Parte xxviii

Java

«

Java: preparazione

Java è un linguaggio di programmazione realizzato da Sun Microsystems, utilizzato originariamente per l'inserzione di programmi all'interno di pagine HTML (applet), un po' come si fa con le immagini. Per questo motivo, il risultato consueto della compilazione di un sorgente Java è una codifica intermedia, indipendente dalla piattaforma, che deve poi essere interpretata localmente dal navigatore o da un altro programma indipendente. Tuttavia, nel tempo sono stati sviluppati anche compilatori alternativi, che producono un programma eseguibile tradizionale (dipendente dalla piattaforma hardware-software).

Per programmare in Java occorre un compilatore, generalmente noto come '**javac**', che sia in grado di generare il formato binario Java, il cosiddetto Java bytecode. Il file che si ottiene non è propriamente un eseguibile, in quanto necessita di un interprete che generalmente è il programma '**java**'.

Esiste una versione ufficiale di questi strumenti, definita JDK (*Java development kit*), e altre versioni indipendenti, come per esempio Kaffe.

Nel capitolo viene descritto in particolare come utilizzare Kaffe. Alla fine del capitolo si trova la descrizione dell'installazione e della configurazione di JDK originale, oltre a una sezione sull'uso di GCJ per la compilazione di sorgenti o binari Java nel formato eseguibile adatto alla propria architettura.

Kaffe

«

Kaffe [1] è un compilatore di sorgenti Java e un interprete di compilati in formato Java (Java bytecode). Attualmente, si tratta di un pacchetto standard delle distribuzioni GNU, per cui non ci dovrebbero essere problemi nella sua installazione. Attualmente, assieme al compilatore e all'interprete, dovrebbero essere disponibili anche le ***classi***, ovvero le librerie Java.

Classi

«

Le classi di Kaffe, che ormai accompagnano questo applicativo, dovrebbero essere contenute in un solo file compresso, che deve rimanere tale. Potrebbe trattarsi di '`/usr/share/kaffe/Klasses.jar`'.

Se si installa Kaffe autonomamente, senza affidarsi a un pacchetto già predisposto per la propria distribuzione GNU, potrebbe essere necessario definire alcune variabili di ambiente. Nell'esempio seguente si fa riferimento a uno script per una shell Bourne o derivata:

```
CLASSPATH=.:/usr/share/kaffe/Klasses.jar
KAFFEHOME=/usr/share/kaffe
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
export CLASSPATH
export KAFFEHOME
export LD_LIBRARY_PATH
```

Se Kaffe fosse stato installato a partire dalla directory '`/usr/local/`', si dovrebbe usare la definizione seguente:

```
CLASSPATH=.:/usr/local/share/kaffe/Klasses.jar
KAFFEHOME=/usr/local/share/kaffe
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
export CLASSPATH
export KAFFEHOME
export LD_LIBRARY_PATH
```

Merita un po' di attenzione la variabile '**`LD_LIBRARY_PATH`**' che potrebbe essere utilizzata anche da altri programmi. '**`LD_LIBRARY_PATH`**' deve contenere i percorsi in cui si trovano i file di libreria; se il proprio sistema utilizza applicazioni che collocano le proprie librerie all'interno di directory inconsuete, queste devono essere aggiunte all'elenco. Segue un esempio esplicativo:

```
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib:/opt/mio_prog/lib:/opt/tuo_prog/lib
```

Compilazione

Per verificare che la compilazione funzioni correttamente, basta preparare il solito programma banale che visualizza un messaggio attraverso lo standard output e poi termina:

```
class CiaoMondoApp
{
    public static void main (String[] args)
      {
        System.out.println ("Ciao Mondo!");
      }
}
```

Il file deve essere salvato con il nome '`CiaoMondoApp.java`'. Kaffe, tra le altre cose, fornisce un collegamento simbolico, denominato '**`javac`**', attraverso cui avviare la compilazione. Così la compilazione avviene nello stesso modo degli strumenti JDK originali:

```
$ javac CiaoMondoApp.java [Invio]
```

Se la sintassi del sorgente Java è corretta, si ottiene un file in formato binario Java, denominato '`CiaoMondoApp.class`'.

Esecuzione

Per eseguire il binario Java generato, ovvero il file '`.class`', occorre un interprete. In questo senso, il binario Java non ha bisogno necessariamente dei permessi di esecuzione, perché viene solo letto dall'interprete.

```
$ kaffe CiaoMondoApp [Invio]

  Ciao Mondo!
```

Come si può osservare dalla riga di comando, il file binario Java deve essere indicato senza l'estensione, che di conseguenza è obbligatoriamente '`.class`'. Kaffe si compone anche dello script '**`java`**', il cui scopo è quello di rendere il comando di interpretazione conforme al JDK; in pratica, '**`java`**' si limita ad avviare il comando '**`kaffe`**'.

`$` **`java CiaoMondoApp`** [*Invio*]

Tuttavia, questo script potrebbe essere modificato in modo da permettere l'avvio di un eseguibile Java anche se è stato fornito il nome del file corrispondente, completo di estensione '`.class`'. L'esempio seguente rappresenta le modifiche che potrebbero essere apportate in tal senso:

```
#!/bin/sh
#
# /usr/bin/java

CLASSE=`/bin/basename $1 .class`
shift
kaffe $CLASSE $@
```

Kernel Linux

Come è noto, uno script viene interpretato automaticamente in base alla convenzione per cui la prima riga inizia con l'indicazione del programma adatto. Per esempio: '**`#!/bin/sh`**', '**`#!/bin/bash`**' e '**`#!/usr/bin/perl`**'. Con i binari Java ciò non è possibile, quindi, per ottenere l'avvio automatico dell'interprete '**`java`**', occorre che il kernel ne sia informato. Per la precisione, occorre attivare la funzionalità generica di riconoscimento dei binari (sezione 8.3.1); inoltre occorre accertarsi che la directory '`/proc/sys/fs/binfmt_misc/`' contenga i file '`register`' e '`status`'.

Se le cose non stanno così, è necessario innestare il file system '**binfmt_misc**':

```
# mount -t binfmt_misc none /proc/sys/fs/binfmt_misc[Invio]
```

Una volta che sono disponibili i file virtuali '`register`' e '`status`', per attivare la funzionalità occorre intervenire con il comando seguente:

```
# echo 1 > /proc/sys/fs/binfmt_misc/status[Invio]
```

Per disattivarla, basta utilizzare il valore zero.

```
# echo 0 > /proc/sys/fs/binfmt_misc/status[Invio]
```

Quando tutto è in ordine per la gestione dei binari eterogenei, si può definire quali file devono essere riconosciuti e quali interpreti devono essere avviati di conseguenza. Nel caso dei binari Java normali, si tratta di eseguire il comando seguente (il percorso dell'interprete, '`/usr/bin/java`' può essere cambiato a seconda delle proprie necessità).

```
# echo ':Java:M::\xca\xfe\xba\xbe::/usr/bin/java:'  ↩
↪> /proc/sys/fs/binfmt_misc/register[Invio]
```

In alternativa, se si è sicuri dell'estensione '`.class`', si può utilizzare il comando seguente:

```
# echo ':Java:E::class::/usr/bin/java:'  ↩
↪> /proc/sys/fs/binfmt_misc/register[Invio]
```

Per verificare che la definizione sia stata recepita correttamente dal kernel, si può leggere il contenuto del file virtuale '`/proc/sys/fs/binfmt_misc/Java`', creato a seguito di uno dei due comandi mostrati sopra.

Quando il kernel è predisposto nel modo appena visto, si possono rendere eseguibili i file binari Java; così, quando si tenta di avviarli, il kernel stesso avvia invece il comando seguente:

```
java file_binario_java argomenti
```

Lo svantaggio di questo sistema sta nel fatto che il nome del file binario Java viene indicato con tutta l'estensione, cosa che normalmente crea dei problemi, sia a Kaffe che al JDK. Per questo, conviene che '`/usr/bin/java`' sia uno script predisposto per risolvere il problema, come già mostrato nella sezione precedente.

Se invece di usare Kaffe si usa il JDK originale, conviene modificare il nome dell'interprete Java, per esempio in '**`java1`**', realizzando poi un file script analogo a quello già visto.

```
#!/bin/sh
#
# /usr/bin/java

CLASSE=`/bin/basename $1 .class`
shift
java1 $CLASSE $@
```

C'è però una cosa che occorre tenere a mente. Con GNU/Linux, così come con altri sistemi, non è possibile avviare un eseguibile se il nome non viene indicato per esteso. In pratica, non è pensabile che succeda quanto accade in Dos in cui i file che finiscono per '`.COM`' o '`.EXE`' sono avviati semplicemente nominandoli senza estensione.

Per chi ha usato GNU/Linux da un po' di tempo ciò dovrebbe essere logico, ma con Java si rischia ancora di essere ingannati: il fatto che, sia l'interprete '**`java`**' originale, sia '**`kaffe`**', vogliano il nome dell'eseguibile Java senza l'estensione '`.class`', non deve fa-

re supporre che ciò valga anche per il kernel. Per cui, se si avvia '`CiaoMondoApp.class`' nel modo seguente,

```
$ java CiaoMondoApp [Invio]
```

quando si vuole che sia il kernel a fare tutto questo in modo automatico, il comando diviene il seguente:

```
$ CiaoMondoApp.class [Invio]
```

Se si tentasse si eseguire il comando seguente, si otterrebbe una segnalazione di errore del tipo: '**`command not found`**'.

```
$ CiaoMondoApp [Invio]
```

Applet

«

Un'applet Java è un programma particolare che può essere incorporato in un documento HTML. Il meccanismo è simile all'inserzione di immagini; l'effetto è quello di un programma grafico che, invece di utilizzare una finestra si inserisce in un'area prestabilita del documento HTML. Un'applet Java non può quindi vivere da sola, richiede sempre l'abbinamento a una pagina HTML.

Il modo migliore per vedere il funzionamento di un programma del genere è attraverso l'utilizzo di un navigatore in grado di eseguire tali applet.

Verifica del funzionamento

«

Per verificare il funzionamento di un'applet si può provare il solito programma banale. In questo caso si comincia con la realizzazione di una pagina HTML che incorpori l'applet che si vuole realizzare.

```
<!-- CiaoMondo.html -->
<HTML>
<HEAD>
    <TITLE>La mia prima applet</TITLE>
</HEAD>
<BODY>
<OBJECT CODETYPE="application/java"
    CLASSID="java:CiaoMondo.class">
Applet Java
</OBJECT>
</BODY>
</HTML>
```

Come si vede, l'elemento '**OBJECT**' dichiara l'utilizzo dell'applet 'CiaoMondo.class'. Segue il sorgente dell'applet:

```
// CiaoMondo.java

import java.applet.Applet;
import java.awt.Graphics;

public class CiaoMondo extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString ("Ciao Mondo!", 50, 25);
    }
}
```

Si compila il sorgente 'CiaoMondo.java' nel solito modo, ottenendo il binario Java 'CiaoMondo.class'

```
$ javac CiaoMondo.java[Invio]
```

Quando si carica il file 'CiaoMondo.html' attraverso un navigatore adatto, incontrando l'elemento '**OBJECT**' che fa riferimento al binario Java '**CiaoMondo.class**', viene caricato il programma 'CiaoMondo.class' nell'area stabilita.

All'interno di quell'area, a partire dall'angolo superiore sinistro, vengono calcolate le coordinate (x=50, y=25) dell'istruzione

‘**`g.drawString("Ciao mondo!", 50, 25)`**’ vista nell’applet.

JDK

«

software non libero

JDK [2] è il pacchetto originale per la compilazione e l’esecuzione di applicativi Java. Viene distribuito in forma binaria, già compilata. Per ottenerlo, si può consultare *http://www.blackdown.org/* o eventualmente si può fare una ricerca attraverso *http://www.google.com* per i file contenenti la stringa ‘`linux-jdk`’ (si potrebbero trovare nomi come ‘`linux-jdk.1.1.3-v2.tar.gz`’). Se si desidera installare il JDK è importante verificare di non avere tracce di Kaffe.

Il JDK può essere installato a partire da qualunque punto del proprio file system. Qui viene proposta l’installazione a partire da ‘`/opt/`’.

Se nel proprio sistema non è presente, la si può creare, quindi al suo interno si può espandere il contenuto del pacchetto JDK. Si ottiene così la directory ‘`jdk`***versione***`/`’, per esempio ‘`jdk1.1.3/`’. Per motivi pratici è opportuno modificare il nome della directory, o creare un collegamento simbolico, in modo che vi si possa accedere utilizzando il nome ‘`/opt/java/`’.

Prima di poter funzionare, il JDK deve essere configurato attraverso delle variabili di ambiente opportune. Nell’esempio seguente si mostra un pezzo di script per una shell Bourne o derivata, in grado di predisporre le variabili necessarie:

```
PATH="/opt/java/bin:$PATH"
CLASSPATH=.:/opt/java/lib/classes.zip:/opt/java/lib/classes
JAVA_HOME=/opt/java
export PATH
export CLASSPATH
export JAVA_HOME
```

Per il funzionamento si può rivedere quanto già indicato per Kaffe. In questo caso, utilizzando il JDK originale, il compilatore è proprio '**javac**' e l'esecutore (o interprete) è '**java**'.

GCJ

GCJ [3] è un programma frontale per il controllo del compilatore GCC e di altri programmi accessori, il cui scopo è quello di compilare sorgenti Java.

La compilazione può avvenire a diversi livelli: da sorgenti Java ('`.java`') o da binari Java ('`.class`') si può arrivare a un file eseguibile per il proprio sistema operativo; in alternativa si possono semplicemente compilare dei sorgenti Java per generare i binari Java corrispondenti ('`.class`'). Semplificando le cose, si possono distinguere questi tre tipi di comandi per la compilazione:

- `gcj -C` ***file_sorgente_java*** ...

  per generare binari Java (file '`.class`');

- `gcj --main=`***classe_principale*** `-o` ***file_da_generare*** ***file_sorgente_java*** ...

  per generare un eseguibile a partire da dei sorgenti Java (file '`.java`');

- `gcj --main=`***classe_principale*** `-o` ***file_da_generare*** ***binario_java***...

  per generare un eseguibile a partire da binari Java (file '`.class`').

Supponendo di avere il solito esempio già visto in precedenza,

```
class CiaoMondoApp
{
    public static void main (String[] args)
    {
        System.out.println ("Ciao Mondo!");
    }
}
```

supponendo questa volta che sia contenuto nel file '`ciao_mondo.java`', si può generare il binario Java '`CiaoMondoApp.class`' con il comando seguente:

`$` **`gcj -C ciao_mondo.java`** [*Invio*]

Per compilare il binario Java in modo da ottenere un binario adatto al sistema operativo e all'architettura del proprio elaboratore, si può usare il comando seguente, generando quindi l'eseguibile '**`ciao`**':

`$` **`gcj --main=CiaoMondoApp -o ciao CiaoMondoApp.class`** [*Invio*]

Infine, per compilare direttamente il sorgente Java, si può agire nello stesso modo:

`$` **`gcj --main=CiaoMondoApp -o ciao ciao_mondo.java`** [*Invio*]

GCJ riconosce la variabile di ambiente '**`CLASSPATH`**', per la ricerca delle classi, fornendo anche la possibilità di indicare tale informazione attraverso la riga di comando, con delle opzioni che qui non vengono mostrate.

Opzione	Descrizione
`-C`	In questo caso, i file in ingresso sono sorgenti Java e vengono compilati generando le classi in forma di binari Java.
`--main=`***classe***	Questa opzione permette di stabilire quale sia la classe da utilizzare come principale, in modo che il programma che si genera inizi da lì il suo funzionamento.
`-o` ***file***	Definisce il nome dell'eseguibile da generare, quando la compilazione non è destinata a ottenere soltanto un binario Java.

Riferimenti

«

- *TransVirtual Technologies Inc.*
  http://www.transvirtual.com
- Riferimenti per ottenere il JDK dalla rete
  http://www.blackdown.org/
- *The source for Java, Documentation*
  http://java.sun.com/docs/index.html
- *The source for Java, Tutorial*
  http://java.sun.com/docs/books/tutorial/index.html

[1] **Kaffe** software libero con licenza speciale

[2] **JDK** software non libero

[3] **GCJ** GNU GPL

Questo capitolo introduce alla programmazione in Java, in modo superficiale, per dare un'idea delle potenzialità di questo linguaggio.

Struttura fondamentale

«

Java è un linguaggio di programmazione strettamente OO (*Object oriented*), cioè a dire che qualunque cosa si faccia, anche un semplice programma che emette un messaggio attraverso lo standard output, va trattato secondo la programmazione a oggetti.

Ciò significa anche che i componenti di questo linguaggio hanno nomi diversi da quelli consueti. Volendo fare un abbinamento approssimativo con un linguaggio di programmazione normale, si potrebbe dire che in Java i programmi sono ***classi*** e le funzioni sono ***metodi***. Naturalmente ci sono anche tante altre cose nuove.

Fatta questa premessa, si può dare un'occhiata alla solita classe banale: quella che visualizza un messaggio e termina.

```
/**
 *  CiaoMondoApp.java
 *  La solita classe banale.
 */

import java.lang.*; // predefinita

class CiaoMondoApp
{
    public static void main (String[] args)
    {
        System.out.println ("Ciao Mondo!"); // visualizza il messaggio
    }
}
```

Il sorgente Java ha molte somiglianze con quello del linguaggio C e qui si intendono segnalare le particolarità rispetto a quel linguaggio.

Commenti

Java ammette l'uso di commenti in stile C, nella solita forma '**/*...*/**', ma ne introduce altri due tipi: uno per la creazione automatica di documentazione, nella forma '**/**...*/**', e uno per fare ignorare tutto ciò che appare a partire dal simbolo di commento fino alla fine della riga, nella forma '***//commento***':

```
/* commento_generico */
```

```
/** documentazione */
```

```
// commento_fino_alla_fine_della_riga
```

Tutti e tre questi tipi di commenti servono a fare ignorare al compilatore una parte del sorgente e questo dovrebbe bastare al prin-

cipiante. Convenzionalmente, è conveniente usare il commento di documentazione per la spiegazione di ciò che fa la classe, all'inizio del sorgente.

Nomi ed estensioni

«

Le estensioni dei file Java sono definite in modo obbligatorio: '`.java`' per i sorgenti e '`.class`' per le classi (i binari Java).

Generalmente, nel sorgente, il nome della classe deve corrispondere alla radice del nome del sorgente e, di conseguenza, anche del binario Java. Per lo stile convenzionale di Java, questo nome inizia con una lettera maiuscola e non contiene simboli strani; se è composto dall'unione di più parole, ognuna di queste inizia con una lettera maiuscola.

Istruzioni

«

Le istruzioni seguono la convenzione del linguaggio C, per cui terminano con un punto e virgola ('`;`') e i raggruppamenti di queste, detti anche blocchi, si fanno utilizzando le parentesi graffe ('`{ }`').

```
istruzione ;
```

```
{ istruzione ;  istruzione ;  istruzione ;  }
```

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale.

Ogni programma in Java deve fare affidamento sull'utilizzo di classi fondamentali che compongono il linguaggio stesso. L'importazione delle classi necessarie viene fatta attraverso l'istruzione '**`import`**', indicando una classe particolare o un gruppo (nel secondo caso si usa un asterisco).

Nell'esempio introduttivo vengono importate tutte le classi del pacchetto '**`java.lang`**', anche se non sarebbe stato necessario dichiararlo, dato che queste classi vengono sempre importate in modo predefinito (senza di queste, nessuna classe potrebbe funzionare).

Le classi standard di Java (cioè queste librerie fondamentali), sono contenute normalmente in un archivio compresso '`.zip`', oppure '`.jar`'. Si è visto nel capitolo u122 che è importante indicare il percorso in cui si trovano, nella variabile di ambiente '**`CLASSPATH`**'.

Osservando il contenuto di questo file, si può comprendere meglio il concetto di pacchetto di classi. Segue solo un breve estratto:

```
Archive:  classes.zip
 Length     Date    Time    Name
 ------     ----    ----    ----
      0  05-19-97  22:46   java/
      0  05-19-97  22:24   java/lang/
   1322  05-19-97  22:24   java/lang/Object.class
   4202  05-19-97  22:24   java/lang/Class.class
    …
   3450  05-19-97  22:24   java/lang/System.class
    …
      0  05-19-97  22:26   java/util/
    …
      0  05-19-97  22:26   java/io/
    …
      0  05-19-97  22:42   java/awt/
    …
```

Ecco che così può diventare più chiaro il fatto che, importare tutte le classi del pacchetto '**`java.lang`**' significa in pratica includere tutte le classi contenute nella directory '`java/lang/`', anche se qui si tratta solo di un file compresso.

Dichiarazione della classe

Generalmente, un file sorgente Java contiene la dichiarazione di una sola classe, il cui nome corrisponde alla radice del file sorgente. La dichiarazione della classe delimita in pratica il contenuto del sorgente, definendo eventuali ***ereditarietà*** da altre classi esistenti.

Quando una classe non eredita da un'altra, si parla convenzionalmente di ***applicazione***, mentre quando eredita dalla classe '**`java.applet.Applet`**' (cioè da '`java/applet/Applet.class`') si usa la definizione ***applet***.

Contenuto della classe

La classe contiene essenzialmente dichiarazioni di variabili e metodi. L'esecuzione di un metodo dipende da una chiamata, detta anche ***messaggio***. Perché una classe si traduca in un programma autonomo, occorre che al suo interno ci sia un metodo che viene eseguito in modo automatico all'avvio.

Nel caso delle classi che non ereditano nulla da altre, come nell'esempio, ci deve essere il metodo '**`main`**' che viene eseguito all'avvio del binario Java contenente la classe stessa. Quando una classe eredita da un'altra, queste regole sono stabilite dalla classe ereditata.

Il metodo '**`main`**' è formato necessariamente come nell'esempio: '**`public static void main(String[] args) {...}`**'.

Variabili e tipi di dati

In Java si distinguono fondamentalmente due tipi di rappresentazione dei dati: primitivi e riferimenti a oggetti. I tipi di dati primitivi sono per esempio i soliti tipi numerici (intero, a virgola mobile, ecc.); gli altri sono ***oggetti***. Un oggetto è quindi una variabile contenente un riferimento a una struttura, più o meno complessa. In Java, gli array e le stringhe sono oggetti; pertanto non esistono tipi di dati primitivi equivalenti.

I nomi delle variabili possono essere composti utilizzando caratteri Unicode. Naturalmente, non è possibile utilizzare nomi coincidenti con parole chiave già utilizzate dal linguaggio stesso. La convenzione stilistica di Java richiede che il nome delle variabili inizi con la lettera minuscola; inoltre, se si tratta di un nome composto, la convenzione richiede di segnalare l'inizio di ogni nuova parola con una lettera maiuscola. Per esempio: '`miaVariabile`', '`dataOdierna`', '`elencoNomiFemminili`'.

Chiamata per valore

In Java, le chiamate dei metodi avvengono trasferendo il valore degli argomenti indicati nella chiamata stessa. Ciò significa che le modifiche che si dovessero apportare all'interno dei metodi non si riflettono all'indietro. Tuttavia, questo ragionamento vale solo per i tipi di dati primitivi, dal momento che quando si utilizzano degli oggetti, essendo questi dei riferimenti, le variazioni fatte al loro interno rimangono anche dopo la chiamata.

Variabili e tipi di dati

«

Si è già accennato al fatto che Java distingue tra due tipi di dati, primitivi e riferimenti a oggetti (o più semplicemente solo oggetti). L'esempio seguente mostra la dichiarazione di un intero all'interno di un metodo e il suo incremento fino a raggiungere un valore predefinito:

```
/**
 *  DieciXApp.java
 *  Un esempio di utilizzo delle variabili.
 */

import java.lang.*; // predefinita

class DieciXApp
{
    public static void main (String[] args)
    {
        int contatore = 0;

        // Inizia un ciclo in cui si emettono 10 «x» attraverso lo
        // standard output.
        while (contatore < 10)
          {
            contatore++;
            System.out.println ("x"); // emette una «x»
          }
    }
}
```

Tipi

«

I tipi di dati primitivi rappresentano un valore singolo. Il loro elenco si trova nella tabella u123.4.

Tabella u123.4. Elenco dei tipi di dati primitivi in Java.

Tipo	Dimensione	Descrizione
`byte`	8 bit, complemento a due.	Intero a 8 bit.
`short`	16 bit, complemento a due.	Intero ridotto.
`int`	32 bit, complemento a due.	Intero normale.
`long`	64 bit, complemento a due.	Intero molto grande.
`float`	32 bit	Virgola mobile, singola precisione.
`double`	64 bit	Virgola mobile, doppia precisione.
`char`	16 bit, carattere Unicode.	Carattere.
`boolean`	*Vero* o *Falso*.	Valore booleano.

Nell'esempio mostrato precedentemente, viene dichiarato un intero normale, '**`contatore`**', inizializzato al valore zero, che poi viene incrementato all'interno di un ciclo:

```
        int contatore = 0;

        // Inizia un ciclo in cui si emettono 10 «x» attraverso lo
        // standard output.
        while (contatore < 10)
        {
            contatore++;
            System.out.println ("x"); // emette una «x»
        }
```

«

Ogni tipo primitivo ha la possibilità di essere rappresentato in forma di costante letterale. La tabella u123.6 mostra l'elenco dei tipi di dati abbinati alla rappresentazione in forma di costante letterale.

Tabella u123.6. Elenco dei tipi di dati primitivi abbinati a una possibile rappresentazione in forma di costante letterale.

Tipo	Esempio di costante	Descrizione o intervallo
`byte`	123	−128..+127
`short`	12345	−32768..+32767
`int`	1234567890	$-(2^{31})..+((2^{31})-1)$
`long`	123456789012345678**90**	$-(2^{63})..+((2^{63})-1)$
`float`	(float)123.456	La costante con virgola è sempre a doppia precisione.
`double`	123.456	
`char`	'A'	Si usano gli apici semplici.
`boolean`	true	Si usano le parole chiave '**`true`**' e '**`false`**'.

È importante osservare che una costante numerica a virgola mobile è sempre a doppia precisione, per cui, se si vuole assegnare a una variabile a singola precisione ('**`float`**') una costante letterale, occorre una conversione di tipo, per mezzo di un cast. In seguito vengono descritte le stringhe, che si delimitano utilizzando gli apici doppi. Per ora è solo il caso ti tenere in considerazione che in Java le stringhe

non sono tipi di dati primitivi, ma oggetti veri e propri.

Campo di azione

Il campo di azione delle variabili in Java viene determinato dalla posizione in cui queste vengono dichiarate. Ciò determina il momento della loro creazione e distruzione. A fianco del concetto del campo di azione, si pone quello della ***protezione***, che può limitare l'accessibilità di una variabile. La protezione viene analizzata in seguito.

A seconda del loro campo di azione, si distinguono in particolare tre categorie più importanti di variabili: variabili appartenenti alla classe (*member variable*), variabili locali e parametri dei metodi.

Variabili appartenenti alla classe

Queste variabili appartengono alle classi e come tali sono dichiarate all'interno delle classi stesse, ma all'esterno dei metodi. L'esempio seguente mostra la dichiarazione della variabile '**`serveAQualcosa`**' come parte della classe '**`FaQualcosa`**'.

```
class FaQualcosa
{
    int serveAQualcosa = 0;

    // Dichiarazione dei metodi
    ...
}
```

Variabili locali

Sono variabili dichiarate all'interno dei metodi. Vengono create alla chiamata del metodo e distrutte alla sua conclusione. Per questo sono visibili solo all'interno del metodo che le dichiara.

Nell'esempio visto in precedenza, quello che visualizza 10 «x», la variabile '**`contatore`**' veniva dichiarata all'interno del metodo '**`main`**'.

Parametri dei metodi

Le variabili indicate in concomitanza con la dichiarazione di un metodo (quelle che appaiono tra parentesi tonde), vengono create nel momento della chiamata del metodo stesso e distrutte alla sua conclusione. Queste variabili contengono la copia degli argomenti utilizzati per la chiamata; in questo senso si dice che le chiamate ai metodi avvengono per valore.

Operatori ed espressioni

«

Gli operatori sono qualcosa che esegue un qualche tipo di funzione, su uno o due operandi, restituendo un valore. Il valore restituito è di tipo diverso a seconda degli operandi utilizzati. Per esempio, la somma di due interi genera un risultato intero.

Gli operandi descritti nelle sezioni seguenti sono solo quelli più comuni e importanti. In particolare, sono stati omessi quelli necessari al trattamento delle variabili in modo binario.

Operatori aritmetici

«

Gli operatori che intervengono su valori numerici sono elencati nella tabella u123.8.

Tabella u123.8. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
++*op*	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
−−*op*	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op−−	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+*op*	Non ha alcun effetto.
−*op*	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 − *op2*	Sottrae dal primo il secondo operando.
op1 ⋆ *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo -- il resto della divisione tra il primo e il secondo operando.
var = *valore*	Assegna alla variabile il valore alla destra.
op1 += *op2*	*op1* = *op1* + *op2*
op1 −= *op2*	*op1* = *op1* − *op2*

Operatore e operandi	Descrizione
op1 `*=` ***op2***	***op1*** `=` ***op1*** `*` ***op2***
op1 `/=` ***op2***	***op1*** `=` ***op1*** `/` ***op2***
op1 `%=` ***op2***	***op1*** `=` ***op1*** `%` ***op2***

Operatori di confronto e operatori logici

«

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è di tipo booleano, rappresentabile in Java dalle costanti letterali '**`true`**' e '**`false`**'. Gli operatori di confronto sono elencati nella tabella u123.9.

Tabella u123.9. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `==` ***op2***	*Vero* se gli operandi si equivalgono.
op1 `!=` ***op2***	*Vero* se gli operandi sono differenti.
op1 `<` ***op2***	*Vero* se il primo operando è minore del secondo.
op1 `>` ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.

Operatore e operandi	Descrizione
op1 `>=` ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici (noti normalmente come: AND, OR, NOT, ecc.). Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare che sia stata valutata effettivamente. Gli operatori logici sono elencati nella tabella u123.10.

Tabella u123.10. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`!` ***op***	Inverte il risultato logico dell'operando.
op1 `&&` ***op2***	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 `||` ***op2***	Se il risultato del primo operando è *Vero* non valuta il secondo.

Concatenamento di stringhe

Si è accennato al fatto che in Java, le stringhe siano oggetti e non tipi di dati primitivi. Esiste tuttavia la possibilità di indicare stringhe letterali nel modo consueto, attraverso la delimitazione con gli apici doppi.

Diverse stringhe possono essere concatenate, in modo da formare una stringa unica, attraverso l'operatore '**+**'.

```
    public static void main (String[] args)
    {
        int contatore = 0;

        while (contatore < 10)
          {
            contatore++;
            System.out.println ("Ciclo n. " + contatore);
          }
    }
```

Nel pezzo di codice appena mostrato, appare in particolare l'istruzione seguente:

```
            System.out.println ("Ciclo n. " + contatore);
```

L'espressione '**"Ciclo n. " + contatore**' si traduce nel risultato seguente:

```
Ciclo n. 1
Ciclo n. 2
…
Ciclo n. 10
```

In pratica, il contenuto della variabile '**contatore**' viene convertito automaticamente in stringa e unito alla costante letterale precedente.

Strutture di controllo del flusso

«

Le strutture di controllo del flusso delle istruzioni sono molto simili a quelle del linguaggio C. In particolare, dove può essere messa un'istruzione si può mettere anche un gruppo di istruzioni delimitate dalle parentesi graffe.

Normalmente, le strutture di controllo del flusso basano questo controllo sulla verifica di una condizione espressa all'interno di

parentesi tonde.

Struttura condizionale: «if»

«

```
if (condizione) istruzione
```

```
if (condizione) istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione (o il gruppo di istruzioni) seguente; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzato '**else**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne segue. Vengono mostrati alcuni esempi.

```
int importo;
…
if (importo > 10000000) System.out.println ("L'offerta è vantaggiosa");
```

```
int importo;
int memorizza;
…
if (importo > 10000000)
  {
    memorizza = importo;
    System.out.println ("L'offerta è vantaggiosa");
  }
else
  {
    System.out.println ("Lascia perdere");
  }
```

```
int importo;
int memorizza;
…
if (importo > 10000000)
  {
    memorizza = importo;
    System.out.println ("L'offerta è vantaggiosa");
  }
else if (importo > 5000000)
  {
    memorizza = importo;
    System.out.println ("L'offerta è accettabile");
  }
else
  {
    System.out.println ("Lascia perdere");
  }
```

Struttura di selezione: «switch»

«

L'istruzione '**`switch`**' è un po' troppo complessa per essere rappresentata in modo chiaro attraverso uno schema sintattico. In generale, l'istruzione '**`switch`**' permette di **saltare** a una certa posizione della struttura, in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero:

```
int mese;
…
switch (mese)
  {
    case 1: System.out.println ("gennaio"); break;
    case 2: System.out.println ("febbraio"); break;
    case 3: System.out.println ("marzo"); break;
    case 4: System.out.println ("aprile"); break;
    case 5: System.out.println ("maggio"); break;
    case 6: System.out.println ("giugno"); break;
    case 7: System.out.println ("luglio"); break;
    case 8: System.out.println ("agosto"); break;
    case 9: System.out.println ("settembre"); break;
    case 10: System.out.println ("ottobre"); break;
    case 11: System.out.println ("novembre"); break;
    case 12: System.out.println ("dicembre"); break;
  }
```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene aggiunta un'istruzione di salto '**`break`**', che serve a uscire dalla struttura, perché altrimenti le istruzioni del caso successivo, se c'è, verrebbero eseguite. Infatti, un gruppo di casi può essere raggruppato assieme, quando si vuole che questi eseguano lo stesso gruppo di istruzioni:

```
int mese;
int giorni;
…
switch (mese)
  {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        giorni = 31;
        break;
    case 4:
    case 6:
```

```
            case 9:
            case 11:
                giorni = 30;
                break;
            case 2:
                if (((anno % 4 == 0) && !(anno % 100 = 0))
                    || (anno % 400 == 0))
                    giorni = 29;
                else
                    giorni = 28;
                break;
          }
```

È anche possibile definire un caso predefinito che si verifichi quando nessuno degli altri si avvera:

```
        int mese;
        …
        switch (mese)
          {
            case 1: System.out.println ("gennaio"); break;
            case 2: System.out.println ("febbraio"); break;
            …
            case 11: System.out.println ("novembre"); break;
            case 12: System.out.println ("dicembre"); break;
            default: System.out.println ("mese non corretto"); break;
         }
```

Iterazione con condizione di uscita iniziale: «while»

«

```
while (condizione) istruzione
```

‘**`while`**’ esegue un’istruzione, o un gruppo di queste, finché la condizione restituisce il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell’esecuzione del successivo. Segue il pezzo dell’e-

sempio già visto, di quella classe che visualizza 10 volte la lettera «x»:

```
        int contatore = 0;

        while (contatore < 10)
          {
            contatore++;
            System.out.println ("x");
          }
```

Nel blocco di istruzioni di un ciclo '**`while`**', ne possono apparire alcune particolari:

- '**`break`**'

  esce definitivamente dal ciclo '**`while`**';

- '**`continue`**'

  interrompe l'esecuzione del gruppo di istruzioni e riprende dalla valutazione della condizione.

L'esempio seguente è una variante del ciclo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento di '**`break`**'. Si osservi che '**`while (true)`**' equivale a un ciclo senza fine, perché la condizione è sempre vera:

```
        int contatore = 0;

        while (true)
          {
            if (contatore >= 10)
              {
                break;
              }
            contatore++;
            System.out.println ("x");
          }
```

Iterazione con condizione di uscita finale: «do-while»

```
do blocco_di_istruzioni while (condizione);
```

'**do**' esegue un gruppo di istruzioni una volta e poi ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

Iterazione enumerativa: «for»

```
for (espressione1; espressione2; espressione3) istruzione
```

Questa è la forma tipica di un'istruzione '**for**', in cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni), mentre la terza serve per l'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, potrebbe esprimersi nella sintassi seguente:

```
for (var = n; condizione; var++) istruzione
```

Il ciclo '**for**' potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

Il vecchio esempio banale, in cui veniva visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un

ciclo ‘**for**’:

```
int contatore;

for (contatore = 0; contatore < 10; contatore++)
  {
    System.out.println ("x");
  }
```

Array e stringhe

In Java, array e stringhe sono oggetti. In pratica, la variabile che contiene un array o una stringa è in realtà un riferimento alla struttura di dati rispettiva.

Array

La dichiarazione di un array avviene in Java in modo molto semplice, senza l’indicazione esplicita del numero di elementi. La dichiarazione avviene come se si trattasse di un tipo di dati normale, con la differenza che si aggiungono una coppia di parentesi quadre a sottolineare che si tratta di un array di elementi di quel tipo. L’esempio seguente dichiara che ‘**arrayDiInteri**’ è un array in cui gli elementi sono di tipo intero (‘**int**’), senza specificare quanti siano:

```
int[] arrayDiInteri;
```

Per fare in modo che l’array esista effettivamente, occorre che questo sia inizializzato, fornendogli gli elementi. Si usa per questo l’operatore ‘**new**’ seguito dal tipo di dati con il numero di elementi racchiuso tra parentesi quadre. L’esempio seguente assegna alla variabile ‘**arrayDiInteri**’ il riferimento a un array composto da sette interi:

```
arrayDiInteri = new int[7];
```

Nella pratica, è normale inizializzare l'array quando lo si dichiara; per cui, quanto già visto si può ridurre all'esempio seguente:

```
int[] arrayDiInteri = new int[7];
```

Il riferimento a un elemento di un array avviene aggiungendo al nome della variabile che rappresenta l'array stesso, il numero dell'elemento, racchiuso tra parentesi quadre. Come nel linguaggio C, il primo elemento si raggiunge con l'indice zero, mentre l'ultimo corrisponde alla dimensione meno uno.

Si è detto che gli array sono oggetti. In particolare, è possibile determinare la dimensione di un array, espressa in numero di elementi, leggendo il contenuto della variabile '**`length`**' dell'oggetto array. Nel caso dell'esempio già visto, si tratta di leggere il contenuto di '**`arrayDiInteri.length`**'.

L'esempio seguente mostra una scansione di un array, indicando una condizione di interruzione del ciclo indipendente dalla conoscenza anticipata della dimensione dell'array stesso. In particolare, la variabile '**`i`**' viene dichiarata contestualmente con la sua inizializzazione, nella prima espressione di controllo del ciclo '**`for`**':

```
for (int i = 0; i < arrayDiInteri.length; i++) {
    arrayDiInteri[i] = i;
}
```

Un array può contenere sia elementi primitivi che riferimenti a oggetti. In questo modo si possono avere gli array multidimensionali. L'esempio seguente rappresenta il modo in cui può essere definito un array 3x2 di interi e anche come scandirne i vari elementi:

```
/**
 *      Matrice3x2App.java
 *      Esempio di uso di array multidimensionali.
 */
```

```
import java.lang.*; // predefinita

class Matrice3x2App
{
    public static void main (String[] args)
    {
        int[][] matrice = new int[3][2];

        for (int i = 0; i < matrice.length; i++)
          {
            for (int j = 0; j < matrice[i].length; j++)
              {
                matrice[i][j] = 1000 + j + i * 10;
                System.out.println ("matrice[" + i + "][" + j + "] = "
                                    + matrice[i][j]);
              }
          }
    }
}
```

L'esecuzione di questo piccolo programma, genera il risultato seguente:

```
matrice[0][0] = 1000
matrice[0][1] = 1001
matrice[1][0] = 1010
matrice[1][1] = 1011
matrice[2][0] = 1020
matrice[2][1] = 1021
```

Stringhe

Le stringhe in Java sono oggetti e se ne distinguono due tipi: stringhe costanti e stringhe variabili. La distinzione è utile perché questi due tipi di oggetti hanno bisogno di una forma di rappresentazione diversa. Così, ciò porta a un'ottimizzazione del programma, che per una stringa costante richiede meno risorse rispetto a una stringa che deve essere variabile, oltre a migliorare altri aspetti legati alla sicurezza.

La dichiarazione di una variabile che possa contente un riferimento a un oggetto stringa-costante, si ottiene con la dichiarazione seguente:

```
String variabile;
```

In pratica, si dichiara che la variabile può contenere un riferimento a un oggetto di tipo '**String**'. La creazione di questo oggetto '**String**' si ottiene come nel caso degli array, utilizzando l'operatore '**new**'.

```
new String (stringa);
```

L'esempio seguente crea la variabile '**stringaCostante**' di tipo '**String**' e la inizializza assegnandoci il riferimento a una stringa:

```
String stringaCostante = new String ("Ciao ciao.");
```

Fortunatamente, si possono utilizzare anche delle costanti letterali pure e semplici. Per cui la stringa '**"Ciao ciao."**' è già di per sé un oggetto stringa-costante.

Si è già accennato al fatto che le stringhe-costanti possono essere concatenate facilmente utilizzando l'operatore '**+**':

```
"Ciao " + "come " + "stai?"
```

L'esempio restituisce un'unica stringa-costante, come quella seguente:

```
"Ciao come stai?"
```

Inoltre, in questi concatenamenti, entro certi limiti, possono essere inseriti elementi diversi da stringhe, come nell'esempio seguente, dove il contenuto numerico intero della variabile '**contatore**'

viene convertito automaticamente in stringa prima di essere emesso attraverso lo standard output.

```
int contatore = 0;

while (contatore < 10)
{
    contatore++;
    System.out.println ("Ciclo n. " + contatore);
}
```

Le stringhe variabili sono oggetti di tipo '**StringBuffer**' e vengono descritte più avanti.

Metodo «main()»

Si è accennato al fatto che una classe che non eredita esplicitamente da un'altra, richiede l'esistenza del metodo '**main()**' e viene detta applicazione. Questo metodo deve avere una forma precisa e si tratta di quello che viene chiamato automaticamente quando si avvia il binario Java corrispondente alla classe stessa. Senza questa convenzione, non ci sarebbe un modo per avviare un programma Java.

```
public static void main (String[] args) { istruzioni }
```

Nella sintassi indicata, le parentesi graffe fanno parte della dichiarazione del metodo e delimitano un gruppo di istruzioni.

args

È Importante osservare l'unico parametro del metodo '**main()**': l'array '**args**' composto da elementi di tipo '**String**'. Questo array contiene gli argomenti passati al programma Java attraverso la riga di comando.

L'esempio seguente, mostra come si può leggere il contenuto di questo array, tenendo presente che non si conosce inizialmente la sua dimensione. L'esempio emette separatamente, attraverso lo standard output, l'elenco degli argomenti ricevuti.

```
/**
 *  LeggiArgomentiApp.java
 *  Legge gli argomenti e gli emette attraverso lo standard output.
 */

import java.lang.*; // predefinita

class LeggiArgomentiApp
{
    public static void main (String[] args)
    {
        int i;

        for (i = 0; i < args.length; i++)
          {
            System.out.println (args[i]);
          }
    }
}
```

Il capitolo precedente ha introdotto l'uso del linguaggio Java per arrivare a scrivere programmi elementari, utilizzando i metodi come se fossero delle funzioni pure e semplici. In questo capitolo si introducono gli oggetti secondo Java.

Creazione e distruzione di un oggetto

«

Un oggetto è un'***istanza*** di una classe, come una copia ottenuta da uno stampo. Così come nel caso della creazione di una variabile contenente un tipo di dati primitivo, si distinguono due fasi: la dichiarazione e l'inizializzazione. Trattandosi di un oggetto, l'inizializzazione richiede prima la creazione dell'oggetto stesso, in modo da poter assegnare alla variabile il riferimento di questo.

Dichiarazione dell'oggetto

«

La dichiarazione di un oggetto è precisamente la dichiarazione di una variabile atta a contenere un riferimento a un particolare tipo di oggetto, specificato dalla classe che può generarlo.

classe variabile

La sintassi appena mostrata dovrebbe essere sufficientemente chiara. Nell'esempio seguente si dichiara la variabile '**`miaStringa`**' predisposta a contenere un riferimento a un oggetto di tipo '**`String`**'.

```
String miaStringa;
```

La semplice dichiarazione della variabile non basta a creare l'oggetto, in quanto così si crea solo il contenitore adatto.

Instanza di un oggetto

L'istanza di un oggetto si ottiene utilizzando l'operatore '**new**' seguito da una chiamata a un metodo particolare il cui scopo è quello di inizializzare opportunamente il nuovo oggetto che viene creato. In pratica, '**new**' alloca memoria per il nuovo oggetto, mentre il metodo chiamato lo prepara. Alla fine, viene restituito un riferimento all'oggetto appena creato.

L'esempio seguente, definisce la variabile '**miaStringa**' predisposta a contenere un riferimento a un oggetto di tipo '**String**', creando contestualmente un nuovo oggetto '**String**' inizializzato in modo da contenere un messaggio di saluto.

```
String miaStringa = new String ("Ciao ciao.");
```

Metodo costruttore

L'inizializzazione di un oggetto viene svolta da un metodo specializzato per questo scopo: il ***costruttore***. Una classe può fornire diversi metodi costruttori che possono servire a inizializzare in modo diverso l'oggetto che si ottiene. Tuttavia, convenzionalmente, ogni classe fornisce sempre un metodo il cui nome corrisponde a quello della classe stessa, ed è senza argomenti. Questo metodo esiste anche se non viene indicato espressamente all'interno della classe.

Java consente di utilizzare lo stesso nome per metodi che accettano argomenti in quantità o tipi diversi, perché è in grado di distingue-

re il metodo chiamato effettivamente in base agli argomenti forniti. Questo meccanismo permette di avere classi con diversi metodi costruttori, che richiedono una serie differente di argomenti.

Utilizzo degli oggetti

«

Finché non si utilizza in pratica un oggetto non si può apprezzare, né comprendere, la programmazione a oggetti. Un oggetto è una sorta di scatola nera a cui si accede attraverso variabili e metodi dell'oggetto stesso.

Si indica una variabile o un metodo di un oggetto aggiungendo un punto ('`.`') al riferimento dell'oggetto, seguito dal nome della variabile o del metodo da raggiungere. Variabili e metodi si distinguono perché questi ultimi possono avere una serie di argomenti racchiusi tra parentesi (se non hanno argomenti, vengono usate le parentesi senza nulla all'interno).

riferimento_all'oggetto **.** ***variabile***

riferimento_all'oggetto **.** ***metodo*** `()`

Prima di proseguire, è bene soffermarsi sul significato si tutto questo. Indicare una cosa come '**`oggetto.variabile`**', significa raggiungere una variabile appartenente a una particolare struttura di dati, che è appunto l'oggetto. In un certo senso, ciò si avvicina all'accesso a un elemento di un array.

Un po' più difficile è comprendere il senso di un metodo di un oggetto. Indicare '**`oggetto.metodo()`**' significa chiamare una funzione

che interviene in un ambiente particolare: quello dell'oggetto.

A questo punto, è necessario chiarire che il riferimento all'oggetto è qualunque cosa in grado di restituire un riferimento a questo. Normalmente si tratta di una variabile, ma questa potrebbe appartenere a sua volta a un altro oggetto. È evidente che sta poi al programmatore cercare si scrivere un programma leggibile.

Nella programmazione a oggetti si insegna comunemente che si dovrebbe evitare di accedere direttamente alle variabili, cercando di utilizzare il più possibile i metodi. Si immagini l'esempio seguente che è solo ipotetico:

```
class Divisione
{
    public int x;
    public int y;
    public calcola ()
    {
        return x/y;
    }
}
```

Se venisse creato un oggetto a partire da questa classe, si potrebbe modificare il contenuto delle variabili e quindi richiamare il calcolo, come nell'esempio seguente:

```
Divisione div = new Divisione ();
div.x = 10;
div.y = 5;
System.out.println ("Il risultato è " + div.calcola ());
```

Però, se si tenta di dividere per zero si ottiene un errore irreversibile. Se invece esistesse un metodo che si occupa di ricevere i dati da inserire nelle variabili, verificando prima che siano validi, si potrebbe evitare di dover prevedere questi inconvenienti.

L'esempio mostrato è volutamente banale, ma gli oggetti (ovvero

le classi che li generano) possono essere molto complessi; pertanto, la loro utilità sta proprio nel fatto di poter inserire al loro interno tutti i meccanismi di filtro e controllo necessari al loro buon funzionamento.

In conclusione, in Java è considerato un buon approccio di programmazione l'utilizzo delle variabili solo in lettura, senza poterle modificarle direttamente dall'esterno dell'oggetto.

La chiamata di un metodo di un oggetto viene anche detta ***messaggio***, per sottolineare il fatto che si invia un'informazione (eventualmente composta dagli argomenti del metodo) all'oggetto stesso.

Distruzione di un oggetto

«

In Java, un oggetto viene eliminato automaticamente quando non esistono più riferimenti alla sua struttura. In pratica, se viene creato un oggetto assegnando il suo riferimento a una variabile, quando questa viene eliminata perché è terminato il suo campo di azione, anche l'oggetto viene eliminato.

Tuttavia, l'eliminazione di un oggetto non può essere presa tanto alla leggera. Un oggetto potrebbe avere in carico la gestione di un file che deve essere chiuso prima dell'eliminazione dell'oggetto stesso. Per questo, esiste un sistema di eliminazione degli oggetti, definito *garbage collector*, o più semplicemente ***spazzino***, che prima di eliminare un oggetto gli permette di eseguire un metodo conclusivo: '`finalize()`'. Questo metodo potrebbe occuparsi di chiudere i file rimasti aperti e di concludere ogni altra cosa necessaria.

Classi

Le classi sono lo stampo, o il prototipo, da cui si ottengono gli oggetti. La sintassi per la creazione di una classe è la seguente. Le parentesi graffe fanno parte dell'istruzione necessaria a creare la classe e ne delimitano il contenuto, ovvero il corpo, costituito dalla dichiarazione di variabili e metodi. Convenzionalmente, il nome di una classe inizia con una lettera maiuscola.

```
[modificatore] class classe [extends classe_superiore] [
implements elenco_interfacce] {…}
```

Il modificatore può essere costituito da uno dei nomi seguenti, a cui corrisponde un valore differente della classe.

Modificatore	Descrizione
`public`	Quando la classe è accessibile anche al di fuori del pacchetto di classi cui appartiene, si utilizza il modificatore '**`public`**'. Se questo non viene indicato, la classe è accessibile solo all'interno del pacchetto cui appartiene.
`abstract`	Quando una classe serve solo come modello astratto per generare altre sottoclassi si utilizza il modificatore '**`abstract`**'.
`final`	Quando si vuole evitare che una classe possa generare altre sottoclassi si indica il modificatore '**`final`**'.

Tutte le classi ereditano automaticamente dalla classe '**`java.lang.Object`**', quando non viene dichiarano espressamente di ereditare da un'altra. La dichiarazione esplicita di volere ereditare da una classe particolare, si ottiene attraverso la parola chiave '**`extends`**' seguita dal nome della classe stessa.

A fianco dell'eredità da un'altra classe, si abbina il concetto di interfaccia, che rappresenta solo un'impostazione a cui si vuole fare riferimento. Questa impostazione non è un'eredità, ma solo un modo per definire una struttura standard che si vuole sia attuata nella classe che si va a creare.

L'eredità avviene sempre solo da una classe, mentre le interfacce che si vogliono utilizzare nella classe possono essere diverse. Se si vogliono specificare più interfacce, i nomi di queste vanno separati con la virgola.

Nel corpo di una classe possono apparire dichiarazioni di variabili e metodi, definiti anche ***membri*** della classe.

Variabili

«

Le variabili dichiarate all'interno di una classe, ma all'esterno dei metodi, fanno parte dei cosiddetti membri, sottintendendo con questo che si tratta di componenti delle classi (anche i metodi sono definiti membri). La dichiarazione di una variabile di questo tipo, può essere espressa in forma piuttosto articolata. La sintassi seguente mostra solo gli aspetti più importanti.

```
[specificatore_di_accesso] [static] [final] tipo variabile [
= valore_iniziale]
```

Lo specificatore di accesso rappresenta la visibilità della variabile ed è qualcosa di diverso dal campo di azione, che al contrario rappresenta il ciclo vitale di questa. Per definire questa visibilità si utilizza una parola chiave il cui elenco e significato è descritto nella sezione i124.2.3.

La parola chiave '**`static`**' indica che si tratta di una variabile appartenente strettamente alla classe, mentre la mancanza di questa indicazione farebbe sì che si tratti di una variabile di istanza. Quando si dichiarano variabili statiche, si intende che ogni istanza (ogni oggetto generato) della classe che le contiene faccia riferimento alle stesse variabili. Al contrario, in presenza di variabili non statiche, ogni istanza della classe genera una nuova copia indipendente di queste variabili.

La parola chiave '**`final`**' indica che si tratta di una variabile che non può essere modificata, in pratica si tratta di una costante. In tal caso, la variabile deve essere inizializzata contemporaneamente alla sua creazione.

Il nome di una variabile inizia convenzionalmente con una lettera minuscola, ma quando si tratta di una costante, si preferisce usare solo lettere maiuscole.

Metodi

I metodi, assieme alle variabili dichiarate all'esterno dei metodi, fanno parte dei cosiddetti membri delle classi. La sintassi seguente mostra solo gli aspetti più importanti della dichiarazione di un metodo. Le parentesi graffe fanno parte dell'istruzione necessaria a creare il metodo e ne delimitano il contenuto, ovvero il corpo.

```
[specificatore_di_accesso] [static] [abstract] [final]
tipo_restituito ↩
↪metodo ( [elenco_parametri] ) [throws elenco_eccezioni] {…}
```

Lo specificatore di accesso rappresenta la visibilità del metodo. Per definire questa visibilità si utilizza una parola chiave il cui elenco e significato è descritto nella sezione i124.2.3.

La parola chiave '**`static`**' indica che si tratta di un metodo appartenente strettamente alla classe, mentre la mancanza di questa indicazione farebbe sì che si tratti di un metodo di istanza. I metodi statici possono accedere solo a variabili statiche; di conseguenza, per essere chiamati non c'è bisogno di creare un'istanza della classe che li contiene. Il metodo normale, non statico, richiede la creazione di un'istanza della classe che lo contiene per poter essere eseguito.

La parola chiave '**`abstract`**' indica che si tratta della struttura di un metodo, del quale vengono indicate solo le caratteristiche esterne, senza definirne il contenuto.

La parola chiave '**`final`**' indica che si tratta di un metodo che non può essere dichiarato nuovamente, nel senso che non può essere modificato in una sottoclasse eventuale.

Il tipo di dati restituito viene indicato prima del nome, utilizzando la stessa definizione che si darebbe a una variabile normale. Nel caso si tratti di un metodo che non restituisce alcunché, si utilizza la parola chiave '**`void`**'.

Il nome di un metodo inizia convenzionalmente con una lettera minuscola, come nel caso delle variabili.

L'elenco di parametri è composto da nessuno o più nomi di variabili precedute dal tipo. Questa elencazione corrisponde implicitamente alla creazione di altrettante variabili locali contenenti il valore corrispondente (in base alla posizione) utilizzato nella chiamata.

La parola chiave '**`throws`**' introduce un elenco di oggetti utili per su-

perare gli errori generati durante l'esecuzione del programma. Tale gestione non viene analizzata in questa documentazione su Java.

Sovraccarico

Java ammette il ***sovraccarico*** dei metodi. Questo significa che, all'interno della stessa classe, si possono dichiarare metodi differenti con lo stesso nome, purché sia diverso il numero o il tipo di parametri che possono accettare. In pratica, il metodo giusto viene riconosciuto alla chiamata in base agli argomenti che vengono forniti.

Chiamata di un metodo

La chiamata di un metodo avviene in modo simile a quanto si fa con le chiamate di funzione negli altri linguaggi. La differenza fondamentale sta nella necessità di indicare l'oggetto a cui si riferisce la chiamata.

Java consente anche di eseguire chiamate di metodi riferiti a una classe, quando si tratta di metodi statici.

Specificatore di accesso

Lo specificatore di accesso di variabili e metodi permette di limitare o estendere l'accessibilità di questi, sia per una questione di ordine (nascondendo i nomi di variabili e metodi cui non ha senso accedere da una posizione determinata), sia per motivi di sicurezza.

La tabella u124.6 mostra in modo sintetico e chiaro l'accessibilità dei componenti in base al tipo di specificatore indicato.

Tabella u124.6. Accessibilità di variabili e metodi in base all'uso di specificatori di accesso.

Specificatore	Classe	Sottoclasse	Pacchetto di classi	Altri
`package`	X		X	
`private`	X			
`protected`	X	X	X	
`public`	X	X	X	X

Se le variabili o i metodi vengono dichiarati senza l'indicazione esplicita di uno specificatore di accesso, viene utilizzato il tipo '**`package`**' in modo predefinito.

Sottoclassi

«

Una sottoclasse è una classe che eredita esplicitamente da un'altra. A questo proposito, è il caso di ripetere che tutte le classi ereditano in modo predefinito da '**`java.lang.Object`**', se non viene specificato diversamente attraverso la parola chiave '**`extends`**'.

Quando si crea una sottoclasse, si ereditano tutte le variabili e i metodi che compongono la classe, salvo quei componenti che risultano oscurati dallo specificatore di accesso. Tuttavia, la classe può dichiarare nuovamente alcuni di quei componenti e si può ancora accedere a quelli della classe precedente, nonostante tutto.

La parola chiave '**super**' rappresenta un oggetto contenente esclusivamente componenti provenienti dalla classe di livello gerarchico precedente. Questo permette di accedere a variabili e metodi che la classe dell'oggetto in questione ha ridefinito. L'esempio seguente mostra la dichiarazione di due classi: la seconda estende la prima.

```
class MiaClasse
{
    int intero;
    void mioMetodo ()
    {
        intero = 100;
    }
}
```

```
class MiaSottoclasse extends MiaClasse
{
    int intero;
    void mioMetodo ()
    {
        intero = 0;
        super.mioMetodo ();
        System.out.println (intero);
        System.out.println (super.intero);
    }
}
```

La coppia di classi mostrata sopra è fatta per generare un oggetto a partire dalla seconda, quindi per eseguire il metodo '**mioMetodo()**' su questo oggetto. Il metodo a essere eseguito effettivamente è quello della sottoclasse.

Quando ci si comporta in questo modo, ridefinendo un metodo in una sottoclasse, è normale che questo richiami il metodo della classe superiore, in modo da aggiungere solo il codice sorgente che serve in più. In questo caso, viene richiamato il metodo omonimo della classe superiore utilizzando '**super**' come riferimento.

Nello stesso modo, è possibile accedere alla variabile '**intero**' della classe superiore, anche se in quella attuale tale variabile viene ridefinita.

È il caso di osservare che la parola chiave '**super**' ha senso solo quando dalla classe si genera un oggetto. Quando si utilizzano metodi e variabili statici per evitare di dover generare l'istanza di un oggetto, non è possibile utilizzare questa tecnica per raggiungere metodi e variabili di una classe superiore.

this

«

La parola chiave '**this**' permette di fare riferimento esplicitamente all'oggetto stesso. Ciò può essere utile in alcune circostanze, come nell'esempio seguente:

```
class MiaClasse
{
    int imponibile;
    int imposta;
    void datiFiscali (int imponibile, int imposta)
    {
        this.imponibile = imponibile;
        this.imposta = imposta;
    }
    …
}
```

La classe appena mostrata dichiara due variabili che servono a conservare le informazioni su imponibile e imposta. Il metodo '**datiFiscali()**' permette di modificare questi dati in base agli argomenti con cui viene chiamato.

Per comodità, il metodo indica con gli stessi nomi le variabili utilizzate per ricevere i valori delle chiamate. Tali variabili diventano locali e oscurano le variabili di istanza omonime. Per poter accedere alle variabili di istanza si utilizza quindi la parola chiave '**`this`**'.

Anche in questa situazione, la parola chiave '**`this`**' ha senso solo quando dalla classe si genera un oggetto.

Interfacce

In Java, l'interfaccia è una raccolta di costanti e di definizioni di metodi senza attuazione. In un certo senso, si tratta di una sorta di prototipo di classe. Le interfacce non seguono la gerarchia delle classi perché rappresentano una struttura indipendente: un'interfaccia può ereditare da una o più interfacce definite precedentemente (al contrario delle classi che possono ereditare da una sola classe superiore), ma non può ereditare da una classe.

Nel caso di interfacce, non è corretto parlare di ereditarietà, ma questo concetto rende l'idea di ciò che succede effettivamente.

La sintassi per la definizione di un'interfaccia, è la seguente:

```
[public] interface interfaccia [extends elenco_interfacce_superiori]
{…}
```

Il modificatore '**`public`**' fa in modo che l'interfaccia sia accessibile a qualunque classe, indipendentemente dal pacchetto di classi

cui questa possa appartenere. Al contrario, se non viene utilizzato, l'interfaccia risulta accessibile solo alle classi dello stesso pacchetto.

La parola chiave '**extends**' permette di indicare una o più interfacce superiori da cui ereditare.

Contenuto di un'interfaccia

Un'interfaccia può contenere solo la dichiarazione di costanti e di metodi astratti (senza attuazione). In pratica, non viene indicato alcuno specificatore di accesso e nessun'altra definizione che non sia il tipo, come nell'esempio seguente:

```
interface Raccoltina
{
    int LIMITEMASSIMO = 1000;

    void aggiungi (Object, obj);
    int conteggio ();
    …
}
```

Si intende implicitamente che le variabili siano '**public**', '**static**' e '**final**', inoltre si intende che i metodi siano '**public**' e '**abstract**'.

Come si può osservare dall'esempio, la definizione dei metodi termina con l'indicazione dei parametri. Il corpo dei metodi, ovvero la loro attuazione, non viene indicato, perché non è questo il compito di un'interfaccia.

Utilizzo di un'interfaccia

Un'interfaccia viene utilizzata in pratica quando una classe dichiara di attuare (realizzare) una o più interfacce. L'esempio seguente

mostra l'utilizzo della parola chiave '**implements**' per dichiarare il legame con l'interfaccia vista nella sezione precedente:

```
class MiaClasse implements Raccoltina
{
    …
    void aggiungi (Object, obj)
    {
        …
    }
    int conteggio ()
    {
        …
    }
    …
}
```

In pratica, la classe che attua un'interfaccia, è obbligata a definire i metodi che l'interfaccia si limita a dichiarare in modo astratto. Si tratta quindi solo di una forma di standardizzazione e di controllo attraverso la stessa compilazione.

Pacchetti di classi

In Java si realizzano delle librerie di classi e interfacce attraverso la costruzione di pacchetti, come già accennato in precedenza. L'esempio seguente mostra due sorgenti Java, '**Uno.java**' e '**Due.java**' rispettivamente, appartenenti allo stesso pacchetto denominato '**PaccoDono**'. La dichiarazione dell'appartenenza al pacchetto viene fatta all'inizio, con l'istruzione '**package**'.

```
/**
 *  Uno.java
 *  Classe pubblica appartenente al pacchetto «PaccoDono».
 */

package PaccoDono;

public class Uno
{
    public void Visualizza ()
    {
        System.out.println ("Ciao Mondo - Uno");
    }
}
```

```
/**
 *  Due.java
 *  Classe pubblica appartenente al pacchetto «PaccoDono».
 */

package PaccoDono;

public class Due
{
    public void Visualizza ()
    {
        System.out.println ("Ciao Mondo - Due");
    }
}
```

Collocazione dei pacchetti

«

Quando si dichiara in un sorgente che una classe appartiene a un certo pacchetto, si intende che il binario Java corrispondente (il file '`.class`') sia collocato in una directory con il nome di quel pacchetto. Nell'esempio visto in precedenza si utilizzava la dichiarazione seguente:

```
package PaccoDono;
```

In tal modo, la classe (o le classi) di quel sorgente deve poi essere collocata nella directory '`PaccoDono/`'. Questa directory, a sua volta, deve trovarsi all'interno dei percorsi definiti nella variabile di ambiente '**`CLASSPATH`**'.

La variabile '**`CLASSPATH`**' è già stata vista in riferimento al file '`classes.zip`' o '`Klasses.jar`' (a seconda del tipo di compilatore e interprete Java), che si è detto contenere le librerie standard di Java. Tali librerie sono in effetti dei pacchetti di classi.

Il file '`classes.zip`' (o il file '`Klasses.jar`') potrebbe essere decompresso a partire dalla posizione in cui si trova, ma generalmente questo non si fa.

Se per ipotesi si decidesse di collocare la directory '`PaccoDono/`' a partire dalla propria directory personale, si potrebbe aggiungere nello script di configurazione della propria shell, qualcosa come l'istruzione seguente (adatta a una shell derivata da quella di Bourne).

```
CLASSPATH="$HOME:$CLASSPATH"
export CLASSPATH
```

Generalmente, per permettere l'accesso a pacchetti installati a partire dalla stessa directory di lavoro (nel caso del nostro esempio si tratterebbe di '`./PaccoDono/`'), si può aggiungere anche questa ai percorsi di '**`CLASSPATH`**'.

```
CLASSPATH=".:$HOME:$CLASSPATH"
export CLASSPATH
```

Utilizzo di classi di un pacchetto

«

L'utilizzo di classi da un pacchetto è già stato visto nei primi esempi, dove si faceva riferimento al fatto che ogni classe importa implicitamente le classi del pacchetto '**`java.lang`**'. Si importa una classe con un'istruzione simile all'esempio seguente:

```
import MioPacchetto.MiaClasse;
```

Per importare tutte le classi di un pacchetto, si utilizza un'istruzione simile all'esempio seguente:

```
import MioPacchetto.*;
```

In realtà, la dichiarazione dell'importazione di una o più classi, non è indispensabile, perché si potrebbe fare riferimento a quelle classi utilizzando un nome che comprende anche il pacchetto, separato attraverso un punto.

L'esempio seguente rappresenta un programma banale che utilizza le due classi mostrate negli esempi all'inizio di queste sezioni dedicate ai pacchetti:

```
/**
 *  MiaProva.java
 *  Classe che accede alle classi del pacchetto «PaccoDono».
 */

import PaccoDono.*;

class MiaProva
{
    public static void main (String[] args)
    {
        // Dichiara due oggetti dalle classi del pacchetto PaccoDono.
        Uno primo = new Uno ();
        Due secondo = new Due ();

        // Utilizza i metodi degli oggetti.
        primo.Visualizza ();
```

```
        secondo.Visualizza ();
    }
}
```

L'effetto che si ottiene è la sola emissione dei messaggi seguenti attraverso lo standard output:

```
Ciao Mondo - Uno
Ciao Mondo - Due
```

Se nel file non fosse stato dichiarato esplicitamente l'utilizzo di tutte le classi del pacchetto, sarebbe stato possibile accedere ugualmente alle sue classi utilizzando una notazione completa, che comprende anche il nome del pacchetto stesso. In pratica, l'esempio si modificherebbe come segue:

```
/**
 *  MiaProva.java
 *  Classe che accede alle classi del pacchetto «PaccoDono».
 */

class MiaProva
{
    public static void main (String[] args)
    {
        // Dichiara due oggetti dalle classi del pacchetto PaccoDono.
        PaccoDono.Uno primo = new PaccoDono.Uno ();
        PaccoDono.Due secondo = new PaccoDono.Due ();

        // Utilizza i metodi degli oggetti.
        primo.Visualizza ();
        secondo.Visualizza ();
    }
}
```

Esempi

«

Gli esempi mostrati nelle sezioni seguenti sono molto semplici, nel senso che si limitano a mostrare messaggi attraverso lo standard output. Si tratta quindi di pretesti per vedere come utilizzare quanto spiegato in questo capitolo. Viene usata in particolare la classe seguente per ottenere degli oggetti e delle sottoclassi:

```
/**
 *      SuperApp.java
 */

class SuperApp
{
    static int variabileStatica = 0; // variabile statica o di classe
    int variabileDiIstanza = 0;  // variabile di istanza

    // Nelle applicazioni è obbligatoria la presenza di questo metodo.
    public static void main (String[] args)
    {
        // Se viene avviata questa classe da sola, viene visualizzato
        // il messaggio seguente.
        System.out.println ("Ciao!");
    }

    // Metodo statico. Può essere usato per accedere solo alla
    // variabile statica.
    public static void metodoStatico ()
    {
        variabileStatica++;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
    }

    // Metodo di istanza. Può essere usato per accedere sia alla
    // variabile statica che a quella di istanza.
    public void metodoDiIstanza ()
    {
        variabileStatica++;
        variabileDiIstanza++;
        System.out.println
```

```
            ("La variabile statica ha raggiunto il valore "
             + variabileStatica);
          System.out.println
            ("La variabile di istanza ha raggiunto il valore "
             + variabileDiIstanza);
    }
}
```

Oggetti e messaggi

Si crea un oggetto a partire da una classe, contenuta generalmente in un pacchetto. Nella sezione precedente è stata presentata una classe che si intende non appartenga ad alcun pacchetto di classi. Ugualmente può essere utilizzata per creare degli oggetti.

L'esempio seguente crea un oggetto a partire da quella classe e quindi esegue la chiamata del metodo '**`metodoDiIstanza`**', che emette due messaggi, per ora senza significato:

```
/**
 *      EsempioOggetti1App.java
 */

class EsempioOggetti1App
{
    public static void main (String[] args)
    {
        SuperApp oSuperApp = new SuperApp ();
        oSuperApp.metodoDiIstanza ();
    }
}
```

Variabili di istanza e variabili statiche

Le variabili di istanza appartengono all'oggetto, per cui, ogni volta che si crea un oggetto a partire da una classe si crea una nuova copia di queste variabili. Le variabili statiche, al contrario, appartengono

a tutti gli oggetti della classe, per cui, quando si crea un nuovo oggetto, per queste variabili viene creato un riferimento all'unica copia esistente.

L'esempio seguente è una variante di quello precedente in cui si creano due oggetti dalla stessa classe, quindi viene chiamato lo stesso metodo, prima da un oggetto, poi dall'altro. Il metodo '**metodoDiIstanza()**' incrementa due variabili: una di istanza e l'altra statica.

```
/**
 *      EsempioOggetti2App.java
 */

class EsempioOggetti2App
{
    public static void main (String[] args)
    {
        SuperApp oSuperApp = new SuperApp ();
        SuperApp oSuperAppBis = new SuperApp ();

        oSuperApp.metodoDiIstanza ();
        oSuperAppBis.metodoDiIstanza ();
    }
}
```

Avviando l'eseguibile Java che deriva da questa classe, si ottiene la visualizzazione del testo seguente:

```
La variabile statica ha raggiunto il valore 1
La variabile di istanza ha raggiunto il valore 1
La variabile statica ha raggiunto il valore 2
La variabile di istanza ha raggiunto il valore 1
```

Le prime due righe sono generate dalla chiamata '**oSuperApp.metodoDiIstanza()**', mentre le ultime due da '**oSuperAppBis.metodoDiIstanza()**'. Si può osservare che l'incremento della variabile statica avvenuto nella prima chiamata riferita all'oggetto '**oSuperApp**' si riflette anche nel secondo ogget-

to, '**oSuperAppBis**', che mostra un valore più grande rispetto alla variabile di istanza corrispondente.

Ereditarietà

Nella programmazione a oggetti, il modo più naturale di acquisire variabili e metodi è quello di ereditare da una classe superiore che fornisca ciò che serve. L'esempio seguente mostra una classe che estende quella dell'esempio introduttivo ('**SuperApp**'), aggiungendo due metodi:

```
/**
 *      SottoclasseApp.java
 */

class SottoclasseApp extends SuperApp
{
    public static void decrementaStatico ()
    {
        variabileStatica--;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
    }

    public void decrementaDiIstanza ()
    {
        variabileStatica--;
        variabileDiIstanza--;
        System.out.println
          ("La variabile statica ha raggiunto il valore "
           + variabileStatica);
        System.out.println
          ("La variabile di istanza ha raggiunto il valore "
           + variabileDiIstanza);
    }
}
```

Se dopo la compilazione si esegue questa classe, si ottiene l'esecu-

zione del metodo '**`main()`**' che è stato definito nella classe superiore. In pratica, si ottiene la visualizzazione di un semplice messaggio di saluto e nulla altro.

Metodi di istanza e metodi statici

«

Il metodo di istanza può accedere sia a variabili di istanza, sia a variabili statiche. Questo è stato visto nell'esempio del sorgente '`EsempioOggetti2App.java`', in cui il metodo '**`metodoDiIstanza()`**' incrementava e visualizzava il contenuto di due variabili, una di istanza e una statica.

I metodi statici possono accedere solo a variabili statiche, che come tali possono essere chiamati anche senza la necessità di creare un oggetto: basta fare riferimento direttamente alla classe. L'esempio mostra in che modo si possa chiamare il metodo '**`metodoStatico()`**' della classe '**`SuperApp`**', senza fare riferimento a un oggetto:

```
/**
 *      EsempioOggetti3App.java
 */

class EsempioOggetti3App
{
    public static void main (String[] args)
    {
        SuperApp.metodoStatico ();
    }
}
```

Nello stesso modo, quando in una classe si vuole chiamare un metodo senza dovere prima creare un oggetto, è necessario che i metodi in questione siano statici.

Java: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

Problemi elementari di programmazione

«

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Java.

Somma tra due numeri positivi

«

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è descritto nella sezione 62.3.1.

```
//
// java SommaApp <x> <y>
// Somma esclusivamente valori positivi.
//
import java.lang.*; // predefinita
//
class SommaApp
{
    //
    static int somma (int x, int y)
    {
        int i;
        int z = x;
        //
        for (i = 1; i <= y; i++)
          {
            z++;
          }
        return z;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        int y;
        //
        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();
        //
        System.out.println (x + "+" + y + "=" + somma (x, y));
    }
}
//
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
    static int somma (int x, int y)
    {
        int z = x;
        int i = 1;
        //
        while (i <= y)
          {
            z++;
            i++;
          }
        return z;
    }
```

Moltiplicazione di due numeri positivi attraverso la somma

«

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è descritto nella sezione 62.3.2.

```
//
// java MoltiplicaApp <x> <y>
// Moltiplica esclusivamente valori positivi.
//
import java.lang.*; // predefinita
//
class MoltiplicaApp
{
    //
    static int moltiplica (int x, int y)
    {
        int i;
        int z = 0;
        //
        for (i = 1; i <= y; i++)
          {
            z = z + x;
          }
        return z;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
```

```
        int y;
        //
        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();
        //
        System.out.println (x + "*" + y + "=" + moltiplica (x, y));
    }
}
//
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
    static int moltiplica (int x, int y)
    {
        int z = 0;
        int i = 1;
        //
        while (i <= y)
          {
            z = z + x;
            i++;
          }
        return z;
    }
```

Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è descritto nella sezione 62.3.3.

```
//
// java DividiApp <x> <y>
// Divide esclusivamente valori positivi.
//
import java.lang.*; // predefinita
//
class DividiApp
{
    //
    static int dividi (int x, int y)
    {
        int z = 0;
        int i = x;
```

```
        //
        while (i >= y)
          {
            i = i - y;
            z++;
          }
        return z;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        int y;
        //
        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();
        //
        System.out.println (x + ":" + y + "=" + dividi (x, y));
    }
}
//
```

Elevamento a potenza

«

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è descritto nella sezione 62.3.4.

```
//
// java ExpApp <x> <y>
// Elevamento a potenza di valori positivi interi.
//
import java.lang.*; // predefinita
//
class ExpApp
{
    //
    static int exp (int x, int y)
    {
        int z = 1;
        int i;
```

```
        //
        for (i = 1; i <= y; i++)
          {
            z = z * x;
          }
        return z;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        int y;
        //
        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();
        //
        System.out.println (x + "**" + y + "=" + exp (x, y));
    }
}
//
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
    static int exp (int x, int y)
    {
        int z = 1;
        int i = 1;
        //
        while (i <= y)
          {
            z = z * x;
            i++;
          }
        return z;
    }
```

Infine, si può usare anche un algoritmo ricorsivo:

```
    static int exp (int x, int y)
    {
        if (x == 0)
          {
            return 0;
          }
        else if (y == 0)
          {
            return 1;
          }
        else
          {
            return (x * exp (x, y-1));
          }
    }
```

Radice quadrata

«

Il problema della radice quadrata è descritto nella sezione 62.3.5.

```
//
// java RadiceApp <x>
// Estrazione della parte intera della radice quadrata.
//
import java.lang.*; // predefinita
//
class RadiceApp
{
    //
    static int radice (int x)
    {
        int z = 0;
        int t = 0;
        //
        while (true)
          {
            t = z * z;

            if (t > x)
              {
                //
                // È stato superato il valore massimo.
                //
                z--;
```

```
                return z;
              }
            z++;
          }
        //
        // Teoricamente, non dovrebbe mai arrivare qui.
        //
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        //
        x = Integer.valueOf(args[0]).intValue ();
        //
        System.out.println ("radq(" + x + ")=" + radice (x));
    }
}
//
```

Fattoriale

«

Il problema del fattoriale è descritto nella sezione 62.3.6.

```
//
// java FattorialeApp <x>
// Calcola il fattoriale di un valore intero.
//
import java.lang.*; // predefinita
//
class FattorialeApp
{
    //
    static int fattoriale (int x)
    {
        int i = x - 1;
        //
        while (i > 0)
          {
            x = x * i;
            i--;
```

```
            }
        return x;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        //
        x = Integer.valueOf(args[0]).intValue ();
        //
        System.out.println (x + "! = " + fattoriale (x));
    }
}
//
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo:

```
    static int fattoriale (int x)
    {
        if (x > 1)
          {
            return (x * fattoriale (x - 1));
          }
        else
          {
            return 1;
          }
        //
        // Teoricamente non dovrebbe arrivare qui.
        //
    }
```

Massimo comune divisore

«

Il problema del massimo comune divisore, tra due numeri positivi, è descritto nella sezione 62.3.7.

```
//
// java MCDApp <x> <y>
// Determina il massimo comune divisore tra due numeri interi positivi.
//
```

```
import java.lang.*; // predefinita
//
class MCDApp
{
    //
    static int mcd (int x, int y)
    {
        int i;
        int z = 0;
        //
        while (x != y)
          {
            if (x > y)
              {
                x = x - y;
              }
            else
              {
                y = y - x;
              }
          }
        return x;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        int y;
        //
        x = Integer.valueOf(args[0]).intValue ();
        y = Integer.valueOf(args[1]).intValue ();
        //
        System.out.println ("Il massimo comune divisore tra " + x
                            + " e " + y + " è " + mcd (x, y));
    }
}
//
```

Numero primo

«

Il problema della determinazione se un numero sia primo o meno, è descritto nella sezione 62.3.8.

```
//
// java PrimoApp <x>
// Determina se un numero sia primo o meno.
//
import java.lang.*; // predefinita
//
class PrimoApp
{
    //
    static boolean primo (int x)
    {
        boolean primo = true;
        int i = 2;
        int j;
        //
        while ((i < x) && primo)
          {
            j = x / i;
            j = x - (j * i);
            //
            if (j == 0)
              {
                primo = false;
              }
            else
              {
                i++;
              }
          }
        return primo;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int x;
        //
        x = Integer.valueOf(args[0]).intValue ();
```

```
        //
        if (primo (x))
          {
            System.out.println (x + " è un numero primo");
          }
        else
          {
            System.out.println (x + " non è un numero primo");
          }
    }
}
//
```

Scansione di array

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in Java.

Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è descritto nella sezione 62.4.1.

```
//
// java RicercaSeqApp
//
import java.lang.*; // predefinita
//
class RicercaSeqApp
{
    //
    static int ricercaseq (int[] lista, int x, int a, int z)
    {
        int i;
        //
        // Scandisce l'array alla ricerca dell'elemento.
        //
        for (i = a; i <= z; i++)
          {
            if (x == lista[i])
              {
```

```
                return i;
              }
          }
        //
        // La corrispondenza non è stata trovata.
        //
        return -1;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int[] lista = new int[args.length-1];
        int x;
        int i;
        //
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //
        x = Integer.valueOf(args[0]).intValue ();
        //
        for (i = 1; i < args.length; i++)
          {
            lista[i-1] = Integer.valueOf(args[i]).intValue ();
          }
        //
        // Esegue la ricerca.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //
        i = ricercaseq (lista, x, 0, lista.length-1);
        //
        // Visualizza il risultato.
        //
        System.out.println (x + " si trova nella posizione "
                            + i + ".");
    }
}
//
```

Esiste anche una soluzione ricorsiva che viene mostrata nella

subroutine seguente:

```
    static int ricercaseq (int[] lista, int x, int a, int z)
    {
        if (a > z)
          {
            //
            // La corrispondenza non è stata trovata.
            //
            return -1;
          }
        else if (x == lista[a])
          {
            return a;
          }
        else
          {
            return ricercaseq (lista, x, a+1, z);
          }
    }
```

Ricerca binaria

«

Il problema della ricerca binaria all'interno di un array, è descritto nella sezione 62.4.2.

```
//
// java RicercaBinApp.java
//
import java.lang.*; // predefinita
//
class RicercaBinApp
{
    //
    static int ricercabin (int[] lista, int x, int a, int z)
    {
        int m;
        //
        // Determina l'elemento centrale.
        //
        m = (a + z) / 2;
        //
        if (m < a)
          {
```

```
            //
            // Non restano elementi da controllare: l'elemento cercato
            // non c'è.
            //
            return -1;
          }
        else if (x < lista[m])
          {
            //
            // Si ripete la ricerca nella parte inferiore.
            //
            return ricercabin (lista, x, a, m-1);
          }
        else if (x > lista[m])
          {
            //
            // Si ripete la ricerca nella parte superiore.
            //
            return ricercabin (lista, x, m+1, z);
          }
        else
          {
            //
            // m rappresenta l'indice dell'elemento cercato.
            //
            return m;
          }
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int[] lista = new int[args.length-1];
        int x;
        int i;
        //
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //
        x = Integer.valueOf(args[0]).intValue ();
        //
        for (i = 1; i < args.length; i++)
          {
```

```
            lista[i-1] = Integer.valueOf(args[i]).intValue ();
          }
        //
        // Esegue la ricerca.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //
        i = ricercabin (lista, x, 0, lista.length-1);
        //
        // Visualizza il risultato.
        //
        System.out.println (x + " si trova nella posizione "
                            + i + ".");
    }
}
//
```

Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Java.

Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 62.5.1. Viene mostrata prima una soluzione iterativa e successivamente il metodo '**bsort**' in versione ricorsiva.

```
//
// java BSortApp
//
import java.lang.*; // predefinita
//
class BSortApp
{
    //
    static int[] bsort (int[] lista, int a, int z)
    {
        int scambio;
        int j;
```

```
        int k;
        //
        // Inizia il ciclo di scansione dell'array.
        //
        for (j = a; j < z; j++)
          {
            //
            // Scansione interna dell'array per collocare nella
            // posizione j l'elemento giusto.
            //
            for (k = j+1; k <= z; k++)
              {
                if (lista[k] < lista[j])
                  {
                    //
                    // Scambia i valori.
                    //
                    scambio = lista[k];
                    lista[k] = lista[j];
                    lista[j] = scambio;
                  }
              }
          }
        //
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento. Qui si restituisce ugualmente un
        // riferimento all'array ordinato.
        //
        return lista;
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int[] lista = new int[args.length];
        int i;
        //
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //
        for (i = 0; i < args.length; i++)
          {
            lista[i] = Integer.valueOf(args[i]).intValue ();
```

```
          }
        //
        // Ordina l'array.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //
        bsort (lista, 0, args.length-1);
        //
        // Visualizza il risultato.
        //
        for (i = 0; i < lista.length; i++)
          {
            System.out.println ("lista[" + i + "] = "
                                + lista[i]);
          }
    }

}
//
```

Segue il metodo '**`bsort`**' in versione ricorsiva:

```
    static int[] bsort (int[] lista, int a, int z)
    {
        int scambio;
        int k;
        //
        if (a < z)
          {
            //
            // Scansione interna dell'array per collocare nella
            // posizione a l'elemento giusto.
            //
            for (k = a+1; k <= z; k++)
              {
                if (lista[k] < lista[a])
                  {
                    //
                    // Scambia i valori.
                    //
                    scambio = lista[k];
                    lista[k] = lista[a];
                    lista[a] = scambio;
                  }
              }
            bsort (lista, a+1, z);
          }
        return lista;
    }
```

Torre di Hanoi

«

Il problema della torre di Hanoi è descritto nella sezione 62.5.3.

```
//
// java HanoiApp <n-anelli> <piolo-iniziale> <piolo-finale>
//
import java.lang.*; // predefinita
//
class HanoiApp
{
    //
    static void hanoi (int n, int p1, int p2)
    {
        if (n > 0)
          {
```

```
            hanoi (n-1, p1, 6-p1-p2);
            System.out.println ("Muovi l'anello " + n
                                + " dal piolo " + p1
                                + " al piolo " + p2 + ".");
            hanoi (n-1, 6-p1-p2, p2);
          }
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int n;
        int p1;
        int p2;
        //
        n = Integer.valueOf(args[0]).intValue ();
        p1 = Integer.valueOf(args[1]).intValue ();
        p2 = Integer.valueOf(args[2]).intValue ();
        //
        hanoi (n, p1, p2);
    }
}
//
```

Quicksort

«

L'algoritmo del Quicksort è stato descritto nella sezione 62.5.4.

```
//
// java QSortApp
//
import java.lang.*; // predefinita
//
class QSortApp
{
    //
    static int part (int[] lista, int a, int z)
    {
        int scambio;
        //
        // Si assume che a sia inferiore a z.
        //
```

```
        int i = a + 1;
        int cf = z;
        //
        // Inizia il ciclo di scansione dell'array.
        //
        while (true)
          {
            while (true)
              {
                //
                // Sposta i a destra.
                //
                if ((lista[i] > lista[a]) || (i >= cf))
                  {
                    break;
                  }
                else
                  {
                    i++;
                  }
              }
            while (true)
              {
                //
                // Sposta cf a sinistra.
                //
                if (lista[cf] <= lista[a])
                  {
                    break;
                  }
                else
                  {
                    cf--;
                  }
              }
            //
            if (cf <= i)
              {
                //
                // È avvenuto l'incontro tra i e cf.
                //
                break;
              }
            else
```

```
            {
              //
              // Vengono scambiati i valori.
              //
              scambio = lista[cf];
              lista[cf] = lista[i];
              lista[i] = scambio;
              //
              i++;
              cf--;
            }
        }
      //
      // A questo punto lista[a..z] è stata ripartita e cf è la
      // collocazione di lista[a].
      //
      scambio = lista[cf];
      lista[cf] = lista[a];
      lista[a] = scambio;
      //
      // A questo punto, lista[cf] è un elemento (un valore) nella
      // giusta posizione.
      //
      return cf;
  }
  //
  static int[] quicksort (int[] lista, int a, int z)
  {
      int cf;
      //
      if (z > a)
        {
          cf = part (lista, a, z);
          quicksort (lista, a, cf-1);
          quicksort (lista, cf+1, z);
        }
      //
      // In Java, gli array sono oggetti e come tali vengono passati
      // per riferimento. Qui si restituisce ugualmente un
      // riferimento all'array ordinato.
      //
      return lista;
  }
  //
```

```
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int[] lista = new int[args.length];
        int i;
        //
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //
        for (i = 0; i < args.length; i++)
          {
            lista[i] = Integer.valueOf(args[i]).intValue ();
          }
        //
        // Ordina l'array.
        // In Java, gli array sono oggetti e come tali vengono passati
        // per riferimento.
        //
        quicksort (lista, 0, args.length-1);
        //
        // Visualizza il risultato.
        //
        for (i = 0; i < lista.length; i++)
          {
            System.out.println ("lista[" + i + "] = "
                                + lista[i]);
          }
    }
}
//
```

Permutazioni

«

L'algoritmo ricorsivo delle permutazioni è descritto nella sezione 62.5.5.

```
//
// java PermutaApp
//
import java.lang.*; // predefinita
//
```

```
class PermutaApp
{
    //
    static void permuta (int[] lista, int a, int z)
    {
        int scambio;
        int k;
        int i;
        //
        // Se il segmento di array contiene almeno due elementi, si
        // procede.
        //
        if ((z - a) >= 1)
          {
            //
            // Inizia un ciclo di scambi tra l'ultimo elemento e uno
            // degli altri contenuti nel segmento di array.
            //
            for (k = z; k >= a; k--)
              {
                //
                // Scambia i valori.
                //
                scambio = lista[k];
                lista[k] = lista[z];
                lista[z] = scambio;
                //
                // Esegue una chiamata ricorsiva per permutare un
                // segmento più piccolo dell'array.
                //
                permuta (lista, a, z-1);
                //
                // Scambia i valori.
                //
                scambio = lista[k];
                lista[k] = lista[z];
                lista[z] = scambio;
              }
          }
        else
          {
            //
            // Visualizza la situazione attuale dell'array.
            //
```

```
            for (i = 0; i < lista.length; i++)
              {
                System.out.print (" " + lista[i]);
              }
            System.out.println ("");
          }
    }
    //
    // Inizio del programma.
    //
    public static void main (String[] args)
    {
        int[] lista = new int[args.length];
        int i;
        //
        // Conversione degli argomenti della riga di comando in
        // numeri.
        //
        for (i = 0; i < args.length; i++)
          {
            lista[i] = Integer.valueOf(args[i]).intValue ();
          }
        //
        // Esegue le permutazioni.
        //
        permuta (lista, 0, args.length-1);
    }
}
//
```