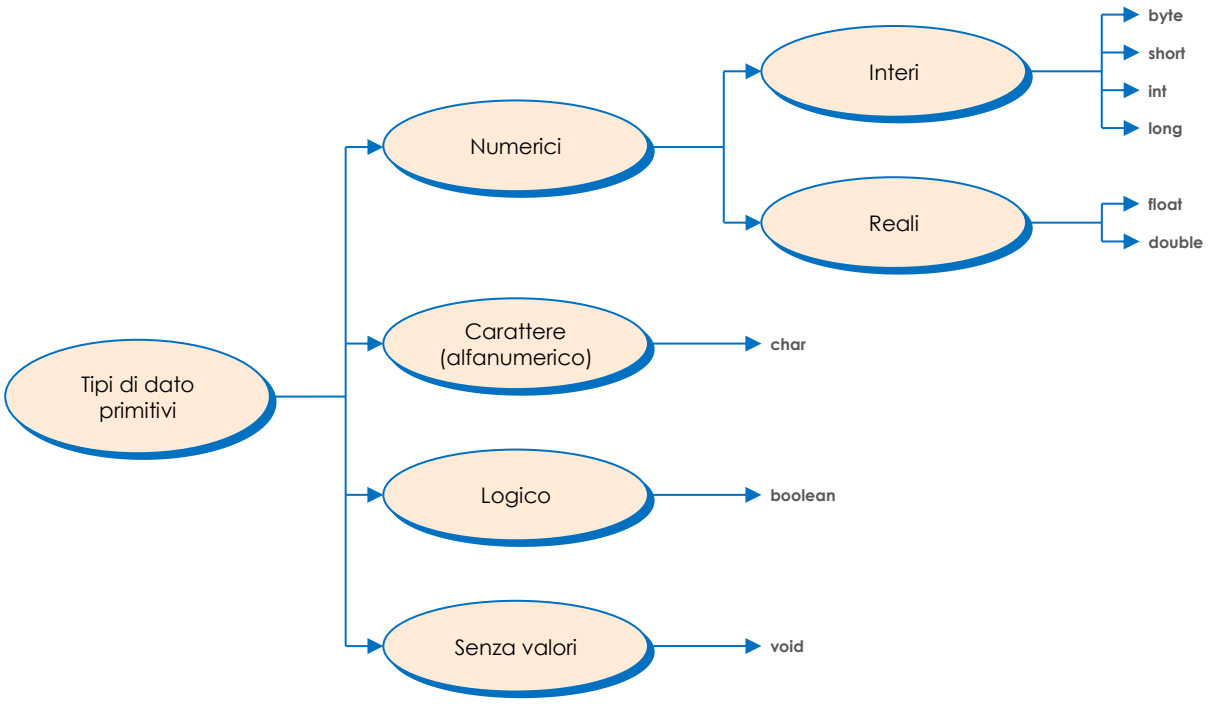
In Java, per introdurre una costante si impiega la keyword `final` scritta nella dichiarazione di una variabile inizializzata con il risultato di un'espressione.
`final` modifica il tipo di una variabile imponendo che il suo valore non possa essere variato nel programma. Secondo una regola informale dei programmatori Java, i nomi delle costanti sono scritti con lettere maiuscole per distinguerli da quelli degli altri elementi di un programma.

Tipi di dato primitivi

In Java, un tipo di dato primitivo (primitive data tipe) definisce:

1. l'insieme dei valori (insieme base) che una variabile può assumere;
2. le operazioni che è possibile applicare alle variabili stesse.

I tipi di dato primitivi sono implementati direttamente nel linguaggio e sono quindi disponibili per essere utilizzati dai programmatori nei loro codici sorgente. Il Java dispone di 9 tipi di dato (o semplicemente tipi) primitivi, classificati in cinque categorie in base ai valori dell'insieme base, come schematizzato nella figura che segue.



Una variabile di uno dei tipi primitivi precedenti può assumere un insieme di valori e/o operazioni limitato. A livello della codifica, le strutture di dati precedenti sono quindi implementazioni approssimative dei tipi di dato astratti (*ADT: Abstract Data Type*) usati durante la progettazione del software. A esempio, come vedremo tra breve, i tipi reali del Java (*float* e *double*) contengono numeri rappresentati in modo approssimato (numero macchina) con 7 o 15 cifre decimali significative, mentre in matematica una variabile reale (ADT) potrebbe avere un numero illimitato di cifre e quindi una precisione infinita.
Per garantire la portabilità del bytecode e l'indipendenza dalla piattaforma, ogni specifico JRE deve garantire che le variabili dei nove tipi di dato primitivo assumano sempre i medesimi insiemi di valori e seguano lo stesso comportamento, definiti dalla tecnologia Java.

Tipi interi

L'insieme base dei tipi di dato primitivi interi (*integer*) del Java è il sottoinsieme dei numeri relativi (interi con segno) contenuti tra un valore minimo (negativo) e uno massimo (positivo).
I numeri interi con segno sono rappresentati, nel programma eseguibile, con la notazione interna in complemento a due, per cui se il compilatore riserva n bit per una variabile, l'intervallo dell'insieme base varia da -2^{n-1} fino a $+2^{n+1}-1$. La tabella seguente presenta l'insieme base dei quattro tipi interi previsti dal Java.

Tipo di dato primitivo	Byte riservati in memoria	I valori dell'insieme base vanno …
byte	1 (8 bit)	da -128 a $+127$
short	2 (16 bit)	da -32768 a $+32767$
int	4 (32 bit)	da -2147483648 a $+2147483647$
long	8 (64 bit)	da -9223372036854775808 a $+9223372036854775807$

Le operazioni disponibili per i tipi interi sono le seguenti.

Operazione	Simbolo operatore
Addizione o segno positivo	+
Sottrazione o cambiamento di segno	-
Moltiplicazione	*
Divisione intera	/
Modulo (resto della divisione tra numeri interi)	%
Incremento (aumenta una variabile di una unità)	++
Decremento (diminuisce una variabile di una unità)	--
Assegnazione	=

L'insieme dei valori assunto da una variabile intera è limitato, per cui si può verificare una condizione di overflow (letteralmente "traboccamento"), ogniqualvolta il risultato di una elaborazione supera la capacità di memorizzazione. La notazione in complemento a due è di tipo "circolare", per cui se un numero supera, a esempio, il massimo valore positivo rappresentabile, la notazione "ritorna indietro" partendo dal minor numero negativo. Allo stesso modo, se un numero trabocca il numero minimo rappresentabile, il valore "riparte" da quello maggiore. A esempio, il seguente frammento di codice sorgente:

```
public class OverflowVarIntLong {
	public static void main(String args[])	{
		int a = 2147483647, b = a + 1;
		long c = 92233720368547758071, d = c + 1;
		System.out.print("\n\n\n\n\n\tVarInt  a      =  " + a);
		System.out.print("\n\t        a + 1 = " + b);
		System.out.print("\n\tVarLong c      =  " + c);
		System.out.print("\n\t        c + 1 = " + d + "\n\n\n\n\n");
	}
}
```

1. assegna alla variabile *a* di tipo `int` il massimo numero intero rappresentabile (con 32 bit);
2. incrementa di una unità il valore di *a* assegnando il risultato a *b* (*b = a + 1*);
3. visualizza sullo schermo il risultato `a = 2147483647  b = -2147483648` che dimostra come il valore di *b* riparta dal minor numero intero rappresentabile.

Il compilatore Java considera tutti i letterali interi (numeri relativi) in formato decimale (base 10) e del tipo `int`, mentre per dichiarare un numero `long` il suo valore deve terminare con la lettera *L* o *l*. I letterali interi possono anche essere rappresentati nei sistemi di numerazione ottale (base 8) ed esadecimale (base 16), facendo precedere una sequenza di cifre, rispettivamente, da uno 0 (zero) e dalla coppia 0x | or 0X. Nel caso di cifre esadecimali, i valori da 10 a 15 sono rappresentati dalle lettere (maiuscole o minuscole) dalla *A* | or *a* alla *F* | or *f*.

Il seguente frammento di codice sorgente:

```
public class VarDecOctHex {
	public static void main(String args[]) {
		int variabileDecimale = 61, variabileOttale = 075, variabileEsadecimale = 0x3D;
		long variabileLong = 0xFFFFFFFFFL;
		System.out.print("\n\n\n\n\n\tVariabile decimale    = " + variabileDecimale);
		System.out.print("\n\tVariabile ottale      = " + variabileOttale);
		System.out.print("\n\tVariabile esadecimale = " + variabileEsadecimale);
		System.out.print("\n\tVariabile Long        = " + variabileLong + "\n\n\n\n\n");
	}
}
```

dichiara e inizializza tre variabili di tipo `int` che rappresentano lo stesso numero, codificato in decimale (*varDec*), ottale (*varOct*) ed esadecimale (*varHex*), come dimostra l'output ottenuto inserendo le linee in un metodo *main()* di un'applicazione console.

```
Variabile decimale    = 61
Variabile ottale      = 61
Variabile esadecimale = 61
Variabile Long        = 68719476735
```

La variabile *varLong* di tipo `long` rappresenta un numero con 9 cifre esadecimali (36 bit), impossibile da rappresentare con un altro tipo primitivo intero.

Tra **byte**, **short**, **int** e **long**, il tipo intero "predefinito" per i programmatori Java è quello **int**. Nelle applicazioni si ricorre a **byte** e **short** per codificare elaborazioni sui numeri binari, mentre si usa **long** quando si devono elaborare numeri interi che superano il valore massimo previsto per il tipo **int**.

Tipi reali

L'insieme base dei tipi float e double è un sottoinsieme limitato dei numeri reali rappresentati interamente in virgola mobile (floating point) in singola (float) e doppia precisione (double).

Tipo di dato primitivo	Byte riservati in memoria	Numero di cifre decimali significative (precisione macchina)	I valori dell'insieme base vanno ...
float	4 (32 bit)	7	circa da $\pm 3{,}40 \cdot 10^{-39}$ a $\pm 3{,}40 \cdot 10^{38}$
double	8 (64 bit)	15	circa da $\pm 1{,}79 \cdot 10^{-309}$ a $\pm 1{,}79 \cdot 10^{308}$

Le operazioni definite sui tipi reali sono tutte quelle previste per gli interi (compresa quella per il calcolo del resto %). L'insieme base delle variabili float e double è un sottoinsieme dei numeri reali, per cui l'esecuzione di una operazione può provocare una condizione di overflow o di underflow, se il risultato dell'elaborazione "trabocca", rispettivamente, il valore massimo (numero maggiore di circa $\pm 10^{38}$ o $\pm 10^{308}$) o minimo rappresentabili (numero minore di circa $\pm 10^{-39}$ o $\pm 10^{-309}$).
Un letterale reale è un numero frazionario, per default in doppia precisione, che può essere scritto nella forma $\pm parte\_intera.parte\_frazionaria$ (questa è per noi una forma usuale della matematica con il punto radice al posto della virgola) oppure in forma esponenziale $\pm intera.frazionariaEsponente$, in cui in luogo della base 10 si usa la lettera *E* |or *e* (maiuscola o minuscola). Per dichiarare esplicitamente una variabile reale in singola precisione si deve far terminare il numero dalla lettera *F* |or *f* mentre una lettera *D* |or *d* finale, contraddistingue, in modo ridondante, un valore in doppia precisione.
Il Java evita alcuni errori, che avvengono durante l'esecuzione (*run time*) dei programmi, legati a problemi di overflow dei numeri reali, assegnando alle variabili le costanti della classe *Double* (che vedremo in seguito): *NaN* (*Not a Number*, per la divisione con lo zero), *NEGATIVE_INFINITY* oppure *POSITIVE_INFINITY* (per codificare l'infinito negativo $-\infty$ e quello positivo $+\infty$).
Nei codici sorgente, è sempre conveniente scrivere i numeri reali con il punto radice e una parte frazionaria (che, se nulla, può essere scritta come intera .0) per non imporre al compilatore la conversione da numero intero a reale. Tra **float** e **double**, il tipo reale "predefinito" per i programmatori Java è quello **double**, per avere a disposizione un maggior numero di cifre significative nei numeri macchina.
Il seguente frammento di codice sorgente presenta la medesima elaborazione (il calcolo di una percentuale) eseguita però con numeri reali in singola e in doppia precisione.

```
public class VarFloatDouble {
	public static void main(String args[]) {
		float percentualeF = 0.758F, capitaleF = 9805566.0F, risultatoF;
		double percentualeD = 0.758, capitaleD = 9805566.0, risultatoD;
		risultatoF = percentualeF*capitaleF;
		risultatoD = percentualeD*capitaleD;
		System.out.print("\n\n\n\n\n\tRisultato in singola precisione = " + risultatoF);
		System.out.print("\n\tRisultato in doppia precisione  = " + risultatoD + "\n\n\n\n\n");
	}
}
```

L'esecuzione del codice precedente, inserito nel corpo del metodo *main()* di un'applicazione console, produce il seguente risultato, che dimostra come l'elaborazione con dati in singola precisione produce un errore pari a 0.028 rispetto al calcolo eseguito con una precisione maggiore.

```
Risultato in singola precisione = 7432619.0
Risultato in doppia precisione  = 7432619.028
```

Tipo carattere

L'insieme base del tipo **char** è formato da numeri interi senza segno a 16 bit (da 0 a $2^{16} - 1$), che rappresentano i simboli del codice alfanumerico Unicode.
In un programma, i valori assunti da una variabile di tipo **char** possono essere:

- ✓ un letterale carattere, formato da un simbolo preceduto e seguito da un singolo apice;
- ✓ il codice numerico Unicode scritto in decimale, ottale (formato '\XXX', con XXX numero in base 8) oppure esadecimale (formato '\uXXXX'. con XXXX 4 cifre in base 16).

Un simbolo rappresentato in ASCII con il codice esadecimale XX diventa quindi in Unicode '\u00XX'.
Il seguente frammento di codice, inserito nel metodo *main()* di un'applicazione console:

```
public class VariabileCarattere {
	public static void main(String args[]) {
		char c1 = 'd', c2 = 'a', c3 = 't', c4 = 'o';

		System.out.print("\n\n\n\n\n\t" + c1 + c2 + c3 + c4 + "\t");
		System.out.print(c1+c2+c3+c4 + "\n");

		c1 = 'd'-32;
		c2 = 97-32;
		c3 = '\164'-32;
		c4 = '\u006F'-32;

		System.out.print("\n\n\n\n\n\t" + c1 + c2 + c3 + c4 + "\t");
		System.out.print(c1+c2+c3+c4 + "\n\n\n\n\n\n");
	}
}
```

stampa sullo schermo la parola `dato`, codificando i suoi caratteri in modi differenti.
L'impiego dell'istruzione di output con l'operatore +:

```
System.out.print(c1+c2+c3+c4 + "\n");
```

provoca la stampa del numero `424` ovvero la somma dei codici Unicode dei quattro caratteri 'd', 'a', 't' e 'o'.
Le operazioni definite sul tipo **`char`** sono le medesime dei tipi interi.
Le linee di codice:

```
c1 = 'd'-32;
c2 = 97-32;
c3 = '\164'-32;
c4 = '\u006F'-32;
System.out.print("\n\n\n\n\n\t" + c1 + c2 + c3 + c4 + "\t");
System.out.print(c1+c2+c3+c4 + "\n\n\n\n\n\n");
```

stampa sullo schermo la parola DATO. La trasformazione delle lettere deriva dall'osservazione che nel codice ASCII (e quindi in quello Unicode), il codice delle lettere minuscole è sempre maggiore di quelle maiuscole di un valore pari a 32.
Per rappresentare i caratteri speciali che non possono essere digitati (a esempio un salto linea, u invio o una tabulazione), si impiega una sequenza di escape definita da una barra rovesciata \ (backslash) seguita da uno oppure più simboli.

Sequenza di escape	Carattere rappresentato
\b	backspace (cancella un carattere a sinistra)
\f	nuova pagina
\n	nuova linea
\r	ritorno a capo
\t	tabulazione orizzontale
\\	barra rovesciata
\'	apice singolo
\"	doppi apici

Dai tipi di dato primitivi alle classi

Una classe (*class*) è un tipo di dato progettato e realizzato appositamente dagli utenti del linguaggio (i programmatori), usando il costrutto del linguaggio **`class`**.
Al nostro livello di studio, possiamo considerare una classe come un tipo di dato in cui:

1. l'insieme base è definito dai valori che può assumere un insieme di variabili denominate attributi;
2. le operazioni sono dichiarate mediante metodi.

Gli attributi sono variabili di tipo primitivo oppure di altre classi, mentre i metodi dichiarano l'elaborazione da effettuare sugli attributi.
Un oggetto è una variabile di uno specifico tipo di dato classe. Dopo aver creato una nuova classe, il programmatore può dichiarare, e allocare, la memoria necessaria per un nuovo oggetto di quel tipo di dato, usando la sintassi generale:

```
NomeClasse nomeOggetto;
nomeOggetto = new nomeClasse();
```

oppure la forma seguente, equivalente, più compatta:

```
NomeClasse nomeOggetto = new NomeClasse();
```

Per richiamare attributi e metodi di un oggetto, si impiega una sintassi basata sull'operatore . del tipo:
nomeOggetto.nomeAttributo
nomeOggetto.nomeMetodo([<paramtroIngresso1>repeat]optional, [<paramtroIngresso1>repeat]optional)
I metodi sono riconoscibili dagli attributi degli oggetti, perché il loro identificatore deve sempre essere seguito dalla coppia di parentesi *()* al cui interno può essere presente una lista di parametri di ingresso che definiscono i dati di input dell'operazione implementata. Il metodo speciale *NomeClasse()*, scritto con lo stesso nome della classe dopo **new**, rappresenta il costruttore e permette di inizializzare un oggetto con dei dati iniziali.
Nelle applicazioni, gli attributi sono "invisibili" per gli utenti della classe (altri programmatori), ma la loro lettura/scrittura avviene mediante la dichiarazione di opportuni metodi denominati: setAttributo(*valoreAttributo*), per scrivere i valori degli attributi, e getAttributo() per leggere i dati delle variabili degli oggetti.
Ogni IDE di programmazione Java dispone di una collezione di classi già dichiarate e quindi pronte per essere richiamate nei programmi con la creazione di nuovi oggetti.
Per risolvere un problema, seguendo il paradigma (modello) object oriented, un programmatore usa di nuovo le classi già disponibili e ne dichiara nuovi tipi specifici per completare la soluzione.

Classe String

La classe *String* permette di gestire, in un programma, oggetti contenenti dati stringa, che non possono modificare la propria dimensione in un programma. Un letterale stringa è formato da una lista lineare di caratteri (Unicode), racchiusi da una coppia (iniziale e finale) di doppi apici. Data l'importanza delle stringhe nell'elaborazione, il Java semplifica la gestione degli oggetti di questa classe in modo tale che i programmatori possono pensare di avere un nuovo tipo di dato primitivo *String*. A esempio, il linguaggio permette di dichiarare nuovi oggetti stringa senza usare **new**, ma seguendo una sintassi analoga a quella delle variabili di tipo primitivo.
Il seguente frammento di codice sorgente:

```
public class ClasseString {
	public static void main(String args[]) {
		String stringa1 = "ITC: Information and Communication Technology";
		String stringa2 = new String("ITC: Information and Communication Technology");
		System.out.print("\n\n\n\n\n\t" + stringa1 + "\n\t" + stringa2 + "\n\n\n\n\n");
	}
}
```

visualizza sulla console il contenuto di un nuovo oggetto *stringa* di tipo *String* assegnandogli come valore "ICT: Information and Communication Technology". La dichiarazione precedente è equivalente a quella successiva, che impiega il costruttore *String()* della classe con l'operatore **new** di allocazione di un nuovo oggetto.

```
String stringa2 = new String("ITC: Information and Communication Technology");
```

Se in un programma sorgente è presente un letterale stringa, il compilatore lo converte automaticamente (in modo trasparente per il programmatore) in un oggetto *String*, il cui identificatore è lo stesso testo della stringa. A esempio, nell'istruzione di output

```
System.out.print("Risultato=");
```

il letterale "Risultato=" è elaborato dalla JVM come un nuovo oggetto *String* di nome "Risultato=".
Per inizializzare un oggetto può essere utile la stringa vuota "", che non contiene alcun carattere.
Gli oggetti di tipo *String* dispongono soltanto dell'operatore di concatenazione +, che applicato a due stringhe produce un testo formato dall'unione di quelli di partenza. Gli oggetti *String* possono essere uniti con altre espressioni, variabili e/o oggetti, che il Java converte automaticamente in stringhe prima dell'operazione di concatenazione.
In un'espressione con l'operatore + è sufficiente che un solo elemento sia una stringa perché tutti gli altri siano convertiti in oggetti *String*. A esempio, nel seguente frammento di codice Java:

```
public class ClasseString2 {
	public static void main(String args[]) {
		double percentuale = 5.4, capitale = 3680.0;
		System.out.print("\n\n\n\n\n\tL\'interesse per un anno del " + percentuale + "% sul capitale
di " + capitale + " è uguale a " + (percentuale*capitale/100) + "\n\n\n\n\n");
	}
}
```

le variabili reali percentuale e capitale e il risultato dell'espressione (percentuale*capitale/100.0) sono convertiti in testo e quindi concatenate tra loro con i letterali "L'interesse per un anno del ", "% sul capitale di " e "è uguale a " per formare il seguente oggetto stringa, creato automaticamente dal compilatore.

```
L'interesse per un anno del 5.4% sul capitale di 3680.0 è uguale a 198.72
```

La seguente tabella descrive alcune operazioni (metodi) utili nell'elaborazione di oggetti stringa. Le stringhe sono liste di caratteri per cui ogni elemento dispone di una posizione data dalla distanza dall'inizio della lista stessa. In Java, al primo carattere della stringa (testa della lista) è assegnata la posizione 0.

Come fare per ...	Carattere rappresentato
... calcolare la lunghezza di un *oggetto* stringa definita dal numero di caratteri presenti.	*oggetto*.lenght()
... estrarre da un *oggetto* stringa una sottostringa definita dai caratteri contenuti tra una posizione iniziale *pIniziale* e una finale *pFinale* (escluso il carattere ella posizione finale).	*oggetto*.substring(*pIniziale*, *pFinale*)
... leggere il carattere posto in una data *posizione* di un *oggetto* stringa (da 0 fino alla lunghezza della stringa -1).	*oggetto*.charAt(*posizione*)
... confrontare tra loro due stringhe *oggetto* e *stringa2* restituendo un valore booleano *true* se le liste di caratteri sono identiche.	*oggetto*.equals(*stringa2*)
... confrontare tra loro due stringhe *oggetto* e *stringa2* ignorando le differenze tra caratteri maiuscoli e minuscoli, restituendo un valore booleano *true* se le liste di caratteri sono identiche.	*oggetto*.equalsIgnoreCase(*stringa2*)
... convertire in maiuscolo (*upper case*) o in minuscolo (*lower case*) tutti i caratteri di un oggetto stringa.	*oggetto*.toUpperCase() *oggetto*.toLowerCase()
... ricercare la prima oppure l'ultima (*last*) occorrenza di un solo *carattere* oppure di una *sottostringa* in un *oggetto* stringa. Il risultato vale -1 se la ricerca non ha avuto successo.	*oggetto*.indexOf(*carattere* |or *sottostringa*) *oggetto*.lastIndexOf(*carattere* |or *sottostringa*)
... sostituire in un *oggetto* stringa tutti i valori di un carattere (*cRicercato*) con un altro (*cSostituito*).	*oggetto*.replace(cRicercato, cSostituito)
... creare una copia di un *oggetto* stringa in cui sono stati eliminati tutti gli spazi iniziali e finali.	*oggetto*.trim()

Il seguente frammento di codice sorgente, che impiega alcuni metodi della classe *String*:

```
public class ClasseString3 {
	public static void main(String args[]) {
		String s1, s2, s3, s4;
		s1 = "informatica";
		s2 = s1.toUpperCase();
		s3 = s2.substring(0, 5);
		s4 = s2.substring(5, 11);

		System.out.print("\n\n\n\n\n\tLunghezza di informatica: " + s1.length());
		System.out.print("\n\t" + s2 + " = " + s3 + "mazione + auto" + s4 + "\n\n\n\n\n");
	}
}
```

visualizza sullo schermo il seguente risultato.

```
Lunghezza di informatica: 11
INFORMATICA = INFORmazione + autoMATICA
```

Classe Math

La classe Math dispone di numerosi metodi statici che permettono di eseguire elaborazioni matematiche, tra le quali ricordiamo quelle descritte nella tabella che segue. I dati di ingresso (indicati nella tabella con *x* e *y*) e i risultati forniti dai metodi sono sempre rappresentati in doppia precisione (*double*).

Come fare per ...	Si usano i metodi statici della classe Math:
... calcolare la potenza y^x, con *x* e *y* numeri reali generici.	Math.pow(*y*, *x*)
... calcolare la radice quadrata di un numero *x*.	Math.sqrt(*x*)
... convertire un angolo *x* da radianti in gradi e viceversa.	Math.toDegrees(*x*) Math.toRadians(*x*)
... calcolare le funzioni trigonometriche seno, coseno e tangente, con l'argomento *x* in radianti.	Math.sin(*x*) Math.cos(*x*) Mat.tan(*x*)
... calcolare le funzioni trigonometriche inverse arcoseno, arcocoseno e arcotangente, restituendo un risultato espresso in radianti.	Math.asin(*x*) Math.acos(*x*) Math.atan(*x*)
... calcolare le funzioni esponenziale (e^x) e logaritmo naturale ($\log_e x$).	Math.exp(*x*) Math.log(*x*)
... trasformare un numero reale *x* in un intero troncando la parte frazionaria dopo averla arrotondata.	Math.round(*x*)
... generare un numero pseudocasuale compreso tra 0 e 1.	Math.random(*x*)

La classe Math dispone anche delle costanti Math.PI e Math.E, che rappresentano in modo approssimato, rispettivamente, le costanti pigreco e il numero di Nepero (base dei logaritmi naturali).