Capitolo 7

Costrutti avanzati

Andrea Gini

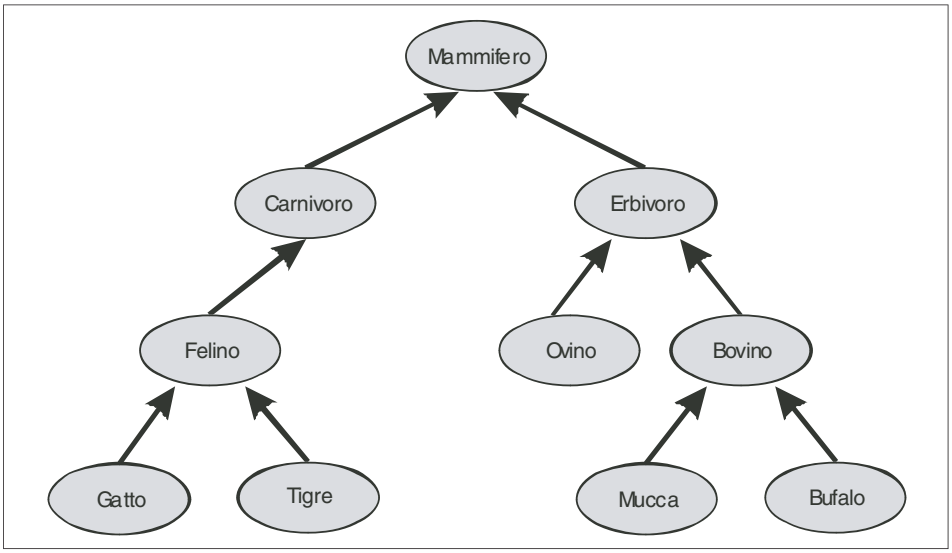
Classi astratte

La classificazione degli oggetti permette di realizzare in modo incrementale entità software dotate di gradi crescenti di specializzazione. Ogni elemento della gerarchia può essere sviluppato e testato in modo indipendente: le migliorie apportate a una classe verranno trasmesse automaticamente a tutte le sottoclassi.

Durante la fase di classificazione capita di creare classi che, pur essendo caratterizzate da attributi e metodi ben precisi, non corrispondono a entità concrete. Nella classificazione degli esseri viventi, per esempio, esiste la categoria dei mammiferi, che racchiude un insieme di attributi comuni a diverse specie viventi (cani, gatti, maiali e così via), ma che di per sé non rappresenta nessun animale. Nella classificazione delle specie animali è possibile trovare ulteriori esempi: in figura 7.1, le categorie Erbivoro, Carnivoro, Felino, Ovino e Bovino non corrispondono ad alcun animale concreto, ma rappresentano comunque passaggi fondamentali nella classificazione.

Nella progettazione del software si presenta spesso un problema simile: le gerarchie molto articolate presentano nodi che corrispondono a categorie astratte di oggetti, indispensabili come passaggi logici di derivazione, ma di fatto non istanziabili. Tali classi vengono dette *astratte* e, a differenza delle classi concrete, possono contenere speciali metodi `abstract`, privi di corpo, che andranno implementati nelle sottoclassi.

Figura 7.1 – *Classificazione delle specie viventi.*



Per definire una classe astratta è necessario aggiungere il modificatore abstract, sia nella dichiarazione della classe sia in quella dei metodi privi di implementazione. Nell'esempio seguente viene definita una classe astratta Mammifero, dotata di tre metodi astratti: ingerisci(), digerisci() ed evacua():

```
public abstract class Mammifero {

 public void mangia(Cibo c) {
   ingerisci(Cibo c);
   digerisci();
   evacua();
 }
 public abstract void ingerisci(Cibo c);
 public abstract void digerisci();
 public abstract void evacua();
}
```

Le modalità di ingestione, digestione ed evacuazione sono differenti in ognuna delle sottocategorie: ogni sottoclasse concreta di Mammifero sarà tenuta a fornire un'implementazione di tali metodi, che rifletta la natura particolare dell'entità rappresentata (nei mammiferi carnivori la digestione è sostanzialmente differente rispetto a quella dei mammiferi erbivori ruminanti). Si noti comunque che una classe astratta può contenere metodi concreti, come il metodo mangia nell'esempio, e che tali metodi possono chiamare liberamente i metodi astratti.

Il contesto statico: variabili e metodi di classe

Gli attributi visti finora sono associati alle istanze di una classe: ogni singola istanza possiede una copia privata di tali attributi, e i metodi della classe lavorano su tale copia privata. Esiste la possibilità di definire attributi statici, ossia variabili legate alla classe, presenti in copia unica e accessibili da tutte le istanze. Allo stesso modo è possibile definire metodi statici, ossia metodi non legati alle singole istanze, che possono operare esclusivamente su variabili statiche. Attributi e metodi statici vengono chiamati anche "attributi e metodi di classe", per distinguerli dagli attributi e dai metodi visti fino a ora, che vengono detti "di istanza".

Metodi e variabili di classe costituiscono il contesto statico di una classe. Per accedere al contesto statico di una classe non è necessario crearne un'istanza; ogni metodo statico è accessibile direttamente mediante la sintassi:

```
NomeClasse.metodoStatico();
```

All'interno di un metodo di classe è possibile accedere solamente a metodi e attributi statici; inoltre, in tale contesto gli identificatori this e super sono privi di significato.

Gli attributi e i metodi di classe, caratterizzati dal modificatore static, sono utili per rappresentare caratteristiche comuni a tutto l'insieme di oggetti appartenenti a una classe. Per esempio, in un'ipotetica classe che denoti le finestre grafiche di un desktop compariranno sia attributi di istanza (come la posizione e la dimensione, che sono caratteristiche di ogni singola finestra) sia attributi di classe (come il colore della barra del titolo, uguale per tutte le finestre presenti nel sistema). Questa situazione può essere rappresentata dal seguente esempio:

```
public class Window {
  private int x;
  private int y;
  private int height ;
  private int width;
  private static Color titleBarColor;

  public void setX(int x) {
     this.x = x;
   }
   ....
   public static void setTitleBarColor(Color c) {
      titleBarColor = c;
   }
}
```

In questo esempio, se si possiede un oggetto di tipo Window e si desidera impostarne le dimensioni, uniche per ogni istanza, si provvederà a effettuare una chiamata a metodo come di consueto:

```
w.setWidth(400);
w.setHeight(300);
```

Al contrario, la chiamata:

```
Window.setTitleBarColor(new Color("Red"));
```

va a modificare l'attributo statico titleBarColor, con la conseguenza di modificare tutti gli oggetti di tipo Window presenti in memoria.

Le variabili e i metodi statici possono essere richiamati anche tramite l'identificatore di una variabile; pertanto, riferendosi all'esempio precedente, le seguenti istruzioni:

```
Window. setTitleBarColor (new Color("Red"));
w.setTitleBarColor (new Color("Red"));
```

sortiranno il medesimo effetto. Dal momento che metodi e variabili di classe operano esclusivamente sul contesto statico, si preferisce seguire la convenzione di richiamarli unicamente mediante l'identificatore della classe, in modo da sottolineare la differenza.

Interfacce

L'ereditarietà è un mezzo di classificazione straordinario, che tuttavia presenta un grandissimo limite: non esiste un unico criterio di classificazione valido in tutte le circostanze. Nella classificazione degli esseri viventi, tanto per fare un esempio, si incontrano grosse difficoltà a trovare il posto giusto per l'ornitorinco, un animale che abbraccia in modo trasversale i principali criteri di divisione (un mammifero che depone le uova, e possiede caratteristiche morfologiche comuni alla marmotta e alla papera). Nel software queste situazioni di parentela trasversali sono molto comuni, e vengono risolte grazie alle interfacce.

Interfacce per definire il comportamento

Un'interfaccia è un costrutto che permette di associare un tipo a una collezione di metodi privi di implementazione. A differenza delle classi, le interfacce supportano l'ereditarietà multipla, una caratteristica estremamente importante che distingue le interfacce dalle classi completamente astratte (classi astratte prive di metodi concreti).

Le interfacce definiscono un protocollo di comportamento che può essere implementato da qualsiasi classe, ovunque si trovi nella gerarchia. I metodi dichiarati all'interno di un'interfaccia non presentano alcun vincolo realizzativo: una classe che implementa un'interfaccia si impegna a fornire un corpo a tutti i metodi definiti dall'interfaccia stessa, al fine di acquisire tale comportamento.

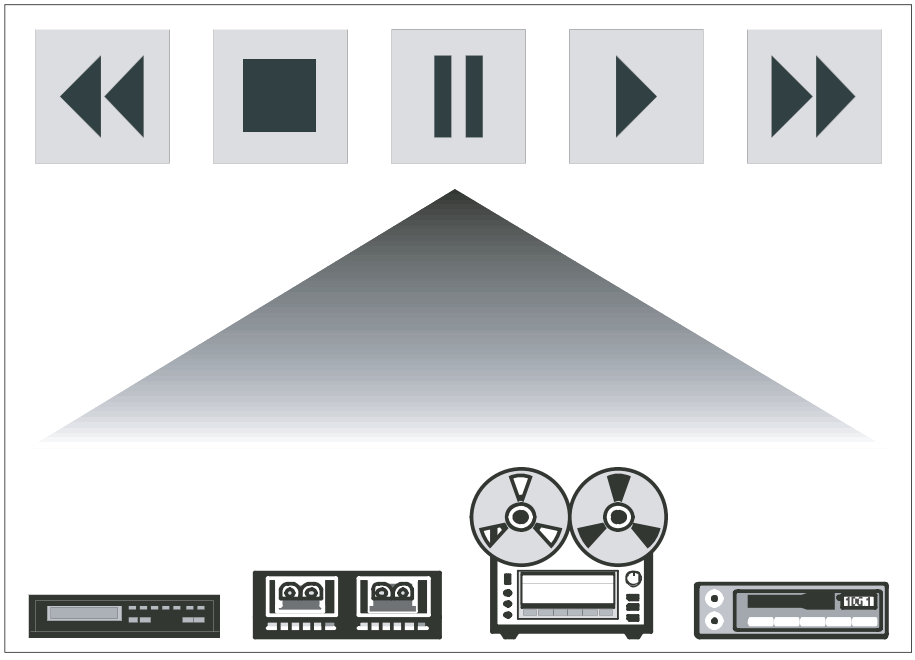
L'uso delle interfacce genera una certa perplessità in chi ha l'abitudine di associare gli oggetti software a una determinata implementazione. Quale può essere l'utilità di un sistema di

classificazione di entità software basato solamente sulle firme dei metodi? La risposta è che l'interfaccia permette di rappresentare una *proprietà* comune a diverse categorie di oggetti, o un *comportamento* presente in oggetti caratterizzati da una differente implementazione.

Un esempio concreto di oggetti diversi dotati di una proprietà in comune può essere trovato in cucina: se durante la preparazione di un dolce la ricetta suggerisce di mischiare gli ingredienti all'interno di un contenitore, si può ricorrere a una marmitta da cucina, realizzata in plastica e con una forma tale da rendere il lavoro di mescolatura particolarmente facile. D'altra parte, in mancanza di una marmitta, è possibile usare qualsiasi altro tipo di contenitore, compresa una pentola per pastasciutta. Nonostante la pentola non sia stata progettata per questo uso, essa ha in comune con la marmitta la proprietà di poter contenere dei fluidi, e per questa ragione potrà essere usata per portare a termine correttamente l'operazione.

Un moderno impianto audio-video presenta invece un esempio lampante di oggetti diversi tra loro dotati della medesima interfaccia utente: CD, DVD, videoregistratore e registratore a cassette sono strumenti completamente diversi dal punto di vista tecnologico. Ciononostante, essi condividono la medesima interfaccia utente (i tasti play, stop, pause, fast forward e rewind). Grazie a questa proprietà, chiunque sappia usare uno qualsiasi di questi strumenti potrà utilizzare senza fatica anche tutti gli altri.

Figura 7.4 – *Oggetti molto diversi tra loro possono condividere un comportamento attraverso un'interfaccia*



Dichiarazione e implementazione di interfacce

Per dichiarare un'interfaccia si ricorre a una sintassi simile a quella usata per le classi:

```
public interface NomeInterfaccia {

  public static final tipo nomeAttributo;

  public  tipo nomeMetodo(tipo par1, tipo par2, …);
}
```

Al posto di class si utilizza la parola chiave interface. Inoltre, i metodi devono per forza essere pubblici e non prevedono un punto e virgola al posto del codice. Nelle interfacce è possibile definire solamente attributi costanti, ossia static e final.

Al pari delle classi, le interfacce possono formare gerarchie, nelle quali è permesso ricorrere all'ereditarietà multipla:

```
public interface MyInterface extends Interface1,Interface2, … {
        ....
}
```

A differenza di quanto avviene per le classi, dove esiste un'unica gerarchia che fa capo a Object, ogni interfaccia può dare vita a una gerarchia di interfacce a sé stante.

Per dichiarare che una classe implementa un'interfaccia, bisogna utilizzare la parola chiave implements nel modo seguente:

```
public class MyClass implements Interface1, Interface2, interface3, ... {
        ....
        corpo della classe
        ....
}
```

dove Interface1, Interface2, Interface3, … sono le interfacce da implementare. È consentita l'implementazione di un numero arbitrario di interfacce.

Se due interfacce contengono metodi con la stessa firma e lo stesso valore di ritorno, la classe concreta dovrà implementare il metodo solo una volta e il compilatore non segnalerà alcun errore. Se i metodi hanno invece lo stesso nome ma firme diverse, la classe concreta dovrà dare un'implementazione per ciascuno dei metodi. Se infine le interfacce dichiarano metodi con lo stesso nome ma con valore di ritorno differente (tipo int getResult() e long getResult()), il compilatore segnalerà un errore, dal momento che il linguaggio Java non permette di dichiarare in una stessa classe metodi la cui firma differisca solo per il tipo del valore di ritorno.

Nel caso degli attributi il problema dei conflitti è molto più semplice: se due interfacce legate

da un qualche grado di parentela dichiarano una costante utilizzando lo stesso nome, sarà sempre possibile accedere all'una o all'altra usando l'identificatore di interfaccia:

```
Interfaccia1.costante;
Interfaccia2.costante;
```

Si noti che in questo caso le costanti omonime possono anche essere di tipo diverso.

Un esempio concreto

Per non restare troppo nell'astratto, ecco un esempio di reale utilità. Le API Java definiscono l'interfaccia Comparable:

```
public interface Comparable {
  public int compareTo(Object o);
}
```

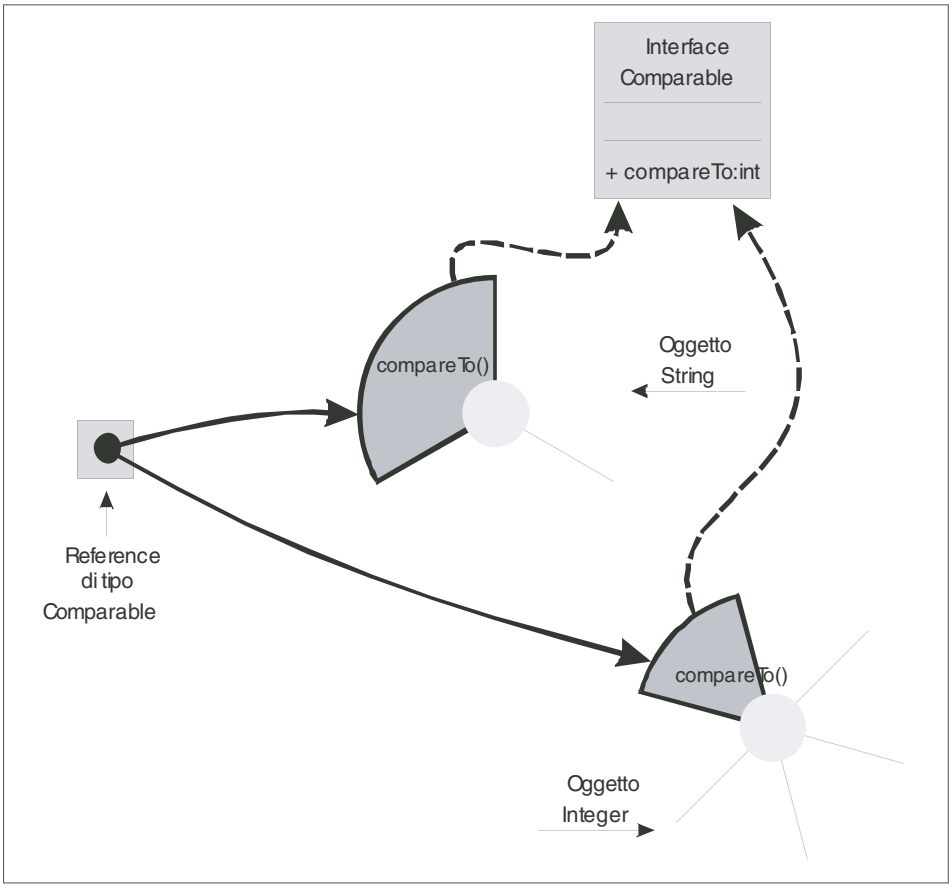
Tale interfaccia viene implementata da un gran numero di classi molto diverse tra loro: BigDecimal, BigInteger, Byte, ByteBuffer, Character, CharBuffer, Charset, CollationKey, Date, Double, DoubleBuffer, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, ObjectStreamField, Short, ShortBuffer, String e URI.

Le classi appena elencate hanno in comune tra di loro soltanto il fatto di essere ordinabili. Dal momento che implementano tutte l'interfaccia Comparable, è possibile scrivere un metodo che permetta di ordinare un array di oggetti di qualsiasi tipo tra quelli elencati:

```
public class ComparableSorter {

  public static Comparable[] sort(Comparable[] list) {
    for(int i = 0 ; i < list.length ; i++ ) {
      int minIndex = i;
      for(int j = i ; j < list.length ; j++) {
        if ( list[j].compareTo(list[minIndex]) < 0  )
          minIndex = j;
      }
      Comparable tmp = list[i];
      list[i] = list[minIndex];
      list[minIndex] = tmp;
    }
    return list;
  }
}
```

Figura 7.2 – *Le interfacce permettono di operare in modo uniforme su oggetti diversi tra loro.*



Il metodo ordina() non è interessato a quale sia il tipo concreto degli oggetti che gli vengono passati: l'unico requisito che gli interessa è che essi implementino l'interfaccia Comparable, in modo da permettere l'esecuzione dell'algoritmo di ordinamento. Dal punto di vista del metodo ordina(), un vettore di Integer è uguale a un vettore di String: il suo comportamento non è influenzato da questa differenza.

Questo metodo funziona su tutti gli oggetti che implementano l'interfaccia Comparable, persino su oggetti che al momento non esistono. Chi realizza le classi concrete ha la responsabilità di stabilire un criterio di confronto e di incorporarlo nel metodo compareTo(): la logica di ordinamento presente nel metodo ordina() trascende il particolare criterio adottato per l'oggetto concreto.

Tipi e polimorfismo

Il polimorfismo è un'importante proprietà dei linguaggi a oggetti: attesta la possibilità di utilizzare un oggetto al posto di un altro, laddove esista una parentela tra i due. Grazie alle interfacce è possibile esprimere a un livello di dettaglio molto profondo l'appartenenza a determinate categorie, e creare procedure in grado di operare in modo trasversale su un gran numero di oggetti accomunati solo da una proprietà o da un comportamento.

Come si è già constatato in precedenza, una classe ha come tipo quello della propria classe e di tutte le sue superclassi. L'interfaccia denota a sua volta un tipo: pertanto, una classe ha tanti tipi quante sono le interfacce implementate (comprese le eventuali super interfacce). Java è un linguaggio *strong typed*: il legame tra un oggetto e i suoi tipi è un aspetto fondamentale e inderogabile, al contrario di linguaggi come C o C++ dove il legame tra tipo e oggetto è lasco, e sono consentite anche operazioni di casting prive di senso. In Java è obbligatorio definire esplicitamente il tipo di una variabile; inoltre, il casting tra oggetti funziona solamente se il tipo dell'oggetto coincide con ciò che viene richiesto dall'operatore di casting. Una buona norma di programmazione è quella di manipolare gli oggetti utilizzando una variabile del tipo dotato dei requisiti più stringenti in relazione al contesto. In questo modo, si garantisce il massimo grado di riutilizzo a ogni singolo elemento del sistema.

Classi e interfacce interne

Java permette di definire classi e interfacce all'interno di altre classi, con un livello di nidificazione arbitrario:

```
public class MyClass {

  public void metodoOmonimo() {
    ....
  }
public interface MyInnerInterface {
  public void m1();
  public void m2();
}
public class MyInnerClass {
  public void metodoOmonimo() {
    ....
  }
}
}
```

Classi e interfacce interne possono essere richiamate nella classe di livello superiore utilizzando il loro nome, senza particolari differenze rispetto al modo in cui si utilizza una qualsiasi altra classe presente nel name space del sorgente. Di contro, se si desidera accedere a esse dal-

l'esterno del sorgente in cui sono state dichiarate, è necessario utilizzare il percorso completo ricorrendo alla dot notation:

```
MyClass.MyInnerClass m = new MyClass.MyInnerClass();
```

Nel codice di esempio, si può notare che la classe interna definisce un metodo utilizzando lo stesso nome di un metodo della classe di livello superiore. In casi come questo, qualora il programmatore desideri chiamare il metodo omonimo della classe di livello superiore, dovrà accedere all'istanza della classe contenitore mediante un uso particolare di this, che viene richiamato come se fosse un attributo accessibile attraverso l'identificatore corrispondente al nome della classe esterna:

```
MyClass.this. metodoOmonimo();
```

Le classi interne sono state introdotte a partire dal JDK 1.1, in particolare per supportare il modello di eventi tipico delle interfacce grafiche. Pertanto esse verranno trattate in modo approfondito solamente in tale contesto.

I package

Il linguaggio Java offre la possibilità di organizzare le classi in package, uno strumento di classificazione gerarchico simile alle directory di un file system. Un package è un contenitore che può raccogliere al suo interno sia classi sia altri package, secondo una struttura gerarchica. Grazie ai package è possibile suddividere un software in moduli, raggruppando insiemi di classi che svolgono un determinato compito. Una simile forma di organizzazione diventa indispensabile nei moderni sistemi software, composti di solito da centinaia o migliaia di classi.

L'organizzazione delle classi in package permette inoltre di risolvere il problema del conflitto di nomi (in inglese name clash). Il conflitto di nomi è un problema molto sentito nella comunità dei programmatori a oggetti: all'interno di progetti di grandi dimensioni, la necessità di dare un nome a ogni cosa conduce a dover riutilizzare più volte nomi comuni tipo "Persona", "Cliente" o "Studente". La divisione in package risolve il problema del conflitto di nomi, dal momento che in questo caso è sufficiente che i nomi siano unici all'interno di un package.

Dichiarazione di package

Per includere una classe in un package è necessario inserire in testa al sorgente la dichiarazione:

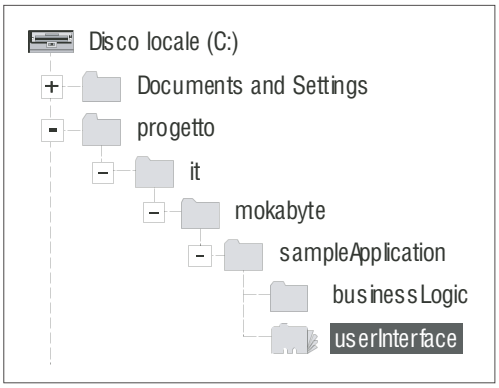
```
package nomepackage;
```

Se si desidera inserire la classe in un sottopackage, bisogna utilizzare la dot notation:

```
package pakage1.package2.myPackage;
```

Oltre alla dichiarazione di appartenenza, è necessario che il file sorgente venga posto fisicamente all'interno di una struttura di directory analoga a quella dei package sottostanti, a partire da una locazione che prenderà il nome di radice. Si osservi la figura 7.3:

Figura 7.3 – *Esempio di organizzazione in package all'interno di un file system.*



Se si volesse creare una classe MainFrame all'interno del package it.mokabyte.sampleApplication.userInterface, sarebbe necessario creare nella directory C:/progetto/it/mokabyte/sampleApplication/userInterface un sorgente "MainFrame.java" dotato di un'intestazione di questo tipo:

```
package it.mokabyte.sampleProject.userInterface;

public class MainFrame {
....
}
```

In uno scenario come questo, la directory "progetto" svolge il ruolo di radice della gerarchia: la posizione dei package all'interno dell'albero viene considerata in relazione a quest'ultima.

Compilazione ed esecuzione

L'organizzazione di un progetto in package richiede una certa attenzione in fase di compilazione ed esecuzione, al fine di evitare errori comuni che generano una grossa frustrazione nei principianti.

In primo luogo, per identificare univocamente una classe è necessario specificarne per esteso il nome assoluto, comprensivo di identificatori di package:

```
it.mokabyte.sampleApplication.userInterface.MainFrame
```

In secondo luogo, prima di invocare i comandi bisogna posizionarsi sulla radice dell'albero di package; in caso contrario, il compilatore e la JVM non riusciranno a trovare i file. Riprendendo l'esempio precedente, prima di compilare o eseguire la classe MainFrame sarà necessario effettuare una chiamata di questo tipo:

```
cd c:/progetto
```

Per compilare la classe MainFrame.java bisogna digitare:

```
javac it\mokabyte\sampleApplication\userInterface\MainFrame.java
```

su piattaforma Windows e:

```
javac it/mokabyte/sampleApplication/userInterface/MainFrame.java
```

in Unix-Linux.
Per eseguire il main della classe, invece, bisogna invocare il comando Java specificando il nome assoluto:

```
java it.mokabyte.sampleApplication.userInterface.MainFrame
```

Quando si comincia a lavorare su progetti di grandi dimensioni organizzati in package, è bene prendere l'abitudine di separare i file sorgenti dalle classi in uscita dal compilatore. Il flag –d del compilatore javac permette di specificare la directory di destinazione del compilatore; pertanto, se si desidera compilare la classe MainFrame in modo tale da porre i file .class nella directory C:\classes, si dovrai invocare il comando:

```
javac -d C:¥classes it\mokabyte\sampleApplication\userInterface\MainFrame.java
```

All'interno della directory C:\classes verranno automaticamente generate le cartelle corrispondenti ai package, con i file .class nelle posizioni corrette.

Dichiarazione import

Per accedere alle classi contenute in un package è necessario specificare il nome della classe per esteso, ricorrendo alla dot notation sia in fase di dichiarazione sia di assegnamento:

```
it.mokabyte.sampleApplication.userInterface.MainFrame m =
                        new it.mokabyte.sampleApplication.userInterface.MainFrame();
```

La scomodità di tale approccio è evidente: l'uso di nomi composti così lunghi può generare altrettanti problemi quanto il conflitto di nomi. Per non essere costretti a specificare ogni volta tutto il percorso verso la classe MainFrame, è possibile importare la classe all'interno del name space del sorgente su cui si sta lavorando, aggiungendo un'apposita clausola import all'intestazione del file:

```
import it.mokabyte.sampleApplication.userInterface.MainFrame;

public class MyClass {
  public static void main(String argv[]) {
      MainFrame f = new MainFrame();
  }
}
```

Grazie alla import, l'identificatore MainFrame entra a far parte dello spazio dei nomi (name space) del sorgente; sarà compito del compilatore associare il nome:

```
MainFrame
```

con la classe:

```
it.mokabyte.sampleApplication.userInterface.MainFrame
```

La import viene spesso utilizzata per importare interi package, utilizzando il carattere * al posto del nome della classe:

```
import it.mokabyte.sampleApplication.userInterface.*;
```

In questo caso, il name space del sorgente comprenderà i nomi di tutte le classi contenute nel package importato. Naturalmente, l'import di interi package può a sua volta generare conflitti sui nomi, dal momento che package differenti possono contenere nomi uguali. Queste eventualità verranno in ogni caso segnalate in fase di compilazione, e potranno essere risolte ricorrendo, solo nei casi di conflitto, all'uso di nomi completi.

Convenzioni di naming dei package

I nomi dei package seguono la consueta convenzione CamelCase, con iniziale minuscola. Per l'organizzazione in sottopackage, le specifiche Sun consigliano di seguire la regola del nome di dominio invertito: per esempio, IBM (dominio ibm.com) inserisce le proprie classi in package che hanno come prefisso com.ibm; allo stesso modo, Sun Microsystems (sun.com) utilizza il prefisso com.sun. Grazie a questa convenzione, è possibile garantire l'unicità dei nomi di classe senza il bisogno di ricorrere a una qualche autorità centralizzata.

Principali package del JDK

A differenza di altri linguaggi di programmazione, le classi di sistema di Java non sono riunite in librerie, ma in package. Ogni package contiene classi che permettono di svolgere un determinato compito: grafica, comunicazione in rete, gestione del file system e così via. Di seguito, vengono presentati i package più importanti del linguaggio:

- java.lang: classi base del linguaggio, tra le quali si trovano Object, String, le wrapper class (Integer, Boolean ecc.) e la classe Math (che contiene le più importanti funzioni matematiche). A differenza degli altri package, java.lang non deve essere importato, dato che fa già parte della dotazione standard del linguaggio.

- java.io: contiene tutte le classi necessarie a gestire il file system e l'input output.

- java.util: classi di utilità, come Vector, Hashtable o Date.

- java.net: qui si trovano tutte le classi di supporto alla comunicazione via rete.

- java.awt: toolkit grafico.

- javax.swing: classi per la gestione della grafica a finestre.

L'uso delle classi presenti in questi e altri package verrà chiarito nei prossimi capitoli.

Modificatori

I modificatori sono parole riservate del linguaggio che permettono di impostare determinate caratteristiche di classi, metodi e attributi. Alcuni di questi modificatori, come abstract e static, sono già stati trattati ampiamente nei paragrafi precedenti; altri, come public, private e protected, verranno ora illustrati in modo completo e formale.

Moficatori di accesso

I modificatori di accesso permettono di impostare il livello di visibilità dei principali elementi di un sorgente Java, ossia classi, metodi e attributi. A ognuno di questi elementi è possibile assegnare uno dei seguenti livelli di protezione:

- private: l'elemento è visibile solo all'interno di una classe.

- nessun modificatore (package protection): l'elemento è visibile a tutte le classi che fanno parte dello stesso package

- protected: l'elemento è accessibile a tutte le classi del package e a tutte le sottoclassi, anche se dichiarate in package differenti.

- public: l'elemento è visibile ovunque.

Il modificatore private è caratteristico degli attributi: il principio dell'incapsulamento raccomanda di dichiarare private tutti gli attributi, e di consentirne l'accesso in modo programmatico

mediante metodi getter. Ci sono comunque diverse occasioni in cui conviene dichiarare private anche un metodo, qualora esso venga chiamato solamente all'interno della classe.

La package protection viene utilizzata soprattutto per classi che hanno una loro utilità all'interno di un package, ma che si desidera restino nascoste al resto del sistema.

Si ricorre a protected in tutte le occasioni in cui si desidera rendere un metodo visibile solo alle sottoclassi: tipicamente, vengono dichiarati protected i metodi getter relativi ad attributi che si desidera restino inaccessibili al di fuori del dominio della classe e delle sue sottoclassi.

Infine, il modificatore public rende un elemento visibile ovunque: si raccomanda di utilizzare tale modificatore con parsimonia e di limitarne l'uso a classi e metodi. Si ricordi che l'insieme dei metodi pubblici di una classe è l'interfaccia attraverso la quale una classe comunica con il mondo esterno. Pertanto, bisogna scegliere con cura quali metodi rendere pubblici e quali no, avendo la stessa accortezza del progettista di elettrodomestici nel non lasciare fili o ingranaggi scoperti.

Final

Il modificatore final assume un significato diverso a seconda del contesto di utilizzo. Se è abbinato a un attributo, esso lo rende immutabile. Tale circostanza impone che l'assegnamento di una variabile final venga effettuato nello stesso momento della dichiarazione. Spesso final viene utilizzato in abbinamento a static per definire delle costanti:

```
public static final float pigreco = 3.14;
```

Bisogna fare attenzione al fatto che, quando si dichiara final un reference a un oggetto, a risultare immutabile è il reference e non l'oggetto in sé. Se si definisce un attributo del tipo:

```
public static final Vector list = new Vector();
```

L'uso di final vieta l'operazione di assegnamento sulla variabile:

```
list = new Vector() // errore: la variabile list è final
```

Al contrario, è consentito chiamare i metodi che modificano lo stato dell'oggetto:

```
list.add("Nuovo Elemento");
```

L'uso di final in abbinamento a un metodo ha invece l'effetto di vietarne l'overriding nelle sottoclassi; se associato a una classe ha l'effetto di proibire del tutto la creazione di sottoclassi (in una classe final, tutti i metodi sono final a loro volta). L'uso di final su metodi e classi, oltre ad avere dei ben precisi contesti di utilizzo, presenta l'ulteriore vantaggio di permettere al compilatore di effettuare ottimizzazioni, rendendo l'esecuzione più efficiente. Per questa ragione alcune classi di sistema, come String e StringBuffer, sono state definite final.

Native

Il modificatore native serve a dichiarare metodi privi di implementazione. A differenza dei metodi abstract, i metodi native vanno implementati in un qualche linguaggio nativo (tipicamente in C o C++). La filosofia di Java non incoraggia la dichiarazione di metodi nativi, ossia dipendenti dalla macchina. Per questa ragione l'effettiva creazione di metodi nativi risulta piuttosto macchinosa.

Strictfp

Il modificatore strictfp, utilizzabile sia nei metodi sia nelle classi, ha lo scopo di forzare la JVM ad attenersi strettamente allo standard IEEE 754 nel calcolo di espressioni in virgola mobile. Java, infatti, utilizza di norma una variante di tale standard che offre maggiore precisione. Vi sono tuttavia situazioni in cui è necessario fare in modo che i risultati delle operazioni siano assolutamente conformi allo standard industriale, anche se questo implica una minore precisione.

Transient, volatile e synchronized

Il modificatore transient permette di specificare che un determinato attributo non concorre a definire lo stato di un oggetto. Esso verrà studiato in profondità nei capitoli relativi alla serializzazione e a JavaBeans. L'uso di volatile e synchronized, due modificatori dotati di una semantica piuttosto complessa, verrà invece illustrato nel capitolo sui thread e sulla concorrenza.