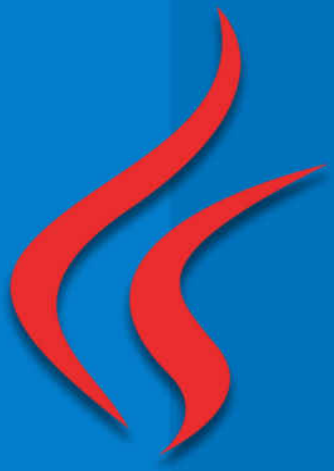
JACK FELLERS

Java



LA GUIDA IN 7 GIORNI
PIU' COMPLETA PER PRINCIPIANTI ALLA
PROGRAMMAZIONE MODERNA CON JAVA.
INCLUSO GLOSSARIO AGGIORNATO AL 2022

WEBHAWK

JAVA

JACK FELLERS

INDICE

Caro lettore, per ringraziarti per la fiducia dimostratami acquistando il mio libro, ecco per te in **regalo**, una guida per fortificare ancora di più la tua conoscenza nella programmazione web!

Scansiona il codice o clicca sul link per riscattarlo in meno di un minuto:

Link alternativo al Qr code:

https://webhawk.tech/optin-it/

Buona lettura!

PREMESSA

Probabilmente hai sentito dire che la programmazione informatica è qualcosa d'incredibilmente difficile. Richiede una laurea in informatica, un sacco di soldi per hardware e software, una mente analitica brillante, tanta pazienza e tanta caffeina. E se ti dicessi che non è proprio così?

Programmare è più facile di quanto si possa pensare, nonostante ciò che i programmatori hanno voluto far credere per anni. Questo è un ottimo momento per imparare a programmare perché vengono resi disponibili innumerevoli strumenti di programmazione in modo gratuito sul Web. Migliaia di programmatori distribuiscono il loro lavoro open source in modo che altre persone possano esaminare come è stato scritto il software, correggere eventuali errori e apportare miglioramenti.

È un ottimo momento per imparare a programmare in Java semplicemente perché è il linguaggio più usato. Miliardi di dispositivi usano Android, un sistema operativo le cui app sono tutte scritte in Java o linguaggi basati su Java. Se usi un telefono Android, sfrutti Java ogni giorno quando guardi un film, ascolti la radio in streaming o usi i social network.

Questo Ebook ha lo scopo d'insegnare la programmazione Java a tre tipi di persone:

1. Principianti che non hanno mai provato a programmare prima
2. Principianti che hanno provato a programmare ma che odiano farlo
3. Coloro che conoscono un altro linguaggio di programmazione e voglio imparare in modo rapido Java

Per raggiungere questo obiettivo, useremo il linguaggio naturale il più possibile al posto del gergo tecnico. Tutti i nuovi termini di

programmazione saranno evidenziati e spiegati.

Alla fine di questo Ebook sarai in grado di scrivere programmi, immergerti in lezioni di programmazione, leggere libri tecnici con maggiore sicurezza e apprendere nuovi linguaggi in modo più facile. Avrai anche delle competenze in Java, fondate su solide basi. Forniremo un'introduzione alla programmazione in modo da spiegare perché Java è così popolare e poi ci immergeremo nella programmazione.

1

PROGRAMMI

Un programma per computer, chiamato anche software, è un modo per dire a un computer di eseguire un'attività. Tutto ciò che fa il computer, dall'avvio allo spegnimento, viene eseguito da un **programma**. Mac OS X è un programma; Minecraft è un programma; il software che controlla la stampante è un programma; anche Windows è un programma.

I programmi per computer sono costituiti da un elenco di comandi gestiti dal computer in un ordine ben definito durante l'esecuzione del programma e ogni comando è detto **istruzione**. Ogni riga di un programma corrisponde a un comando quindi a un'istruzione.

In linguaggi come il BASIC, i numeri delle righe vengono utilizzati per mettere le istruzioni nell'ordine corretto mentre altri linguaggi come Java non usano i numeri delle righe bensì modi diversi per comunicare al computer come eseguire un programma.

Essendo noi stessi a scrivere i programmi, non è possibile incolpare il computer quando qualcosa va storto durante l'esecuzione del programma. Il computer fa esattamente quello che gli diciamo di fare, quindi la colpa di eventuali errori di solito ricade sul programmatore. Questa è una cattiva notizia ma vi è anche una buona notizia: non puoi fare alcun danno permanente. Puoi sperimentare come preferisci con Java, non si tratta di esperimenti di laboratorio con attrezzature e kit di preparazione costosissimi. Puoi installare Java sul tuo PC, installare un ambiente di sviluppo e iniziare a programmare.

In Java

La raccolta d'istruzioni che compongono un programma per computer è chiamata **codice sorgente**. La maggior parte dei programmi per computer sono scritti nello stesso modo in cui si scrivono le e-mail, digitando ciascuna istruzione in una finestra di testo. Alcuni strumenti di programmazione sono dotati di un proprio editor di codice sorgente e altri possono essere utilizzati con qualsiasi editor di testo.

Al termine della scrittura di un programma per computer, il file viene salvato sul disco. I programmi per computer hanno spesso la propria estensione per indicare il tipo di file, i programmi Java, ad esempio, devono avere l'estensione .java, come in Esempio.java. Per eseguire un programma che hai salvato come file, hai bisogno di aiuto e questo dipende dal linguaggio di programmazione che stai utilizzando. Alcuni linguaggi richiedono un interprete per eseguire i propri programmi, altri un compilatore.

L'interprete esamina ogni riga di un programma e la esegue, quindi procede alla riga successiva. Il più grande vantaggio dei linguaggi interpretati è che sono più veloci da testare. Quando si scrive un programma, è possibile provarlo immediatamente, correggere gli errori e riprovare. Lo svantaggio principale, invece, è che funzionano più lentamente di altri programmi perché ogni riga deve essere tradotta in istruzioni che il computer può eseguire, una alla volta.

Altri linguaggi di programmazione richiedono un compilatore che ha il compito di recuperare un programma e tradurlo in una forma che il computer possa capire. Inoltre, il programma viene eseguito nel modo più efficiente possibile. Il programma compilato può essere eseguito direttamente senza la necessità di un interprete. I programmi compilati vengono eseguiti in modo più rapido dei programmi interpretati ma

richiedono più tempo per la fase di test. Devi scrivere il tuo programma e compilare il tutto prima di provare il codice e se trovi un errore e lo risolvi, devi compilare nuovamente il programma.

Java è insolito perché richiede sia un compilatore che un interprete. Il compilatore converte le istruzioni che compongono il programma in **bytecode**. Una volta che questo bytecode è stato creato correttamente, può essere eseguito da un interprete chiamato **Java Virtual Machine**. La Java Virtual Machine, chiamata anche JVM, è quello strumento che consente allo stesso programma Java di funzionare senza modifiche su diversi sistemi operativi e diversi tipi di dispositivi. La macchina virtuale trasforma il bytecode in istruzioni che il sistema operativo di un determinato dispositivo può eseguire.

Molti nuovi programmatori si scoraggiano quando iniziano a testare i loro programmi perché compaiono errori ovunque. Alcuni di questi sono errori di sintassi, che vengono identificati dal computer mentre legge il programma. Altri errori sono di tipo logico, che vengono rilevati dal programmatore solo quando il programma viene testato.

Quando inizi a scrivere i tuoi programmi, acquisisci anche una certa familiarità con gli errori, diventando parte naturale del processo di sviluppo. Gli errori di programmazione sono chiamati **bug**, un termine che risale a un secolo o più per descrivere errori nei dispositivi tecnici. Il processo di correzione degli errori viene chiamato **debug**.

2

INSTALLAZIONE

Per iniziare a scrivere dei programmi in Java, è necessario disporre di uno strumento di programmazione Java. Sono disponibili diversi programmi di questo tipo: a partire dalla *Java Development Kit* fino ai più sofisticati *Eclipse*, *IntelliJ IDEA* e *NetBeans*. Questi ultimi tre strumenti sono ciascuno un ambiente di sviluppo integrato (**IDE**) ovvero potenti strumenti utilizzati dai programmatori professionisti per svolgere il proprio lavoro.

Ogni volta che Oracle rilascia una nuova versione di Java, il primo strumento che la supporta è il Java Development Kit (JDK). Per creare i programmi in questo Ebook, abbiamo usato la JDK versione 9 ma qualunque versione superiore dovrebbe garantire il funzionamento dei programmi creati.

La JDK è un insieme di strumenti da riga di comando gratuiti per la creazione di software Java. Manca completamente un'interfaccia grafica quindi, se non hai mai lavorato in un ambiente non grafico come il prompt dei comandi di Windows o l'interfaccia della riga di comando di Linux, troverai difficile usare la JDK.

NetBeans, offerto gratuitamente da Oracle, è un modo molto più semplice per scrivere e testare il codice Java rispetto alla cruda JDK. Questo strumento include un'interfaccia grafica, un editor di codice sorgente, delle funzionalità per disegnare interfacce utente e un gestore di progetti.

Ogni IDE funziona a complemento della JDK eseguendo i suoi comandi dietro le quinte quindi è necessario disporre di entrambi gli strumenti sul sistema quando si inizia a sviluppare programmi Java. La maggior parte dei programmi di questo Ebook sono stati creati con NetBeans, che è possibile scaricare e installare insieme o separatamente dalla JDK.

Nota bene che non è obbligatorio usare NetBeans per replicare gli esempi forniti in questo Ebook, sei libero di utilizzare Eclipse, IntelliJ IDEA, Atom o addirittura editor di testo come Sublime o Notepad++.

Se hai già installato un IDE per altri linguaggi, verifica se può essere usato anche per sviluppare in Java. Alcuni strumenti come IntelliJ IDEA consentono di programmare in diversi linguaggi semplicemente aggiungendo un plug-in. In caso contrario collegati all'indirizzo https://netbeans.org/ infatti sebbene NetBeans abbia funzionalità avanzate che richiedono tempo per l'apprendimento, semplifica molto la creazione e l'esecuzione di semplici applicazioni scritte in Java.

Sino a ora ti è stato presentato il concetto di programmazione di un computer, hai anche imparato qualcosa in più sul funzionamento di Java.

Se sei ancora confuso su programmi, linguaggi di programmazione o Java in generale, non farti prendere dal panico. Tutto inizierà ad avere un senso a partire dal prossimo capitolo, che ti guiderà nel processo di creazione di un programma Java.

3

HELLO WORLD

Con la maggior parte dei linguaggi di programmazione, i programmi per computer vengono scritti inserendo il testo in un editor di testo (chiamato anche editor di codice sorgente). NetBeans include un proprio editor per la scrittura di programmi Java che, essenzialmente, sono semplici file di testo senza alcuna formattazione (non hanno testo centrato o testo in grassetto).

L'editor del codice sorgente di NetBeans funziona come un semplice editor di testo con alcuni utili miglioramenti per i programmatori. Il testo trasforma in colori diversi il codice durante la digitazione per identificare diversi elementi o parole chiave del linguaggio. NetBeans effettua il conteggio delle righe e fornisce un'utile documentazione all'interno dell'editor. Poiché i programmi Java sono file di testo, è possibile aprirli e modificarli con qualsiasi editor di testo. È possibile scrivere un programma Java con NetBeans, aprirlo con *Blocco Note* di Windows e apportare modifiche per poi riaprirlo successivamente in NetBeans senza problemi.

Il primo programma Java che creerai mostrerà un benaugurante saluto nel mondo dell'informatica: "Hello World!". Per preparare il primo progetto in NetBeans, se non l'hai ancora fatto, crea un nuovo progetto assegnando un nome a scelta seguendo questi passaggi:

1. Scegli il comando di menu *File*, *Nuovo progetto* in modo da aprire la relativa finestra di dialogo;
2. Scegli la categoria di progetto *Java* e il tipo di progetto *Applicazione Java*, quindi fai clic su *Avanti*;
3. Digita il nome del progetto;
4. Deseleziona la casella di controllo *Crea classe principale*;
5. Fai clic su *Fine*.

Il progetto sarà creato nella sua cartella con il nome scelto. Puoi usare questo progetto per tutti i programmi Java proposti in questo e-book. Il progetto verrà visualizzato nel riquadro *Progetti* accanto a un segno + che serve per vedere i file e le cartelle contenuti nel progetto. Per aggiungere un nuovo programma Java al progetto attualmente aperto, selezionare *File*, *Nuovo file*.

Il riquadro *Categorie* elenca i diversi tipi di programmi Java che è possibile creare, fai clic sulla cartella *Java* in questo riquadro per visualizzare i tipi di file che appartengono a questa categoria. Per questo primo progetto, seleziona il tipo di file *Empty Java* e fare clic su *Avanti*. Ci verranno richiesti il nome della classe e il nome del pacchetto, inseriamo rispettivamente Saluto e com.esercizio. Dopo aver completato questi passaggi il risultato sarà una classe simile alla seguente:

```
package com.esercizio;


class Saluto {
    public static void main(String[] arguments) {
    // Il mio primo programma Java
    }
    }
```

Assicurati che le lettere maiuscole e minuscole siano esattamente come mostrato nell'esempio e usa la barra spaziatrice o il tasto Tab per inserire gli spazi vuoti davanti alle righe. A questo punto, il file Saluto.java contiene lo scheletro di un programma Java.

Analisi del programma

Un package è un modo per raggruppare insieme i programmi Java. La prima riga del file precedente indica al computer d'impostare com.esercizio come nome del pacchetto del programma.

Dopo una linea vuota troviamo il nome della classe e ti ricordo che ogni istruzione che dai a un computer è chiamata istruzione. L'istruzione class è il modo in cui dai un nome al tuo programma e viene anche utilizzata per determinare altre cose sul programma stesso, come vedrai in seguito.

Un programma Java deve avere un nome che corrisponda alla prima parte del suo nome file, la parte prima del "." E dovrebbe essere scritto nello stesso modo. Ad esempio, il file Saluto.java deve contenere una classe di nome Saluto. Se il nome del programma non corrisponde al nome del file, viene visualizzato un errore quando si tenta di compilare, a seconda di come viene utilizzata l'istruzione class per configurare il programma. La riga successiva è fondamentale:

public static void main**(**String**[]** arguments**) {**

Questa riga può essere tradotta come: "La parte principale del programma inizia qui". I programmi Java sono organizzati in diverse sezioni, quindi deve esserci un modo per identificare la parte di un programma che viene eseguita per prima durante l'esecuzione. L'istruzione main è il punto di accesso alla maggior parte dei programmi Java, a eccezione delle applet, dei programmi eseguiti su una pagina Web e qualche altro caso. Per differenziarli da questi altri tipi, i programmi eseguiti direttamente sul tuo computer sono chiamati **applicazioni**.

Nel programma che abbiamo scritto, le righe 3, 4, 6 e 7 contengono delle parentesi graffe aperte ({) o chiuse (}). Queste parentesi sono un modo

per raggruppare le righe del tuo programma (allo stesso modo in cui le parentesi sono usate in una frase per raggruppare delle parole). Tutto ciò che è racchiuso tra la parentesi graffa aperta e quella chiusa fa parte dello stesso gruppo, questi raggruppamenti sono chiamati **blocchi**.

I blocchi possono essere posizionati all'interno di altri blocchi (proprio come in questa frase vengono utilizzate le parentesi (e qui viene utilizzato un secondo insieme di parentesi)). Il programma creato ha parentesi sulla linea 4 e sulla linea 6 che stabiliscono un altro blocco. Questo blocco inizia con l'istruzione main. Le righe all'interno del blocco dell'istruzione main verranno eseguite all'avvio del programma. L'unica riga che non abbiamo ancora analizzato è la più semplice ovvero:

// Il mio primo programma Java

Questa linea è una sorta di segnaposto infatti il simbolo // dice al computer d'ignorare tutto quello che segue su quella riga perché è stata inserita nel programma solo a beneficio di coloro che guardano il codice sorgente. Le righe che servono a questo scopo sono chiamate **commenti**.

In questo momento, hai scritto un programma Java completo infatti può essere compilato senza errori ma, se lo esegui, non succede nulla. Il motivo è che non hai ancora detto al computer di fare qualcosa. Il blocco d'istruzioni main contiene solo un singolo commento, che, tra l'altro, viene ignorato dal computer. È necessario aggiungere alcune istruzioni tra le parentesi del blocco main.

Memorizzare informazioni

Nei programmi che scrivi, hai bisogno di un luogo dove conservare le informazioni per un breve periodo di tempo. Puoi farlo usando una **variabile** cioè una parte di memoria che può contenere informazioni come numeri interi, numeri in virgola mobile, valori vero o falso, caratteri e righe di testo. Le informazioni memorizzate in una variabile possono cambiare, ed è così che si ottiene il nome “variabile”. Nel file Saluto.java, sostituisci la riga 5 (quella con il commento) con la seguente:

String benvenuto = "Hello World!"**;**

Questa istruzione dice al computer di memorizzare il testo "Hello World!" in una variabile chiamata benvenuto. In un programma Java, è necessario indicare al computer quale tipo d'informazioni conterrà una variabile. In questo programma, il messaggio di benvenuto è una stringa: una riga di testo che può includere lettere, numeri, punteggiatura e altri caratteri. Inserendo String nell'istruzione si imposta la variabile per poter contenere i valori di tipo stringa.

Nota bene che questa istruzione nel programma include un punto e virgola alla fine della riga. Il punto e virgola termina ogni istruzione in un programma Java e possiamo compararlo al punto alla fine di una frase. Il computer utilizza questo carattere di terminazione per determinare quando termina un'istruzione e inizia quella successiva.

Se esegui il programma a questo punto, sembra comunque che non accada nulla. Il comando per memorizzare il testo nella variabile di benvenuto viene eseguito dietro le quinte. Per fare in modo che il computer mostri che sta facendo qualcosa, è possibile visualizzare il contenuto di quella variabile.

Inserisci un'altra riga vuota nel programma dopo aver dichiarato il messaggio di benvenuto e inserisci la seguente istruzione:

System.out.println(benvenuto);

Questa istruzione indica al computer di visualizzare il valore memorizzato nella variabile benvenuto. L'istruzione System.out.println() consente al computer di visualizzare le informazioni sulla console del dispositivo di output: il monitor.

Ecco come si presenta il programma funzionante:

```
package com.esercizio;


class Saluto {
    public static void main(String[] arguments) {
    String benvenuto = "Hello World!";
    System.out.println(benvenuto);
    }
    }
```

Probabilmente fino ad ora hai eseguito il programma premendo il tasto “Run” dell’IDE. In realtà, prima di poter eseguire un programma Java, è necessario compilarlo e, in fase di compilazione, le istruzioni fornite al computer nel programma vengono convertite in una forma che il computer può comprendere meglio.

NetBeans compila i programmi automaticamente non appena vengono salvati. Se il tuo programma è simile a quello mostrato sopra, il programma verrà compilato correttamente. Automaticamente verrà creata una versione compilata del programma ovvero un nuovo file chiamato Saluto.class. Tutti

i programmi Java sono compilati in file di classe, a cui viene data l'estensione .class.

Un programma Java può essere composto da diverse classi che lavorano insieme, ma in un programma semplice come questo è necessaria solo una classe. Il compilatore trasforma il codice sorgente Java in bytecode ovvero un modulo che può essere eseguito dalla Java Virtual Machine (JVM).

Identificare gli errori

La potenza di un IDE ti consente di essere più produttivo e soprattutto identificare in modo rapido gli eventuali errori. In fase di battitura ho scritto queste righe:

```
String benvenuto = "Hello World!";
System.out.println(bevenuto);
```

Riesci a identificare l'errore? L'errore è un classico errore di battitura nel nome della variabile, che dovrebbe essere benvenuto invece di bevenuto. Prova a replicare questo errore per vedere come si comporta il tuo IDE, dovresti vedere delle finestre di dialogo che ti comunicano l'errore.

Quando incontri errori di questo tipo, assicurati che le lettere maiuscole e minuscole siano corrette e che tutti i caratteri necessari, come parentesi graffe, tonde e punti e virgola, siano inclusi.

4

TIPI DI VARIABILI

Nel programma precedente abbiamo memorizzato una sequenza di lettere in una variabile. Le informazioni memorizzate nelle variabili possono essere modificate durante l'esecuzione di un programma. Le stringhe sono solo uno dei tipi di informazioni che possono essere archiviate nelle variabili, che possono anche contenere caratteri, numeri interi, numeri in virgola mobile e oggetti.

Sappiamo bene cosa sono le istruzioni e i blocchi ma quando le istruzioni sono di tipo matematico diventano **espressioni**. Un esempio di espressione è la seguente riga:

int prova = 7 * 4**;**

Le variabili sono il modo principale con cui un computer ricorda qualcosa durante l'esecuzione di un programma. In un programma Java, le variabili vengono create con un'istruzione che deve includere due cose: il nome della variabile ed il tipo di informazioni che la variabile memorizzerà.

La dichiarazione di una variabile può includere anche il valore delle informazioni archiviate.

Per vedere i diversi tipi di variabili e il modo in cui vengono create, avvia NetBeans e crea un nuovo file Java con il nome della classe Variabile.

Interi e Reali

Finora il programma Variabile ha solo un blocco main() che non contiene alcuna istruzione, si tratta dello scheletro dell'applicazione. Inserisci la seguente dichiarazione:

int ruote**;**

Questa istruzione crea una variabile denominata ruote che non specifica un valore, quindi per il momento questa variabile è uno spazio di archiviazione vuoto.

Il testo int all'inizio dell'istruzione indica che la variabile sarà utilizzata per memorizzare numeri interi. È possibile utilizzare il tipo int per memorizzare la maggior parte dei numeri non decimali necessari nei programmi per computer. Questo tipo di variabile può contenere qualsiasi numero intero compreso tra circa -2,14 miliardi e 2,14 miliardi.

Aggiungi una riga vuota dopo l'istruzione appena inserita e aggiungi la seguente istruzione:

float motore**;**

Questa istruzione crea una variabile con nome motore mentre il testo float indica che conterrà numeri in virgola mobile. Le variabili a virgola mobile vengono utilizzate per memorizzare numeri che potrebbero contenere un punto decimale. Il tipo di variabile float contiene numeri decimali fino a 38 cifre mentre il tipo double è più grande e può contenere numeri decimali fino a 300 cifre.

Caratteri e Stringhe

Come abbiamo visto nel capitolo precedente è possibile utilizzare le variabili per memorizzare il testo. Due tipi di testo possono essere memorizzati come variabili: caratteri e stringhe.

Un carattere è una singola lettera, numero, segno di punteggiatura o simbolo mentre una stringa è un gruppo di caratteri. Il prossimo passo nella creazione del programma Variabile è quello di creare una variabile char e una variabile String. Aggiungi queste due istruzioni dopo la riga dove abbiamo definito il float:

```
char tipo = 'S';
String marchio = "Audi";
```

Come avrai notato, queste due istruzioni usano simboli diversi attorno ai propri valori. Quando si utilizzano i caratteri, è necessario inserire delle virgolette singole su entrambi i lati del valore assegnato ad una variabile. Per le stringhe è necessario racchiudere il valore tra virgolette doppie.

Le virgolette impediscono che il carattere o la stringa vengano confusi con il nome di una variabile o un'altra parte delle istruzioni. Dopo aver aggiunto le istruzioni con char e String, il programma dovrebbe assomigliare a questo:

```
package com.esercizio;


class Variabile {
    public static void main(String[] arguments) {
    int ruote;
    float motore;
```

```
char categoria = 'S';
String marchio = "Audi";
}
}
```

Nell’esempio le ultime due variabili nel programma Variabile usano il segno = per assegnare un valore iniziale quando le variabili vengono create. Puoi usare questa opzione per tutte le variabili che crei in un programma Java.

I tipi di variabili che sono stati introdotti finora sono quelli principali e che userai per la maggior parte della programmazione in Java. È bene sapere che ci sono altri tipi meno comuni: uno di questi è byte, che contiene numeri interi che spaziano da -128 a 127. Il secondo, short, può essere utilizzato per numeri interi di dimensioni inferiori rispetto al tipo int. Un numero di tipo short può variare da -32.768 a 32.767.

L'ultimo dei tipi di variabili numeriche, long, viene utilizzato per numeri interi troppo grandi da contenere per il tipo int. Un intero long può variare da -9,22 quintilioni a 9,22 quintilioni che è un numero molto ma molto grande.

Booleani

Java dispone di un tipo di variabile chiamata boolean che può essere utilizzata solo per memorizzare il valore vero o il valore falso. A prima vista, una variabile booleana potrebbe non sembrare particolarmente utile, tuttavia, le variabili booleane vengono utilizzate molto spesso nei programmi.

Supponiamo di voler continuare con la creazione di un oggetto di tipo "Automobile", abbiamo dichiarato alcuni valori come ruote, motore, categoria e marchio. Potremmo utilizzare i valori booleani per effettuare dei controlli e vedere se l'auto ha tutto il necessario:

boolean haVolante = **false;**

boolean haMotore = **true;**

Vedremo in seguito come questi valori booleani sono fondamentali per modificare il normale flusso d'esecuzione di un programma.

Nome delle variabili

I nomi delle variabili in Java possono iniziare con una lettera, un carattere underscore (_) o il simbolo del dollaro ($). Il resto del nome può essere qualsiasi lettera o numero.

Puoi dare alle tue variabili quasi tutti i nomi che ti piacciono, ma accertati che siano coerenti tra loro.

Nota bene che Java fa distinzione tra maiuscole e minuscole quando si tratta dei nomi delle variabili, è necessario utilizzare sempre le maiuscole nello stesso modo. Ad esempio, se la variabile haVolante viene referenziata con HaVolante da qualche parte nel programma, si genererà un errore che impedirà la compilazione del programma.

Il nome di una variabile dovrebbe descrivere il suo contenuto in qualche modo. Per convenzione, la prima lettera deve essere minuscola e se il nome della variabile ha più di una parola, si trasforma la prima lettera di ogni parola successiva in maiuscolo. Questa convenzione è detta camelCase, esiste anche un'altra convenzione che usa un underscore (_) per separare le parole ed è detta snake_case.

Se provi ad inserire un nome di variabile non valido in un programma, NetBeans risponderà segnalando l'errore con un'icona rossa accanto alla linea dove si è verificato l'errore. Anche le parole chiave di Java, come public, class, true, false ecc. non possono essere utilizzate come nomi di variabili. Java 9 ha aggiunto un'altra limitazione: un nome di variabile non può essere un singolo carattere underscore (_).

5

OPERATORI

Le istruzioni possono usare espressioni matematiche impiegando gli operatori +, -, *, / e %. Questi operatori vengono utilizzati per manipolare i numeri in tutti i programmi Java. Un'addizione in Java utilizza l'operatore +, come vedi in queste istruzioni:

```
int peso = 25;
peso = peso + 15;
```

La seconda istruzione utilizza l'operatore + per impostare la variabile peso uguale al valore corrente più 15 unità. Tutti gli altri operatori matematici possono essere usati allo stesso modo:

```
int peso = 25;


// Sottrazione
    peso = peso - 10;


// Moltiplicazione
    peso = peso * 2;


// Divisione
    peso = peso / 5;


// Modulo
    peso = peso % 5;
```

Per trovare il resto di una divisione, si può utilizzare l'operatore % detto anche **operatore modulo**. Nell’esempio si avrà il risultato di peso % 5 pari a 1 perché prima vengono eseguite tutte le operazioni precedenti a partire dal peso pari a 25.

Incremento e decremento

Assumiamo che un'attività nel tuo programma stia cambiando il valore di una variabile di un’unità. È possibile aumentare il valore di un’unità ovvero **incrementare** la variabile o, diminuire il valore di un’unità quindi **decrementare** la variabile. Ci sono operatori per svolgere entrambe le attività infatti per incrementare il valore di una variabile si usa l'operatore ++, come nella seguente istruzione:

```
peso++;
```

Per decrementare una variabile di un’unità si usa l’operatore --:

```
peso--;
```

Esistono due forme per incrementare o decrementare una variabile, la **forma prefissa** e quella **postfissa**. Nel primo caso la variabile viene incrementata o decrementata prima di essere usata in un’espressione. Nel secondo caso la variabile viene incrementata dopo aver valutato l’espressione. Ecco un esempio per chiarire le idee:

```
int x = 5, y = 5;


System.out.println(++x); // stampa 6
    System.out.println(x); // stampa 6


System.out.println(y++); // stampa 5
    System.out.println(y); // stampa 6
```

So bene che questo concetto non è semplice da capire e può portare un po' di confusione pertanto sappi che puoi raggiungere lo stesso obiettivo con il codice seguente:

```
x = x + 1;


// equivale a


x++;
```

Precedenza degli operatori

Quando si utilizza un'espressione con più di un operatore, è necessario sapere quale ordine utilizza il computer nel momento in cui elabora l'espressione. Considera le seguenti affermazioni:

```
int y = 10;
x = y * 3 + 5;
```

A meno che tu non sappia quale ordine utilizza il computer quando risolve l'espressione matematica in questione, non puoi essere sicuro di quale sarà il risultato memorizzato nella variabile x. Potrebbe essere 35 oppure 80, a seconda che sia valutato per primo y * 3 o 3 + 5.

In realtà esiste un ordine di valutazione per eliminare eventuali dubbi:

1. L'incremento e il decremento vengono valutati per primi;
2. Successivamente si verificano la moltiplicazione, la divisione e l'operatore modulo;
3. Seguono addizione e sottrazione;
4. In seguito, ci sono gli operatori di confronto;
5. Per ultimo viene usato il segno di uguale = per impostare il valore di una variabile.

Poiché la moltiplicazione ha luogo prima dell'addizione, è possibile rivisitare l'esempio precedente e trovare la risposta: y viene prima moltiplicato per 3, quindi vengono aggiunte 5 unità. La variabile x sarà pari a 35. Alla luce di ciò, dovresti essere in grado di capire il risultato delle seguenti istruzioni:

```
int x = 5;
```

int numero = x++ * 6 + 4 * 10/2;

Quale valore sarà memorizzato nella variabile numero?

Il valore sarà pari a 50 perché innanzitutto, viene gestito l'operatore di incremento e x++ imposta il valore della variabile x a 6. Poiché si tratta di un operatore in forma postfissa verrà usato il valore originale di x e l'espressione diventa la seguente:

int numero = 5 * 6 + 4 * 10/2;

Adesso si ha 30 + 40/2, quindi si esegue la divisione e si ottiene 30 + 20 ovvero 50.

6

USARE LE STRINGHE

I tuoi programmi Java sono in grado di fare tranquillamente il loro lavoro e di non fermarsi mai ma quando un programma deve comunicare, il modo più semplice per farlo è attraverso l'uso delle stringhe. I programmi Java usano le stringhe come mezzo principale per comunicare con gli utenti. Le stringhe sono raccolte di testo: lettere, numeri, punteggiatura e altri caratteri.

Abbiamo già visto come memorizzare una stringa o un carattere all'interno di una variabile ma come dobbiamo comportarci se vogliamo creare una stringa con una nuova riga o con un ritorno a capo? Quando viene creata o visualizzata una stringa, il suo testo deve essere racchiuso tra virgolette doppie e Java dispone di un carattere speciale che può essere inserito in una stringa: (\).

Ogni volta che viene rilevato questo carattere in una stringa, esso assume un significato ben preciso come mostrato dalla tabella che segue:

Carattere speciale	**Cosa mostra**
\'	Singolo apice
\"	Doppio apice
\\	Backslash
\t	Tab
\b	Backspace

\r
Ritorno a capo
\n
Nuova riga

Assumiamo che il programma Java consigli un paio di scarpe in base ai gusti dell'utente. Il programma potrebbe suggerire qualcosa del genere:

```
System.out.println("Ti potrebbero piacere le scarpe Tod\'s");
// Ti potrebbero piacere le scarpe Tod's
```

```
System.out.println("Ti potrebbero piacere le seguenti scarpe: \n Tod\'s \n Gucci \n Hogan");
// Ti potrebbero piacere le scarpe:
// Tod's
// Gucci
// Hogan
```

Nota bene che abbiamo usato solo una volta la funzione println() che serve a scrivere una nuova riga nella console di output.

Concatenare le stringhe

Quando si utilizza System.out.println() e si lavora con le stringhe in altri modi, è possibile unire due stringhe utilizzando l'operatore + cioè lo stesso operatore utilizzato per sommare dei numeri.

L'operatore + ha un significato diverso in relazione alle stringhe infatti non addiziona il contenuto delle stringhe ma lo unisce formando una nuova stringa, tale processo si chiama **concatenazione**. La concatenazione di stringhe è molto utile perché al posto di inserire l'intera stringa su una sola riga, rendendo più difficile la comprensione del programma, puoi usare l'operatore + per spezzare il testo su due o più righe del codice sorgente Java.

Ecco un esempio:

```
System.out.println("\"Siate affamati, siate folli.\"\n"
+ "\"Stay hungry, Stay foolish.\"\n"
+ "\t-- Steve Jobs, Apple\'s CEO");
```

In questo esempio puoi notare che ho usato diverse volte la concatenazione e il carattere speciale \ per riportare una famosa citazione di Steve Jobs.

Non puoi concatenare solo stringhe con l'operatore + ma anche altri tipi di variabili. Prendiamo in considerazione il codice seguente:

```
int ruote = 4;
char tipo = 'S';
System.out.println("Il tipo " + tipo + " ha " + ruote + "ruote.");
```

Questo esempio mostra un aspetto unico di come l'operatore + lavora con le stringhe. Può far sì che le variabili che non sono stringhe vengano trattate

come stringhe quando vengono visualizzate.

Il linguaggio Java offre questa funzionalità per facilitare la visualizzazione delle informazioni infatti il risultato di questo codice sarà:

Il tipo S ha 4 ruote.

Funzioni utili

Attenzione, potresti pensare che sia possibile comparare due stringhe tramite il doppio uguale (==), come abbiamo già visto per numeri e caratteri. In realtà questo è un errore comune in Java, è necessario usare il metodo equals() invocato su una delle due stringhe.

Immaginiamo di dover implementare un quiz:

```
String risposta = "Boxe";
System.out.println("In quale sport si è distinto Muhammad Ali?");
System.out.println("Hai risposto " + risposta);
System.out.println("La tua risposta è " + risposta.equals("Boxe"));
```

Il risultato di questo codice è:

```
In quale sport si è distinto Muhammad Ali?
Hai risposto Boxe
La tua risposta è true
```

Sarebbe meglio vero o falso piuttosto che un valore booleano ma vedremo a breve come usare dei costrutti che ovviano a questo problema.

Abbiamo parlato di metodo ma cos'è esattamente? Un metodo è un modo per eseguire un'attività in un programma Java. Il compito di questo metodo, ad esempio, è determinare se una stringa ha lo stesso valore di un'altra. Se le due variabili di tipo stringa hanno lo stesso valore, chiamando il metodo equals() viene visualizzato il valore booleano true. In caso contrario, viene visualizzato false.

Poiché siamo italiani e non ha senso dire che la risposta è true, risolviamo questo problema.

```
    boolean valoreRisp = risposta.equals("Boxe");
    if (valoreRisp == true) {
    System.out.println("La tua risposta è vera");
    }


if (valoreRisp == false) {
    System.out.println("La tua risposta è falsa");
    }
```

Come vedi, abbiamo assegnato il valore del confronto alla variabile valoreRisp e, successivamente, abbiamo confrontato il valore con true e false per stampare la frase appropriata. Abbiamo utilizzato anche un altro costrutto per comparare i valori ma lo vedremo nel dettaglio nel prossimo capitolo.

Può anche essere utile determinare la lunghezza di una stringa e puoi farlo con il metodo length().

Questo metodo funziona allo stesso modo del metodo equals(), a parte il fatto che è coinvolta solo una variabile di tipo stringa. Osserva il seguente esempio:

```
    String citta = "Bologna";
    int cittaLung = citta.length();
```

Questo esempio imposta cittaLung, una variabile intera con valore pari a 7. Il metodo length() conta il numero di caratteri nella variabile stringa denominata citta e memorizza questo conteggio nella variabile intera cittaLung.

Un altro compito comune quando si usano le stringhe è verificare se una stringa è contenuta all'interno di un'altra. Per effettuare questa verifica, usa il metodo indexOf() ed inserisci la stringa che stai cercando tra parentesi.

```
int posizione = testo.indexOf("Mi piace Bologna");
```

Se la stringa non viene trovata, indexOf() produce il valore -1 altrimenti indexOf() produce un numero intero che rappresenta la posizione in cui inizia la stringa. Le posizioni in una stringa sono numerate a partire da 0, che rappresenta il primo carattere nella stringa.

```
String testo = "Ho visitato Bologna e l'ho trovata stupenda. Mi piace Bologna e la sua cucina!";
int posizione = testo.indexOf("Mi piace Bologna");
System.out.println(posizione);
// 45
```

Se stai cercando una stringa all'interno di un'altra ma non ti interessa la posizione, il metodo contains() restituisce un valore booleano. Restituisce true se la stringa cercata viene trovata e false altrimenti.

Nota bene che i metodi indexOf() e contains() fanno distinzione tra maiuscole e minuscole, il che significa che cercano solo il testo in maiuscolo esattamente come la stringa di ricerca. Se la stringa contiene lo stesso testo ma in maiuscolo, indexOf() produce il valore –1 e contains() restituisce false.

7

TEST CONDIZIONALI

Quando si scrive un programma per computer, si fornisce al computer un elenco di istruzioni che vengono seguite alla lettera. Puoi dire al computer di elaborare alcune complesse formule matematiche e lui le risolverà.

Tuttavia, ci sono momenti in cui è necessario che il computer sia più selettivo su ciò che fa. Ad esempio, se hai scritto un programma per verificare il saldo del tuo conto corrente, è possibile che il computer visualizzi un avviso se il conto è stato chiuso. In caso contrario, il messaggio sarebbe inaccurato e addirittura sconvolgente.

Il modo per eseguire questa attività in un programma Java è utilizzare una **condizione**, un'istruzione che fa accadere qualcosa in un programma solo se viene soddisfatta una condizione specifica. Abbiamo già visto nel capitolo precedente come usare il condizionale if ma vedremo anche else e switch.

Quando un programma Java prende una decisione, lo fa impiegando un'**istruzione condizionale**. Puoi controllare la condizione nei tuoi programmi Java usando le parole chiave if, else, switch, case e break. Puoi anche usare gli operatori condizionali ==, !=, <, >, <=, >= e ?, insieme alle variabili booleane.

if

Il modo più semplice per testare una condizione in Java è usando un'istruzione if. Questa istruzione verifica se una condizione è vera o falsa e agisce solo se la condizione è vera. Si utilizza if insieme alla condizione da testare, come nella seguente istruzione:

```
long saldo = -450.00F;
if (saldo < 0) {
System.out.println("Saldo negativo!");
}
```

L'istruzione if verifica se la variabile saldo è inferiore a 0 utilizzando l'operatore minore di (<). In tal caso, il blocco all'interno dell'istruzione if viene eseguito, visualizzando un messaggio. Il blocco viene eseguito perché la condizione è vera. Nell'esempio precedente, se la variabile saldo ha un valore pari o superiore a 0, l'istruzione println() è ignorata.

Si noti che la condizione testata deve essere racchiusa tra parentesi, come in (saldo < 0).

Puoi usare diversi tipi di operatori con cui eseguire il confronto, potresti verificare se il saldo è proprio pari a 0 (saldo == 0), se è diverso da 0 (saldo != 0), se maggiore di 500 (saldo > 500) ecc.

Presta sempre attenzione perché l'operatore utilizzato per condurre test di uguaglianza ha due segni uguali: (==). È facile confondere questo operatore con l'operatore (=), che viene utilizzato per assegnare un valore ad una variabile. Usa sempre due segni uguali in un'istruzione condizionale.

if-else

Ci sono momenti in cui vuoi eseguire delle istruzioni se una condizione è vera ed eseguirne altre se la condizione è falsa. Puoi farlo usando l'istruzione else insieme all'istruzione if, che abbiamo già visto. Riprendiamo il codice del capitolo precedente:

```
boolean valoreRisp = risposta.equals("Boxe");


if (valoreRisp == true) {
    System.out.println("La tua risposta è vera");
    }
    else {
    System.out.println("La tua risposta è falsa");
    }
```

L'istruzione else non ha una condizione da verificare, a differenza dell'istruzione if. Questo avviene perché l'istruzione else è abbinata all'istruzione if che la precede immediatamente.

In questo caso i valori restituiti dal metodo equals() sono soltanto due: true o false. Alla luce di ciò, se il valore restituito non è true, sarà necessariamente false quindi è corretto usare un blocco else.

È possibile anche usare più condizioni if o else if per verificare altre condizioni. Allo stesso modo si possono innestare dei blocchi if:

```
if (val > 0) {
System.out.println("Il valore è maggiore di 0");
} else if (val < 0) {
System.out.println("Il valore è minore di 0");
```

```
} else {
System.out.println("Il valore è 0");
}
```

switch

Le istruzioni if e else sono utili per situazioni con due possibili condizioni, ma ci sono momenti in cui ne hai più di due. Hai già visto che le istruzioni if e else possono essere concatenate per gestire diverse condizioni ma un modo più efficace è con l'istruzione switch, che può verificare una varietà di condizioni diverse ed eseguire azione di conseguenza.

Immaginiamo di avere un cantiere in corso e vogliamo mostrare dei messaggi in base al valore della percentuale sull'avanzamento dei lavori:

```
int percentuale = 25;


switch (percentuale) {
    case 0:
    System.out.println("Non ancora iniziato.");
    break;
    case 25:
    System.out.println("Iniziato da poco");
    break;
    case 50:
    System.out.println("Siamo a metà!");
    break;
    case 75:
    System.out.println("Quasi completato!");
    break;
    case 100:
    System.out.println("Finito!");
    break;
    default:
```

```
System.out.println("Verifica il cantiere");
}
```

Ogni istruzione case verifica la variabile di test nell'istruzione switch rispetto ad un valore specifico. Il valore utilizzato in un'istruzione case può essere un carattere, un numero intero o una stringa. Nell'esempio precedente, ci sono diverse condizioni tutte di tipo numerico e ogni case ha due istruzioni che lo seguono. Quando una di queste istruzioni case corrisponde alla variabile valutata con switch, il computer esegue tutte le istruzioni dopo l'istruzione case fino a quando non incontra un'istruzione break.

In questo caso il risultato sarà:

Iniziato da poco

L'istruzione successiva è break, quindi non viene eseguita nessun'altra istruzione all'interno di switch. L'istruzione break dice al computer di uscire dal blocco switch. Ricorda di ponderare bene l'uso di break limitandolo a questi costrutti perché potrebbe portare a risultati indesiderati.

Queste parole chiave sono comunque preziose perché se non ci fossero, in questo esempio avresti ottenuto tutti i messaggi su console:

Iniziato da poco

Siamo a metà!

Quasi completato!

Finito!

Verifica il cantiere

L'istruzione default viene utilizzata come predefinita ovvero come se nessuna delle precedenti istruzioni case è vera. In questo esempio, prova a cambiare la percentuale con qualsiasi altro valore non previsto.

Non è necessario utilizzare un'istruzione default con il blocco switch ma è consigliato infatti quando viene omessa, non verrà eseguita nessuna

azione se nessuna delle istruzioni case è stata verificata.

Operatore ternario

L'istruzione condizionale più complicata in Java è l'operatore ternario contrassegnato dal simbolo ?. L'operatore ternario assegna un valore o visualizza un valore basandosi su una condizione. Ad esempio, considera un videogioco che imposta la variabile numeroNemici su uno dei due valori in base alla difficoltà. Puoi creare questo codice con un'istruzione if-else:

```
if (difficolta > 5) {
numeroNemici = 20;
} else {
numeroNemici = 10;
}
```

Lo stesso identico codice è possibile scriverlo con l'operatore ternario in modo molto più conciso e snello:

```
numeroNemici = difficolta > 5 ? 20 : 10;
```

Un'espressione ternaria è composta da cinque parti:

- Condizione da testare;
- Operatore (?);
- Valore da usare in caso la condizione sia vera;
- Due punti (:);
- Valore da usare in caso la condizione sia falsa.

L'operatore ternario può essere utile ma è anche il più difficile in Java per i principianti.

8

CICLI

Da piccolo ero un bambino vivace quindi una delle punizioni più usate consisteva nel farmi scrivere diverse volte una frase, un po' come nei Simpsons. Questa punizione può funzionare con i bambini ma sarebbe completamente inutile per un computer.

Un computer può ripetere un'attività come questa con estrema facilità infatti i programmi sono ideali per fare sempre la stessa operazione a causa dei cicli. Un **ciclo** è un'istruzione o un blocco di istruzioni che si ripete in un programma.

Alcuni cicli vengono eseguiti un determinato numero di volte, altri possono essere infiniti. Esistono tre istruzioni di ciclo in Java: for, do-while e while. Potrai usare ognuno di questi in modo equivalente ma è utile imparare come funzionano tutti e tre. Spesso puoi semplificare una sezione del ciclo di un programma scegliendo quello più appropriato.

for

Durante la programmazione trovi molte circostanze in cui un ciclo è utile. Puoi usarli per continuare a fare qualcosa più e più volte, ad esempio quando un antivirus effettua una scansione di tutti i file presenti nel computer. L'istruzione più complessa per creare dei cicli di Java è for, che essenzialmente ripete una sezione di un programma un numero fisso di volte. La sua complessità è data dalla sua forma:

```
for (int i = 0; i < 1000; i++) {
if (i % 11 == 0) {
System.out.println(i);
}
}
```

Questo ciclo mostra ogni numero compreso tra 0 e 999 che è divisibile per 11. Un ciclo for ha una variabile che determina quando il ciclo deve iniziare e finire. Questa variabile è chiamata **contatore** (o indice). Il contatore nel ciclo precedente è la variabile i.

Un ciclo for è composto da tre parti tra parentesi che seguono la parola chiave for: inizializzazione, condizione e modifica del contatore. Queste sezioni sono separate dal punto e virgola (;).

L'esempio illustra queste tre sezioni:

- La sezione di inizializzazione: nella prima parte, alla variabile i viene assegnato un valore iniziale pari a 0;
- La sezione per la condizione: nella seconda parte, esiste un test condizionale come quello che potresti usare in un if: i < 1000;

- La sezione per la modifica del contatore: la terza parte è un'istruzione che modifica il valore della variabile i, in questo esempio utilizzando l'operatore di incremento.

Nella sezione di inizializzazione, si imposta la variabile contatore ed è possibile usare una variabile già esistente o creare la variabile nell'istruzione for, come nell'esempio precedente con la variabile i.

La seconda sezione contiene un test che deve rimanere vero affinché il ciclo possa continuare. Quando questa condizione diventa falsa, il ciclo termina. In questo esempio, il ciclo termina quando la variabile i è uguale o maggiore di 1000.

L'ultima sezione dell'istruzione for contiene un'istruzione che modifica il valore della variabile contatore. Questa istruzione viene eseguita ogni volta quindi ad ogni **iterazione**. Il contatore deve cambiare in qualche modo altrimenti il ciclo non finirebbe mai. Nell'esempio, i viene incrementata di un’unità nella sezione di modifica del valore. Se non cambiasse, rimarrebbe al suo valore originale di 0 e la condizione i<1000 sarebbe sempre vera.

while

Il ciclo while è più semplice di un ciclo for, l'unica cosa di cui ha bisogno è una condizione che accompagni l'istruzione while. Creiamo lo stesso codice precedente usando un while:

```
int i = 0;
while (i < 1000) {
if (i % 11 == 0) {
System.out.println(i);
}
i++;
}
```

Questo ciclo continua a ripetersi fino a quando la variabile i non è più minore di 1000. L'istruzione while verifica la condizione all'inizio del ciclo prima di eseguire qualunque istruzione nel ciclo.

Quando il programma incontra un ciclo while, se la condizione è falsa, le istruzioni all'interno del ciclo vengono ignorate. Se la condizione while è vera, il ciclo viene eseguito una volta e verifica nuovamente la condizione while. Se la condizione testata non cambia mai all'interno del ciclo, si creerà un ciclo infinito.

do-while

Il ciclo do-while è simile al ciclo while ma la verifica della condizione viene eseguita alla fine quindi le istruzioni vengono eseguite almeno una volta. Creiamo lo stesso codice precedente usando un do-while:

```
int i = 0;
do {
if (i % 11 == 0) {
System.out.println(i);
}
i++;
} while (i < 1000);
```

Come per il ciclo while, questo ciclo continua fino a quando la variabile i è pari o superiore a 1000. Il ciclo do-while è diverso perché il test condizionale viene eseguito dopo le istruzioni all'interno del ciclo, anziché prima di esse.

Quando viene raggiunto il blocco do per la prima volta durante l'esecuzione di un programma, le istruzioni tra il do e while vengono eseguite automaticamente, successivamente viene testata la condizione while per determinare se il ciclo deve essere ripetuto.

Se la condizione while è vera, il ciclo viene ripetuto ancora una volta; se la condizione è falsa, il ciclo termina. Qualcosa deve accadere all'interno delle istruzioni do e while affinché la condizione testata con while restituisca false, altrimenti il ciclo continuerà infinitamente.

È importante ricordare che le istruzioni all'interno di un ciclo do-while vengono sempre eseguite almeno una volta.

break e continue

Il classico modo per uscire da un ciclo è che la condizione testata diventi falsa e questo è vero per tutti e tre i tipi di cicli in Java.

Tuttavia, ci sono dei momenti in cui vuoi che un ciclo termini immediatamente, anche se la condizione da testare è ancora vera.

Puoi farlo con l'istruzione break, come mostrato:

```
int i = 0;
while (i <= 1000) {
i = i + 5;
if (i == 400) break;
}
```

Un'istruzione break termina il ciclo che contiene l'istruzione. In questo esempio, il while è progettato per eseguire il ciclo fino a quando la variabile i è minore di 1000, tuttavia, un caso speciale fa in modo che il ciclo termini prima. Se il valore di i è uguale a 400, viene eseguita l'istruzione break, che termina immediatamente il ciclo. All'uscita di questo ciclo il valore di i è esattamente 400.

Un'altra istruzione speciale che è possibile utilizzare all'interno di un ciclo è continue. L'istruzione continue fa sì che il ciclo esca dall'iterazione corrente passando all'elemento successivo. Vediamo come usarlo:

```
int i = 0;
while (i <= 1000) {
i = i + 100;
if (i == 400) continue;
System.out.println("Il valore di i è " + i);
}
```

L’output di questo programma è il seguente:

Il valore di i è 100
Il valore di i è 200
Il valore di i è 300
Il valore di i è 500
Il valore di i è 600
Il valore di i è 700
Il valore di i è 800
Il valore di i è 900
Il valore di i è 1000
Il valore di i è 1100

All'interno di questo ciclo, le istruzioni vengono eseguite normalmente a meno che il valore della variabile i non sia uguale a 400. In tal caso, l'istruzione continue fa ritornare il ciclo all'istruzione while anziché procedere normalmente con l'istruzione System.out.println().

A causa dell'istruzione continue, il ciclo non visualizza mai il seguente testo:

Il valore di i è 400

È possibile utilizzare le istruzioni break e continue con tutti e tre i tipi di cicli.

9

ARRAY

Un array è un gruppo di variabili correlate che condividono lo stesso tipo. Qualsiasi tipo di informazione che può essere memorizzata come variabile può essere memorizzata come elemento in un array.

Gli array possono essere utilizzati per tenere traccia di tipi di informazioni più sofisticati rispetto a una singola variabile ma sono altrettanto facili da creare e manipolare. In sostanza si tratta di variabili raggruppate sotto un nome comune.

Come le variabili, gli array vengono creati dichiarando il tipo di variabile e il nome. Una coppia di parentesi quadre ([]) segue il tipo per distinguerli dalle variabili. È possibile creare array per qualsiasi tipo di informazione che può essere memorizzata come variabile. La seguente istruzione crea un array di variabili di tipo stringa:

String**[]** nomi**;**

Ecco due istruzioni che creano rispettivamente array di numeri interi e valori booleani:

int**[]** voti**;**

boolean**[]** presenze**;**

Negli esempi precedenti abbiamo creato variabili per contenere array ma non memorizzano alcun valore. Per memorizzare dei valori, è possibile utilizzare la parola chiave new insieme al tipo di variabile o archiviare i valori nell'array tra parentesi graffe.

Quando si utilizza new, è necessario specificare quanti oggetti diversi saranno memorizzati nell'array. Ogni oggetto in un array è chiamato **elemento**. La seguente istruzione crea un array e alloca lo spazio per i valori che contiene:

int**[]** voti = **new** int**[**200**];**

Questo esempio crea un array di numeri interi chiamati voti. L'array ha 200 elementi che possono memorizzare i voti in matematica di tutti gli studenti di un istituto.

Quando si crea un array con l'istruzione new, è necessario specificare il numero di elementi. A ciascun elemento dell'array viene assegnato un valore iniziale che dipende dal tipo dell'array.

Tutti gli array numerici hanno come valore iniziale 0, gli array di caratteri sono inizializzati con "\0" e gli array booleani hanno il valore false. Un array di tipo String e tutti gli altri oggetti vengono creati con il valore iniziale null.

Per array che non sono estremamente grandi, è possibile impostare i loro valori iniziali nel momento della creazione. L'esempio seguente crea un array di stringhe ed assegna i valori iniziali:

String**[]** nomi = **{**"Antonio"**,** "Bea"**,** "Filippo"**,** "Marco"**,** "Vincenzo"**};**

Le informazioni che devono essere archiviate negli elementi dell'array vengono inserite tra parentesi graffe ({) e (}) con virgole che separano ciascun elemento. Il numero di elementi nell'array è impostato dal numero di elementi presenti nell'elenco e separati da virgole.

Gli elementi dell'array sono numerati, a partire da 0 che indica il primo elemento. È possibile accedere ad un elemento specifico facendo riferimento a questo numero tra parentesi quadre ([) e (]).

L'istruzione precedente può essere riscritta con il seguente codice:

```
String[] nomi = new String[5];
nomi[0] = "Antonio";
nomi[1] = "Bea";
nomi[2] = "Filippo";
nomi[3] = "Marco";
nomi[4] = "Vincenzo";
```

Ogni elemento dell'array deve essere dello stesso tipo. In questo caso, viene utilizzata una stringa per contenere il nome dei membri di una squadra. Dopo aver creato l'array, non è più possibile creare elementi. Il compilatore Java non consentirà l'accesso ad altri elementi perché è stato riservato uno spazio di memoria solo per cinque elementi in questo caso.

Possiamo verificare quanto esposto inserendo un altro elemento all'array:

```
nomi[5] = "Mirko";
```

Proviamo ad eseguire il programma e otterremo il seguente errore:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
    at Main.main(Main.java:9)
    exit status 1
```

La parola "eccezione" è un sinonimo di errore nei programmi Java.

In questo caso, il compilatore ci sta informando che siamo andati oltre la dimensione prestabilita per l'array quindi l'indice 5 per questo array non

è valido.

Se si desidera controllare il limite superiore di un array in modo da evitare un'eccezione di questo tipo, possiamo usare la variabile chiamata length che è associata ad ogni array. Questa variabile è un numero intero che è pari al numero di elementi contenuti in un array. L'esempio seguente crea un array e ne riporta la lunghezza:

```
String[] nomi = {"Antonio", "Bea", "Filippo", "Marco", "Vincenzo"};


System.out.println("Ci sono " + nomi.length + " nomi nell'array.");
```

In questo caso il risultato sarà:

Ci sono 5 nomi nell'array.

Puoi lavorare con il testo in Java usando una stringa o un array di caratteri. Quando lavori con le stringhe, può essere utile inserire ogni carattere di una stringa come elemento di un array di caratteri. Per fare ciò, puoi invocare il metodo toCharArray() della stringa, che produce un array di caratteri con lo stesso numero di elementi della lunghezza della stringa.

Vediamo un esempio con questo metodo per inserire due spazi tra ogni parola del testo:

```
String testo = "Ho visitato Bologna e l'ho trovata stupenda. Mi piace Bologna e la sua cucina!";
char[] arr = testo.toCharArray();


for (int i = 0; i < arr.length; i++) {
    char corrente = arr[i];
    if (corrente != ' ') System.out.print(corrente);
    else System.out.print("  ");
```

```
    }

System.out.println();
```

In questo esempio iteriamo sugli elementi dell'array restituito dalla funzione toCharArray() e lo confrontiamo con il carattere "spazio vuoto". Se il valore corrente non è uno spazio vuoto lo scriviamo sulla console di output, altrimenti scriviamo una stringa contenente due spazi vuoti nella console di output.

Potremmo utilizzare l'operatore ternario per semplificare ulteriormente questo blocco if-else:

```
String testo = "Ho visitato Bologna e l'ho trovata stupenda. Mi piace Bologna e la sua cucina!";
char[] arr = testo.toCharArray();

for (int i = 0; i < arr.length; i++) {
    char corrente = arr[i];
    System.out.print(corrente != ' ' ? corrente : " ");
    }

System.out.println();
```

Array multidimensionali

Gli array visti sino ad ora hanno tutti una sola dimensione quindi è possibile recuperare un elemento utilizzando un singolo numero. Alcuni tipi di informazioni richiedono più dimensioni per essere archiviate come array, ad esempio i punti in un sistema di coordinate (x, y). Una dimensione dell'array può memorizzare la coordinata x e l'altra dimensione può memorizzare la coordinata y.

Per creare un array con due dimensioni, è necessario utilizzare un altro paio di parentesi quadre durante la creazione e l'utilizzo dell'array.

Supponiamo di voler creare una griglia per il gioco della dama in Java:

```
String[][] griglia = new String[8][8];


//Inizializziamo ogni cella
    for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
    griglia[i][j] = " ";
    }
    }


//Inizializziamo le celle con pedine bianche
    for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 8; j++) {
    griglia[i][j] = "B";
    }
    }
```

```
//Inizializziamo le celle con pedine nere
    for (int i = 6; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
    griglia[i][j] = "N";
    }
    }


//Mostro la griglia
    for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
    System.out.print(griglia[i][j] + " ");
    }


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

Il risultato di questo codice è una schiera di pedine bianche da una parte e una schiera di pedine nere dall'altra:

B B B B B B B B
B B B B B B B B
N N N N N N N N
N N N N N N N N

L'uso di array multidimensionali segue gli standard visti finora per le variabili e gli array ma possono risultare leggermente più complessi, tuttavia, con la pratica vedrai che ben presto saprai usarli in modo eccellente.

10

CLASSI

La classe è al centro di Java, è il costrutto logico su cui è costruito l'intero linguaggio perché definisce la forma e la natura di un oggetto. In quanto tale, la classe costituisce la base per la programmazione orientata agli oggetti in Java. Qualsiasi concetto che desideri implementare in un programma Java deve essere incapsulato all'interno di una classe.

Poiché la classe è così fondamentale per Java, questo e i prossimi capitoli saranno dedicati ad essa. Questo capitolo è un'introduzione agli elementi di base di una classe e approfondiremo come una classe può essere utilizzata per creare oggetti. Imparerai anche i metodi, i costruttori e la parola chiave this.

Forse la cosa più importante da capire su una classe è che definisce un nuovo tipo di dati. Una volta definito, questo nuovo tipo può essere utilizzato per creare oggetti di quel tipo. Pertanto, una classe è un modello per un oggetto e un oggetto è un'istanza di una classe. Poiché un oggetto è un'istanza di una classe, vedrai spesso le due parole oggetto e istanza usate in modo intercambiabile.

Quando si definisce una classe, ne dichiari la forma e la natura in modo esatto. Puoi farlo specificando i dati che contiene e il codice che opera su quei dati. Se le classi molto semplici possono contenere solo codice o solo dati, la maggior parte delle classi del mondo reale contiene entrambi. Come vedrai, il codice di una classe definisce l'interfaccia con i suoi dati. Una classe viene dichiarata mediante l'uso della parola chiave class. Una forma generale semplificata di una definizione di classe è mostrata qui:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
```

```
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

I dati, o variabili, definiti all'interno di una classe sono chiamati variabili di istanza. Il codice è contenuto all'interno dei metodi. Collettivamente, i metodi e le variabili definiti all'interno di una classe sono chiamati membri della classe. Nella maggior parte delle classi, vengono gestite le variabili di istanza e vi si accede dai metodi definiti per quella classe. Pertanto, come regola generale, sono i metodi che determinano come possono essere utilizzati i dati di una classe. Le variabili definite all'interno di una classe sono chiamate variabili di istanza perché ogni istanza della classe (ovvero ogni oggetto della classe) contiene la propria copia di queste variabili. Pertanto, i dati per un oggetto sono separati e univoci dai dati per un altro.

Tutti i metodi hanno la stessa forma generale di main(), tuttavia, la maggior parte dei metodi non verrà specificata come static o pubblic. Si noti che la forma generale di una classe non specifica un metodo main(). Le classi Java non hanno bisogno di avere un metodo main(), ne specifichi uno solo se quella classe è il punto di partenza per il tuo programma. Inoltre,

alcuni tipi di applicazioni Java, come le applet, non richiedono affatto un metodo main().

I programmatori C++ noteranno che la dichiarazione di classe e l'implementazione dei metodi sono archiviate nello stesso luogo e non sono definite separatamente. Questo a volte crea file .java molto grandi, poiché qualsiasi classe deve essere interamente definita in un unico file sorgente. Questa funzionalità di progettazione è stata incorporata in Java perché si riteneva che, a lungo termine, avere specifiche, dichiarazioni e implementazioni in un unico posto rendesse il codice più facile da mantenere.

Un semplice esempio

Iniziamo il nostro studio della classe con un semplice esempio. Ecco una classe chiamata Box che definisce tre variabili di istanza: larghezza, altezza e profondità. Attualmente, Box non contiene alcun metodo (ma alcuni verranno aggiunti a breve).

```
class Box {
double width;
double height;
double depth;
}
```

Come indicato, una classe definisce un nuovo tipo di dati. In questo caso, il nuovo tipo di dati è chiamato Box, utilizzerai questo nome per dichiarare oggetti di tipo Box. È importante ricordare che una dichiarazione di classe crea solo un modello; non crea un oggetto reale. Pertanto, il codice precedente non determina l'esistenza di alcun oggetto di tipo Box. Per creare effettivamente un oggetto Box, utilizzerai un'istruzione come la seguente:

```
Box mybox = new Box(); // create a Box object called mybox
```

Dopo l'esecuzione di questa istruzione, mybox sarà un'istanza di Box, pertanto, avrà una realtà "fisica". Per il momento, non preoccuparti dei dettagli di questa affermazione. Come accennato in precedenza, ogni volta che crei un'istanza di una classe, stai creando un oggetto che contiene la propria copia di ciascuna variabile di istanza definita dalla classe. Pertanto, ogni oggetto Box conterrà le proprie copie delle variabili di istanza relative

a larghezza, altezza e profondità. Per accedere a queste variabili, utilizzerai l'operatore punto (.). L'operatore punto collega il nome dell'oggetto con il nome di una variabile di istanza. Ad esempio, per assegnare alla variabile larghezza di mybox il valore 100, dovresti utilizzare la seguente istruzione:

```
mybox.width = 100;
```

Questa istruzione dice al compilatore di assegnare alla copia di width che è contenuta all'interno dell'oggetto mybox il valore di 100. In generale, si utilizza l'operatore punto per accedere sia alle variabili di istanza che ai metodi all'interno di un oggetto. Sebbene comunemente indicato come operatore punto, la specifica formale per Java classifica il . come un separatore. Tuttavia, poiché l'uso del termine "operatore punto" è molto diffuso, useremo tale terminologia in questo libro. Ecco un programma completo che utilizza la classe Box:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
```

```
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Dovresti chiamare il file che contiene questo programma BoxDemo.java, perché il metodo main() è nella classe chiamata BoxDemo, non nella classe chiamata Box. Quando compili questo programma, scoprirai che sono stati creati due file .class, uno per Box e uno per BoxDemo. Il compilatore Java inserisce automaticamente ogni classe nel proprio file .class e non è necessario che sia la classe Box che la classe BoxDemo si trovino effettivamente nello stesso file sorgente. Puoi inserire ogni classe nel proprio file, chiamati rispettivamente Box.java e BoxDemo.java. Per eseguire questo programma, è necessario eseguire BoxDemo.class. Quando lo esegui, vedrai il seguente output:

```
Volume is 3000.0
```

Come affermato in precedenza, ogni oggetto ha le proprie copie delle variabili di istanza. Ciò significa che se hai due oggetti Box, ognuno ha la propria copia di depth, width e height. È importante comprendere che le modifiche alle variabili di istanza di un oggetto non hanno effetto sulle variabili di istanza di un altro. Ad esempio, il seguente programma dichiara due oggetti Box:

```
// This program declares two Box objects.
class Box {
```

```
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

L'output prodotto da questo programma è mostrato qui:

```
Volume is 3000.0
```

Volume is 162.0

Come puoi vedere, i dati di mybox1 sono completamente separati dai dati contenuti in mybox2.

Dichiarare un oggetto

Come appena spiegato, quando crei una classe, stai creando un nuovo tipo di dati. Ottenere oggetti di una classe è un processo composto da due fasi. Innanzitutto, devi dichiarare una variabile del tipo di classe. Questa variabile non definisce un oggetto. È semplicemente una variabile che può fare riferimento a un oggetto. In secondo luogo, è necessario acquisire una copia fisica effettiva dell'oggetto e assegnarla a quella variabile. Puoi farlo usando l'operatore new. L'operatore new alloca dinamicamente (ovvero alloca in fase di esecuzione) la memoria per un oggetto e restituisce un riferimento ad esso. Questo riferimento è, più o meno, l'indirizzo in memoria dell'oggetto allocato da new, quindi viene memorizzato nella variabile. Pertanto, in Java, tutti gli oggetti di classe devono essere allocati dinamicamente. Diamo un'occhiata ai dettagli di questa procedura. Nei programmi di esempio precedenti viene utilizzata una riga simile alla seguente per dichiarare un oggetto di tipo Box:

```
Box mybox = new Box();
```

Questa dichiarazione combina i due passaggi appena descritti. Può essere riscritto in questo modo per mostrare ogni passaggio in modo più chiaro:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

La prima riga dichiara mybox come riferimento a un oggetto di tipo Box. Dopo l'esecuzione di questa riga, mybox contiene il valore null, che indica che non punta ancora a un oggetto reale. Qualsiasi tentativo di utilizzare mybox a questo punto risulterà in un errore in fase di compilazione. La riga

successiva assegna un oggetto reale e assegna un riferimento ad esso a mybox. Dopo l'esecuzione della seconda riga, puoi utilizzare mybox come se fosse un oggetto Box. Ma in realtà, mybox contiene semplicemente l'indirizzo di memoria dell'oggetto Box effettivo.

Coloro che hanno familiarità con C/C++ hanno probabilmente notato che i riferimenti agli oggetti sembrano essere simili ai puntatori. Questo sospetto è, essenzialmente, corretto. Un riferimento a un oggetto è simile a un puntatore di memoria. La differenza principale, e la chiave per la sicurezza di Java, è che non puoi manipolare i riferimenti come puoi fare con i puntatori effettivi. Pertanto, non è possibile fare in modo che un riferimento a un oggetto punti a una posizione di memoria arbitraria o manipolarlo come un numero intero.

Come appena spiegato, l'operatore new alloca dinamicamente la memoria per un oggetto. Ha questa forma generale:

class-var = **new** classname **( );**

Qui, class-var è una variabile del tipo di classe che viene creato. Il classname è il nome della classe di cui viene creata un'istanza. Il nome della classe seguito da parentesi specifica il costruttore per la classe. Un costruttore definisce cosa succede quando viene creato un oggetto di una classe. I costruttori sono una parte importante di tutte le classi e hanno molti attributi significativi. La maggior parte delle classi del mondo reale definisce esplicitamente i propri costruttori all'interno della definizione di classe. Tuttavia, se non viene specificato alcun costruttore esplicito, Java fornirà automaticamente un costruttore predefinito. Questo è il caso di Box. Per ora, useremo il costruttore predefinito. Presto vedrai come definire i tuoi costruttori.

A questo punto, ti starai chiedendo perché non è necessario utilizzare new per cose come numeri interi o caratteri. La risposta è che i tipi primitivi

di Java non sono implementati come oggetti, sono implementati come variabili "normali". Questo è stato fatto per garantire l'efficienza. Come vedrai, gli oggetti hanno molte caratteristiche e attributi che richiedono a Java di trattarli in modo diverso rispetto ai tipi primitivi. Non applicando lo stesso sovraccarico ai tipi primitivi che si applica agli oggetti, Java può implementare i tipi primitivi in modo più efficiente. Successivamente, vedrai le versioni degli oggetti dei tipi primitivi che sono disponibili per il tuo uso in quelle situazioni in cui sono necessari oggetti completi di questi tipi.

È importante comprendere che new alloca memoria per un oggetto durante il runtime. Il vantaggio di questo approccio è che il tuo programma può creare tutti gli oggetti necessari durante l'esecuzione del tuo programma. Tuttavia, poiché la memoria è finita, è possibile che new non sia in grado di allocare memoria per un oggetto perché la memoria è insufficiente. In questo caso, si verificherà un'eccezione in fase di esecuzione. Per i programmi di esempio in questo libro, non dovrai preoccuparti della memoria, ma dovrai considerare questa possibilità nei programmi del mondo reale.

Esaminiamo ancora una volta la distinzione tra una classe e un oggetto. Una classe crea un nuovo tipo di dati che può essere utilizzato per creare oggetti cioè, una classe crea una struttura logica che definisce la relazione tra i suoi membri. Quando dichiari un oggetto di una classe, stai creando un'istanza di quella classe. Quindi, una classe è un costrutto logico. Un oggetto ha una realtà fisica cioè, un oggetto occupa spazio nella memoria. È importante tenere ben presente questa distinzione.

Le variabili di riferimento dell'oggetto agiscono in modo diverso rispetto a quanto ci si potrebbe aspettare quando viene eseguita un'assegnazione. Ad esempio, cosa pensi che faccia il seguente codice?

```
Box b1 = new Box();
Box b2 = b1;
```

Si potrebbe pensare che a b2 venga assegnato un riferimento a una copia dell'oggetto a cui fa riferimento b1. Cioè, potresti pensare che b1 e b2 si riferiscano a oggetti separati e distinti. Tuttavia, questo sarebbe sbagliato. Dopo l'esecuzione di questo frammento, b1 e b2 faranno entrambi riferimento allo stesso oggetto. L'assegnazione di b1 a b2 non ha allocato memoria né copiato alcuna parte dell'oggetto originale. Fa semplicemente in modo che b2 si riferisca allo stesso oggetto di b1. Pertanto, qualsiasi modifica apportata all'oggetto tramite b2 influenzerà l'oggetto a cui si riferisce b1, poiché sono lo stesso oggetto. Sebbene b1 e b2 si riferiscano entrambi allo stesso oggetto, non sono collegati in nessun altro modo. Ad esempio, una successiva assegnazione a b1 scollegherà semplicemente b1 dall'oggetto originale senza influenzare l'oggetto o influenzare b2. Per esempio:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Qui b1 è stato impostato su null, ma b2 punta ancora all'oggetto originale. Ricorda che quando si assegna una variabile di riferimento oggetto a un'altra variabile di riferimento oggetto, non si crea una copia dell'oggetto, si esegue solo una copia del riferimento.

Metodi

Come accennato all'inizio di questo capitolo, le classi di solito consistono in due parti: variabili di istanza e metodi. L'argomento dei metodi è ampio perché Java offre loro così tanta potenza e flessibilità. Ci sono alcuni fondamenti che devi imparare ora in modo da poter iniziare ad aggiungere metodi alle tue classi. Questa è la forma generale di un metodo:

```
type name(parameter-list) {
// body of method
}
```

Qui, type specifica il tipo di dati restituiti dal metodo. Può essere qualsiasi tipo valido, inclusi i tipi di classe creati dall'utente. Se il metodo non restituisce un valore, il tipo restituito deve essere void. Il nome del metodo è specificato per name, può essere qualsiasi identificatore valido diverso da quelli già utilizzati da altri elementi nell'ambito di applicazione corrente. L'elenco dei parametri è una sequenza di coppie di tipi e identificatori separate da virgole. I parametri sono essenzialmente variabili che ricevono il valore degli argomenti passati al metodo quando viene chiamato. Se il metodo non ha parametri, l'elenco dei parametri sarà vuoto. I metodi che hanno un tipo restituito diverso da void restituiscono un valore alla routine chiamante utilizzando la seguente forma dell'istruzione return:

```
return value;
```

Qui, value è il valore restituito. Nelle prossime sezioni vedrai come creare vari tipi di metodi, inclusi quelli che accettano parametri e quelli che restituiscono valori. Sebbene sia perfettamente corretto creare una classe

che contenga solo dati, ciò accade raramente. Nella maggior parte dei casi, utilizzerai i metodi per accedere alle variabili di istanza definite dalla classe. In effetti, i metodi definiscono l'interfaccia per la maggior parte delle classi. Ciò consente all'implementatore della classe di nascondere il design specifico delle strutture dati interne dietro astrazioni di metodi più pulite ed eleganti. Oltre a definire metodi che forniscono accesso ai dati, puoi anche definire metodi utilizzati internamente dalla classe stessa. Iniziamo aggiungendo un metodo alla classe Box.

Potrebbe esserti venuto in mente guardando i programmi precedenti che il calcolo del volume di una scatola era qualcosa che veniva gestito meglio dalla classe Box piuttosto che dalla classe BoxDemo. Dopotutto, poiché il volume di una scatola dipende dalle dimensioni della scatola stessa, ha senso che la classe Box lo calcoli. Per fare ciò, devi aggiungere un metodo a Box, come mostrato qui:

```
// This program includes a method inside the box class.
class Box {
double width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
```

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

Questo programma genera il seguente output, che è lo stesso della versione precedente.

```
Volume is 3000.0
Volume is 162.0
```

Guarda attentamente le due righe di codice:

```
mybox1.volume();
mybox2.volume();
```

La prima riga qui richiama il metodo volume() su mybox1. Cioè, chiama volume() relativo all'oggetto mybox1, usando il nome dell'oggetto seguito dall'operatore punto. Pertanto, la chiamata a mybox1.volume() visualizza il

volume della casella definita da mybox1 e la chiamata a mybox2.volume() visualizza il volume della casella definita da mybox2. Ogni volta che volume() viene richiamato, viene visualizzato il volume per la casella specificata. Se non si ha familiarità con il concetto di chiamare (o invocare) un metodo, chiariamo subito la questione.

Quando viene eseguito mybox1.volume(), il sistema di runtime Java trasferisce il controllo al codice definito all'interno di volume(). Dopo che le istruzioni all'interno di volume() sono state eseguite, il controllo viene restituito alla routine chiamante e l'esecuzione riprende con la riga di codice successiva alla chiamata. Nel senso più generale, un metodo è il modo in cui Java implementa le subroutine.

C'è qualcosa di molto importante da notare all'interno del metodo volume(): le variabili di istanza sono riferite direttamente, senza precederle con il nome di un oggetto o con l'operatore punto. Quando un metodo utilizza una variabile di istanza definita dalla sua classe, lo fa in modo diretto, senza riferimento esplicito a un oggetto e senza l'uso dell'operatore punto. Questo è facile da capire se ci pensi. Un metodo viene sempre invocato in relazione a qualche oggetto della sua classe. Una volta che si è verificata questa chiamata, l'oggetto è noto. Pertanto, all'interno di un metodo, non è necessario specificare l'oggetto una seconda volta. Ciò significa che width, height e depth all'interno di volume() si riferiscono implicitamente alle copie di quelle variabili trovate nell'oggetto che richiama volume().

Esaminiamo meglio: quando si accede a una variabile di istanza tramite codice che non fa parte della classe in cui è definita quella variabile di istanza, è necessario farlo tramite un oggetto, utilizzando l'operatore punto. Tuttavia, quando si accede a una variabile di istanza tramite codice che fa parte della stessa classe della variabile di istanza, è possibile fare riferimento direttamente a tale variabile. La stessa cosa vale per i metodi.

Ritornare un valore

Sebbene l'implementazione di volume() sposti il calcolo del volume di una scatola all'interno della classe Box a cui appartiene, non è il modo migliore per farlo. Ad esempio, cosa succede se un'altra parte del programma volesse conoscere il volume di una scatola, ma non visualizzarne il valore? Un modo migliore per implementare volume() consiste nel calcolare il volume della casella e restituire il risultato al chiamante. L'esempio seguente, una versione migliorata del programma precedente, fa proprio questo:

```
// Now, volume() returns the volume of a box.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
```

```
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Come puoi vedere, quando viene chiamato volume(), viene messo sul lato destro di un'istruzione di assegnazione. Sulla sinistra c'è una variabile, in questo caso vol, che riceverà il valore restituito da volume(). Così, dopo aver eseguito:

```
vol = mybox1.volume();
```

il valore di mybox1.volume() è 3.000 e questo valore viene quindi memorizzato in vol.

Ci sono due cose importanti da capire sulla restituzione dei valori:

- Il tipo di dati restituiti da un metodo deve essere compatibile con il tipo restituito specificato dal metodo. Ad esempio, se il tipo restituito di un metodo è booleano, non è possibile restituire un numero intero;

- Anche la variabile che riceve il valore restituito da un metodo (come vol, in questo caso) deve essere compatibile con il tipo restituito specificato per il metodo.

Un altro punto: il programma precedente può essere scritto in modo un po' più efficiente perché in realtà non c'è bisogno della variabile vol. La chiamata a volume() avrebbe potuto essere utilizzata direttamente nell'istruzione println(), come mostrato qui:

```
System.out.println("Volume is" + mybox1.volume());
```

In questo caso, quando viene eseguito println(), mybox1.volume() verrà chiamato in modo automatico e il suo valore verrà passato a println().

Mentre alcuni metodi non necessitano di parametri, la maggior parte ne ha bisogno per poter ricevere qualcosa in input, elaborarlo e restituire un dato di output. I parametri consentono di generalizzare un metodo. Cioè, un metodo parametrizzato può operare su una varietà di dati e/o essere utilizzato in un numero maggiore di situazioni. Per illustrare questo aspetto, utilizziamo un esempio molto semplice. Ecco un metodo che restituisce il quadrato del numero 10:

```
int square()
{
return 10 * 10;
}
```

Sebbene questo metodo restituisca effettivamente il valore di 10 al quadrato, il suo utilizzo è molto limitato. Tuttavia, se modifichi il metodo in modo che prenda in input un parametro, come mostrato di seguito, puoi rendere square() molto più utile.

```
int square(int i)
{
return i * i;
}
```

Ora, square() restituirà il quadrato di qualsiasi valore con cui viene chiamato. Cioè, square() è ora un metodo generico in grado di calcolare il quadrato di qualsiasi valore intero, anziché solo del numero 10. Ecco un esempio:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

Nella prima chiamata a square(), il valore 5 verrà passato al parametro i. Nella seconda chiamata, riceverà il valore 9. La terza invocazione passa il valore di y, che è 2 in questo esempio. Come mostrano questi esempi, square() è in grado di restituire il quadrato di tutti i dati passati in input. È importante mantenere i concetti di parametro e argomento. Un parametro è una variabile definita da un metodo che riceve un valore quando il metodo viene chiamato. Ad esempio, in square(), i è un parametro. Un argomento è un valore che viene passato a un metodo quando viene richiamato. Ad esempio, square(100) passa 100 come argomento. All'interno di square(), il parametro i riceve quel valore. È possibile utilizzare un metodo parametrizzato per migliorare la classe Box. Negli esempi precedenti, le dimensioni di ciascuna scatola dovevano essere impostate separatamente utilizzando una sequenza di istruzioni, come ad esempio:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

Sebbene questo codice funzioni, è preoccupante per due motivi. In primo luogo, è goffo e soggetto a errori. Ad esempio, sarebbe facile dimenticare di impostare una delle dimensioni. In secondo luogo, nei programmi Java ben progettati, è necessario accedere alle variabili di istanza solo tramite metodi definiti dalla loro classe. In futuro, puoi modificare il comportamento di un metodo, ma non puoi modificare il comportamento di una variabile di istanza esposta. Pertanto, un approccio migliore all'impostazione delle dimensioni di una scatola consiste nel creare un metodo che prenda in input le dimensioni di una scatola nei suoi parametri e imposti ciascuna variabile di istanza in modo appropriato. Questo concetto è implementato dal seguente programma:

```
// This program uses a parameterized method.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
```

```
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Come puoi vedere, il metodo setDim() viene utilizzato per impostare le dimensioni di ogni scatola. Ad esempio, quando viene eseguito

```
mybox1.setDim(10, 20, 15);
```

10 viene copiato nel parametro w, 20 viene copiato in h e 15 viene copiato in d. All'interno di setDim( ) i valori di w, h e d vengono quindi assegnati rispettivamente a width, height e depth. Per molti lettori, i concetti presentati nelle sezioni precedenti risulteranno familiari. Tuttavia, se cose come chiamate (o invocazione) di un metodo, argomenti e parametri sono

nuovi per te, potresti voler dedicare del tempo a sperimentare prima di andare avanti con nuovi concetti. I concetti di chiamata al metodo, parametri e valori di ritorno sono fondamentali per la programmazione Java pertanto prenditi il tempo di cui hai bisogno per capirli a fondo.

11

CLASSI IN DETTAGLIO

Può essere noioso inizializzare tutte le variabili in una classe ogni volta che viene creata un'istanza. Anche quando si aggiungono funzioni utili come setDim(), sarebbe più semplice e conciso eseguire tutte le assegnazioni al momento della creazione dell'oggetto. Poiché il requisito per l'inizializzazione è così comune, Java consente agli oggetti di inizializzarsi da soli quando vengono creati. Questa inizializzazione automatica viene eseguita tramite l'uso di un costruttore.

Un costruttore inizializza un oggetto immediatamente dopo la creazione, ha lo stesso nome della classe in cui risiede ed è sintatticamente simile a un metodo. Una volta definito, il costruttore viene chiamato automaticamente subito dopo la creazione dell'oggetto, prima del completamento del nuovo operatore. I costruttori sembrano un po' strani perché non hanno alcun tipo di ritorno, nemmeno void, questo perché il tipo restituito del costruttore di una classe è il tipo di classe stesso. È compito del costruttore inizializzare lo stato interno di un oggetto in modo che il codice che crea un'istanza abbia immediatamente un oggetto completamente inizializzato e utilizzabile. È possibile rielaborare l'esempio Box in modo che le dimensioni di una scatola vengano inizializzate automaticamente quando viene costruito un oggetto. Per farlo, sostituisci setDim() con un costruttore. Iniziamo definendo un semplice costruttore che semplicemente imposta le dimensioni di ogni scatola sugli stessi valori. Questa versione è mostrata qui:

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
```

```
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Quando questo programma viene eseguito, genera i seguenti risultati:

Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0

Come puoi vedere, sia mybox1 che mybox2 sono stati inizializzati dal costruttore Box() quando sono stati creati. Poiché il costruttore fornisce a tutte le scatole le stesse dimensioni, 10 per 10 per 10, sia mybox1 che mybox2 avranno lo stesso volume. L'istruzione println() all'interno di Box() è solo a scopo illustrativo. La maggior parte dei costruttori non visualizzerà nulla, inizializzeranno semplicemente un oggetto. Prima di procedere, riesaminiamo l'operatore new. Come sai, quando assegni un oggetto, usi la seguente forma generale:

class-var = **new** classname **( );**

Ora puoi capire perché le parentesi sono necessarie dopo il nome della classe. Ciò che sta effettivamente accadendo è che viene chiamato il costruttore per la classe. Così, alla riga:

Box mybox = **new** Box**();**

new Box() sta chiamando il costruttore Box(). Quando non si definisce in modo esplicito un costruttore per una classe, Java crea un costruttore predefinito per la classe stessa. Questo è il motivo per cui la riga di codice precedente funzionava nelle versioni precedenti di Box che non definivano un costruttore. Il costruttore predefinito inizializza automaticamente tutte le variabili di istanza con il valore zero. Il costruttore predefinito è spesso sufficiente per le classi semplici, ma di solito non va bene per quelle più

sofisticate. Una volta definito il proprio costruttore, il costruttore predefinito non viene più utilizzato. Sebbene il costruttore Box() nell'esempio precedente inizializzi un oggetto Box, non è molto utile: tutti i box hanno le stesse dimensioni. Quello che serve è un modo per costruire oggetti Box di varie dimensioni. La soluzione semplice è aggiungere parametri al costruttore, come probabilmente puoi intuire, questo lo rende molto più utile. Ad esempio, la versione seguente di Box definisce un costruttore parametrizzato che imposta le dimensioni di una scatola come specificato da tali parametri. Presta particolare attenzione a come vengono creati gli oggetti Box.

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo7 {
public static void main(String args[]) {
```

```
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);
    double vol;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
    }
    }
```

L’output del programma è il seguente:

```
    Volume is 3000.0
    Volume is 162.0
```

Come puoi vedere, ogni oggetto viene inizializzato come specificato nei parametri del suo costruttore. Ad esempio, nella riga seguente,

```
    Box mybox1 = new Box(10, 20, 15);
```

i valori 10, 20 e 15 vengono passati al costruttore Box() quando l’operatore new crea l'oggetto. Pertanto, la copia di width, height e depth di mybox1 avrà rispettivamente i valori 10, 20 e 15.

La parola chiave this

A volte un metodo dovrà fare riferimento all'oggetto che lo ha invocato. Per consentire ciò, Java definisce la parola chiave this. La parola chiave this può essere utilizzata all'interno di qualsiasi metodo per fare riferimento all'oggetto corrente. Cioè, this è sempre un riferimento all'oggetto su cui è stato invocato il metodo. Puoi usarlo ovunque sia consentito un riferimento a un oggetto del tipo della classe corrente. Per capire meglio a cosa si riferisce, si consideri la seguente versione di Box( ):

```
// A redundant use of this.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Questa versione di Box() funziona esattamente come la versione precedente. L'uso di this è ridondante, ma perfettamente valido. All'interno di Box(), this farà sempre riferimento all'oggetto invocante. Sebbene in questo caso sia ridondante, è utile in altri contesti, uno dei quali sarà spiegato a breve.

Come sapete, è illegale in Java dichiarare due variabili locali con lo stesso nome all'interno dello stesso ambito o che lo racchiudono. È interessante notare che puoi avere variabili locali, inclusi parametri formali per i metodi, che si sovrappongono ai nomi delle variabili di istanza della classe. Tuttavia, quando una variabile locale ha lo stesso nome di una variabile di istanza, la variabile locale nasconde la variabile di istanza. Questo è il motivo per cui width, height e depth non sono state utilizzate

come nomi dei parametri per il costruttore Box() all'interno della classe Box. Se lo fossero stati, width, ad esempio, si sarebbe riferita al parametro formale, nascondendo la width della variabile di istanza. Mentre di solito è più facile usare semplicemente nomi diversi, c'è un altro modo per aggirare questa situazione.

Poiché this consente di fare riferimento direttamente all'oggetto, è possibile utilizzarlo per risolvere eventuali collisioni del namespace che potrebbero verificarsi tra le variabili di istanza e le variabili locali. Ad esempio, ecco un'altra versione di Box(), che utilizza width, height e depth per i nomi dei parametri e quindi utilizza this per accedere alle variabili di istanza con lo stesso nome:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

Un avvertimento: l'uso di this in un tale contesto a volte può creare confusione e alcuni programmatori stanno attenti a non usare variabili locali e nomi di parametri formali che nascondono le variabili di istanza. Naturalmente, altri programmatori credono sia giusto il contrario, ovvero che sia una buona convenzione usare gli stessi nomi per chiarezza, e usare this per superare l'occultamento della variabile di istanza. È una questione di gusti, non esiste un approccio corretto e uno sbagliato.

Poiché gli oggetti vengono allocati dinamicamente utilizzando l'operatore new, ci si potrebbe chiedere in che modo tali oggetti vengono distrutti e in che modo viene rilasciata la loro memoria per una successiva riallocazione. In alcuni linguaggi, come C++, gli oggetti allocati

dinamicamente devono essere rilasciati manualmente mediante l'uso dell'operatore delete. Java adotta un approccio diverso; gestisce automaticamente la deallocazione. La tecnica che esegue questa operazione è chiamata Garbage Collection. Funziona così: quando non esistono riferimenti a un oggetto, si presume che quell'oggetto non sia più necessario e la memoria occupata dall'oggetto può essere recuperata. Non c'è bisogno esplicito di distruggere oggetti come in C++. Questa fase si verifica solo sporadicamente durante l'esecuzione del programma. Non si verificherà semplicemente perché esistono uno o più oggetti che non vengono più utilizzati. Inoltre, diverse implementazioni di runtime Java adotteranno approcci diversi alla garbage collection, ma per la maggior parte non devi nemmeno pensarci mentre scrivi i tuoi programmi.

A volte un oggetto dovrà eseguire alcune azioni quando viene distrutto, ad esempio, se un oggetto contiene una risorsa non Java come un handle di file o un font, è possibile assicurarsi che queste risorse vengano liberate prima che un oggetto venga distrutto. Per gestire tali situazioni, Java fornisce un meccanismo chiamato finalizzazione. Utilizzando la finalizzazione, puoi definire azioni specifiche che si verificheranno quando un oggetto sta per essere recuperato dal Garbage Collector. Per aggiungere un finalizzatore a una classe, definisci semplicemente il metodo finalize(). Il runtime Java chiama quel metodo ogni volta che sta per distruggere un oggetto di quella classe. All'interno del metodo finalize(), specificherai le azioni che devono essere eseguite prima che un oggetto venga distrutto. Il Garbage Collector viene eseguito periodicamente, verificando la presenza di oggetti a cui non viene più fatto riferimento da alcuno stato di esecuzione o indirettamente tramite altri oggetti referenziati. Subito prima che un asset venga liberato, il runtime Java chiama il metodo finalize() sull'oggetto. Il metodo finalize() ha questa forma generale:

```
protected void finalize( )
{
```

```
// finalization code here
}
```

Qui, la parola chiave protected è uno specificatore che impedisce l'accesso a finalize() tramite codice definito al di fuori della sua classe. È importante capire che finalize() viene chiamato appena prima della garbage collection. Ad esempio, non viene chiamato quando un oggetto esce dallo scope (ambito). Ciò significa che non puoi sapere quando, o anche se, finalize() verrà eseguito, pertanto, il tuo programma dovrebbe fornire altri mezzi per rilasciare le risorse di sistema, ecc., utilizzate dall'oggetto. Non deve basarsi su finalize() per il normale funzionamento del programma.

Se si ha familiarità con C++, allora saprai che C++ consente di definire un distruttore per una classe, che viene chiamato quando un oggetto esce dall'ambito. Java non supporta questa idea né fornisce dei veri e propri distruttori. Il metodo finalize() approssima la funzione di un distruttore e man mano che acquisirai maggiore esperienza con Java, vedrai che la necessità di funzioni di distruzione è minima a causa del sottosistema di Garbage Collection di Java.

Stack

Sebbene la classe Box sia utile per illustrare gli elementi essenziali di una classe, ha scarso valore pratico. Per mostrare il vero potere delle classi, questo capitolo si concluderà con un esempio più sofisticato. Come ricorderete, uno dei vantaggi più importanti di OOP (Object Oriented Programming) è l'incapsulamento dei dati e il codice che manipola quei dati. Come hai visto, la classe è il meccanismo con cui si ottiene l'incapsulamento in Java. Creando una classe, stai creando un nuovo tipo di dati che definisce sia la natura dei dati manipolati che le routine utilizzate per manipolarli. Inoltre, i metodi definiscono un'interfaccia coerente e controllata con i dati della classe. Pertanto, puoi utilizzare la classe attraverso i suoi metodi senza doverti preoccupare dei dettagli della sua implementazione o di come i dati vengono effettivamente gestiti all'interno della classe.

In un certo senso, una classe è come un "motore di dati". Non è richiesta alcuna conoscenza di ciò che accade all'interno del motore per utilizzare il motore attraverso i suoi comandi. Infatti, poiché i dettagli sono nascosti, il suo funzionamento interno può essere modificato secondo le proprie necessità. Finché il tuo codice usa la classe attraverso i suoi metodi, i dettagli interni possono cambiare senza causare effetti collaterali al di fuori della classe. Per vedere un'applicazione pratica, sviluppiamo uno degli esempi archetipici di incapsulamento: lo stack.

Uno stack memorizza i dati utilizzando l'ordine first-in, last-out cioè, come una pila di piatti su un tavolo: il primo piatto posato sul tavolo è l'ultimo piatto da usare. Gli stack sono controllati attraverso due operazioni tradizionalmente chiamate push e pop. Per mettere un oggetto in cima alla pila, utilizzerai push. Per togliere un oggetto dalla pila, utilizzerai pop. Come vedrai, è facile incapsulare l'intero meccanismo dello stack. Ecco una

classe chiamata Stack che implementa uno stack per un massimo di dieci numeri interi:

```
// This class defines an integer stack that can hold 10 values
class Stack {
int stck[] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
```

Come puoi vedere, la classe Stack definisce due elementi di dati e tre metodi. Lo stack di numeri interi è contenuto nello stack dell'array. Questo array è indicizzato dalla variabile tos, che contiene sempre l'indice della parte superiore dello stack. Il costruttore Stack() inizializza tos con il valore –1, che indica uno stack vuoto. Il metodo push() mette un elemento nello stack mentre per recuperare un elemento, si dovrà chiamare pop(). Poiché l'accesso allo stack avviene tramite push() e pop(), il fatto che lo stack sia contenuto in un array in realtà non è rilevante per l'utilizzo dello stack. Ad esempio, lo stack potrebbe essere contenuto in una struttura dati più complicata, come una linked list, ma l'interfaccia definita da push() e pop() rimarrebbe la stessa. La classe TestStack mostra la classe Stack, crea due stack interi, inserisce alcuni valori su ciascuno di essi e quindi li rimuove.

```
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
}
}
```

Ecco il risultato:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

Come puoi vedere, i contenuti di ogni pila sono separati. Un ultimo punto sulla classe Stack. Per come è attualmente implementato, è possibile che l'array che contiene lo stack, stck, venga modificato dal codice al di fuori della classe Stack. Questo lascia Stack aperto a un uso improprio o dannoso ma vedrai a breve come porre rimedio a questa situazione.

Overloading

In Java è possibile definire due o più metodi all'interno della stessa classe che condividono lo stesso nome, purché le loro dichiarazioni di parametri siano diverse. In questo caso, si dice che i metodi sono sovraccaricati e il processo viene definito sovraccarico del metodo (overloading). L'overloading del metodo è uno dei modi in cui Java supporta il polimorfismo. Se non hai mai usato un linguaggio che consenta di sovraccaricare i metodi, all'inizio il concetto potrebbe sembrare strano ma come vedrai, il sovraccarico del metodo è una delle funzionalità più interessanti e utili di Java.

Quando viene richiamato un metodo di overload, Java utilizza il tipo e/o il numero di argomenti come guida per determinare quale versione del metodo di overload chiamare effettivamente. Pertanto, i metodi sovraccaricati devono differire per il tipo e/o il numero dei relativi parametri. Sebbene i metodi sottoposti a overload possano avere tipi restituiti diversi, il tipo restituito da solo non è sufficiente per distinguere due versioni di un metodo. Quando Java incontra una chiamata a un metodo overloaded, esegue semplicemente la versione del metodo i cui parametri corrispondono agli argomenti utilizzati nella chiamata. Ecco un semplice esempio che illustra il sovraccarico del metodo:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
```

```
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

Questo programma genera il seguente risultato:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
```

Result of ob.test(123.25): 15190.5625

Come puoi vedere, test() è sovraccaricato quattro volte. La prima versione non accetta parametri, la seconda un parametro intero, la terza due parametri interi e la quarta un parametro double. Il fatto che anche la quarta versione di test() restituisca un valore non ha alcuna conseguenza rispetto all'overloading, poiché i tipi restituiti non svolgono un ruolo nella risoluzione dell'overload. Quando viene chiamato un metodo sovraccaricato, Java cerca una corrispondenza tra gli argomenti utilizzati per chiamare il metodo e i parametri del metodo, tuttavia, questa corrispondenza non deve essere sempre esatta. In alcuni casi, le conversioni automatiche dei tipi di Java possono svolgere un ruolo nella risoluzione del sovraccarico. Si consideri ad esempio il seguente programma:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
```

```
    OverloadDemo ob = new OverloadDemo();
    int i = 88;
    ob.test();
    ob.test(10, 20);
    ob.test(i); // this will invoke test(double)
    ob.test(123.2); // this will invoke test(double)
    }
    }
```

Ecco il risultato:

```
    No parameters
    a and b: 10 20
    Inside test(double) a: 88
    Inside test(double) a: 123.2
```

Come puoi vedere, questa versione di OverloadDemo non definisce test(int). Pertanto, quando test() viene chiamato con un argomento intero all'interno di Overload, non viene trovato alcun metodo corrispondente. Tuttavia, Java può convertire automaticamente un intero in un double e questa conversione può essere utilizzata per risolvere la chiamata. Pertanto, dopo che test(int) non viene trovato, Java eleva i a double e quindi chiama test(double). Naturalmente, se test(int) fosse stato definito, sarebbe stato chiamato quel metodo.

Java utilizza le sue conversioni di tipo automatiche solo se non viene trovata una corrispondenza esatta. L'overloading dei metodi supporta il polimorfismo perché è un modo in cui Java implementa il paradigma "un'interfaccia, più metodi". Per capire come, considera quanto segue. Nei linguaggi che non supportano l'overloading dei metodi, a ogni metodo deve

essere assegnato un nome univoco. Tuttavia, spesso vorrai implementare essenzialmente lo stesso metodo per diversi tipi di dati. Considera la funzione del valore assoluto. Nei linguaggi che non supportano l'overloading, di solito esistono tre o più versioni di questa funzione, ciascuna con un nome leggermente diverso. Ad esempio, in C, la funzione abs() restituisce il valore assoluto di un intero, labs() restituisce il valore assoluto di un intero long e fabs() restituisce il valore assoluto di un valore a virgola mobile. Poiché C non supporta l'overloading, ogni funzione deve avere il proprio nome, anche se tutte e tre le funzioni fanno essenzialmente la stessa cosa. Questo rende la situazione più complessa, concettualmente, di quanto non sia in realtà. Sebbene il concetto alla base di ciascuna funzione sia lo stesso, hai tre nomi da ricordare. Questa situazione non si verifica in Java, poiché ogni metodo di valore assoluto può utilizzare lo stesso nome. In effetti, la libreria di classi standard di Java include un metodo per il valore assoluto, chiamato abs(). Questo metodo è sovraccaricato dalla classe Math di Java per gestire tutti i tipi numerici. Java determina quale versione di abs() chiamare in base al tipo di argomento. Il valore dell'overloading è che consente l'accesso ai metodi correlati tramite l'uso di un nome comune. Pertanto, il nome abs rappresenta l'azione generale che viene eseguita. Spetta poi al compilatore scegliere la versione specifica giusta per una particolare circostanza ma tu, programmatore, devi solo ricordare l'operazione generale che viene eseguita. Attraverso l'applicazione del polimorfismo, diversi nomi sono stati ridotti a uno solo.

Sebbene questo esempio sia abbastanza semplice, se espandi il concetto, puoi vedere come il sovraccarico può aiutarti a gestire una maggiore complessità. Quando sovraccarichi un metodo, ogni versione di quel metodo può eseguire qualsiasi attività tu desideri. Non esiste una regola che stabilisca che i metodi sovraccaricati debbano essere correlati tra loro. Tuttavia, da un punto di vista stilistico, il sovraccarico di un metodo implica una relazione. Pertanto, puoi usare lo stesso nome per sovraccaricare

metodi non correlati ma non dovresti. Ad esempio, è possibile utilizzare il nome sqr per creare metodi che restituiscono il quadrato di un numero intero e la radice quadrata di un valore a virgola mobile, ma queste due operazioni sono fondamentalmente diverse. L'applicazione del sovraccarico del metodo, in questo modo, vanifica il suo scopo originale. In pratica, dovresti sovraccaricare solo le operazioni strettamente correlate.

Oltre a sovraccaricare i metodi normali, puoi anche sovraccaricare i metodi del costruttore. In effetti, per la maggior parte delle classi che crei nel mondo reale, i costruttori sovraccaricati saranno la norma, non l'eccezione. Per capire perché, torniamo alla classe Box sviluppata prima. Di seguito è riportata l'ultima versione di Box:

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

Come puoi vedere, il costruttore Box() richiede tre parametri. Ciò significa che tutte le dichiarazioni di oggetti Box devono passare tre argomenti al

costruttore Box(). Ad esempio, la seguente affermazione non è attualmente valida:

```
Box ob = new Box();
```

Poiché Box() richiede tre argomenti, è un errore chiamarlo senza di essi. Ciò solleva alcune domande importanti. E se volessi semplicemente una scatola e non ti importasse (o sapessi) quali fossero le sue dimensioni iniziali? Oppure, cosa succede se si desidera essere in grado di inizializzare una scatola specificando un solo valore da usare per tutte e tre le dimensioni? Per come è scritta ora la classe Box, queste opzioni non sono disponibili per te. Fortunatamente, la soluzione a questi problemi è abbastanza semplice: è sufficiente sovraccaricare il costruttore Box in modo che gestisca le situazioni appena descritte. Ecco un programma che contiene una versione migliorata di Box che fa proprio questo:

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
```

```
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
```

```
}
```

Ecco l'output del programma:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Come puoi vedere, il costruttore corretto viene chiamato in base ai parametri specificati quando viene eseguito l'operatore new.

Finora, abbiamo utilizzato solo tipi semplici come parametri per metodi. Tuttavia, è sia corretto che comune passare oggetti ai metodi. Si consideri ad esempio il seguente programma:

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
```

```
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

Questo è il risultato:

```
ob1 == ob2: true
ob1 == ob3: false
```

Come puoi vedere, il metodo equals() all'interno di Test confronta due oggetti per verificarne l'uguaglianza e restituisce il risultato. Cioè, confronta l'oggetto invocante con quello che gli è stato passato, se contengono gli stessi valori, il metodo restituisce true. In caso contrario, restituisce false. Si noti che il parametro o in equals() specifica Test come tipo. Sebbene Test sia un tipo di classe creato dal programma, viene utilizzato esattamente allo stesso modo dei tipi integrati di Java. Uno degli usi più comuni dei parametri oggetto riguarda i costruttori. Spesso, vorrai costruire un nuovo oggetto in modo che inizialmente sia lo stesso di un oggetto esistente. Per fare ciò, è necessario definire un costruttore che accetta un oggetto della sua classe come parametro. Ad esempio, la seguente versione di Box consente a un oggetto di inizializzarne un altro:

```
// Here, Box allows one object to initialize another.
class Box {
double width;
double height;
double depth;
```

```
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
```

```
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

Come vedrai quando inizierai a creare le tue classi, di solito è necessario fornire molte forme differenti di costruttori per consentire la costruzione di oggetti in modo conveniente ed efficiente.

In generale, ci sono due modi in cui un linguaggio informatico può passare un argomento a una subroutine. Il primo modo è detto “passaggio per valore” e questo approccio copia il valore di un argomento nel parametro formale della subroutine. Pertanto, le modifiche apportate al parametro della subroutine non hanno alcun effetto sull'argomento. Il secondo modo in cui un argomento può essere passato è detto “passaggio

per riferimento" e in questo approccio, al parametro viene passato un riferimento a un argomento (non il valore dell'argomento). All'interno della subroutine, questo riferimento viene utilizzato per accedere all'argomento effettivo specificato nella chiamata. Ciò significa che le modifiche apportate al parametro influiranno sull'argomento utilizzato per chiamare la subroutine. Come vedrai, sebbene Java utilizzi un approccio di tipo "passaggio per valore" per passare tutti gli argomenti, l'effetto preciso differisce se viene passato un tipo primitivo o un tipo di riferimento. Quando si passa un tipo primitivo a un metodo, viene passato per valore, pertanto, viene creata una copia dell'argomento e ciò che accade al parametro che riceve l'argomento non ha effetto al di fuori del metodo. Si consideri ad esempio il seguente programma:

```
// Primitive types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}


class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
```

```
}
}
```

L’output del programma sarà:

```
a and b before call: 15 20
a and b after call: 15 20
```

Come puoi vedere, le operazioni che avvengono all'interno di meth() non hanno effetto sui valori di a e b utilizzati nella chiamata; i loro valori qui non sono cambiati in 30 e 10. Quando si passa un oggetto a un metodo, la situazione cambia drasticamente, perché gli oggetti vengono passati da ciò che è effettivamente un “passaggio per riferimento”. Tieni presente che quando crei una variabile di un tipo di classe, stai solo creando un riferimento a un oggetto. Pertanto, quando si passa questo riferimento a un metodo, il parametro che lo riceve farà riferimento allo stesso oggetto a cui fa riferimento l'argomento. Ciò significa effettivamente che gli oggetti agiscono come se fossero passati ai metodi mediante l'uso di un passaggio per riferimento. Le modifiche sull'oggetto all'interno del metodo influiscono sull'oggetto utilizzato come argomento. Si consideri ad esempio il seguente programma:

```
// Objects are passed through their references.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
```

```
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
class PassObjRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

Ecco l'output del programma:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

Come puoi vedere, in questo caso, le azioni all'interno di meth() hanno influenzato l'oggetto utilizzato come argomento. Ricorda: quando un riferimento a un oggetto viene passato a un metodo, il riferimento stesso viene passato mediante l'uso di passaggio per valore. Tuttavia, poiché il valore passato si riferisce a un oggetto, la copia di quel valore farà comunque riferimento allo stesso oggetto che fa il suo argomento corrispondente.

Un metodo può restituire qualsiasi tipo di dati, inclusi i tipi di classe creati dall'utente. Ad esempio, nel programma seguente, il metodo incrByTen() restituisce un oggetto in cui il valore di a è maggiore di 10 unità rispetto a quello dell'oggetto invocante.

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
}
}
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "
+ ob2.a);
}
}
```

Il risultato sarà:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Come puoi vedere, ogni volta che incrByTen() viene invocato, viene creato un nuovo oggetto e viene restituito un riferimento ad esso alla routine chiamante. Nel programma precedente inoltre, poiché tutti gli oggetti sono allocati dinamicamente usando new, non devi preoccuparti che un oggetto esca dall'ambito perché termina il metodo in cui è stato creato. L'oggetto continuerà ad esistere finché c'è un riferimento ad esso da qualche parte nel tuo programma. Quando non ci sono riferimenti ad esso, l'oggetto verrà recuperato dalla prossima esecuzione del Garbage Collection.

Ricorsione

Java supporta la ricorsione, ovvero il processo di definizione di qualcosa in termini di sé stesso. Per quanto riguarda la programmazione Java, la ricorsione è l'attributo che consente a un metodo di chiamare sé stesso. Un metodo che chiama sé stesso è detto ricorsivo e il classico esempio di ricorsione è il calcolo del fattoriale di un numero. Il fattoriale di un numero N è il prodotto di tutti i numeri interi compresi tra 1 e N. Ad esempio, 3 fattoriale è 1 × 2 × 3, ovvero 6. Ecco come può essere calcolato un fattoriale mediante un metodo ricorsivo:

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

L’output del programma è:

Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Se non hai familiarità con i metodi ricorsivi, l'uso di fact() può sembrare un po' confuso. Ecco come funziona: quando fact() viene chiamato con argomento 1, la funzione restituisce 1; in caso contrario, restituisce il prodotto di fact(n–1)*n. Per valutare questa espressione, fact() viene chiamato con n–1. Questo processo si ripete finché n è uguale a 1 e le chiamate al metodo iniziano a restituire un valore. Per capire meglio come funziona il metodo fact(), facciamo un breve esempio. Quando si calcola il fattoriale di 3, la prima chiamata a fact() farà sì che venga effettuata una seconda chiamata con un argomento 2. Questa invocazione farà sì che fact() venga chiamata una terza volta con un argomento 1. Questa chiamata restituirà 1, che viene poi moltiplicato per 2 (il valore di n nella seconda invocazione). Questo risultato (che è 2) viene quindi restituito all'invocazione originale di fact() e moltiplicato per 3 (il valore originale di n). Questo produce la risposta, 6.

Potresti trovare interessante inserire le istruzioni println() in fact(), che mostra a quale livello si trova ciascuna chiamata e quali sono le risposte intermedie. Quando un metodo chiama sé stesso, vengono allocati in memoria nello stack nuove variabili e parametri locali; quindi, il codice del metodo viene eseguito con queste nuove variabili dall'inizio. Quando ogni chiamata ricorsiva restituisce un valore, le vecchie variabili e i parametri locali vengono rimossi dallo stack e l'esecuzione riprende nel punto della chiamata all'interno del metodo.

Le versioni ricorsive di molte routine possono essere eseguite un po' più lentamente rispetto all'equivalente iterativo a causa del sovraccarico

aggiuntivo delle chiamate di funzione aggiuntive. Molte chiamate ricorsive a un metodo potrebbero causare un sovraccarico dello stack. Poiché l'archiviazione per i parametri e le variabili locali avviene nello stack e ogni nuova chiamata crea una nuova copia di queste variabili, è possibile che lo stack si esaurisca. In questo caso, il sistema di runtime Java causerà un'eccezione, tuttavia, probabilmente non dovrai preoccuparti di questo a meno che non si scateni una routine ricorsiva.

Il vantaggio principale dei metodi ricorsivi è che possono essere utilizzati per creare versioni più chiare e più semplici di diversi algoritmi rispetto alle loro versioni iterative. Ad esempio, l'algoritmo di ordinamento QuickSort è piuttosto difficile da implementare in modo iterativo. Inoltre, alcuni tipi di algoritmi relativi all'IA vengono implementati più facilmente utilizzando soluzioni ricorsive. Quando si scrivono metodi ricorsivi, è necessario disporre di un'istruzione if da qualche parte per forzare la restituzione del metodo senza che la chiamata ricorsiva venga eseguita. Se dimentichi ciò, una volta chiamato il metodo, non restituirà mai più un valore e questo è un errore molto comune quando si lavora con la ricorsione. Usa liberamente le istruzioni println() durante lo sviluppo in modo da poter guardare cosa sta succedendo e poter interrompere l'esecuzione se vedi che hai commesso un errore. Ecco un altro esempio di ricorsione. Il metodo ricorsivo printArray() stampa i primi elementi i nei valori dell'array.

```
// Another example that uses recursion.
class RecTest {
int values[];
RecTest(int i) {
values = new int[i];
}
// display array -- recursively
void printArray(int i) {
```

```
if(i==0) return;
else printArray(i-1);
System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}
class Recursion2 {
public static void main(String args[]) {
RecTest ob = new RecTest(10);
int i;
for(i=0; i<10; i++) ob.values[i] = i;
ob.printArray(10);
}
}
```

Ecco l'output del programma:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

12

VISIBILITÀ

Come sapete, l'incapsulamento collega i dati con il codice che li manipola. Tuttavia, l'incapsulamento fornisce un altro importante attributo: il controllo dell'accesso. Attraverso l'incapsulamento, puoi controllare quali parti di un programma possono accedere ai membri di una classe. Controllando l'accesso, puoi prevenire un uso improprio, ad esempio, consentendo l'accesso ai dati solo attraverso un insieme ben definito di metodi, è possibile prevenire l'uso improprio di tali dati. Pertanto, se correttamente implementata, una classe crea una “scatola nera” che può essere utilizzata, ma il cui funzionamento interno non è soggetto a manomissioni. Tuttavia, le classi presentate in precedenza non soddisfano completamente questo obiettivo. Ad esempio, si consideri la classe Stack mostrata in precedenza, sebbene sia vero che i metodi push() e pop() forniscano un'interfaccia controllata allo stack, questa interfaccia non viene applicata. In altre parole, è possibile che un'altra parte del programma ignori questi metodi e acceda direttamente allo stack. Naturalmente, nelle mani sbagliate, questo potrebbe portare a problemi.

In questa sezione verrà introdotto il meccanismo mediante il quale è possibile controllare con precisione l'accesso ai vari membri di una classe. La modalità di accesso a un membro è determinata dal modificatore di accesso allegato alla sua dichiarazione. Java fornisce un ricco set di modificatori di accesso. Alcuni aspetti del controllo degli accessi sono legati principalmente all'ereditarietà o ai pacchetti, un pacchetto è essenzialmente un raggruppamento di classi.

Adesso, iniziamo esaminando il controllo di accesso in quanto si applica a una singola classe. Una volta compresi i fondamenti del controllo degli accessi, il resto sarà facile. I modificatori di accesso di Java sono public, private e protected. Java definisce anche un livello di accesso predefinito. Il

livello protected si applica solo quando è coinvolta l'eredità. Gli altri modificatori di accesso sono descritti di seguito.

Iniziamo definendo public e private. Quando un membro di una classe viene modificato da public, è possibile accedere a quel membro con qualsiasi altro codice. Quando un membro di una classe viene specificato come private, è possibile accedere a quel membro solo da altri membri della sua classe. Ora puoi capire perché main() è sempre stato preceduto dal modificatore public. Viene chiamato da codice esterno al programma, ovvero dal sistema di runtime Java. Quando non viene utilizzato alcun modificatore di accesso, per impostazione predefinita il membro di una classe è public all'interno del proprio pacchetto, ma non è possibile accedervi al di fuori del suo pacchetto. Nelle classi sviluppate finora, tutti i membri di una classe hanno utilizzato la modalità di accesso predefinita, che è essenzialmente public, tuttavia, questo non è ciò che in genere vorresti. Di solito, vorrai limitare l'accesso ai membri dati di una classe, consentendo l'accesso solo tramite metodi. Inoltre, ci saranno momenti in cui vorrai definire metodi privati per una classe. Un modificatore di accesso precede il resto della specifica del tipo di un membro, cioè, deve iniziare lo statement di dichiarazione di un membro. Ecco un esempio:

```
public int i;
private double j;
private int myMethod(int a, char b) { //...
```

Per comprendere gli effetti dell'accesso public e private, si consideri il seguente programma:

```
/* This program demonstrates the difference between
public and private.
*/
class Test {
```

```
    int a; // default access
    public int b; // public access
    private int c; // private access
    // methods to access c
    void setc(int i) { // set c's value
    c = i;
    }
    int getc() { // get c's value
    return c;
    }
    }


class AccessTest {
    public static void main(String args[]) {
    Test ob = new Test();
    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;
    // This is not OK and will cause an error
    // ob.c = 100; // Error!
    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
    ob.b + " " + ob.getc());
    }
    }
```

Come puoi vedere, all'interno della classe Test, utilizza l'accesso predefinito, che per questo esempio equivale a specificare public. Per quanto riguarda b è esplicitamente specificato come public. Il membro c ha

accesso private. Ciò significa che non è possibile accedervi tramite codice al di fuori della sua classe. Quindi, all'interno della classe AccessTest, c non può essere utilizzato direttamente. È necessario accedervi tramite i suoi metodi pubblici: setc() e getc(). Se dovessi rimuovere il simbolo del commento dall'inizio della riga successiva,

```
// ob.c = 100; // Error!
```

non saresti in grado di compilare questo programma a causa della violazione di accesso. Per vedere come applicare il controllo di accesso a un esempio più pratico, si consideri la seguente versione (rivisitata) della classe Stack:

```
// This class defines an integer stack that can hold 10 values.
class Stack {
/* Now, both stck and tos are private. This means
that they cannot be accidentally or maliciously
altered in a way that would be harmful to the stack.
*/
private int stck[] = new int[10];
private int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
```

```
    }


// Pop an item from the stack
    int pop() {
    if(tos < 0) {
    System.out.println("Stack underflow.");
    return 0;
    }
    else
    return stck[tos--];
    }
    }
```

Come puoi vedere, ora sia stck, che contiene lo stack, sia tos, che è l'indice della parte superiore dello stack, sono specificati come private. Ciò significa che non è possibile accedervi o modificarli se non tramite push() e pop(). Rendere tos private, ad esempio, impedisce ad altre parti del programma di settarlo inavvertitamente su un valore che è oltre la fine dell'array stck. Il programma seguente mostra la classe Stack nella sua versione migliorata. Prova a rimuovere le righe commentate per dimostrare a te stesso che i membri stck e tos sono, in effetti, inaccessibili.

```
    class TestStack {
    public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();
    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);
```

```
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
// these statements are not legal
// mystack1.tos = -2;
// mystack2.stck[3] = 100;
}
}
```

Sebbene i metodi di solito forniscano l'accesso ai dati definiti da una classe, questo non deve sempre essere vero. È perfettamente corretto consentire a una variabile di istanza di essere pubblica quando vi sono buone ragioni per farlo. Ad esempio, la maggior parte delle classi semplici in questo libro sono state create con poca preoccupazione per il controllo dell'accesso alle variabili di istanza per motivi di semplicità. Tuttavia, nella maggior parte delle classi del mondo reale, dovrai consentire le operazioni sui dati solo tramite i metodi. Come vedrai, è particolarmente importante quando è coinvolta l'eredità, ci saranno momenti in cui vorrai definire un membro di classe che verrà utilizzato indipendentemente da qualsiasi oggetto di quella classe.

Static

Normalmente, è necessario accedere a un membro della classe solo insieme a un oggetto della sua classe, tuttavia, è possibile creare un membro che può essere utilizzato da solo, senza riferimento a un'istanza specifica. Per creare un tale membro, fai precedere la sua dichiarazione con la parola chiave static. Quando un membro è dichiarato statico, è possibile accedervi prima della creazione di qualsiasi oggetto della sua classe e senza riferimento a nessun oggetto. È possibile dichiarare statici sia i metodi che le variabili. L'esempio più comune di membro statico è main(). Il metodo main() è dichiarato statico perché deve essere chiamato prima che esistano oggetti. Le variabili di istanza dichiarate come statiche sono essenzialmente variabili globali. Quando gli oggetti della sua classe vengono dichiarati, non viene eseguita alcuna copia di una variabile statica, invece, tutte le istanze della classe condividono la stessa variabile statica. I metodi dichiarati come statici hanno diverse restrizioni:

- Possono solo chiamare direttamente altri metodi statici.
- Possono accedere solo direttamente ai dati statici.
- Non possono fare riferimento a questo o super in alcun modo.

Se devi eseguire calcoli per inizializzare le tue variabili statiche, puoi dichiarare un blocco statico che viene eseguito esattamente una volta, quando la classe viene caricata per la prima volta. L'esempio seguente mostra una classe che ha un metodo statico, alcune variabili statiche e un blocco di inizializzazione statico:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
```

```
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

Non appena la classe UseStatic viene caricata, vengono eseguite tutte le istruzioni statiche. Innanzitutto, a viene impostato sul valore 3, quindi viene eseguito il blocco statico, che stampa un messaggio e quindi inizializza b su a*4 ovvero 12. Quindi viene chiamato il metodo main(), che chiama meth(), passando 42 a x. Le tre istruzioni println() si riferiscono alle due variabili statiche a e b, nonché alla variabile locale x.

Ecco il risultato del programma:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Al di fuori della classe in cui sono definiti, i metodi e le variabili statiche possono essere utilizzati indipendentemente da qualsiasi oggetto. Per fare ciò, devi solo specificare il nome della loro classe seguito dall'operatore punto. Ad esempio, se desideri chiamare un metodo statico dall'esterno della sua classe, puoi farlo utilizzando la seguente formula generica:

```
classname.method( )
```

Qui, classname è il nome della classe in cui è dichiarato il metodo statico. Come puoi vedere, questa invocazione è simile a quella usata per chiamare metodi non statici tramite variabili di riferimento a oggetti. È possibile accedere a una variabile statica allo stesso modo, utilizzando l'operatore punto sul nome della classe. Questo è il modo in cui Java implementa una versione controllata di metodi globali e variabili globali. Ecco un esempio. All'interno di main(), è possibile accedere al metodo statico callme() e alla variabile statica b tramite il nome della classe StaticDemo.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Final

Un campo può essere dichiarato final. In questo modo si evita che il suo contenuto venga modificato, rendendolo, essenzialmente, una costante. Ciò significa che è necessario inizializzare un campo final quando viene dichiarato. Puoi farlo in due modi: in primo luogo, puoi assegnargli un valore quando viene dichiarato. In secondo luogo, puoi assegnargli un valore all'interno di un costruttore. Il primo approccio è il più comune, ecco un esempio:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Le parti successive del tuo programma possono ora utilizzare FILE_OPEN, ecc., come se fossero costanti, senza timore che un valore sia stato modificato. È una convenzione comune scegliere tutti gli identificatori maiuscoli per i campi final, come mostra questo esempio. Oltre ai campi, possono essere dichiarati final sia i parametri del metodo che le variabili locali. La dichiarazione di un parametro final ne impedisce la modifica all'interno del metodo. La dichiarazione di una variabile locale final evita che le venga assegnato un valore più di una volta. La parola chiave final può essere applicata anche ai metodi, ma il suo significato è sostanzialmente diverso rispetto a quando viene applicata alle variabili.

Classi innestate

È possibile definire una classe all'interno di un'altra classe; tali classi sono note come classi nidificate o innestate. L'ambito di una classe nidificata è limitato dall'ambito della sua classe che lo racchiude. Pertanto, se la classe B è definita all'interno della classe A, B non esiste indipendentemente da A. Una classe nidificata ha accesso ai membri, inclusi i membri privati, della classe in cui è nidificata, tuttavia, la classe di inclusione non ha accesso ai membri della classe nidificata. Una classe nidificata dichiarata direttamente all'interno dell'ambito della sua classe di inclusione è un membro della sua classe di inclusione. È anche possibile dichiarare una classe nidificata che è locale a un blocco.

Esistono due tipi di classi nidificate: statiche e non statiche. Una classe nidificata statica è quella a cui è applicato il modificatore static. Poiché è statica, deve accedere ai membri non statici della sua classe che lo racchiude tramite un oggetto cioè, non può fare riferimento direttamente ai membri non statici della sua classe che lo racchiude. A causa di questa restrizione, le classi nidificate statiche vengono utilizzate raramente. Il tipo più importante di classe nidificata è la classe interna. Una classe interna è una classe nidificata non statica. Ha accesso a tutte le variabili e ai metodi della sua classe esterna e può fare riferimento ad esse direttamente nello stesso modo in cui lo fanno altri membri non statici della classe esterna. Il programma seguente illustra come definire e utilizzare una classe interna. La classe denominata Outer ha una variabile di istanza denominata external_x, un metodo di istanza denominato test() e definisce una classe interna chiamata Inner.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
```

```
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Ecco il risultato del programma:

```
display: outer_x = 100
```

Nel programma, una classe interna denominata Inner è definita nell'ambito della classe Outer. Pertanto, qualsiasi codice della classe Inner può accedere direttamente alla variabile external_x. Un metodo di istanza denominato display() è definito all'interno di Inner, questo metodo visualizza outer_x sul flusso di output standard. Il metodo main() di InnerClassDemo crea un'istanza della classe Outer e richiama il suo metodo test(). Quel metodo crea un'istanza della classe Inner e viene chiamato il metodo display(). È

importante rendersi conto che un'istanza di Inner può essere creata solo nell'ambito della classe Outer. Il compilatore Java genera un messaggio di errore se un codice al di fuori della classe Outer tenta di creare un'istanza della classe Inner.

In generale, un'istanza di classe interna deve essere creata da un ambito di inclusione. Come spiegato, una classe interna ha accesso a tutti i membri della sua classe che la racchiude, ma non è vero il contrario. I membri della classe interna sono noti solo nell'ambito della classe interna e non possono essere utilizzati dalla classe esterna. Per esempio:

```
// This program will not compile.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
int y = 10; // y is local to Inner
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
void showy() {
System.out.println(y); // error, y not known here!
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
```

```
outer.test();
}
}
```

Qui, y è dichiarata come una variabile di istanza di Inner. Pertanto, non è noto al di fuori di quella classe e non può essere utilizzato da showy(). Sebbene ci siamo concentrati sulle classi interne dichiarate come membri all'interno di un ambito di classe esterno, è possibile definire classi interne all'interno di qualsiasi scope di blocco. Ad esempio, puoi definire una classe nidificata all'interno del blocco definito da un metodo o anche all'interno del corpo di un ciclo for, come mostra il prossimo programma:

```
// Define an inner class within a for loop.
class Outer {
int outer_x = 100;
void test() {
for(int i=0; i<10; i++) {
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
Inner inner = new Inner();
inner.display();
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
```

```
outer.test();
}
}
```

Ecco l'output del programma:

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

Sebbene le classi nidificate non siano applicabili a tutte le situazioni, sono particolarmente utili durante la gestione degli eventi, possono essere usate per semplificare il codice necessario per gestire certi tipi di eventi. Un ultimo punto: le classi nidificate non erano consentite dalla specifica originale 1.0 per Java ma sono state aggiunte a partire da Java 1.1.

13

EREDITARIETÀ

L'ereditarietà è uno dei capisaldi della programmazione orientata agli oggetti perché consente la creazione di classificazioni gerarchiche. Utilizzando l'ereditarietà, puoi creare una classe generale che definisce i tratti comuni a un insieme di elementi correlati. Questa classe può quindi essere ereditata da altre classi più specifiche, ognuna aggiungendo quelle cose che la rendono sono unica. Nella terminologia di Java, una classe ereditata è chiamata superclasse. La classe che eredita è chiamata sottoclasse. Pertanto, una sottoclasse è una versione specializzata di una superclasse, eredita tutti i membri definiti dalla superclasse e aggiunge i propri elementi unici. Per ereditare una classe, devi semplicemente incorporare la definizione di una classe in un'altra usando la parola chiave extends. Per vedere come, iniziamo con un breve esempio. Il seguente programma crea una superclasse chiamata A e una sottoclasse chiamata B. Nota come la parola chiave extends viene usata per creare una sottoclasse di A.

```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
```

```
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args []) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

Ecco l’output del programma:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

Come puoi vedere, la sottoclasse B include tutti i membri della sua superclasse A. Questo è il motivo per cui subOb può accedere ad i e j e chiamare showij(). Inoltre, all'interno di sum(), i e j possono essere riferiti direttamente, come se facessero parte di B. Anche se A è una superclasse per B, è anche una classe indipendente e completamente indipendente. Essere una superclasse per una sottoclasse non significa che la superclasse non possa essere utilizzata da sola. Inoltre, una sottoclasse può essere una superclasse per un'altra sottoclasse. La forma generale di una dichiarazione di classe che eredita una superclasse è mostrata qui:

```
class subclass-name extends superclass-name {
// body of class
}
```

Puoi specificare solo una superclasse per qualsiasi sottoclasse che crei. Java non supporta l'ereditarietà di più superclassi in una singola sottoclasse. È possibile, come affermato, creare una gerarchia di ereditarietà in cui una sottoclasse diventa una superclasse di un'altra sottoclasse. Tuttavia, nessuna classe può essere una superclasse di per sé.

Sebbene una sottoclasse includa tutti i membri della sua superclasse, non può accedere a quei membri della superclasse che sono stati dichiarati private. Ad esempio, considera la seguente semplice gerarchia di classi:

```
/* In a class hierarchy, private members remain
private to their class.
This program contains an error and will not
compile.
*/
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
```

```
}
}
```

Questo programma non verrà compilato perché l'uso di j all'interno del metodo sum() di B provoca una violazione di accesso. Poiché j è dichiarato private, è accessibile solo da altri membri della propria classe. Le sottoclassi non vi hanno accesso. Ricorda: un membro di una classe che è stato dichiarato privato rimarrà privato per la sua classe. Non è accessibile da alcun codice al di fuori della sua classe, comprese le sottoclassi.

Diamo un'occhiata a un esempio più pratico che aiuterà a illustrare il potere dell'ereditarietà. Qui, la versione finale della classe Box sviluppata nei capitoli precedenti verrà estesa per includere un quarto componente chiamato weight (peso). Pertanto, la nuova classe conterrà la larghezza, l'altezza, la profondità e il peso di una scatola.

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
```

```
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Here, Box is extended to include weight.
class BoxWeight extends Box {
double weight; // weight of box
// constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}
class DemoBoxWeight {
public static void main(String args[]) {
```

```
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}
}
```

Ecco il risultato:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

BoxWeight eredita tutte le caratteristiche di Box e ad esse aggiunge la componente weight. Non è necessario che BoxWeight ricrei tutte le funzionalità presenti in Box, può semplicemente estendere Box per soddisfare i propri scopi. Uno dei principali vantaggi dell'ereditarietà è che una volta creata una superclasse che definisce gli attributi comuni a un insieme di oggetti, può essere utilizzata per creare un numero qualsiasi di sottoclassi più specifiche. Ogni sottoclasse può personalizzare con precisione la propria classificazione. Ad esempio, la classe seguente eredita Box e aggiunge un attributo color:

```
// Here, Box is extended to include color.
class ColorBox extends Box {
int color; // color of box
ColorBox(double w, double h, double d, int c) {
width = w;
height = h;
depth = d;
color = c;
}
}
```

Ricorda, una volta creata una superclasse che definisce gli aspetti generali di un oggetto, quella superclasse può essere ereditata per formare classi specializzate. Ogni sottoclasse aggiunge semplicemente i propri attributi univoci. Questa è l'essenza dell'eredità.

A una variabile di riferimento di una superclasse può essere assegnato un riferimento a qualsiasi sottoclasse derivata da quella superclasse. Troverai questo aspetto dell'ereditarietà molto utile in una varietà di situazioni. Ad esempio, considera quanto segue:

```
class RefDemo {
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
```

```
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

Qui, weightbox è un riferimento agli oggetti BoxWeight e plainbox è un riferimento agli oggetti Box. Poiché BoxWeight è una sottoclasse di Box, è consentito assegnare plainbox un riferimento all'oggetto weightbox. È importante comprendere che è il tipo della variabile di riferimento, non il tipo dell'oggetto a cui si riferisce, che determina a quali membri è possibile accedere cioè, quando un riferimento a un oggetto di sottoclasse viene assegnato a una variabile di riferimento di superclasse, avrai accesso solo a quelle parti dell'oggetto definite dalla superclasse. Questo è il motivo per cui plainbox non può accedere a weigth anche quando si riferisce a un oggetto BoxWeight. Se ci pensi, questo ha senso, perché la superclasse non sa cosa aggiunge una sottoclasse. Questo è il motivo per cui l'ultima riga di codice nel frammento precedente è commentata. Non è possibile che un riferimento Box acceda al campo weigth, perché Box non ne definisce uno.

CONCLUSIONI

Dopo aver finito questo e-book, potresti chiederti dove o come migliorare le tue capacità di programmazione in Java. In realtà questo e-book è solo un'introduzione al mondo di Java e puoi usare altri e-book, libri, forum su Internet, corsi on-line e altre risorse per espandere le tue conoscenze su Java.

Però ricorda con quanta paura o timore ti sei approcciato a questo e-book, probabilmente non immaginavi nemmeno che il tuo computer potesse trovare tutti i numeri divisibili per 11 in pochi secondi. Programmare in Java è molto più semplice di quanto sembri, non dovrei dirtelo perché migliaia di programmatori hanno usato le loro abilità Java per ottenere lavori ben pagati nello sviluppo di software, nella programmazione di server o nella creazione di app Android.

Ben presto, e con molta pratica, diventerai un bravo programmatore Java infatti chiunque può imparare a scrivere programmi per computer. Ricorda, però, che Java è solo uno dei linguaggi di programmazione da imparare. Molti lo preferiscono perché è un linguaggio utile, potente e moderno che viene utilizzato dalle aziende di tutto il mondo.

Utilizza sempre l'ultima versione disponibile di Java e cerca di essere sempre aggiornato sulle novità e su cosa è cambiato dall'ultima versione.

Prova a creare dei programmi che dispongono di un'interfaccia utente grafica, prova a sviluppare servizi Web, crea app Android o altro ancora. Per curiosità sappi che il videogioco Minecraft, che ha avuto un notevole successo, è interamente scritto in Java.

In questo e-book ti ho insegnato la programmazione Java da zero ma sono fiducioso nelle tue capacità e so che proseguirai per saperne sempre di più.

Come già detto il miglior modo per apprendere è la pratica quindi immagina un progetto e mettilo in opera, qualunque cosa sia purché sia

realizzabile con un computer. Non hai bisogno di un computer ultrapotente per sviluppare, non hai bisogno di laboratori o altri apparecchi. Puoi sviluppare di giorno, di notte, nel tempo libero, l'essenziale è imparare il linguaggio e se dovessi accorgerti che proprio non ti piace, potrai sempre passare ad un altro linguaggio.

Continua ad impegnarti e vedrai che ben presto il tuo progetto sarà realtà!

IL GLOSSARIO JAVA

Applet: le applet sono programmi Java basati sulle classi Applet o JApplet. Sono più strettamente associati alla capacità di fornire contenuto attivo all'interno delle pagine Web. Hanno diverse caratteristiche che li distinguono dalle normali applicazioni grafiche Java, come la mancanza di un metodo principale definito dall'utente e le restrizioni di sicurezza che limitano le loro capacità di eseguire alcune normali attività.

Baseline di sicurezza: la baseline di sicurezza è l'aggiornamento minimo consigliato per Java.

CA (Autorità di certificazione, Certificate Authority): il certificato digitale viene aggiunto alle applicazioni del computer per confermare che l'applicazione è stata effettivamente fornita dal proprietario del certificato.

Cache del browser Web: un'area di memorizzazione temporanea che contiene copie delle pagine del browser Web. Per risolvere alcuni problemi d'installazione o di configurazione di Java, a volte può rendersi necessario svuotare la cache manualmente, accedendo a un'apposita finestra di dialogo.

Classe: un progetto o un prototipo da cui vengono creati gli oggetti. Un esempio potrebbe essere una classe "Cliente". La classe può definire gli attributi come "nome", "numero cliente", "indirizzo", ecc. La classe definisce anche i comportamenti (metodi) per gli oggetti creati con la classe. È possibile utilizzare metodi di esempio per impostare l'indirizzo del cliente, modificare l'indirizzo del cliente e ottenere l'indirizzo.

Classi Bootstrap: classi che compongono la Java Platform Core Application Programming Interface (API), come quelle che si trovano in java.lang, java.io e pacchetti java.io.

Corpo di classe: il corpo di una definizione di classe. Il corpo raggruppa le definizioni dei membri di una classe: campi, metodi e classi nidificate.

Costante di classe: una variabile definita sia finale che statica.

CPU (Aggiornamento patch critiche, Critical Patch Update): gli aggiornamenti di patch critiche sono set di patch con correzioni per la sicurezza forniti secondo una pianificazione fissa.

CVE (Vulnerabilità ed esposizioni comuni, Common Vulnerabilities and Exposures): i numeri CVE sono identificativi univoci comuni per le informazioni note pubblicamente sulle vulnerabilità di sicurezza.

Data di scadenza di Java: Java fornirà altre avvertenze e promemoria per l'aggiornamento alla versione più recente.

Eccezione: un oggetto che rappresenta il verificarsi di una circostanza eccezionale - in genere, qualcosa che è andato storto nell'esecuzione di un programma. Gli oggetti Exception vengono creati da classi che estendono la classe Throwable.

Errore di confine: errori che derivano da errori di programmazione commessi ai margini di un problema: indicizzazione fuori dal bordo di un array, gestione di elementi di dati, terminazione del ciclo e così via. Gli errori di confine sono un tipo molto comune di errore logico.

Identificatore: un nome definito dal programmatore per una variabile, un metodo, una classe o un'interfaccia.

IFTW: la procedura d'installazione diretta da Web o da linea.

Intestazione di classe: l'intestazione di una definizione di classe. L'intestazione dà un nome alla classe e ne definisce l'accesso. Descrive anche se la classe estende una super classe o implementa qualsiasi interfaccia.

Manifest: un file speciale che contiene informazioni sui file all'interno di un package JAR.

Mappa immagine: un'immagine divisa in aree logiche, ognuna delle quali ha un punto caldo.

Memoria fisica: molto spesso nel sito java.com con questo termine si fa riferimento al dispositivo semiconduttore di memorizzazione presente nei computer noto come RAM (memoria ad accesso casuale).

Metodo nativo: un metodo scritto in un linguaggio diverso da Java, ma accessibile a un programma Java.

Modulo: un gruppo di componenti del programma, in genere con visibilità limitata ai componenti del programma in altri moduli. Java usa i pacchetti per implementare questo concetto.

Parametro: un valore passato a una funzione che può utilizzare il parametro o eseguire un'operazione su di esso.

Programmazione parallela: uno stile di programmazione in cui le istruzioni non vengono necessariamente eseguite in una sequenza ordinata ma in parallelo. Semplificano la creazione di programmi progettati per essere eseguiti su hardware multiprocessore, ad esempio.

SDK Java 2: una particolare implementazione della funzionalità astratta descritta nelle specifiche Sun della piattaforma Java 2.

Segnalibro: utilizzato da un browser Web per ricordare i dettagli di un URL (Uniform Resource Locator).

Valore chiave: l'oggetto utilizzato per generare un codice hash associato per la ricerca in una struttura dati associativa.

Variabile di ciclo: una variabile utilizzata per controllare il funzionamento di un ciclo, ad esempio un ciclo for. In genere, a una variabile di ciclo verrà assegnato un valore iniziale e lo è quindi incrementato dopo ogni iterazione finché non raggiunge o supera un valore finale.