Capitolo 1

Introduzione al linguaggio

Java è un moderno linguaggio di programmazione *object-oriented*, *general-purpose*, *concorrente* e *class-based*, le cui origini si possono far risalire al lontano 1991, quando, presso Sun Microsystems, un team di straordinari programmatori e hacker del codice, formato principalmente da James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan, iniziò a lavorare al suo sviluppo.
In principio il linguaggio fu chiamato OAK ("quercia" in inglese, in onore dell'albero che Gosling vedeva dalla finestra del suo ufficio) e fu sviluppato in seno a un progetto chiamato Green Project (i progettisti furono chiamati il *Green Team*).
Lo scopo del progetto era quello di dotare svariati dispositivi elettronici di consumo di un meccanismo tramite il quale fosse possibile far eseguire, in modo indipendente dalla loro differente architettura, i programmi scritti per essi.
Questo meccanismo si sarebbe dovuto concretizzare nella scrittura di un componente software, denominato *virtual machine* (macchina virtuale), da implementare per il particolare hardware del dispositivo e che sarebbe stato in grado di far girare il *codice intermedio*, denominato *bytecode*, generato da un compilatore del linguaggio Java.
Detto in altro modo, l'obiettivo era quello di permettere agli sviluppatori di scrivere i programmi in Java una sola volta e con la certezza che sarebbe stato possibile eseguirli su tutti i dispositivi hardware dotati, per l'appunto, di una macchina virtuale in grado di interpretare ed eseguire il bytecode (da qui l'acronimo WORA, *Write Once, Run Anywhere*, o anche WORE, *Write Once, Run Everywhere*).

In questo capitolo

- **Cenni sull'architettura di un elaboratore**
- **Paradigmi di programmazione**
- **Concetti introduttivi allo sviluppo in Java**
- **Elementi di un ambiente Java**
- **Il primo programma in Java**
- **Compilazione ed esecuzione del codice**
- **La Java Virtual Machine: un breve dettaglio**

Nel maggio del 1995 fu completato lo sviluppo di OAK, con l'annuncio alla conferenza Sun World '95. Con l'occasione, visto che il nome OAK apparteneva già a un altro linguaggio di programmazione, si decise di cambiare il nome del linguaggio in Java.

NOTA STORICA

Sun Microsystems era una società multinazionale, produttrice di software e di hardware, fondata nel 1982 da tre studenti dell'università di Stanford: Vinod Khosla, Andy Bechtolsheim e Scott McNealy. Il nome è infatti l'acronimo di *Stanford University Network*. Nel gennaio del 2010 Sun è stata acquistata dal colosso informatico Oracle Corporation per la considerevole cifra di 7,4 miliardi di dollari. Tra i suoi prodotti software ricordiamo il sistema operativo Solaris e il file system di rete NFS, mentre tra i prodotti hardware le workstation e i server basati sui processori RISC SPARC.

CURIOSITÀ

Le notizie "leggendarie" che circolano in merito alla nascita del linguaggio Java narrano che questo nome fu scelto durante un incontro tra i suoi progettisti in un bar mentre bevevano caffè americano; infatti *java* è un termine utilizzato nello slang per indicare una bevanda fatta con miscele di chicchi di caffè.

Successivamente, nel 1996, alla prima *JavaOne Developer Conference*, venne rilasciata la prima versione di Java (la 1.0) che suscitò, da subito, interesse e attenzione nel mondo dell'*information technology* perché prometteva di essere un linguaggio di programmazione semplice, robusto, sicuro, portabile e, come già detto, indipendente da qualsiasi dispositivo hardware (*architecture neutral*). A partire da quel momento iniziò una capillare e profonda diffusione di Java, soprattutto grazie all'esplosione di Internet e del World Wide Web dove, all'epoca, non esistevano programmi che permettessero di fornire contenuto dinamico alle pagine HTML (se si escludevano gli script CGI). Con Java invece fu possibile, attraverso particolari programmi (*applet*), generare contenuti web dinamici e allo stesso tempo indipendenti dal sistema operativo e dal browser sottostante (il browser di allora più famoso, Netscape Navigator, incorporò, infatti, la *tecnologia* Java).
Dopo di ciò il successo di Java è stato davvero prorompente e inarrestabile e per l'utilizzo con il linguaggio è stato creato un vasto insieme di librerie software. Solo per citarne alcune: per l'accesso indipendente ai database (JDBC); per lo sviluppo di applicazioni web sia *semplici* sia di livello *enterprise* (Servlet/JSP/JSF); per lo sviluppo di sofisticate interfacce utente (Swing, Java FX); per la programmazione di rete; per la progettazione di applicazioni multithreading e così via.

NOTA

Nelle versioni 1.0 e 1.1, le release di Java avevano il prefisso JDK (*Java Development Kit*); dalla versione 1.2 fino alla versione 1.5, J2SE (*Java 2 Standard Edition*); dalla versione 1.6 in poi, Java SE (*Java Standard Edition*). Inoltre, a partire dalla release con nome in codice Tiger si ha un numero di versione interna (1.5, 1.6 e così via), denominata *developer version*, e un numero di versione esterna (5.0, 6 e così via), denominata *product version*. Infine, dalla versione 9 è scomparso il numero di versione interna e dalla versione 10, come già detto nell'Introduzione, si è adottato un nuovo sistema di versioning *time-based* a cadenza semestrale. Nel momento di pubblicazione del presente libro l'ultima versione disponibile è la 23 del 17 settembre 2024 (Java SE 23).

Tabella 1.1 Release di Java dalle origini fino alla versione 9, pre-cambiamento time-based.

Prefisso e versione	Nome in codice	Data di rilascio finale
JDK 1.0	Oak	23 gennaio 1996
JDK 1.1	Sparkler	19 febbraio 1997
JDK 1.1.4	Sparkler	12 settembre 1997
JDK 1.1.5	Pumpkin	3 dicembre 1997
JDK 1.1.6	Abigail	24 aprile 1998
JDK 1.1.7	Brutus	28 settembre 1998
JDK 1.1.8	Chelsea	8 aprile 1999
J2SE 1.2	Playground	4 dicembre 1998
J2SE 1.2.1	(nessuno)	30 marzo 1999
J2SE 1.2.2	Cricket	8 luglio 1999
J2SE 1.3	Kestrel	8 maggio 2000
J2SE 1.3.1	Ladybird	17 maggio 2001
J2SE 1.4.0	Merlin	13 febbraio 2002
J2SE 1.4.1	Hopper	16 settembre 2002
J2SE 1.4.2	Mantis	26 giugno 2003
J2SE 1.5.0 (5.0)	Tiger	29 settembre 2004
Java SE 1.6 (6)	Mustang	11 dicembre 2006
Java SE 1.7 (7)	Dolphin	28 luglio 2011
Java SE 1.8 (8)	(nessuno)	18 marzo 2014
Java SE 9	(nessuno)	27 luglio 2017

Oggi, dunque, possiamo asserire tranquillamente, e con un certo grado di certezza, che Java è un importante e potente linguaggio di programmazione *mainstream*; ciò lo dimostra anche il fatto che la sua conoscenza è una competenza molto richiesta nel mondo del lavoro e che il suo rating di popolarità è senza dubbio tra i primi posti come linguaggio maggiormente preferito dagli sviluppatori software, come dimostrato anche dal famoso indicatore di popolarità TIOBE (Figura 1.1 e 1.2).

DETTAGLIO

L'indicatore *TIOBE Programming Community index* misura la popolarità di un linguaggio di programmazione *Touring completo*. Viene aggiornato mensilmente in base al numero dei risultati delle ricerche effettuate con 25 search engine di una query avente il pattern `+"<language> programming"`.

CURIOSITÀ

Oracle, in merito all'attuale capillarità e diffusione del linguaggio Java nel mondo, enumera i seguenti dati: è studiato da milioni di studenti con un bacino di milioni di sviluppatori software, è il "motore" di 60 miliardi di device hardware ed è la piattaforma principale per il *cloud development*.

Apr 2024	Apr 2023	Change	Programming Language	Ratings	Change
1	1		Python	16.41%	+1.90%
2	2		C	10.21%	-4.20%
3	4	^	C++	9.76%	-3.20%
4	3	v	Java	8.94%	-4.29%
5	5		C#	6.77%	-1.44%
6	7	^	JavaScript	2.89%	+0.79%
7	10	^	Go	1.85%	+0.57%
8	6	v	Visual Basic	1.70%	-2.70%
9	8	v	SQL	1.61%	-0.06%
10	20	^^	Fortran	1.47%	+0.88%
11	11		Delphi/Object Pascal	1.47%	+0.24%
12	12		Assembly language	1.30%	+0.26%
13	18	^^	Ruby	1.24%	+0.58%
14	17	^	Swift	1.23%	+0.51%
15	15		Scratch	1.14%	+0.35%
16	14	v	MATLAB	1.11%	+0.25%
17	9	vv	PHP	1.09%	-0.26%
18	38	^^	Kotlin	1.05%	+0.80%

Figura 1.1 Tabella risultati TIOBE (rating aggiornato a settembre 2024).

Programming Language	2024	2019	2014	2009	2004	1999	1994	1989
Python	1	4	8	6	10	29	22	-
C	2	2	1	2	2	1	1	1
C++	3	3	4	3	3	2	2	3
Java	4	1	2	1	1	14	-	-
C#	5	6	5	7	8	25	-	-
JavaScript	6	7	9	9	9	19	-	-
Visual Basic	7	19	-	-	-	-	-	-
SQL	8	9	-	-	7	-	-	-
PHP	9	8	6	5	6	-	-	-
Go	10	18	41	-	-	-	-	-
Objective-C	30	10	3	38	47	-	-	-
Lisp	33	29	14	20	15	16	6	2
(Visual) Basic	-	-	7	4	5	3	3	7

Figura 1.2 Tabella risultati TIOBE (scostamento di posizione rispetto a determinate annualità aggiornato a settembre 2024).

Cenni sull'architettura di un elaboratore

Un linguaggio di programmazione, indipendentemente dal fatto che sia categorizzato come linguaggio di basso, medio o alto livello, deve sempre far affidamento al sottostante hardware per il suo funzionamento e per la produzione del codice eseguibile specifico.

TERMINOLOGIA

Un linguaggio di programmazione è definibile di *alto livello* se offre un alto livello di astrazione e indipendenza rispetto ai dettagli hardware di un elaboratore. Ciò implica che un programmatore utilizzerà keyword e costrutti sintattici di facile comprensione che gli permetteranno di scrivere un programma in modo relativamente semplificato, con la possibilità di concentrarsi solo sulla logica dell'algoritmo da implementare. I linguaggi di alto livello sono definibili come linguaggi *closer to humans*. Al contrario, un linguaggio di programmazione è definibile di *basso livello* quando non offre alcun layer di intermediazione/astrazione rispetto all'hardware da programmare: il programmatore deve non solo avere una profonda conoscenza di tale hardware, ma anche lavorare direttamente con esso (registri, memoria e così via). I linguaggi di basso livello sono definibili come linguaggi *closer to computers*. Infine, un linguaggio di programmazione è definibile di *medio livello* quando mette a disposizione del programmatore sia funzionalità di *alto livello* (per esempio il costrutto di *funzione* o il costrutto di *struttura* o *classe*), che garantiscono una maggiore efficienza e flessibilità nella costruzione dei programmi, sia funzionalità di *basso livello* (come il costrutto di *puntatore*), che garantiscono, al pari dei linguaggi *machine-oriented* come l'assembly, una maggiore efficienza nello sfruttamento diretto dell'hardware sottostante. In base a quanto detto, Java può considerarsi un linguaggio di programmazione di alto livello principalmente perché: i dettagli dell'hardware sottostante non gli sono disponibili; non permette di agire direttamente sulla memoria, non fornendo al programmatore costrutti per la sua allocazione o deallocazione; non consente costrutti *unsafe* (non sicuri) che permettano, per esempio, di accedere a un elemento di un array fuori dai suoi limiti.

Quindi, prima di addentrarci nello studio sistematico del linguaggio Java, conviene delineare alcuni concetti teorici che riguardano la struttura di un generico elaboratore elettronico, soffermandoci in modo più approfondito su due componenti in particolare: la CPU e la memoria centrale.
Ciò si rende opportuno soprattutto quando un linguaggio di programmazione consente di sfruttare a basso livello l'hardware di un sistema target. Pertanto, comprendere, seppure a grandi linee, come un computer è progettato e costituito può sicuramente aiutare a scrivere programmi che *dialogano* con l'hardware sottostante, con maggiore consapevolezza dei loro effetti (capire, per esempio, com'è strutturata la memoria centrale può sicuramente aiutare a gestirla in modo più sicuro ed efficiente).
In ogni caso, quanto diremo resterà comunque valido anche per un linguaggio di alto livello come Java che, pur non consentendo normalmente di sfruttare in modo diretto con i suoi costrutti l'hardware sottostante, permette comunque, tramite un'apposita interfaccia (JNI, *Java Native Interface*), di utilizzare codice scritto nativamente in C o C++ che può far uso, per esempio, di costrutti propri per l'accesso diretto alla memoria o all'hardware sottostante.

Il modello di von Neumann

I linguaggi imperativi, fra i quali Java, condividono un modello computazionale che rappresenta un'astrazione del sottostante calcolatore elettronico, dove la computazione procede modificando valori memorizzati all'interno di locazioni di memoria. Questo modello è definito come *architettura di von Neumann*, dal nome dello scienziato ungherese

John von Neumann che la ideò nel 1945. Tale modello è alla base della progettazione e costruzione dei computer e quindi dei linguaggi imperativi che vi si rifanno e che rappresentano, dunque, astrazioni della macchina di von Neumann (Figura 1.3).

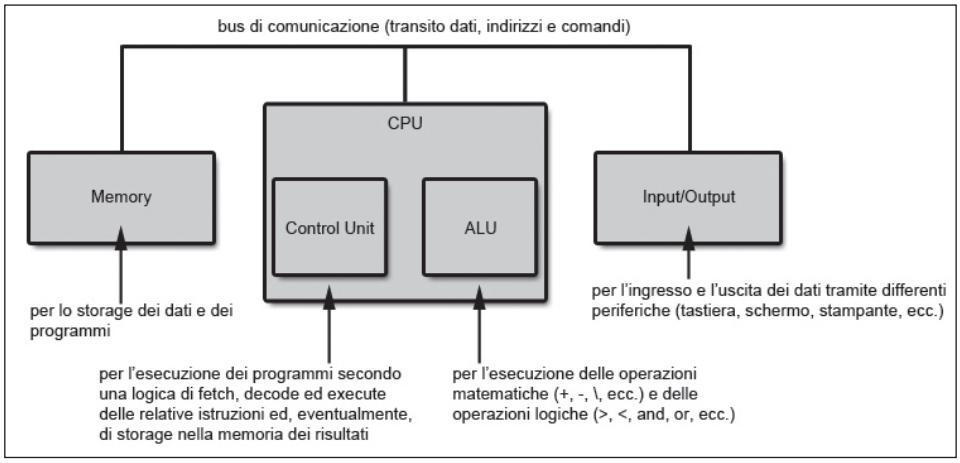

Figura 1.3 Architettura semplificata di un computer, basata sul modello di von Neumann.

In sostanza, nel modello di von Neumann un computer è costituito dai seguenti elementi: una CPU (*Central Processing Unit*) per il controllo e l'esecuzione dell'elaborazione, al cui interno si trovano l'ALU (*Arithmetic Logic Unit*) e una *Control Unit*; celle di memoria identificate da un indirizzo numerico, atte a ospitare i dati coinvolti nell'elaborazione; dispositivi per l'input e l'output dei dati da e verso l'elaboratore; un bus di comunicazione tra le varie parti per il transito di dati, indirizzi e segnali di controllo.
Sia i dati, sia le istruzioni di programmazione sono collocati in memoria. In pratica, nel modello di von Neumann abbiamo due elementi caratterizzanti: la memoria, che memorizza le informazioni, e il processore, che fornisce operazioni per modificare il contenuto, ossia lo stato della memoria.
In definitiva, un computer digitale modellato secondo l'architettura di von Neumann non è altro che un sistema di componenti quali processori, memorie, device di input/output e così via tra di loro interconnessi (tramite *bus*) che cooperano congiuntamente al fine di svolgere i processi computazionali per i quali sono stati progettati.

La CPU

La CPU (*Central Processing Unit*) è il "cervello" di ogni computer ed è deputata principalmente a interpretare ed eseguire le istruzioni elementari che rappresentano i programmi e a effettuare operazioni di coordinamento tra le varie parti di un computer.
Questa unità di elaborazione, dal punto di vista fisico, è un circuito elettronico formato da un elevato numero di transistor (da diverse centinaia di milioni fino ai miliardi delle più potenti CPU) ubicati in un circuito integrato (chip) di dimensioni ridotte (pochi centimetri quadrati).
Dal punto di vista logico, invece, una CPU è formata dalle seguenti unità (Figura 1.4).

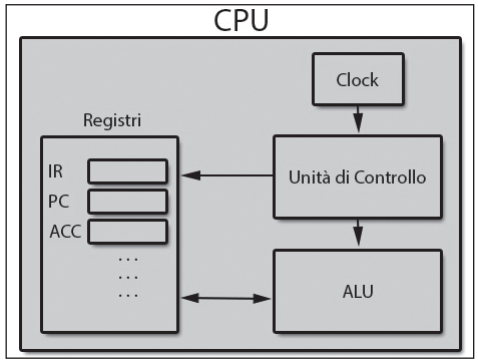

Figura 1.4 Struttura logica di una CPU.

- ALU (*Arithmetic Logic Unit*), ossia l'unità aritmetico-logica. È costituita da un insieme di circuiti deputati a effettuare calcoli aritmetici (somme, sottrazioni e così via) e operazioni logiche (boolean AND, boolean OR e così via) sui dati. Il suo funzionamento si può sintetizzare ponendo, per esempio, un'operazione di somma tra due numeri: preleva da appositi registri di memoria di input gli operandi trasmessi (diciamo X e Y); effettua l'operazione di somma (X + Y); scrive il risultato della somma in un registro di memoria di output. In pratica possiamo considerare la ALU come una sorta di calcolatrice "primitiva", che riceve i dati e vi effettua dei calcoli al fine di produrre un risultato.
- *Control Unit*, ossia l'unità di controllo. Coordina e dirige le varie parti di un calcolatore in modo da consentire la corretta esecuzione dei programmi. In pratica, in una sorta di ciclo infinito, preleva (*fetch*), decodifica (*decode*) ed esegue (*execute*) le istruzioni dei programmi. Attraverso la fase di prelievo acquisisce un'istruzione dalla memoria, la carica nel registro delle istruzioni e rileva la successiva istruzione da prelevare. Attraverso la fase di decodifica interpreta l'istruzione da eseguire. Attraverso la fase di esecuzione esegue ciò che l'istruzione indica. È un'azione di input? Allora incarica l'unità di input di trasferire dei dati nella memoria centrale. È un'azione di output? Allora incarica l'unità di output di trasferire dei dati dalla memoria centrale verso l'esterno. È un'azione di elaborazione dei dati? Allora richiede il trasferimento dei dati nell'ALU, incarica l'ALU di elaborarli e trasferisce il risultato nella memoria centrale. È un'azione di salto? Allora aggiorna il registro *contatore di programma* con l'indirizzo cui saltare. Le operazioni svolte dall'unità di controllo sono regolate da un orologio interno di sistema (*system clock*) che genera segnali o impulsi regolari a una certa frequenza, espressa in *hertz*, che consentono alle varie parti di operare in modo coordinato e sincronizzato. Maggiori sono questi impulsi al secondo, detti anche cicli di clock, maggiore è la quantità di istruzioni per secondo che una CPU può elaborare e quindi la sua velocità.
- *Registers*, ossia i registri. Sono unità di memoria estremamente veloci (possono essere letti e scritti ad alta velocità perché sono interni alla CPU) e di una certa dimensione, utilizzati per specifiche funzionalità. Vi sono numerosi registri; quelli più importanti sono l'*Instruction Register* (registro istruzione), che contiene l'istruzione che si sta eseguendo, il *Program Counter* (contatore di programma), che contiene l'indirizzo della

successiva istruzione da eseguire; gli *Accumulator* (accumulatori), che contengono, temporaneamente, gli operandi di un'istruzione e alla fine della computazione il risultato dell'operazione eseguita dall'ALU.

La memoria centrale

La memoria centrale, detta anche primaria o principale, è quella parte del computer dove sono memorizzate le istruzioni e i dati dei programmi.
Ha diverse caratteristiche: è *volatile*, perché il contenuto si perde nel momento in cui il calcolatore viene spento; è *veloce*, ossia il suo accesso in lettura/scrittura può avvenire in tempi estremamente ridotti; è ad *accesso casuale*, perché il tempo di accesso alle relative celle è indipendente dalla loro posizione e dunque costante per tutte le celle (da questo punto di vista la memoria centrale è definita come RAM, *Random Access Memory*).
L'unità di base di memorizzazione è la cifra binaria o *bit*, che è il composto aplologico delle parole *binary digit*. Un bit può contenere solo due valori: la cifra `0` oppure la cifra `1`; il relativo sistema di numerazione binario richiede, pertanto, solo quei due valori per la codifica dell'informazione digitale.

NOTA

All'indirizzo `https://github.com/thp1972/Libro_Java21` è disponibile il link `Sistemi Numerici` che apre una breve guida sui sistemi numerici e le principali operazioni effettuabili.

La memoria primaria, dal punto di visto logico, è rappresentabile come una sequenza di celle o locazioni di memoria (Figura 1.5), ciascuna delle quali può memorizzare una certa quantità di informazioni. Ogni cella, detta *parola* (*memory word*), ha una dimensione fissa e tale dimensione, espressa in multipli di 8, ne indica la lunghezza, ossia il suo numero di bit (possiamo avere, per esempio, word di 8 bit, di 16 bit, di 32 bit e così via). Così se una cella ha k bit allora può contenere una delle 2^k combinazioni differenti di bit.
Inoltre, la lunghezza della word indica anche la quantità di informazioni che un computer durante un'operazione può elaborare, contemporaneamente e in parallelo.

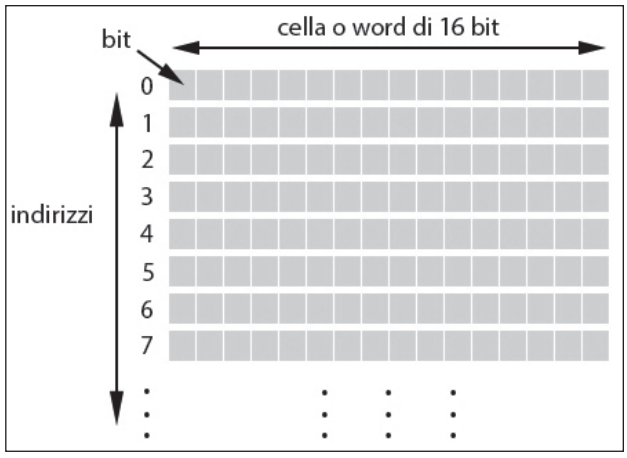

Figura 1.5 Memoria primaria come sequenza di celle.

Altra caratteristica essenziale di una cella di memoria è che ha un indirizzo, ossia un numero binario che ne consente la localizzazione da parte della CPU, al fine di reperire o scrivervi contenuti informativi, anch'essi binari. La quantità di indirizzi referenziabili dipende dal numero di bit di un indirizzo: generalizzando, se un indirizzo ha *n* bit, allora il massimo numero di celle indirizzabili sarà 2^n.
Per esempio, la Figura 1.6 mostra tre differenti layout per una memoria di 160 bit rispettivamente con celle di 8 bit, 16 bit e 32 bit. Nel primo caso sono necessari almeno 5 bit per esprimere 20 indirizzi da 0 a 19 (infatti, 2^5 ne permette fino a 32); nel secondo caso sono necessari almeno 4 bit per esprimere 10 indirizzi da 0 a 9 (infatti, 2^4 ne permette fino a 16); nel terzo caso sono necessari almeno 3 bit per esprimere 5 indirizzi da 0 a 4 (infatti, 2^3 ne permette fino a 8).

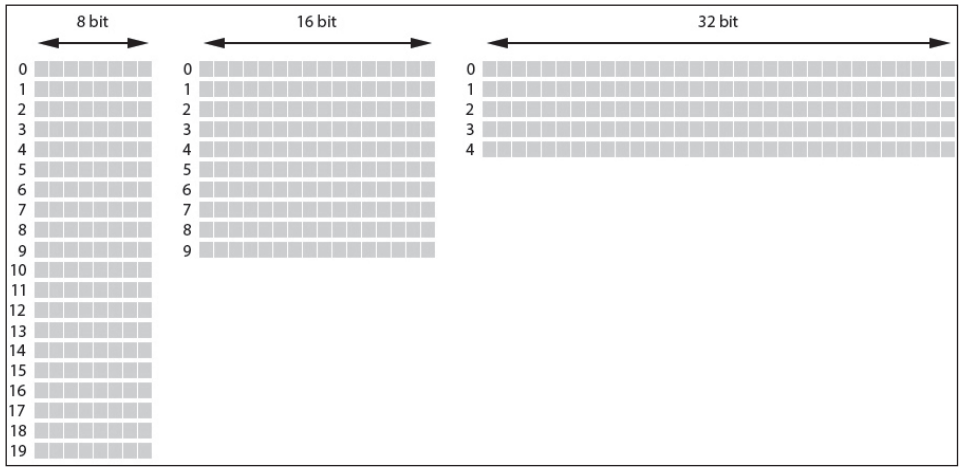

Figura 1.6 Tre differenti layout per una memoria di 160 bit.

In più è importante dire che il numero di bit di un indirizzo è indipendente dal numero di bit per cella: una memoria con 2^{10} celle di 8 bit ciascuna e una memoria con 2^{10} celle di 16 bit ciascuna necessiteranno entrambe di indirizzi di 10 bit.
La dimensione di una memoria è dunque data dal numero di celle per la loro lunghezza in bit. Nei nostri tre casi è sempre di 160 bit (correntemente, un computer moderno avrà una dimensione come minimo di 4 GB di memoria se avrà almeno 2^{32} celle indirizzabili di 1 byte di lunghezza ciascuna).

Ordinamento dei byte

Quando una word è composta da più di 8 bit (1 byte) e quindi 16 bit (2 byte), 32 bit (4 byte) e così via vi è la necessità di decidere come ordinare o disporre in memoria i relativi byte. Oggi esistono due modalità di ordinamento largamente utilizzate dai moderni elaboratori elettronici denominate di seguito.

- *Big endian* (architetture Motorola 68k, SPARC e così via): data una word, il byte più significativo (*most significant byte*) è memorizzato all'indirizzo più basso (*smallest address*) e a partire da quello, da sinistra a destra (*left-to-right*), sono memorizzati i byte successivi.

- *Little endian* (architetture Intel x86, x86-64 e così via): data una word, il byte meno significativo (*least significant byte*) è memorizzato all'indirizzo più basso (*smallest address*) e a partire da quello, da destra a sinistra (*right-to-left*), sono memorizzati i byte successivi.

CURIOSITÀ

I termini sopraindicati si devono allo scrittore e poeta irlandese Jonathan Swift, che nel suo famoso libro *I Viaggi di Gulliver* prendeva in giro i politici che si facevano guerra per il giusto modo di rompere le uova sode: dalla punta più piccola (*little end*) oppure dalla punta più grande (*big end*)? Questo termine fu comunque usato per la prima volta da Danny Cohen, uno scienziato informatico, in un significativo articolo del 1980 dal titolo *On Holy Wars and a Plea for Peace*, rintracciabile al seguente URL: `http://www.ietf.org/rfc/ien/ien137.txt`.

Per comprendere quanto detto, consideriamo come viene memorizzato un numero come 2854123 (binario, 00000000001010111000110011101011) in una word di 32 bit secondo un'architettura big endian e secondo un'architettura little endian (Figura 1.7): nel primo caso il byte più a sinistra (più significativo) è memorizzato nell'indirizzo 0 e poi a seguire, da sinistra a destra, gli altri byte sono memorizzati negli indirizzi 1, 2 e 3; nel secondo caso il byte più a destra (meno significativo) è memorizzato nell'indirizzo 0 e poi a seguire, da destra a sinistra, gli altri byte sono memorizzati negli indirizzi 1, 2 e 3.

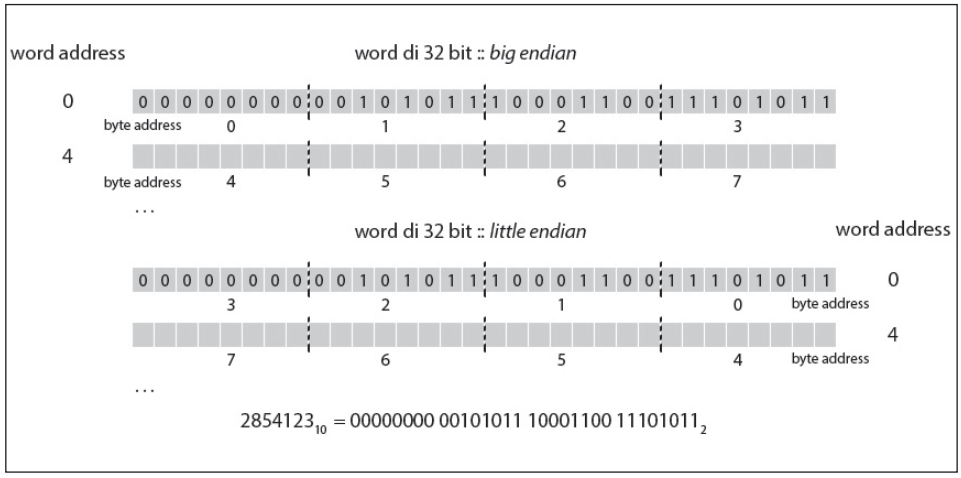

Figura 1.7 Ordinamento della memoria: big endian vs little endian.

TERMINOLOGIA

Se vediamo una word come una sequenza di bit (Figura 1.8) piuttosto che una sequenza di byte, possiamo altresì dire che essa avrà un bit più significativo (*most significant bit*) che sarà ubicato al limite sinistro di tale sequenza (*high-order end*) e un bit meno significativo (*least significant bit*) che sarà ubicato al limite destro, sempre della stessa sequenza (*low-order end*).

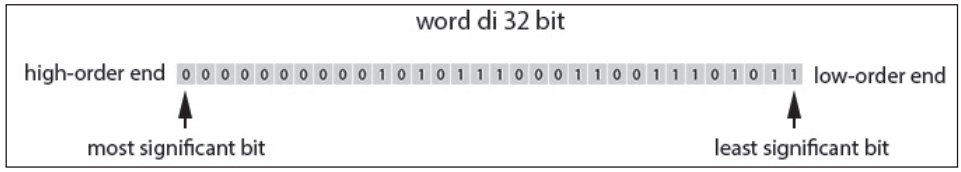

Figura 1.8 Una word come una sequenza di bit.

In definitiva, se due sistemi adottano questi due metodi di ordinamento in memoria dei byte e si devono scambiare dei dati, vi possono essere problemi di congruità tra i dati inviati da un sistema low endian verso un sistema big endian se non sono previsti appositi accorgimenti oppure se non sono forniti idonei meccanismi di conversione.
Per esempio, se un'applicazione scritta su un sistema SPARC memorizza dei dati binari in un file e poi lo stesso file è aperto in lettura su un sistema x86, si avranno problemi di congruità perché il file è stato scritto in modo *endian-dependent*.
Per evitare tale problema si possono adottare vari accorgimenti come, per esempio, quello di scrivere i dati in un formato *neutrale* che preveda file testuali e stringhe oppure adottare idonee routine di conversione (*byte swapping*) che, a seconda del sistema in uso, forniscano la corretta rappresentazione dell'informazione binaria.
In pratica, quando si deve scrivere software per diverse piattaforme hardware che hanno sistemi di *endianness* incompatibili, tale software deve essere sempre pensato in modo portabile, non presupponendo mai un particolare ordinamento in memoria dei byte.

Paradigmi di programmazione

Un *paradigma* o stile di programmazione indica un determinato modello concettuale e metodologico, offerto in termini concreti da un linguaggio di programmazione, al quale fa riferimento un programmatore per progettare e scrivere un programma informatico e dunque per risolvere un determinato problema algoritmico. Si conoscono molti differenti paradigmi di programmazione, ma quelli che seguono sono i più comuni.

Il paradigma procedurale

L'unità principale di programmazione è, per l'appunto, la procedura o la funzione, che ha lo scopo di manipolare i dati del programma. Questo paradigma è talune volte indicato anche come *imperativo*, perché consente di costruire un programma indicando dei comandi (assegna, chiama una procedura, esegui un loop e così via) che esplicitano quali azioni si devono eseguire, e in quale ordine, per risolvere un determinato compito. Questo paradigma si basa, dunque, su due aspetti di rilievo: il primo è riferito al cambiamento di stato del programma che è causa delle istruzioni eseguite (si pensi al cambiamento del valore di una variabile in un determinato tempo durante l'esecuzione del programma); il secondo è inerente allo stile di programmazione adottato, che è orientato al "come fare o come risolvere" piuttosto che al "cosa si desidera ottenere o cosa risolvere". Esempi di linguaggi che supportano il paradigma procedurale sono: FORTRAN, COBOL, Pascal, C e così via.

Il paradigma a oggetti

L'unità principale di programmazione è l'oggetto (nei sistemi basati sui prototipi) oppure la classe (nei sistemi basati sulle classi). Questi oggetti, definibili come *virtuali*, sono astrazioni concettuali degli oggetti reali, del mondo fisico, che intendono modellare. Questi ultimi possono essere oggetti più generali (per esempio un computer) oppure oggetti più specifici, maggiormente specializzati (per esempio una scheda madre, una scheda video e così via). Noi utilizziamo tali oggetti senza sapere nulla della complessità con cui sono costruiti e comunichiamo con essi attraverso messaggi (sposta il puntatore, digita dei caratteri) e interfacce (mouse, tastiera). Inoltre, essi sono dotati di attributi o caratteristiche (velocità del processore, colore del *case* e così via) che possono essere letti e, in alcuni casi, modificati. Questi oggetti reali vengono presi come modello per la costruzione di sistemi software a oggetti, dove l'oggetto (o la classe) avrà dei metodi per l'invio di messaggi e degli attributi che rappresenteranno i dati da manipolare. Principi fondamentali di tale paradigma sono i seguenti.

- *Incapsulamento*: un meccanismo attraverso il quale i dati e il codice di un oggetto sono protetti da accessi arbitrari (*information hiding*). Per dati e codice intendiamo tutti i membri di una classe, ovvero sia i membri dati (definiti anche, nel gergo della OOP, *Object Oriented Programming*, semplicemente come *campi*), sia i membri funzione (definiti anche, nel gergo della OOP, semplicemente come *metodi*). La protezione dell'accesso viene effettuata applicando ai membri della classe degli specificatori o modificatori di accesso, definibili come: *pubblico*, con cui si consente l'accesso a un membro di una classe da parte dei metodi di altre classi; *protetto*, con cui si consente l'accesso a un membro di una classe solo da parte di metodi appartenenti alle sue classi derivate; *privato*, con cui un membro di una classe non è accessibile né da metodi di altre classi né da quelli delle sue classi derivate, ma soltanto dai metodi della sua stessa classe.
- *Ereditarietà*: un meccanismo attraverso il quale una classe può avere relazioni di ereditarietà nei confronti di altre classi. Per relazione di ereditarietà intendiamo una relazione gerarchica di parentela *padre-figlio*, dove una classe figlio (definita *classe derivata* o *sottoclasse*) deriva da una classe padre (definita *classe base* o *superclasse*) i metodi e le proprietà pubbliche e protette, e dove essa stessa ne definisce di proprie. Con l'ereditarietà si può costruire, di fatto, un modello orientato agli oggetti che in principio è generico e minimale (ha solo classi base) e poi, man mano che se ne presenta l'esigenza, può essere esteso attraverso la creazione di sottomodelli sempre più specializzati (ha anche classi derivate).
- *Polimorfismo*: un meccanismo attraverso il quale si può scrivere codice in modo generico ed estendibile grazie al potente concetto che una classe base può riferirsi a tutte le sue classi derivate cambiando, di fatto, la sua *forma*. Ciò si traduce, in pratica, nella possibilità di assegnare a una variabile `A` (istanza di una classe base) il riferimento di una variabile `B` (istanza di una classe derivata da `A`) e, successivamente, riassegnare alla stessa variabile `A` il riferimento di una variabile `C` (istanza di un'altra classe derivata da `A`). La caratteristica appena indicata consentirà, attraverso il riferimento `A`, di invocare i metodi di `A` che `B` o `C` hanno ridefinito in modo specializzato, con la garanzia che il sistema di *runtime* del linguaggio di programmazione a oggetti saprà sempre a quale classe derivata appartengono. La discriminazione automatica, effettuata dal sistema di *runtime* di un tale linguaggio, di quale oggetto (istanza di una classe derivata) è

contenuto in una variabile (istanza di una classe base) è effettuata con un meccanismo definito *dynamic binding* (binding dinamico).

TERMINOLOGIA

Nel gergo della OOP, i membri di una classe sono chiamati in molteplici modi. Per i membri che rappresentato dei "dati" viene usata la seguente terminologia: campi, membri dati, membri di dati, dati membro, variabili membro. Per i membri che invece rappresentano "funzionalità" possono essere impiegati i seguenti termini: metodi, membri funzione, membri di funzioni, funzioni membro. Per quanto riguarda il presente testo utilizzeremo, in generale, i termini così come sono stati indicati nel documento di specifica del linguaggio Java, ossia campi (*field*) per i dati e metodi (*method*) per le funzionalità.

Esempi di linguaggi che supportano il paradigma a oggetti sono: Java, C#, C++, JavaScript, Smalltalk, Python e così via.

- *Paradigma funzionale*: l'unità principale di programmazione è la funzione vista in puro senso matematico. Infatti, il flusso esecutivo del codice è guidato da una serie di valutazioni di funzioni che, trasformando i dati che elaborano, conducono alla soluzione di un problema. Gli aspetti rilevanti di questo paradigma sono: nessuna mutabilità di stato (le funzioni sono *side-effect free*, ossia non modificano alcuna variabile); il programmatore non si deve preoccupare dei dettagli implementativi del "come" risolvere un problema, ma piuttosto di "cosa" si vuole ottenere dalla computazione. Esempi di linguaggi che supportano il paradigma funzionale sono: Lisp, Haskell, F#, Erlang e Clojure.
- *Paradigma logico*: l'unità principale di programmazione è il *predicato logico*. In pratica con questo paradigma il programmatore dichiara solo i "fatti" e le "proprietà" che descrivono il problema da risolvere, lasciando al sistema il compito di "inferirne" la soluzione e dunque raggiungerne il "goal" (l'obiettivo). Esempi di linguaggi che supportano il paradigma logico sono: Datalog, Mercury, Prolog, ROOP e così via.

Per esempio, un linguaggio come il C supporta pienamente il paradigma procedurale, dove l'unità principale di astrazione è rappresentata dalla funzione attraverso la quale si manipolano i dati di un programma. Da questo punto di vista, esso si differenzia dai linguaggi di programmazione che sposano il paradigma a oggetti come, per esempio, C#, C++ o Java, perché in quest'ultimo paradigma ci si concentra prima sulla creazione di nuovi tipi di dato (le classi) e poi sui membri funzione e i membri dati a essi relativi. In altre parole, mentre in un linguaggio procedurale come il C la modularità di un programma viene fondamentalmente descritta dalle procedure o funzioni che manipolano i dati, nella programmazione a oggetti la modularità viene descritta dalle classi che incapsulano al loro interno membri funzione e membri dati. Per questa ragione si suole dire che nel mondo a oggetti la dinamica (metodi) è subordinata alla struttura (classi).

NOTA

Il linguaggio Java supporta, principalmente, il paradigma di programmazione a oggetti, dove l'unità principale di astrazione è rappresentata dalla classe e dove vi è piena conformità con i principi fondamentali di tale paradigma: incapsulamento, ereditarietà e polimorfismo. In ogni caso Java è considerabile un linguaggio di programmazione *multiparadigma* poiché, di fatto, supporta anche quello procedurale e, a partire dalla

versione 8, ha iniziato a supportare, seppure limitatamente all'introduzione delle *lambda expression*, anche quello funzionale.

TERMINOLOGIA

Nei linguaggi di programmazione si usano termini come funzione (*function*), metodo (*method*), procedura (*procedure*), sottoprogramma (*subprogram*), sottoroutine (*subroutine*) e così via per indicare un blocco di codice posto a un determinato indirizzo di memoria che è invocabile, richiamabile, per eseguire le istruzioni che vi si trovano. Dal punto di vista pratico, pertanto, significano tutti la stessa cosa anche se, in letteratura, talune volte sono evidenziate delle differenze soprattutto in base al linguaggio di programmazione che si prende in esame. Per esempio: in Pascal una funzione restituisce un valore mentre una procedura non restituisce nulla; in C una funzione può agire anche come una procedura, mentre il termine "metodo" non esiste; in Java un metodo è una funzionalità "associata" all'oggetto o alla classe in cui è stato definito.

Concetti introduttivi allo sviluppo in Java

Prima di affrontare lo studio sistematico della programmazione in Java e di descrivere un semplice programma introduttivo, appare opportuno soffermarsi su alcuni concetti propedeutici allo sviluppo Java che sono trasversali al linguaggio e che servono a inquadrarlo meglio nel suo complesso.

Fasi di sviluppo di un programma in Java

Un programma scritto in Java, prima di poter essere eseguito, passa attraverso le seguenti fasi, che ne definiscono un generico ciclo operativo.

- *Analisi*. Questa è la fase che sottende all'individuazione delle informazioni preliminari allo sviluppo di un software, le quali possono riguardare la sua fattibilità in senso tecnico ed economico (analisi costi/benefici), il suo dominio applicativo, i suoi requisiti funzionali (cosa il software deve offrire) e così via.
- *Progettazione*. In questa fase si inizia a ideare in modo più concreto come si può sviluppare il software che è stato oggetto della precedente analisi. In pratica il software viene scomposto in moduli e componenti e si definisce la loro interazione e anche il loro contenuto (dettaglio interno). La progettazione indica, pertanto, *come* il software deve essere implementato piuttosto di *cosa* deve fare il software (appannaggio della fase di analisi).

TERMINOLOGIA

La scomposizione di un programma in moduli o componenti segue tipicamente una logica *divide et impera* (dividi e domina o dividi e conquista). In sostanza, un problema viene diviso in vari sotto-problemi i quali sono a loro volta divisi in altri sotto-problemi e questa divisione continua finché i più piccoli sotto-problemi non sono in grado di ottenere la relativa soluzione nel modo più semplice possibile. Dopodiché, tutte le soluzioni semplici trovate vengono combinate per produrre la soluzione generale del problema. Questo approccio alla soluzione dei problemi è anche detto *top-down* (dall'alto verso il basso) perché si parte

dal formulare un problema generale e in modo poco dettagliato (posto, cioè, in cima a una piramide ideale) e poi lo si decompone, rifinisce (*stepwise refinement*), in sotto-problemi più specializzati e maggiormente dettagliati (posti, cioè, alla base della piramide ideale).

- *Codifica*. Questa è la fase in cui si implementa concretamente il software oggetto della progettazione. In pratica, attraverso l'utilizzo di un editor di testo, si scrivono gli algoritmi, le funzionalità e le istruzioni del programma codificate secondo la sintassi propria del linguaggio Java. Il codice scritto nell'editor è detto codice sorgente (*source code*), e il file prodotto è un file di codice sorgente (*source code file*).
- *Compilazione*. Questa è la fase in cui un apposito programma, il compilatore (*compiler*), traduce, converte, il codice sorgente nel relativo file in codice intermedio (*intermediate language code*), ovvero in codice non direttamente eseguibile nel sistema target, ma comprensibile e interpretabile solo dal sistema di *runtime* (*virtual machine*), laddove, poi, un apposito compilatore Just-In-Time (JIT) lo compila nel codice nativo (*machine code*) del sistema. Questo codice intermedio viene scritto in un apposito file (*intermediate language code file*). Inoltre, in questa fase, il compilatore si occupa anche di verificare che il codice sorgente sia scevro da errori sintattici che violerebbero le regole di Java; in caso contrario, avvisa l'utente della presenza di tali errori, interrompe la compilazione e non procede alla generazione del codice intermedio.
- *Esecuzione*. Questa è la fase in cui il file contenente il codice intermedio viene caricato nella memoria, convertito in codice nativo per la specifica piattaforma ed eseguito, ovvero produce gli effetti computazionali per i quali è stato progettato e sviluppato. In linea generale, in una shell testuale, per caricare in memoria ed eseguire un programma scritto in Java è sufficiente digitare nel relativo ambiente di esecuzione il nome del file di codice intermedio preceduto dal comando `java`.
- *Test e debug*. Questa è la fase in cui si verifica la correttezza funzionale del programma in esecuzione, ovvero se presenta errori o comportamenti non pertinenti con la logica che avrebbe dovuto seguire. A tal fine esistono appositi programmi, chiamati *debugger*, che consentono di analizzare il flusso di esecuzione di un programma *step by step*, fermare la sua esecuzione al raggiungimento di un determinato *breakpoint* per esaminarne lo stato corrente, rilevare il valore delle variabili in un certo momento e così via.
- *Mantenimento*. Questa è la fase in cui un programma che è in produzione, a seguito di richieste o osservazioni degli utilizzatori, può subire modifiche migliorative, per esempio in termini di prestazioni, oppure modifiche integrative che riguardano l'aggiunta di moduli che offrono funzionalità supplementari rispetto a quelle previste in origine.

Elementi di un ambiente Java

Per poter programmare in Java è necessario scaricare un JDK e installarlo nel sistema operativo in uso (un dettaglio si trova consultando l'apposita guida reperibile all'indirizzo `https://github.com/thp1972/Libro_Java21`, link `Installazione Java`).
Dopo l'installazione del JDK avremo a disposizione tutti gli strumenti per compilare ed eseguire i programmi creati, incluse svariate librerie di classi dette API (*Application Programming Interface*). La Figura 1.9 riporta un esempio di struttura di directory e file creata da un'installazione tipica.

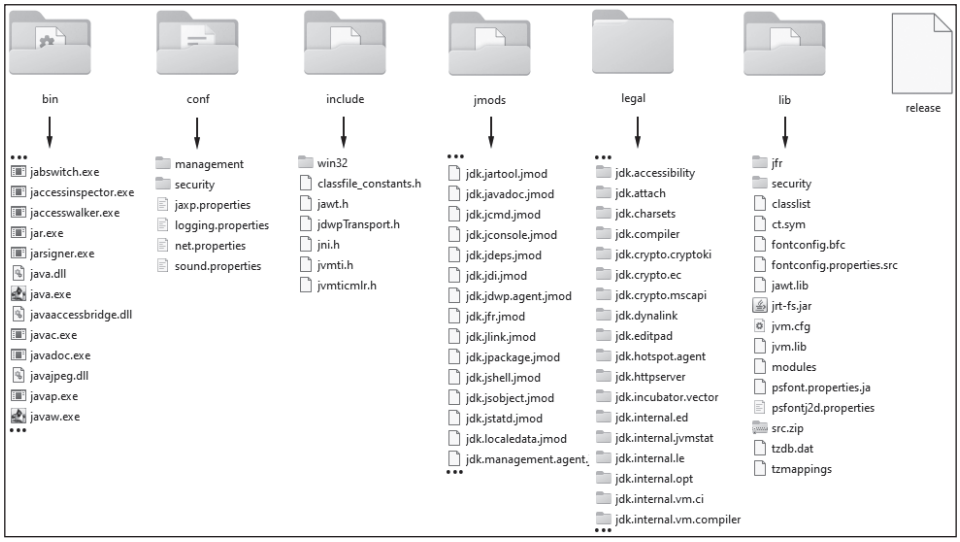

Figura 1.9 Struttura ad albero del JDK 21 a 64 bit creata in un sistema Windows.

In pratica, a partire dalla directory radice del JDK 21 avremo varie cartelle.

- Una cartella di nome `bin` che contiene tutti i file eseguibili dei tool di sviluppo di Java come `javac.exe` (il compilatore dal codice sorgente Java in codice intermedio bytecode), `java.exe` (il *launcher* di un'applicazione Java), `jar.exe` (il gestore per file di archivi basati sul formato Java ARchive o JAR), `javap.exe` (il disassemblatore di file `.class`) e così via.
- Una cartella di nome `conf` contenente file `.properties`, `.policy` e così via utilizzabili per editare appositi *file di configurazioni* (per esempio, il file `net.properties` è impiegabile per editare delle impostazioni per un proxy di rete).
- Una cartella di nome `include` che contiene file header in linguaggio C/C++ per l'integrazione di codice nativo scritto con i predetti linguaggi con il codice scritto con il linguaggio Java. Questa integrazione avviene mediante l'utilizzo della *Java Native Interface* (JNI) e della *Java Virtual Machine Tools Interface* (JVMTI).

> **ATTENZIONE**
>
> Il termine *codice nativo*, in questo specifico contesto, è riferito al codice scritto nativamente in C o C++ i cui compilatori producono *applicazioni native*, ossia applicazioni che girano nativamente per una piattaforma hardware e un sistema operativo specifici (per un determinato *host environment*). Tuttavia, in contesti più generali, con il termine *codice nativo* si intende anche il cosiddetto *machine language code*, che è prodotto, per esempio, da un compilatore C, per una specifica piattaforma di esecuzione e dunque da essa dipendente. Il compilatore Java, invece, nel momento in cui compila codice sorgente Java, produce un codice in linguaggio "intermedio" (detto *bytecode*) che è interpretabile ed eseguibile da qualsiasi macchina virtuale Java (JVM, *Java Virtual Machine*) e quindi risulta indipendente dalla specifica piattaforma di destinazione.

- Una cartella di nome `jmods` contenente i *moduli* del JDK corrente e in formato JMOD (estensione `.jmod`). Precisiamo che i moduli e il relativo progetto *Jigsaw*, introdotto dalla versione 9 di Java, sarà esaminato in dettaglio nel Capitolo 13, *Moduli*.
- Una cartella di nome `legal` contenente per ogni modulo una relativa sottocartella (per esempio, `java.sql`) al cui interno sono presenti informazioni legali, come le licenze e i termini di utilizzo associati al modulo relativo (per esempio il file `LICENSE`).
- Una cartella di nome `lib` che contiene file che rappresentano i dettagli implementativi del sistema di *runtime*. È importante dire che i file di classi e risorse memorizzati negli "storici" file che si trovavano nella medesima directory fino a Java 8 come, per esempio, `rt.jar`, `tools.jar` e `dt.jar` sono stati ora memorizzati sempre nella stessa directory ma in file più specifici e con un formato più efficiente (sono stati dunque eliminati dalla directory `lib` i citati `.jar`).
- Un file di nome `release` che contiene varie informazioni di servizio codificate come coppie di chiave/valore (per esempio, `IMPLEMENTOR="Oracle Corporation"`, `JAVA_VERSION="21.0.2"`, `OS_ARCH="x86_64"`, `OS_NAME="Windows"`, `OS_ARCH="x86_64"` e così via).

NOTA

Fino alla versione 9 di Java, nella root directory dell'installazione, era anche presente un file di archivio basato sul formato ZIP di nome `src.zip` che conteneva il codice sorgente di tutte quelli classi che rappresentano il core delle API di Java (per esempio quelle che si trovano nei package `java.*`, `javax.*` e così via). A partire da Java 10, invece, la predetta directory è stata spostata nella directory `lib`.

Ambiente di sviluppo di Java: terminologia essenziale

Quando si parla, in linea generale, di *ambiente di sviluppo di Java*, è opportuno comprendere i seguenti termini e sapere pertanto differenziarli in modo opportuno.

- *Java Development Kit* (JDK): rappresenta quell'ambiente software che comprende tool e utility per sviluppare ed eseguire le applicazioni scritte per Java (`java`, `javac`, `javap`, `jdb`, `jar` e così via).
- *Java Runtime Environment* (JRE): rappresenta quell'ambiente software entro il quale i programmi Java sono eseguiti e comprende una Java Virtual Machine e le librerie software di Java.
- *Java Virtual Machine* (JVM): rappresenta quell'ambiente software che permette di interpretare ed eseguire il bytecode proprio di Java. L'implementazione di riferimento della JVM per Java è quella prodotta da Oracle stessa ed è definita come *Java HotSpot Virtual Machine*, ma ce ne sono anche altre come, per esempio, OpenJ9 prodotta da Eclipse Foundation.

NOTA

Fino a Java 8, il JDK conteneva anche un JRE.

Il primo programma in Java

Vediamo, attraverso la disamina del Listato 1.1, gli elementi basilari per strutturare e scrivere un programma Java. Le funzionalità impiegate saranno poi trattate con dovizia di particolari nei capitoli di pertinenza. In questa fase, abbiamo solo inteso illustrare

la struttura di massima degli elementi costitutivi di un programma in Java e fornire la terminologia di base applicabile ai principali costrutti del linguaggio.

Listato 1.1 (FirstProgram.java).

```
package LibroJava.Capitolo1;

/*
Primo programma in Java
 */
public class FirstProgram
{
    private static int counter = 10;
    private static final int multiplicand = 10;
    private static final int multiplier = 20;

    public static void main(String[] args)
    {
        // dichiarazione e inizializzazione contestuale
        // di più variabili di diverso tipo
        String text_1 = "Primo programma in Java:",
               text_2 = " Buon divertimento!";
        int a = 10, b = 20;

        float f; // dichiarazione
        f = 44.5f; // inizializzazione

        // stampa qualcosa...
        System.out.printf("%s%s%n", text_1, text_2);
        System.out.printf("Stamperò un test condizionale tra a=%d e b=%d%n", a, b);

        if (a < b) // se a < b stampa quello che segue...
        {
            System.out.print("a < b VERO!");
        }
        else /* altrimenti stampa quest'altra stringa */
        {
            System.out.print("a > b VERO!");
        }

        System.out.print("\nStamperò un ciclo iterativo, dove leggerò ");
        System.out.println("per 10 volte il valore di a");

        /*
         * ciclo for
         */
        for (int i = 0; i < 10; i++)
        {
            System.out.printf("Passo %d ", i);
            System.out.printf("--> a=%d%n", a);
        }

        /*
            // esecuzione della moltiplicazione
        */
        System.out.printf("Ora eseguirò una moltiplicazione tra %d e %d%n",
                          multiplicand, multiplier);
```

```
        int res = mult(multiplicand, multiplier); // invocazione di un metodo
        System.out.printf("Il risultato di %d x %d è: %d%n", multiplicand, multiplier, res);
    }

    /**************************************
    * Metodo: mult                        *
    * Scopo: moltiplicazione di due valori *
    * Parametri: a, b -> int              *
    * Restituisce: int                    *
    **************************************/
    private static int mult(int a, int b)
    {
        return a * b;
    }
}
```

Il Listato 1.1 inizia con l'istruzione `package`, che consente di creare librerie di tipi correlati. Nel nostro caso la classe denominata `FirstProgram`, definita con la keyword `class`, è un nuovo tipo di dato che apparterrà al package (o libreria) denominato `LibroJava.Capitolo1`. Da questo punto di vista, quindi, un package può essere visto come una sorta di "contenitore" di tipi di dato e infatti possiamo dire che la classe `FirstProgram` è "contenuta" nel package `LibroJava.Capitolo1`.
Notiamo poi la definizione di un'istruzione di commento: tra il carattere di inizio commento `/*` e il carattere di fine commento `*/` viene specificato del testo che verrà ignorato dal compilatore e che serve a documentare o chiarire specifiche parti del codice sorgente (questo tipo di commento è simile a quello usato per i linguaggi C e C++).
Il testo di questo tipo di *commento* può anche essere suddiviso su più righe e, infatti, per tale ragione è spesso anche definito come commento *multiriga* (*multiline comment*) oppure, in accordo con la specifica terminologia del linguaggio Java, come commento *tradizionale* (*traditional comment*).
In ogni caso non è possibile annidare commenti multiriga all'interno di altri commenti (Snippet 1.1) e ciò perché il marker di chiusura `*/` del commento annidato finisce per "chiudere" il marker di commento iniziale `/*`. In questo modo, quindi, il secondo marker di chiusura commento `*/` rimarrebbe senza un marker di apertura commento con cui "accoppiarsi", inducendo il compilatore a generare un errore del tipo: `illegal start of expression */`.

Snippet 1.1 (Snippet_1.1.java) Commenti multiriga annidati.

```
...
public class Snippet_1_1
{
    public static void main(String[] args)
    {
        /* // commento che annida
            AAAA
            /* // commento annidato
                BBB
            */
        CCCC
        */
    }
}
```

Infine, i commenti multiriga possono essere anche scritti con diversi stili di preferenza, per esempio: di fianco, a lato di una porzione di codice (*winged comment*), come mostra quello definito dopo l'istruzione `else`; in forma di riquadro di contenimento (*boxed comment*), come mostra quello scritto prima della definizione del metodo `mult`; scrivendo degli asterischi `*` per ogni riga di separazione, come mostra quello posto prima del ciclo `for`.
Quindi, il programmatore è libero di scegliere per i commenti la formattazione che preferisce, a condizione, però, che vi sia sempre una corrispondenza tra il marcatore di apertura commento `/*` e il marcatore di chiusura commento `*/`.
In Java si può comunque utilizzare anche un secondo tipo di commento, simile a quello usato per il linguaggio C++, che è indicato tramite l'utilizzo di due caratteri slash `//` cui si fa seguire il testo di commento che sarà ignorato dal compilatore.
Questo commento, poiché termina automaticamente alla fine della riga, è spesso definito commento a *singola riga* (*single-line-comment*), oppure, in accordo con la specifica di Java, anche come *end-of-line comment*. Ha la caratteristica che può essere anche annidato all'interno di un commento multiriga, come mostra in modo evidente il commento `// esecuzione della moltiplicazione` posto prima dell'invocazione del metodo `printf`.
In più, anche con questo tipo di commento, è possibile scrivere commenti su più righe semplicemente scrivendo su ogni riga i caratteri `//` e la porzione di testo di commento.
Un esempio di quanto detto è visibile nei primi due commenti, posti subito dopo la parentesi graffa di apertura del metodo `main`.

IMPORTANTE

Nel Capitolo 15, *Documentazione del codice sorgente*, vedremo che è anche possibile usare un'altra tipologia di commenti, definiti come *documentation comments* (commenti di documentazione), che consentono di documentare il codice sorgente scrivendo, tra i caratteri `/**` e `*/`, semplice testo, tag HTML e tag specifici riconosciuti solo dal tool `javadoc`.

Dopo il commento multiriga che segue la dichiarazione del package abbiamo la definizione di una *classe*, effettuata tramite la keyword `class`; la classe è denominata `FirstProgram` e tra le parentesi graffe aperta e chiusa sono indicati i suoi membri, ossia i suoi campi (membri dati) e suoi metodi (membri funzione).
Una classe, ricordiamo, è il *blocco costitutivo* di un qualsiasi programma che segua il paradigma della programmazione orientata agli oggetti; essa definisce una sorta di modello, di tipo, per delle "entità", definite come *oggetti*, che ne rappresentano specifiche e concrete *istanze* o *esempi*; possiamo così dire che mentre, per esempio, una classe `Car` può rappresentare un modello che astrae una generica "automobile", un oggetto `ford` può rappresentare un'istanza concreta, un esempio specifico di quella generica automobile.
La nostra classe `FirstProgram` ha dunque tre campi, denominati `counter`, `multiplicand` e `multiplier`, e due metodi, denominati `main` e `mult`.
Il campo `counter` è una *variabile*, ovvero una locazione di memoria che può contenere valori che possono cambiare nel tempo. A ogni variabile deve essere associato l'insieme di valori che può contenere, tramite la specificazione di un tipo di dato di appartenenza. La variabile `counter` ha infatti associato, tramite la keyword `int`, un tipo di dato intero, ovvero potrà solo contenere valori propri del sottoinsieme matematico dei numeri interi (per esempio, `-220`, `45600` e così via).
I campi `multiplicand` e `multiplier` rappresentano, invece, delle *costanti* (keyword `final`) ossia delle locazioni di memoria che contengono valori che non possono cambiare nel tempo; in pratica, dopo aver assegnato un valore a una costante, questa potrà essere usata

nell'ambito del programma solo in modalità "lettura" (per ottenerne il valore) ma non in modalità "scrittura" (per alterarne il valore).
Il metodo `main`, invece, è il *metodo* principale di un qualsiasi programma Java, laddove per metodo si intende un blocco di codice contenente una o più istruzioni che rappresentano una funzionalità ed eseguono un compito specifico; un metodo `main` deve essere sempre presente nell'ambito di un programma, perché ne rappresenta l'*entry point*, ossia il "punto di ingresso" attraverso il quale il programma viene eseguito.
In pratica il metodo `main` viene invocato automaticamente dall'ambiente di esecuzione (dalla JVM, *Java Virtual Machine*) quando il programma viene avviato (*application startup*); poi attraverso il metodo `main` vengono eseguite le istruzioni in esso contenute e invocati gli altri metodi indicati; da questo punto di vista, possiamo asserire che un programma in Java "deve" essere sempre composto da almeno una classe che contiene il metodo `main` e "può" definire o utilizzare una o più classi ausiliarie. Quindi non è altro che una collezione di una o più classi, tra di loro interagenti. Questo fondamentale metodo può essere definito utilizzando una delle due seguenti modalità (Snippet 1.2 e 1.3).

Snippet 1.2 (Snippet_1_2.java) Modalità di definizione di main; I definizione.

```
...
public class Snippet_1_2
{
    // args è espresso come array di String
    public static void main(String[] args) { /* ... */ }
}
```

Snippet 1.3 Snippet_1_3.java) Modalità di definizione di main; II definizione.

```
...
public class Snippet_1_3
{
    // args è espresso come parametro a lunghezza variabile
    public static void main(String... args) { /* ... */ }
}
```

In pratica, il metodo `main` deve essere definito nel seguente modo.

- Con il *modificatore di accesso* (*access modifier*) `public`, il quale indica che il metodo è accessibile da client esterni alla classe in cui il metodo stesso è stato definito.
- Con il *modificatore* `static`, il quale stabilisce che il metodo è un *membro statico di una classe*, piuttosto che un *membro di istanza di un oggetto*, e dunque può essere invocato senza creare il relativo oggetto; questa keyword è importante, perché permette al metodo `main` di essere invocato direttamente dall'ambiente di esecuzione (la JVM) e di avviarne il relativo programma.

  NOTA

  In definitiva, le keyword `static` e `public` permettono al metodo `main` di essere invocato pubblicamente dalla JVM in quanto *client esterno* e di avviarne il relativo programma.

- Con la keyword `void`, la quale indica che il metodo non restituisce alcun valore. Ovviamente un metodo può restituire un valore e in questo caso occorre indicarne il tipo di appartenenza, per esempio `int` per la restituzione di un tipo intero.

- Con una coppia di parentesi tonde `(...)`, all'interno delle quali è posta una variabile denominata `args` che rappresenta il *parametro* del metodo. Ogni metodo, infatti, può avere zero o più parametri che rappresentano, se presenti, delle variabili che saranno riempite con valori (detti *argomenti*) passati al metodo all'atto della sua invocazione. I parametri di un metodo devono avere, inoltre, un tipo di dato associato; infatti, il parametro `args` è dichiarato come di tipo array di stringhe (`String[]` o `String...`). Il parametro `args` permette dunque al metodo `main` di ottenere in input gli argomenti eventualmente passati quando si invoca dalla riga di comando il programma che lo contiene.

> **NOTA**
>
> Il nome del parametro formale di `main` è arbitrario. È pertanto possibile utilizzare un nome diverso da `args`, ma il suo tipo deve sempre essere espresso come `String[]` oppure `String...` laddove quest'ultima modalità di scrittura di un tipo di un parametro è relativa alla dichiarazione di metodi che possono accettare *argomenti di lunghezza variabile*, come studieremo in dettaglio nel Capitolo 6, *Metodi*.

- Con una coppia di parentesi graffe `{...}`, all'interno delle quali sono scritte delle istruzioni che nel loro complesso rappresentano le operazioni che il metodo deve eseguire.

Abbiamo poi la definizione del metodo `mult`, la quale evidenzia come esso restituisca un tipo di dato intero e accetti due argomenti sempre di tipo intero. Nell'ambito del metodo `main` notiamo, infatti, come il metodo `mult` sia invocato con due argomenti di tipo intero (i valori `10` e `20`) e restituisca un valore di tipo intero assegnato alla variabile `res`. Lo stesso metodo `mult` ha, infatti, una sua implementazione algoritmica che evidenzia come restituisca un valore di tipo intero che è il risultato della moltiplicazione tra i parametri `a` e `b`, sempre di tipo intero. Per quanto attiene al contenuto del metodo `main`, notiamo subito una serie di istruzioni che definiscono delle variabili che rappresentano locazioni di memoria modificabili, deputate a contenere un valore di un determinato tipo di dato. Così gli identificatori `text_1` e `text_2` indicano variabili che possono contenere caratteri; gli identificatori `a`, `b` e `res` indicano variabili che possono contenere numeri interi; l'identificatore `f` indica una variabile che può contenere numeri decimali, ossia numeri formati da una parte intera e una parte frazionaria separati da un determinato carattere (per Java tale carattere è il punto).
Una variabile, in linea generale, può essere prima dichiarata e poi inizializzata (è il caso della variabile `f`) oppure può essere dichiarata e inizializzata contestualmente in un'unica istruzione (è il caso delle altre variabili).
A parte le varie istruzioni di stampa su console dei valori espressi dalle rispettive stringhe dei metodi `printf`, `print` e `println` del tipo `PrintStream` notiamo: l'impiego di un'istruzione di selezione doppia `if/else` che valuta se una data espressione è vera o falsa, eseguendone, a seconda del risultato, il codice corrispondente; l'impiego di un'istruzione di iterazione `for` che consente di eseguire ciclicamente una serie di istruzioni finché una data espressione è vera.
Pertanto, l'istruzione `if/else` valuta se il valore della variabile `a` è minore del valore della variabile `b` e, nel caso, stampa la relativa stringa; altrimenti, in caso di valutazione falsa, stampa l'altra stringa. L'istruzione `for`, invece, stampa su console per 10 volte informazioni sul valore delle variabili `i` e `a`.

NOTA

In un linguaggio di programmazione a oggetti, come Java, ogni metodo "appartiene" alla classe nella quale è stato definito e pertanto per poterlo invocare è sempre necessario usare una particolare sintassi che prevede: se è di tipo metodo di classe, l'uso del nome della sua classe, l'operatore punto e il suo nome; se è di tipo metodo di istanza, l'uso del nome dell'oggetto istanza della sua classe, l'operatore punto e il suo nome.

Infine, concludiamo con alcune brevi indicazioni:

- ogni istruzione deve terminare con il carattere ; (punto e virgola);
- le parentesi graffe aperta { e chiusa } delimitano un blocco di codice contenente delle istruzioni;
- il codice può essere scritto secondo il proprio personale stile di indentazione, utilizzando i caratteri di spaziatura (*spazio*, *tabulazione*, *invio* e così via) desiderati.

Un "assaggio" dei metodi printf, println e print

Il metodo `printf`, definito nella classe `PrintStream`, la quale è definita nel package `java.io` esportato dal modulo `java.base`, consente di scrivere (visualizzare) una *stringa formattata* nel corrente *stream di output* che, tipicamente, è rappresentato dallo *stream di standard output*, il quale è associato a una destinazione di output convenzionale che in linea generale è uno schermo o monitor (ossia un *display device*).

TERMINOLOGIA

È comune usare anche il termine *stampare* per indicare la procedura con cui il metodo `printf` inoltra la relativa stringa al corrente stream di output (lo stesso vale anche per i metodi `println` e `print`).

In Java lo stream di standard output è aperto in automatico dalla Java Virtual Machine quando un programma è avviato per la sua esecuzione, ed è poi dalla stessa "associato" al campo `out` di tipo `PrintWriter` definito nella classe `System` (package `java.lang`, modulo `java.base`).

DETTAGLIO

La dichiarazione completa del campo `out` è la seguente: `public static final PrintStream out`. Esso rappresenta, cioè, una costante (`final`) di classe (`static`) con visibilità pubblica (`public`).

Esistono due definizioni in *overloading* del metodo `printf`, ma per ora accenneremo solo quella con la segnatura `public PrintStream printf(String format, Object... args)`, che consente, di fatto, di specificare i seguenti argomenti.

- Come primo argomento, un letterale stringa definito come *stringa di formato* (*format string*). Questa stringa di formato è tipicamente costituita da zero o più istanze di *testo fisso* (*fixed test*), alternate a uno o più *specificatori di formato* (*format specifier*) scritti secondo una particolare sintassi che fa uso, nella sua forma semplificata, del simbolo `%` (percento) seguito da un altro carattere che ne indica la *modalità di conversione*. Questi specificatori di formato rappresentano in sostanza una sorta di *segnaposti* (*placeholder*) ossia "locazioni" all'interno della stringa di formato il cui contenuto sarà sostituito, a

runtime, da un valore che è una rappresentazione *convertita* e *formattata* di tipo stringa del relativo oggetto passato al metodo come argomento successivo.

- Come argomenti successivi, un *elenco di oggetti*.

Così nel caso dell'istruzione `System.out.printf("Stamperò un test condizionale tra a=%d e b=%d%n", a, b);` del Listato 1.1 possiamo dire che l'argomento `"Stamperò un test condizionale tra a=%d e b=%d%n"` rappresenta la stringa di formato, mentre `a` e `b` rappresentano gli argomenti successivi, ossia l'elenco di oggetti da formattare.
Nell'ambito della stringa di formato avremo che: `Stamperò un test condizionale tra a=` e `b=` rappresenta il testo fisso; `%d` e `%n` rappresentano gli specificatori di formato, laddove il primo `%d` è il segnaposto per l'argomento `a`, il secondo `%d` è il segnaposto per l'argomento `b` e `%n` indica semplicemente il *separatore di riga* della specifica piattaforma.
Così, a *runtime*, la stringa `"Stamperò un test condizionale tra a=%d e b=%d%n"` sarà visualizzata nel seguente modo unitamente all'invio di un separatore di riga: `Stamperò un test condizionale tra a=10 e b=20` dove appare evidente come il primo segnaposto `%d` sia stato sostituito dalla rappresentazione stringa del valore intero proprio della variabile `a` (il primo oggetto dell'elenco) mentre il secondo segnaposto `%d` sia stato sostituito dalla rappresentazione stringa del valore intero proprio della variabile `b` (il secondo oggetto dell'elenco). In questo contesto, infatti, il carattere `d` che segue il simbolo `%` sta a indicare che l'argomento corrispondente deve essere formattato come un numero intero in base 10.

IMPORTANTE

Lo specificatore di formato `%n` restituisce un separatore di riga indipendente dal corrente sistema in uso, così come restituito dall'invocazione del metodo `System.lineSeparator()`. Per esempio, in un sistema Unix restituirà il carattere `\n` (*line feed*), mentre in un sistema Windows restituirà i caratteri `\r\n` (*carriage return* più *line feed*). Quindi, per ragioni di portabilità tra sistemi differenti, consigliamo di non usare nella stringa di formato direttamente il carattere `\n` come indicatore del separatore di riga.

TERMINOLOGIA

L'*overloading* (sovraccarico), trattato in modo dettagliato nel Capitolo 6, *Metodi*, è una caratteristica implementativa che consente di definire metodi con lo stesso nome ma con una diversa *segnatura*. Per segnatura di un metodo si intende quell'insieme di informazioni che lo identificano univocamente fra gli altri metodi della sua stessa classe di appartenenza. Tali informazioni includono il suo nome, il numero, il tipo e l'ordine dei suoi parametri e mai il tipo del valore da esso restituito.

Il metodo `println`, definito nella classe `PrintStream`, la quale è definita nel package `java.io` esportato dal modulo `java.base`, consente di scrivere (visualizzare) una rappresentazione stringa, nel corrente stream di output (tipicamente, lo stream di standard output), dell'argomento fornitogli, che può essere di tipo booleano (`boolean`), intero (`int`), in virgola mobile (`float`) e così via. In più genera anche un carattere di separatore di riga.

Tabella 1.2 Definizioni in overloading del metodo println.

Segnatura	Semantica
`public void println()`	Scrive il corrente separatore di riga.
`public void println(boolean x)`	Scrive un valore booleano e un separatore di riga.

`public void println(char x)`	Scrive un carattere e un separatore di riga.
`public void println(int x)`	Scrive un numero intero di 32 bit e un separatore di riga.
`public void println(long x)`	Scrive un numero intero di 64 bit e un separatore di riga.
`public void println(float x)`	Scrive un numero in virgola mobile di 32 bit e un separatore di riga.
`public void println(double x)`	Scrive un numero in virgola mobile di 64 bit e un separatore di riga.
`public void println(char[] x)`	Scrive un array di caratteri[1] e un separatore di riga.
`public void println(String x)`	Scrive una stringa e un separatore di riga.
`public void println(Object x)`	Scrive un oggetto[2] e un separatore di riga.

[1] Dato un array di caratteri ne scrive in sequenza i relativi caratteri (gli array saranno trattati nel Capitolo 3, *Array*).

[2] Dato un oggetto ne scrive la rappresentazione di tipo stringa così come implementata dal relativo metodo `toString` (il metodo `toString` sarà trattato nel Capitolo 7, *Programmazione basata sugli oggetti*).

Nel caso, per esempio, dell'istruzione `System.out.println("per 10 volte il valore di a")`, verrà stampata su schermo la stringa `"per 10 volte il valore di a"` unitamente a un separatore di riga. Infine, il metodo `print`, definito anch'esso nella classe `PrintStream` (package `java.io`, modulo `java.base`), si comporta allo stesso modo del metodo `println` appena esaminato, ma vi differisce perché non genera alcun separatore di riga (Tabella 1.3).

Tabella 1.3 Definizioni in overloading del metodo print.

Segnatura	Semantica
`public void print(boolean b)`	Scrive un valore booleano.
`public void print(char c)`	Scrive un carattere.
`public void print(int i)`	Scrive un numero intero di 32 bit.
`public void print(long l)`	Scrive un numero intero di 64 bit.
`public void print(float f)`	Scrive un numero in virgola mobile di 32 bit.
`public void print(double d)`	Scrive un numero in virgola mobile di 64 bit.
`public void print(char[] s)`	Scrive un array di caratteri.
`public void print(String s)`	Scrive una stringa.
`public void print(Object obj)`	Scrive un oggetto.

Per esempio, l'istruzione `System.out.print("a < b VERO!")` stamperà su schermo la stringa `"a < b VERO!"` senza, però, alcun separatore di riga.
In definitiva, la discriminante sull'utilizzo del metodo `printf` rispetto al metodo `println` (o `print`) risiede nel porsi la domanda se si desidera avere una totale flessibilità su come formattare l'output di una stringa (è il caso di `printf`) oppure se si è soddisfatti della formattazione di default (è il caso di `println` o `print`).

Elementi strutturali di un programma in Java

Un programma in Java è costituito da uno o più file di codice sorgente definiti in modo rigoroso e formale come un'unità di compilazione (*compilation unit*), la quale è costituita, in linea generale e a un più alto livello, dalle seguenti parti fondamentali:

- una dichiarazione di *package*, che fornisce un nome completamente qualificato (*fully qualified name*) del package cui la predetta unità di compilazione appartiene;

- delle dichiarazioni di *importazione*, che consentono di far riferimento ai tipi appartenenti ad altri package e ai membri statici dei tipi impiegando direttamente il loro nome semplice (*simple name*);
- delle dichiarazioni, *top level*, dei tipi (classi e interfacce).

Ogni unità di compilazione è dunque modellata, idealmente, in più parti strutturali e semantiche che sono combinate insieme con lo scopo di formare un programma.
Abbiamo quattro elementi di base che formano la struttura lessicale di un file sorgente Java: terminatori di riga, spazi, commenti e token; gli spazi, i commenti e i token sono definiti come elementi di input (*input element*) dell'unità di compilazione. Degli elementi indicati, solo i token hanno un impatto significativo sulla struttura sintattica di un programma Java; infatti, gli spazi e i commenti sono impiegati solo per separare i token. Formano i token gli identificatori, le keyword, i letterali (*integer*, *floating-point*, *boolean*, *character*, *string* e *null*), gli operatori e i segni di punteggiatura.
Per Java il numero di caratteri di spaziatura (spazi, tabulazione e così via) impiegati come separatori tra i token è libero: ogni programmatore può scegliere quello che preferisce, secondo il suo personale stile di scrittura. Tuttavia un token non può essere "diviso" senza causare un errore e cambiarne la semantica. Lo stesso vale per un letterale stringa, con il seguente distinguo: al suo interno è sempre possibile inserire caratteri di spazio, ma è un errore separarlo all'interno dell'editor su più righe premendo il tasto Invio.
A partire dagli elementi lessicali significativi si costruiscono, principalmente, le *statement*, ossia le istruzioni proprie di un programma Java, che sono rappresentate dalle dichiarazioni, dalle espressioni, dalle selezioni, dalle iterazioni e così via (Figura 1.10).

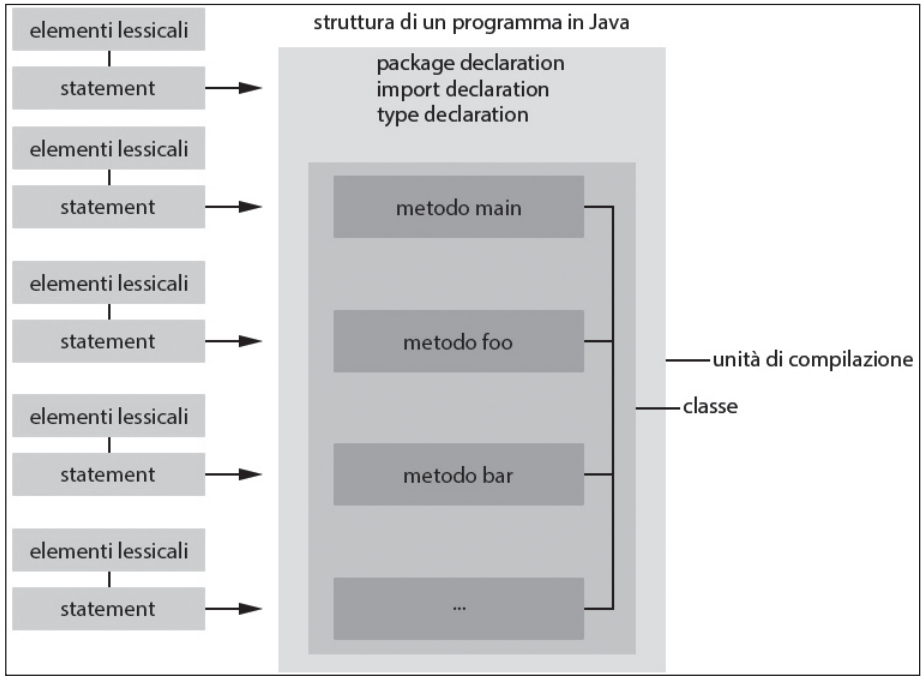

Figura 1.10 Costituzione strutturale di un generico programma in Java con l'entry-point main.

Quanto alle istruzioni, esse rappresentano azioni o comandi che devono essere eseguiti durante l'esecuzione di un programma.
Infine, per Java, come già anticipato, ogni istruzione singola (*single-line statement*) deve terminare con il carattere punto e virgola ; mentre due o più istruzioni (*multi-line statement*) possono essere raggruppate insieme a formare un'unica entità sintattica, definita blocco (*block statement*), se incluse tra le parentesi graffe aperta `{` e chiusa `}`.

NOTA

A volte nel codice sorgente si può rilevare la scrittura di una sola istruzione posta in un blocco di istruzioni. Ciò non è sintatticamente errato e viene fatto, comunemente, per ragioni di stile e indentazione.

Le Tabelle 1.4 e 1.5 mostrano tutte le keyword del linguaggio, aggiornate alla versione 21 di Java; la Tabella 1.6 mostra i segni di punteggiatura; la Tabella 1.7 mostra gli operatori utilizzabili; lo Snippet 1.4 evidenzia alcune keyword, operatori, identificatori, letterali e così via.

Tabella 1.4 Keyword riservate del linguaggio Java.

`abstract`	`assert`	`boolean`	`break`	`byte`	`case`
`catch`	`char`	`class`	`const`[1]	`continue`	`default`
`do`	`double`	`else`	`enum`	`extends`	`final`
`finally`	`float`	`for`	`if`	`goto1`	`implements`
`import`	`instanceof`	`int`	`interface`	`long`	`native`
`new`	`package`	`private`	`protected`	`public`	`return`
`short`	`static`	`strictfp`[3]	`super`	`switch`	`synchronized`
`this`	`throw`	`throws`	`transient`	`try`	`void`
`volatile`	`while`	`_ (underscore)`[2]			

[1] Le keyword `const` e `goto` sono riservate anche se nel linguaggio non sono utilizzate.

[2] Il singolo carattere underscore (_) è, dalla versione 9 di Java, una keyword del linguaggio. Non è dunque più possibile utilizzarlo, singolarmente, come identificatore di una variabile (per esempio, scrivere un'istruzione come `int _ = 10;` darà il seguente messaggio di compilazione: `error: as of release 9, '_' is a keyword, and may not be used as an identifier`.

[3] La keyword `strictfp` è ritenuta obsoleta e non dovrebbe essere utilizzata nel nuovo codice.

Tabella 1.5 Keyword contestuali del linguaggio Java.

`exports`	`module`	`non-sealed`	`open`	`opens`	`permits`
`provides`	`record`	`requires`	`sealed`	`to`	`transitive`
`uses`	`var`	`when`	`with`	`yield`	

TERMINOLOGIA

Una keyword riservata (*reserved keyword*) è una sequenza di caratteri riservata che non è permesso impiegare come identificatore per denominare, per esempio, una variabile o un metodo. Una keyword contestuale (*contextual keyword*) è una sequenza di caratteri che può essere utilizzata come identificatore nell'ambito di un programma Java, ma solo se è al di fuori del suo contesto semantico di attribuzione, laddove assume il ruolo di keyword riservata. Così, per esempio, le keyword `module` e `open` sono riservate solo se appaiono nel contesto di dichiarazione di un *modulo*.

Tabella 1.6 Segni di punteggiatura.

{	}	[	]	(	)	.	,	;	...
@	::								

Tabella 1.7 Operatori.

=	>	<	!	~	?	:	->
==	>=	<=	!=	&&	||	++	--
+	-	*	/	&	|	^	%
<<	>>	>>>	<<=	>>=	>>>=	+=	-=
*=	/=	&=	|=	^=	%=		

TERMINOLOGIA

Gli operatori sono simboli utilizzati nell'ambito di espressioni, atti a descrivere operazioni a uno o più operandi. Così, un'espressione come `x * z` utilizza l'operatore * per moltiplicare l'operando `x` per l'operando `z`. I segni di punteggiatura, invece, sono impiegati per eseguire raggruppamenti e separazioni. Per esempio, i segni `{ }` servono per raggruppare le istruzioni in un blocco; il segno `,` serve per separare più variabili in una singola dichiarazione e così via.

Snippet 1.4 (Snippet_1_4.java) Esempi di keyword, operatori, identificatori, letterali etc.

```
...
public class Snippet_1_4
{
    // public, static, void -> keyword
    // main, args           -> identificatori
    // []                   -> segno di punteggiatura di separazione
    // ()                   -> segno di punteggiatura di raggruppamento
    public static void main(String[] args)
    // { -> segno di punteggiatura di raggruppamento
    {
        // int                  -> keyword
        // number, temp, status -> identificatori
        // ,                    -> segno di punteggiatura di separazione
        // ;                    -> segno di punteggiatura di separazione
        int number, temp, status;

        // int, float, char -> keyword
        // a, f, c          -> identificatori
        // =                -> operatore
        // 100              -> letterale intero
        // 120.78f          -> letterale in virgola mobile
        // 'A'              -> letterale carattere
        // ;                -> segno di punteggiatura di separazione
        int a = 100;
        float f = 120.78f;
        char c = 'A';

        // string       -> keyword
        // name         -> identificatore
        // =            -> operatore
        // "Pellegrino" -> letterale stringa
```

```
        // ;            -> segno di punteggiatura di separazione
        String name = "Pellegrino";
        // } -> segno di punteggiatura di raggruppamento
    }
}
```

ATTENZIONE

Alcuni simboli, a seconda del contesto, possono essere trattati come segni di punteggiatura oppure come operatori. Per esempio, le parentesi tonde, quando sono usate nell'ambito della definizione di un metodo, agiscono come segni di punteggiatura per raggruppare i parametri; ma quando sono impiegate nell'ambito di utilizzo di un metodo agiscono come un operatore di invocazione. Allo stesso modo le parentesi quadre quando sono usate nell'ambito della dichiarazione di un array agiscono come segni di punteggiatura, per separare il tipo di dato dell'array dal suo identificatore; ma quando sono impiegate nell'ambito di utilizzo di un array agiscono come un operatore di accesso indicizzato a un elemento dell'array. In pratica, anche se in accordo con la specifica del linguaggio Java i simboli `( )` e `[ ]` sono tecnicamente segni di punteggiatura, possono agire anche da operatori.

Compilazione ed esecuzione del codice

Dopo aver scritto il programma del Listato 1.1 con un qualunque editor di testo o IDE di preferenza, vediamo come eseguirne la compilazione che, lo ricordiamo, è quel procedimento mediante il quale un compilatore Java legge un file sorgente (nel nostro caso `FirstProgram.java`) e lo trasforma in un file (per esempio `FirstProgram.class`) che conterrà le istruzioni (bytecode) proprie della Java Virtual Machine, cioè da essa comprensibili e pronte per essere elaborate tramite un apposito compilatore Just-In-Time, che le trasformerà in linguaggio macchina del sistema hardware di riferimento.
Prima di vedere come utilizzare manualmente un compilatore Java (Shell 1.1), verifichiamo che esso sia disponibile nel sistema; da una shell (*bash*, *command prompt* e così via) digitiamo, semplicemente, il comando `javac -version` e verifichiamo che appaiano, in output, le informazioni sull'attuale versione del compilatore in uso (per esempio, può essere visualizzato qualcosa come `javac 21.0.2`).

NOTA

Per compilare ed eseguire direttamente i listati e gli snippet di codice ho creato la seguente struttura "parlante" di directory, posta a partire dalla directory radice del sistema operativo scelto, ossia Windows 11: `MY_JAVA_SOURCES`, `MY_JAVA_CLASSES`, `MY_JAVA_PACKAGES`, `MY_JAVA_JARS`, `MY_JAVA_MODS`, `MY_JAVA_RUNTIMES` e `MY_JAVA_DOCUMENTATION`. Tale struttura non è obbligatoria, ma è consigliabile replicarla al fine di seguire nel miglior modo possibile quanto indicato nel presente testo.

Shell 1.1 Invocazione del comando di compilazione.

```
C:\MY_JAVA_SOURCES> javac -d \MY_JAVA_CLASSES FirstProgram.java
```

Dopo la fase di compilazione possiamo avviare ed eseguire il file prodotto (Shell 1.2 tramite il comando `java` (il relativo output è mostrato in Output 1.1).

> **NOTA**
>
> L'opzione di compilazione `-d <directory>` è impiegata per specificare una directory di destinazione per i file `.class` generati durante la fase di compilazione. Se una classe è parte di un package, il compilatore `javac` produrrà, a partire dalla directory indicata, delle sotto-directory che mapperanno il nome del package specificato. Così, tornando al nostro esempio, dato che la classe `FirstProgram` è parte del package `LibroJava.Capitolo1`, `javac` produrrà la seguente struttura di directory: `C:\MY_JAVA_CLASSES\LibroJava\Capitolo1\FirstProgram.class`.

Shell 1.2 Avvio del programma.

```
C:\MY_JAVA_CLASSES> java LibroJava.Capitolo1.FirstProgram
```

In breve, il comando `java` lancia un'applicazione Java (legge il relativo file `.class`) avviando l'ambiente di *runtime* di Java laddove la Java Virtual Machine, tramite il compilatore JIT, traduce il bytecode in codice nativo del sistema in cui eseguire il programma, che, infine, viene eseguito.

Output 1.1 Esecuzione di Shell 1.2.

```
Primo programma in Java: Buon divertimento!
Stamperò un test condizionale tra a=10 e b=20
a < b VERO!
Stamperò un ciclo iterativo, dove leggerò per 10 volte il valore di a
Passo 0 --> a=10
Passo 1 --> a=10
Passo 2 --> a=10
Passo 3 --> a=10
Passo 4 --> a=10
Passo 5 --> a=10
Passo 6 --> a=10
Passo 7 --> a=10
Passo 8 --> a=10
Passo 9 --> a=10
Ora eseguirò una moltiplicazione tra 10 e 20
Il risultato di 10 x 20 è: 200
```

Dalla Shell 1.2 vediamo che il comando `java` esegue il programma `FirstProgram` che stampa quanto mostrato nell'Output 1.1. È utile sottolineare alcuni aspetti:

- il nome del programma `FirstProgram` è il nome della classe contenuta nel file omonimo;
- il nome del file `FirstProgram.class` contenente il programma da eseguire viene passato al comando `java` senza l'indicazione dell'estensione `.class`;
- la classe `FirstProgram` invocata è preceduta dal nome del package di appartenenza `LibroJava.Capitolo1`, poiché quando una classe appartiene a un package, il suo nome deve sempre farne parte.

Problemi di compilazione ed esecuzione?

Elenchiamo alcuni problemi che si potrebbero incontrare durante la fase di compilazione o di esecuzione del programma appena esaminato.

- Il comando di compilazione `javac` è inesistente? Verificare che il path del sistema operativo in uso contenga tra i percorsi di risoluzione la directory `bin` del JDK. Si rimanda a quanto illustrato all'indirizzo `https://github.com/thp1972/Libro_Java21`, link `Installazione Java` per i dettagli su come impostare correttamente il path.
- Il compilatore `javac` non trova il file `FirstProgram.java`? Verificare che la directory corrente sia `C:\MY_JAVA_SOURCES`.
- Il *launcher* `java` non trova il file `FirstProgram.class`? Verificare che la directory corrente sia `C:\MY_JAVA_CLASSES`.

La Figura 1.11 mostra in dettaglio cosa accade dopo aver creato con un qualsiasi editor di testo un file contenente codice scritto in accordo con la sintassi del linguaggio Java, dalla fase di compilazione del predetto file fino alla fase di avvio del relativo file prodotto, che conterrà le istruzioni eseguibili del corrispettivo programma.

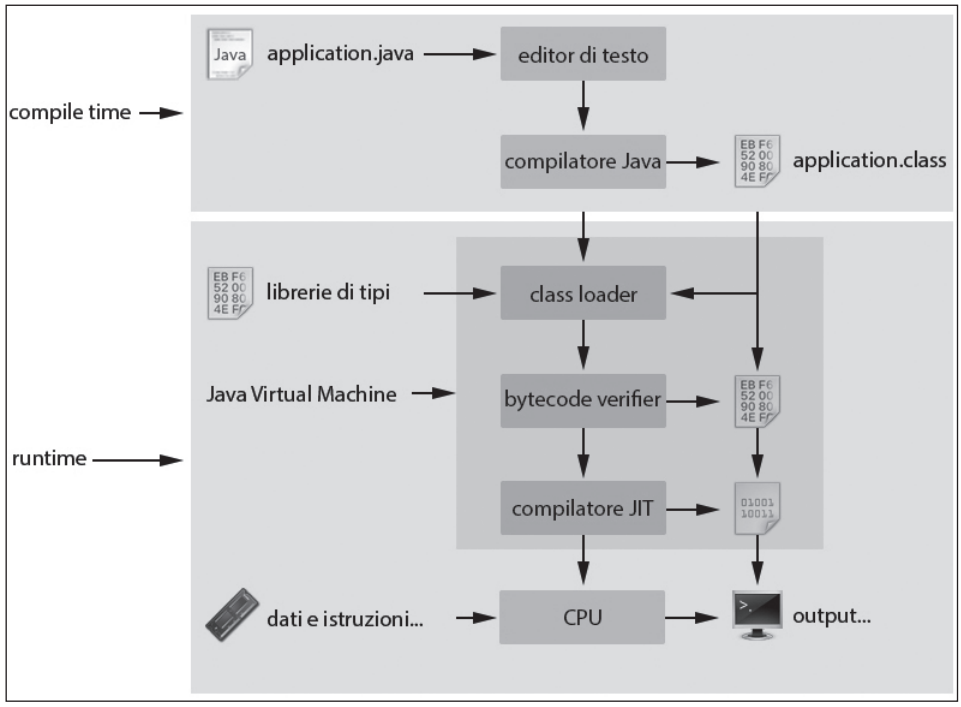

Figura 1.11 Flusso completo di generazione ed esecuzione di un programma Java.

1. Il compilatore Java verifica la correttezza sintattica del codice sorgente scritto all'interno di un file `.java`. Se il processo di verifica va a buon fine converte quindi quel

codice sorgente presente nel predetto file `.java` in un apposito codice di linguaggio intermedio (il bytecode) scrivendolo in un altro file avente estensione `.class`.
2. La Java Virtual Machine, tramite un *class loader*, carica in memoria i tipi utilizzati nel programma.
3. La Java Virtual Machine, tramite un *bytecode verifier*, esamina il bytecode delle classi caricate in memoria, al fine di verificare che sia valido, sicuro e che risponda a determinati vincoli (*static* e *structural constraint*) così come espressamente regolamentati nel documento della *Java Virtual Machine Specification*.
4. La Java Virtual Machine avvia un apposito compilatore Just-In-Time, il quale converte il bytecode in codice di linguaggio macchina, ossia nel codice nativo della piattaforma di esecuzione, che viene poi eseguito (in pratica, alla fine del processo, la CPU preleva, decodifica ed esegue le istruzioni che compongono il programma).

DETTAGLIO

Come abbiamo detto, un compilatore JIT converte a *runtime* (al momento dell'esecuzione) il bytecode in codice nativo. Tuttavia questo processo di conversione è effettuato *just in time*, ossia "appena prima dell'esecuzione", quando cioè essa è effettivamente necessaria. Avviene, nella pratica, quanto segue: quando il flusso di esecuzione del codice raggiunge un punto in cui si invoca un metodo, il compilatore JIT ne traduce il bytecode in codice macchina e tale codice viene eseguito. Successivamente, se il programma invoca di nuovo lo stesso metodo, esso non viene tradotto un'altra volta, ma viene subito eseguito. Al contempo, i metodi non invocati non vengono mai tradotti. Il processo descritto rende evidente come un compilatore JIT sia in effetti efficiente e performante, perché non traduce in blocco, al momento dell'esecuzione, tutto il bytecode in codice macchina senza una reale necessità.

NOTA

Allo stato attuale, la Java Virtual Machine, così come implementata nell'OpenJDK, non offre anche la possibilità di utilizzare un cosiddetto compilatore *Ahead-Of-Time* (*AOT compiler*). In alcune circostanze, infatti, per migliorare le prestazioni di un'applicazione, si può voler usare un compilatore che effettui, per l'appunto, una compilazione anticipata (*ahead-of-time compilation*) di un file `.class`, ossia ne compili tutto il bytecode in codice nativo prima di eseguire l'applicazione; questa compilazione avviene, in genere, *at install time*, ovvero quando l'applicazione viene installata su una piattaforma di destinazione.

La Java Virtual Machine: un breve dettaglio

Prima di continuare lo studio del linguaggio Java appare opportuno fornire dettagli maggiori sulla Java Virtual Machine e sul bytecode generato da un compilatore Java.
Partiamo subito dal dire, in modo più rigoroso di quanto finora fatto, che una Java Virtual Machine è una cosiddetta *abstract computing machine*, ossia un'implementazione software di un computer reale (*real computing machine*) che al pari di quest'ultimo è dotata di un proprio set di istruzioni ed è dunque in grado di eseguire dei programmi.
Il set di istruzioni della Java Virtual Machine è denominato *bytecode* ed è, a tutti gli effetti, il *linguaggio macchina* della macchina virtuale ossia il linguaggio che essa è in grado,

nativamente, di comprendere (così come il linguaggio macchina proprio di una CPU Intel o ARM è quel linguaggio da esse nativamente comprensibile).
Il set di istruzioni della Java Virtual Machine è costituito da 256 *opcode* (*operation code*, codici operativi) che specificano le operazioni eseguibili ed, eventualmente, anche gli operandi impiegabili.

TERMINOLOGIA

Il termine *bytecode* si riferisce al fatto che il set di istruzioni della macchina virtuale Java è composto da massimo 256 opcode ossia ogni opcode ha la lunghezza di 1 byte.

Ogni opcode ha anche un relativo *mnemonico*, il quale rappresenta una sorta di identificatore *umanamente* rammentabile che è impiegabile per esprimere quella specifica operazione (oppure, detto in altro modo, un codice mnemonico è una descrizione testuale breve dell'istruzione propria dell'opcode, che invece è espresso in valore binario).
Possiamo quindi dire che il file `.class` prodotto da un compilatore Java come `javac` genera un *file binario* contenente i relativi bytecode, espressi in un formato non umanamente comprensibile; sono però in un formato comprensibile dalla macchina virtuale Java (rappresentano, come detto, il suo linguaggio macchina).
È tuttavia possibile, grazie al comando `javap` fornito dal JDK, *disassemblare* il file `.class` al fine di ottenere come output i codici mnemonici umanamente comprensibili dei bytecode che rappresentano, da questo punto di vista, le istruzioni nel *linguaggio assembly* della macchina virtuale Java (*virtual machine assembly language*).
In tal senso, quindi, se nel JDK fosse disponibile un *assemblatore* (*assembler*), sarebbe possibile scrivere direttamente *codice Java* nel linguaggio assembly proprio della JVM e darlo poi in pasto all'assemblatore, il quale provvederà a trasformarlo in codice macchina (bytecode) proprio della macchina virtuale Java.

TERMINOLOGIA

Il linguaggio assembly (*assembly language*) è un linguaggio di programmazione a basso livello specifico per una particolare architettura hardware. Esso permette la programmazione di tali architetture attraverso un determinato set di istruzioni (*instruction set*) che sono rappresentate da codici operativi mnemonici delle equivalenti istruzioni in codice macchina, che sono espresse come sequenze di 1 e 0 e che sono le uniche *forme* di istruzioni realmente e direttamente comprensibili dalle architetture hardware. Un assemblatore (*assembler*) infine è il programma che si occupa di convertire il codice scritto in linguaggio assembly nel corrispettivo codice macchina dell'architettura hardware di destinazione.

NOTA

Esistono, tra gli altri, i seguenti progetti open source: Jasmin (`http://jasmin.sourceforge.net/`) e Krakatau Bytecode Tools (`https://github.com/Storyyeller/Krakatau`). Essi forniscono degli assemblatori per il linguaggio assembly della Java Virtual Machine e consentono pertanto di creare file `.class` contenenti il rispettivo bytecode in formato binario.

Ritornando, per esempio, al Listato 1.1, il disassemblato della classe `FirstProgram` è ottenibile con l'invocazione del comando `javap` con il flag `-c` (Shell 1.3) che produce come risultato le istruzioni assembly proprie della JVM (Output 1.2).

Shell 1.3 Utilizzo del comando javap con il file FirstProgram.class.

```
C:\MY_JAVA_CLASSES> javap -c LibroJava\Capitolo1\FirstProgram.class
```

Output 1.2 Esecuzione di Shell 1.3.

```
public class LibroJava.Capitolo1.FirstProgram
{
  public LibroJava.Capitolo1.FirstProgram();
    Code:
       0: aload_0
       1: invokespecial #1           // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: ldc           #2           // String Primo programma in Java:
       2: astore_1
       3: ldc           #3           // String Buon divertimento!
       5: astore_2
       6: bipush        10
       8: istore_3
       9: bipush        20
      11: istore        4
      13: ldc           #4           // float 44.5f
      ...
     108: istore        6
     110: iload         6
     112: bipush        10
     114: if_icmpge     164
     117: getstatic     #5           // Field java/lang/System.out:Ljava/io/PrintStream;
     120: ldc           #17          // String Passo %d
     122: iconst_1
     123: anewarray     #7           // class java/lang/Object
     126: dup
     127: iconst_0
     ...
  static {};
    Code:
       0: bipush        10
       2: putstatic     #23          // Field counter:I
       5: return
}
```

L'Output 1.2 è un estratto del codice assembly prodotto dal comando `javap` che ci mostra per le righe sotto la dicitura `Code:` tre informazioni essenziali e in sequenza: un numero intero come `0`, `2`, `3` e così via, che rappresenta un offset in byte dove si trova la relativa istruzione assembly rispetto all'inizio del metodo di appartenenza; l'istruzione assembly da elaborare (l'opcode mnemonico testuale) con, se presenti, gli operandi che utilizza; gli eventuali commenti scritti con la stessa sintassi usata per definire i commenti nel codice sorgente di un qualsiasi programma Java (*single-line-comment*).