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.