# A3

# Il linguaggio di programmazione Java

Java è un linguaggio di programmazione orientato agli oggetti sviluppato sotto la guida di James Gosling in Sun Microsystem (in seguito acquisita da Oracle Corporation) a partire dalle ricerche effettuate presso l'università di Stanford in California agli inizi degli anni '90 del secolo scorso. La sintassi di base del linguaggio (strutture di controllo, espressioni, …) è volutamente simile a quella del linguaggio C/C++ allo scopo di favorire il passaggio a Java dei programmatori che lo utilizzavano in precedenza.
Concepito inizialmente come un linguaggio finalizzato allo sviluppo di applicazioni complesse per dispositivi elettronici *embedded*, ha avuto una grande diffusione a partire dal 1995 grazie all'espansione su scala mondiale della rete Internet come strumento di programmazione per l'ambiente web. La figura che segue raffigura il logo ufficiale del linguaggio di programmazione Java:


I punti di forza che hanno decretato il grande successo di questo linguaggio – ormai uno dei più utilizzati linguaggi di programmazione – sono i seguenti:

- **Orientamento agli oggetti**. Java è un linguaggio *object-oriented* e in quanto tale permette di sviluppare software che modella la realtà mediante entità (oggetti) dotate di specifiche d'uso e di funzionamento definite (classi); in ogni caso Java non è un linguaggio *object-oriented* «puro», ad esempio i valori dei tipi primitivi, come i numeri interi, non sono oggetti.
- **Portabilità**. Il motto dei progettisti di Java è stato «*Write once, run everywhere*» («scrivi una volta, esegui ovunque»): l'idea alla base di questo approccio è legata alla tecnologia denominata **JVM** (*Java Virtual Machine*) o **JRE** (*Java Runtime Environment*); il compilatore Java infatti non produce codice eseguibile per una specifica piattaforma hardware/software, ma il cosiddetto *bytecode* che viene interpretato dalla JVM. Ogni piattaforma hardware/software ha una propria JVM specifica per cui il *bytecode* può essere indifferentemente eseguito su ciascuna piattaforma

### Curiosità sul nome del linguaggio Java

Alcune voci mai confermate vogliono che il nome del linguaggio sia stato inteso dagli stessi creatori come acronimo per «*Just Another Vacuum Acronym*» («soltanto un altro acronimo vuoto») con ironico riferimento al grande numero di abbreviazioni utilizzate dagli sviluppatori software.
Dato che i creatori del linguaggio si riunivano spesso in un caffè dove discutevano del progetto, pare più probabile che il linguaggio abbia preso il nome dalla qualità di caffè Java dell'omonima isola dell'Indonesia. A conferma di questa interpretazione vi è il logo ufficiale del linguaggio e il fatto che il *magic number* che identifica i file di *bytecode* (il formato eseguibile dei programmi Java) è in esadecimale CAFEBABE, cioè letteralmente «ragazza del caffè», forse un omaggio alla cameriera del caffè.

perché per ognuna di esse è la macchina virtuale a farsi carico della sua esecuzione.

**OSSERVAZIONE** Queste caratteristiche hanno reso il Java fin dall'inizio il linguaggio ideale per le applicazioni web: in questo contesto infatti non è dato conoscere allo sviluppatore la piattaforma hardware/software di esecuzione di un'applicazione che può di volta in volta essere diversa.

La figura che segue esemplifica come un file di codice sorgente avente estensione «.java» viene trasformato dal compilatore in un file di *bytecode* con estensione «.class» che può essere eseguito su qualsiasi piattaforma hardware/software se dotata della specifica JVM (FIGURA 1).

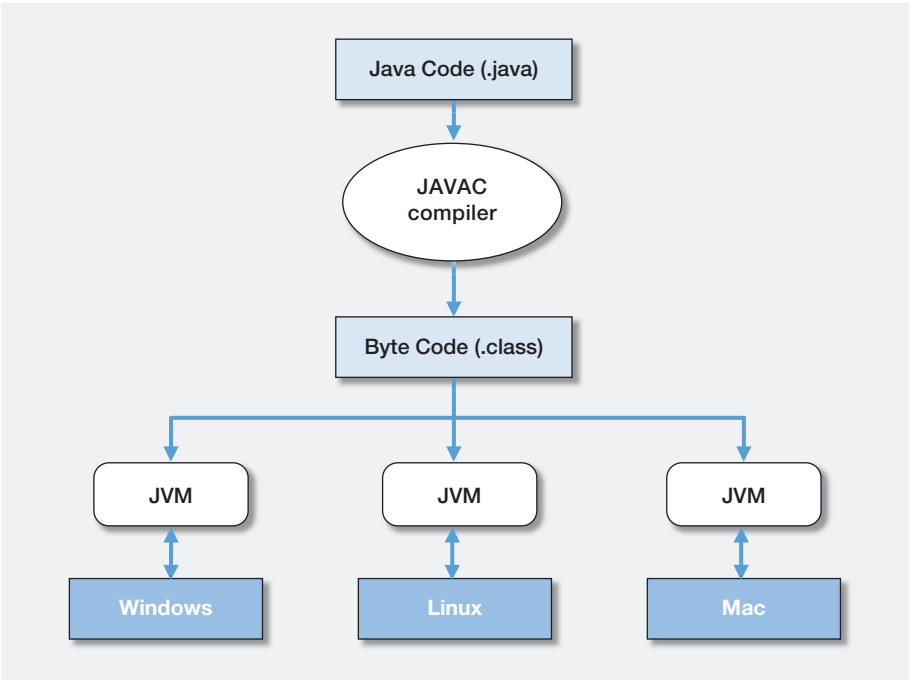



FIGURA 1

- **Disponibilità di strumenti e librerie per la programmazione**. Uno degli aspetti che spesso portano all'adozione di Java come linguaggio per la realizzazione di un prodotto software è dato dalla quantità e qualità delle librerie di cui il linguaggio è dotato che lo rendono facilmente integrabile con altre tecnologie software (accesso a database relazionali, gestione di documenti XML, supporto nativo per la programmazione di rete e web sia lato *client* che *server*, strumenti per la grafica e multimedialità, …).
- **Sicurezza dell'esecuzione del codice**. L'ambiente di esecuzione della piattaforma Java è stato uno dei primi a consentire l'esecuzione automatica di codice proveniente da sorgenti remote: un codice potenzialmente incontrollato deve poter essere eseguito con la certezza che non possa interagire – per errore, o per intento malevolo – con l'ambiente del computer su cui viene eseguito.

### *Bytecode*

Il *bytecode* è un linguaggio intermedio – più astratto del linguaggio macchina di un processore reale – usato per definire le operazioni che costituiscono un programma.
Un linguaggio intermedio come il *bytecode* è utile nella implementazione dei linguaggi di programmazione perché riduce la dipendenza dall'hardware e facilita la creazione degli interpreti del linguaggio stesso.
Un programma in *bytecode* viene infatti «eseguito» da un altro programma che ne interpreta le istruzioni.
Questo interprete è spesso indicato con il termine «macchina virtuale», in quanto può essere visto come un computer astratto che simula buona parte delle funzionalità di un computer reale.
Questa astrazione genera la possibilità di scrivere programmi portabili in modo che risultino eseguibili in diversi ambienti operativi e con diverse architetture hardware.
Questo è un vantaggio che hanno anche i linguaggi di programmazione interpretati, tuttavia un interprete di *bytecode* risulta essere molto più performante di un interprete per un linguaggio di alto livello, perché ha poche e semplici istruzioni che simulano il funzionamento dello hardware del computer.

- ***Multithreading* nativo**. Il linguaggio di programmazione Java è stato uno dei primi a definire un modello di *multithreading* nativo e indipendente dal sistema operativo di compilazione e di esecuzione; questa caratteristica originale, per il fatto che consente al programmatore di sfruttare i moderni processori *multi-core*, si sta dimostrando ogni giorno più importante.

# 1 Compilazione ed esecuzione di programmi Java; memoria *heap* e *garbage-collector*

L'elemento fondamentale di un programma Java è la ***classe***, un «modello» a partire dal quale è possibile istanziare oggetti indipendenti con caratteristiche simili. In una classe sono normalmente presenti due sezioni, una riservata alla componente informativa che si realizza nella definizione degli attributi e l'altra relativa all'aspetto operativo concretizzato nei metodi, ciascuno dei quali definisce una funzione. Dato che ogni oggetto è istanza di una classe, un programma Java è di fatto una collezione di oggetti che interagiscono mediante la loro interfaccia pubblica.
Quello che segue è un esempio minimo di programma Java; esso visualizza la stringa «Hello, world»:

```
public class Hello {
  public static void main (String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Analizziamolo nel dettaglio:

- il nome della classe deve coincidere con quello del file in cui viene definita; la convenzione adottata da tutti i programmatori Java e assunta dalla maggior parte degli ambienti di sviluppo consiste nell'avere una sola classe per file il cui nome ha estensione «.java» (in questo caso «Hello.java»);
- un programma Java è costituito da un insieme di classi: nel nostro esempio abbiamo una sola classe la cui dichiarazione inizia con la parola riservata `class` seguita dal nome della classe – «`Hello`» – e dal simbolo «{» a cui corrisponde il simbolo «}» alla fine della dichiarazione della classe;
- la classe *Hello* non ha attributi e contiene un solo metodo, denominato ***main***; diversamente da quanto avviene nel linguaggio C++, in Java un metodo può essere invocato solo da parte del codice di un altro metodo – della stessa classe, o di una classe distinta – per cui è necessario un metodo iniziale da cui ha inizio l'esecuzione del programma: il metodo iniziale è il metodo *main* che ha sempre la seguente firma:

```
public static void main (String[] args);
```

### JRE e JDK

Per eseguire programmi Java su un computer è sufficiente installare il JRE (*Java Runtime Environment*) che comprende la JVM (*Java Virtual Machine*) per la specifica piattaforma hardware/software.
Per compilare programmi Java è indispensabile anche il JDK (*Java Development Kit*) che comprende il compilatore.
Entrambi sono liberamente e gratuitamente scaricabili dal sito dedicato della Oracle Corporation. I programmatori Java utilizzano IDE (*Integrated Development Environment*) avanzati – i più diffusi sono *Eclipse* e *Netbeans* – ma anche in questo caso in fase di compilazione del codice viene richiamato il compilatore del JDK.

- il metodo *main* si caratterizza per le seguenti proprietà:
  - è pubblico (`public`), ovvero visibile da ogni punto del codice;
  - è statico (`static`), è cioè invocabile indipendentemente dall'esistenza di oggetti istanza della classe, come in questo caso;
  - non restituisce nulla (`void`);
  - gli eventuali parametri in input forniti dall'utente sulla riga di comando costituiscono un vettore di stringhe (cioè di oggetti della classe predefinita `String`).

**OSSERVAZIONE** Ogni classe di un programma Java può avere un suo metodo *main*, in ogni caso l'esecuzione del programma ha inizio con l'invocazione del metodo *main* di una classe specificata come «classe principale» (*main class*).

Dopo aver editato il codice sorgente Java in un file di testo la cui denominazione rispetta la convenzione indicata è possibile compilare il codice eseguibile (*bytecode*) invocando il comando **javac** sulla riga di comando:

```
>javac Hello.java
```

Se il codice è sintatticamente corretto, il compilatore genera il file «Hello.class» che contiene il *bytecode*.
Per eseguire il programma è necessario eseguire la macchina virtuale (il programma *java*) e fornire come argomento il *bytecode* da interpretare:

```
>java Hello
```

Se non vi sono errori in fase di esecuzione viene visualizzato l'output del programma:

```
Hello, world!
```

**OSSERVAZIONE** La visualizzazione della stringa «Hello, world!» è causata dalla seguente riga di codice del metodo *main*

```
System.out.println("Hello, world!");
```

che invoca il metodo *println* della classe predefinita *PrintStream* di cui è istanza l'attributo statico *out* della classe *System*; l'invocazione del metodo *print* al posto di *println* causa la visualizzazione della stringa fornita come argomento senza il ritorno a capo.

Consideriamo la seguente classe *Punto* che permette di istanziare oggetti che rappresentano punti del piano cartesiano aventi come unici attributi il valore dell'ascissa e dell'ordinata. Il diagramma UML di tale classe è quello di FIGURA **2**.

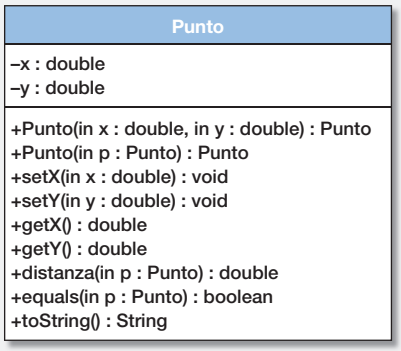



FIGURA **2**

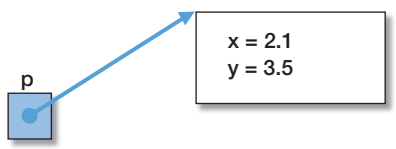
Se implementiamo per essa il seguente metodo *main*:

```
  public static void main (String[] args) {
    Punto p;

    p = new Punto(2.1,3.5);
    System.out.println(p.getX());
    System.out.println(p.getY());
}
```

possiamo istanziare un oggetto della classe e verificare su di esso la funzionalità dei metodi della classe che restituiscono il valore delle sue coordinate.

La dichiarazione `Puntp p;` consente di definire un riferimento a un oggetto istanza della classe *Punto*, ma non istanzia un oggetto.

L'istruzione `p = new Punto(2.1,3.5);` istanzia un oggetto di classe *Punto* nello *heap* che sarà riferito da *p* come illustrato nel seguito:




**OSSERVAZIONE** In Java un oggetto viene sempre creato mediante l'uso esplicito della parola chiave `new`; nell'esempio precedente la dichiarazione della variabile di tipo *Punto* e la creazione di un oggetto di classe *Punto* potevano anche formulate in un'unica riga di codice:

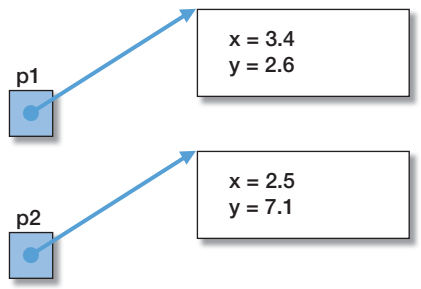
```
Punto p = new Punto(2.1,3.5);
```

**OSSERVAZIONE** In Java non esiste il concetto esplicito di «puntatore», ma di fatto tutte le variabili cui è possibile assegnare oggetti istanza di classi sono puntatori impliciti.

Se nel metodo *main* della classe *Punto* vi fossero le seguenti righe di codice:

```
…
Punto p1, p2;
p1 = new Punto(3.4,2.6);
p2 = new Punto(2.5,7.1);
…
```

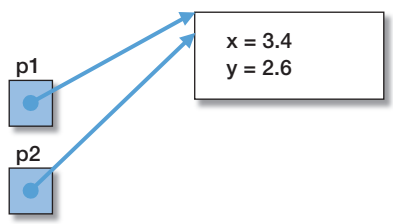
la situazione nello *heap* sarebbe la seguente:




Ma nel caso in cui il codice fosse invece quello che segue:

```
…
Punto p1, p2;
p1 = new Punto(3.4,2.6);
p2 = p1;
…
```

si avrebbe che le variabili *p1* e *p2* riferiscono lo stesso oggetto (l'unico che viene effettivamente creato):




**OSSERVAZIONE** Tutte le variabili aventi come tipo una classe, come ad esempio *p1* e *p2*, al momento della loro dichiarazione sono inizializzate con il valore `null` che indica che non riferiscono alcuno oggetto nello *heap*. Nell'esempio precedente la variabile *p1* assume come valore il riferimento all'oggetto creato nello *heap* solo al momento dell'esecuzione dell'istruzione

```
p1 = new Punto(3.4,2.6);
```

mentre *p2* assume il valore del riferimento allo stesso oggetto con l'esecuzione dell'istruzione

```
p2 = p1;
```

**OSSERVAZIONE** Quando due variabili riferiscono lo stesso oggetto nello *heap* la variazione degli attributi dell'oggetto è ovviamente visibile attraverso entrambi i riferimenti; ad esempio il seguente frammento di codice nel metodo *main* degli esempi precedenti

```
…
Punto p1, p2;
p1 = new Punto(3.4,2.6);
p2 = p1;
p1.setX(1.5);
…
```

ha come conseguenza che anche l'ordinata di *p2* vale 1.5 – e non più 3.4 come inizialmente impostato – in quanto è un attributo dello stesso oggetto che è stato modificato!

# 2 Struttura di un programma Java e fondamenti del linguaggio

Sotto l'aspetto sintattico un programma Java è un insieme di classi ognuna delle quali è definita in un file con estensione «.java» avente lo stesso nome della classe. Almeno la classe principale deve avere un metodo *main* da cui avrà inizio l'esecuzione del programma.

## 2.1 *Package*

Le classi che costituiscono un programma Java sono normalmente organizzate in gruppi denominati ***package***: un *package* raggruppa classi logicamente interdipendenti permettendone la gestione in modo ordinato. Nei file delle classi che appartengono a uno stesso *package* è riportata come prima riga una definizione del tipo:

```
package nome_package;
```

dove *nome_package* è comune a tutte le classi del *package* e deve essere univoco almeno nel contesto del progetto software.
I *package* di classi, una volta compilati, possono essere memorizzati in file compressi con estensione «.JAR» che comprendono tutti i file di *bytecode* delle singole classi del *package*.

## 2.2 Classi

Ogni file che contiene la definizione di una classe dovrebbe avere la seguente struttura di massima:

- commento di intestazione del file;
- dichiarazione del *package*;

- sezione di *import* dei *package* (librerie, estensioni, ecc.);
- definizione della classe:
  - attributi,
  - costruttori,
  - metodi.

L'importazione dei *package* necessari per la corretta compilazione della classe si effettua con la parola chiave `import`.

**ESEMPIO**

Dovendo utilizzare la classi della libreria per l'uso dei file di testo è necessario premettere alla definizione della classe le seguenti righe di codice:

```
…
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
…
```

**OSSERVAZIONE** È possibile dichiarare l'importazione di un intero gruppo di *package* utilizzando il simbolo «*»:

```
import java.io.*;
```

## 2.3 Membri di una classe

Gli attributi, sia variabili che costanti, costituiscono il contenuto informativo di una classe: essi sono gli elementi che descriveranno lo stato dei singoli oggetti istanziati a partire dalla classe.

**OSSERVAZIONE** Oltre agli attributi – o **variabili di istanza** – definiti a livello di classe il linguaggio Java consente la definizione di **variabili locali** all'interno dei singoli metodi.

Le costanti sono identificate dalla parola chiave `final`.

**ESEMPIO**

Il valore di $\pi$ può essere così definito:

```
final double PI_GRECO = 3.1416;
```

I metodi realizzano i comportamenti che gli oggetti implementano: essi sono definiti in modo analogo alle funzioni del linguaggio C/C++, ma non si possono avere parametri passati per riferimento.
Le strutture di controllo del flusso di esecuzione sono le stesse del linguaggio C/C++:

- `if-else`
- `switch-case`
- `while`
- `do-while`
- `for`

I membri di una classe – siano essi attributi o metodi – sono caratterizzati, oltre che dal loro tipo, anche dalla:

- visibilità data dal contesto di dichiarazione dell'attributo o del metodo o da una eventuale omonimia di un parametro o variabile locale che lo nasconde;
- accessibilità stabilita tramite i modificatori `private`, `protected`, `public` e `package`.

La TABELLA **1** riporta quali metodi possono accedere agli attributi o invocare i metodi di una classe in base al tipo di accessibilità specificata nella dichiarazione.

TABELLA **1**

| Accessibilità | Metodi della classe | Metodi di una classe derivata | Metodi di una classe dello stesso *package* | Metodi di tutte le classi |
|---|---|---|---|---|
| `private` | Sì | No | No | No |
| `protected` | Sì | Sì | Sì | No |
| `public` | Sì | Sì | Sì | Sì |
| | Sì | No | Sì | No |

**OSSERVAZIONE** Anche se è possibile fare direttamente riferimento a un attributo pubblico di una classe direttamente nella forma *oggetto.attributo*, è generalmente preferibile definire tutti gli attributi come privati o protetti lasciando che la loro gestione sia consentita esclusivamente in maniera controllata mediante specifici metodi.

## 2.4 Membri statici

Con il modificatore `static` è possibile qualificare attributi o metodi come appartenenti alla classe in cui sono definiti invece che agli oggetti istanziati a partire della classe.

In particolare se la dichiarazione di un attributo è preceduta dalla parola chiave `static` esso è un attributo di classe e non di una determinata istanza di essa ed è condiviso da tutte le istanze della classe: una eventuale modifica al suo valore è quindi automaticamente condivisa da tutti gli oggetti istanziati a partire dalla classe.

ESEMPIO

Il diagramma UML che segue è rappresenta una classe *Libro* per la quale il prezzo di ogni libro è dato da un costo fisso costante di 5,5 € più un costo variabile in funzione delle pagine che lo compongono (FIGURA **3**).

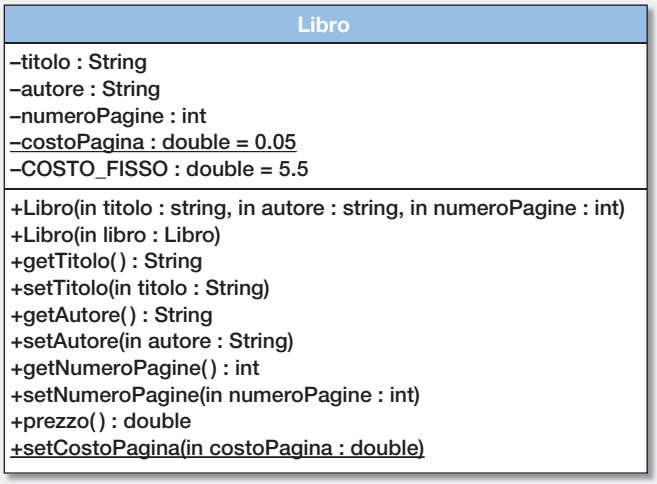



FIGURA **3**

Gli attributi e i metodi sottolineati sono, in base alla convenzione UML, statici. La classe è implementata in linguaggio Java con il seguente codice:

```
public class Libro {
   private String titolo;
   private String autore;
   private int numeroPagine;
   private static double costoPagina = 0.05;
   final double COSTO_FISSO = 5.5;

public Libro(String titolo, String autore, int numeroPagine) {
   this.titolo = titolo;
   this.autore = autore;
   this.numeroPagine = numeroPagine;
}
public Libro(Libro libro) {
   this.titolo = libro.titolo;
   this.autore = libro.autore;
   this.numeroPagine = libro.numeroPagine;
}
public String getTitolo() {
   return titolo;
}
public void setTitolo(String titolo) {
   this.titolo = titolo;
}
public String getAutore() {
   return autore;
}
```

▶

```
  public void setAutore(String autore) {
    this.autore = autore;
  }
  public int getNumeroPagine() {
    return numeroPagine;
  }
  public void setNumeroPagine(int numeroPagine) {
    this.numeroPagine = numeroPagine;
  }
  public double prezzo() {
    return COSTO_FISSO + numeroPagine*costoPagina;
  }
  public static void setCostoPagina(double costo){
    costoPagina = costo;
  }
}
```

Se si istanziano due oggetti distinti con il seguente frammento di codice

```
Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
```

le istruzioni

```
System.out.println(l1.getTitolo()+": "+l1.prezzo());
System.out.println(l2.getTitolo()+": "+l2.prezzo());
```

producono il seguente output

```
Pinocchio: 13.0
Pollicino: 9.5
```

se ripetiamo le stesse istruzioni dopo aver modificato il costo della pagina

```
Libro.setCostoPagina(0.1);
System.out.println(l1.getTitolo()+": "+l1.prezzo());
System.out.println(l2.getTitolo()+": "+l2.prezzo());
```

si ottiene in output la corrispondente variazione per entrambi i libri

```
Pinocchio: 20.0
Pollicino: 13.5
```

perché il metodo statico *setCostoPagina* ha modificato l'attributo statico *costoPagina* condiviso da tutte le istanze della classe *Libro*.
Nel codice precedente il metodo statico è stato invocato a partire dall'identificatore della classe stessa, ma è possibile farvi riferimento da un qualsiasi oggetto istanza della classe ottenendo esattamente lo stesso risultato:

```
l1.setCostopagina(0.1);
```

Nell'esempio illustrato è stata utilizzata la parola chiave `this`, il cui significato è quello di fare esplicito riferimento all'oggetto corrente.

Nel caso specifico ha permesso di riferire gli attributi dei parametri del costruttore per i quali sono stati usati gli stessi nomi: l'uso del riferimento `this` ha evitato l'ambiguità tra il parametro e l'attributo pertanto l'assegnazione

```
this.titolo = titolo;
```

viene interpretata come assegnamento del valore del parametro *titolo* all'attributo *titolo* dell'oggetto corrente identificato da «`this.titolo`».

Analogamente agli attributi, un metodo statico è un metodo della classe che può essere invocato indipendentemente dagli oggetti istanziati. In generale per invocare un metodo si riferisce un'istanza della classe con la notazione *oggetto.metodo*. Questa notazione può rappresentare una limitazione perché alcuni metodi sono semplici funzioni che restituiscono un output a partire dall'input fornito senza interessare lo stato dell'oggetto. È possibile dichiarare metodi statici che non richiedono la creazione di un oggetto per essere invocati: metodi di questo tipo possono infatti essere invocati utilizzando direttamente il nome della classe, senza che occorra applicarli ad alcuna istanza e in pratica corrispondono alle funzioni del linguaggio C/C++.

**ESEMPIO**

L'istruzione

```
int n = Integer.parseInt("123");
```

assegna alla variabile *n* il valore 123 restituito dal metodo statico *parseInt* della classe *Integer* che trasforma in un valore numerico il contenuto di una stringa di caratteri.

La strutturazione del linguaggio Java obbliga a definire i metodi statici all'interno di un classe, ma per invocarli è sufficiente premettere il nome della classe al loro nome.

I metodi statici hanno comunque una importante, ma giustificata limitazione: non possono accedere agli attributi non statici della classe in quanto questi assumono valori specifici per ciascuna istanza.

**OSSERVAZIONE** Il metodo *main* di una classe è statico: infatti l'interprete del linguaggio Java inizia l'esecuzione di un programma a partire dal nome della classe principale di cui inizialmente non viene creata alcuna istanza.

## 2.5 Costruttori

Tra i metodi di una classe ne esiste uno, o – come vedremo in seguito – più di uno, che ha lo stesso nome della classe: il **metodo costruttore** viene invocato automaticamente al momento della creazione di un oggetto istanza della classe utilizzando l'operatore `new`. La sua funzione principale è quella di assegnare un valore iniziale agli attributi del nuovo oggetto creato.

Anche se il linguaggio Java non richiede esplicitamente la sua dichiarazione (viene in questo caso invocato un costruttore implicito) è buona norma prevederlo per ogni classe in modo da assicurare la corretta inizializzazione di un oggetto quando viene creato. Il costruttore non ha tipo di ritorno perché l'oggetto che restituisce assume sempre il tipo della classe stessa e – dovendo necessariamente essere invocato per la costruzione di oggetti – generalmente il suo livello di accessibilità è `public`.
Una classe può avere più di un costruttore se questi si differenziano nella firma (*overloading*): il compilatore selezionerà automaticamente il costruttore corretto in funzione del numero e del tipo dei parametri forniti nell'invocazione.
Una buona pratica di programmazione prevede che ogni oggetto abbia due costruttori particolari:

- il **costruttore di copia** è un costruttore a cui viene fornito come argomento un oggetto istanza della stessa classe per realizzarne un clone copiandone i singoli attributi;

**ESEMPIO**

Il costruttore di copia della classe *Libro* di un esempio precedente può essere utilizzato per copiare un oggetto istanza della classe:

```
Libro l1 = new Libro("L'isola misteriosa", "J. Verne", 500);
Libro l2 = new Libro(l1);
```

Si noti che nello *heap* sono stati creati due oggetti distinti copia uno dell'altro, ma indipendenti rispetto alle successive variazioni.

- il **costruttore di default** privo di argomenti che inizializza gli attributi a un valore predefinito.

**ESEMPIO**

Il costruttore di default della classe *Libro* dell'esempio precedente può essere così implementato:

```
public Libro() {
   this.titolo = "";
   this.autore = "";
   this.numeroPagine = 0;
}
```

### *Javabeans*

Una classe Java che rispetti le seguenti convenzioni:

- ha un costruttore pubblico senza parametri
- gli attributi sono accessibili mediante metodi get/set
- è serializzabile (*)

è un componente software riutilizzabile in contesti diversi comunemente definito *Javabean*.

(*) Questo concetto sarà illustrato nel seguito del capitolo.

# 3 La struttura di base di una classe e il metodo *main*

Sotto l'aspetto sintattico un programma Java è un insieme di classi ciascuna delle quali è definita in un file con estensione «.java» avente lo stesso nome della classe. Come abbiamo già detto, almeno la classe principale del programma deve avere un metodo *main* da cui avrà inizio l'esecuzione. Ma una buona pratica di programmazione prevede di dotare ogni singola classe sviluppata di un metodo *main* che esegua un test dei metodi della classe indipendentemente dal contesto di utilizzazione finale; senza sostituire la necessaria documentazione della classe il metodo *main* costituisce anche un esempio di invocazione dei metodi per gli sviluppatori che devono utilizzare la classe nei propri programmi.
Riprendendo il diagramma UML della classe *Punto* visto in precedenza, il codice Java che implementa la classe è il seguente:

```
1  package geometria;
2
3  public class Punto {
4    private double x;
5    private double y;
6
7    public Punto(double x, double y) {
8      setX(x);
9      setY(y);
10   }
11
12   public Punto(Punto p) {
13     x = p.getX();
14     y = p.getY();
15   }
16
17   public void setX(double x) { this.x = x; }
18   public void setY(double y) { this.y = y; }
19   public double getX() { return x; }
20   public double getY() { return y; }
21
22   public double distanza(Punto p) {
23     double dx = x - p.getX();
24     double dy = y - p.getY();
25     return Math.sqrt((dx*dx)+(dy*dy));
26   }
27
28   public boolean equals(Punto p) {
29     return ((x==p.x) && (y==p.y));
30   }
31
```

▶

### *Java Enterprise e POJO*

La diffusione del linguggio di programmazione Java è in larga parte dovuta al suo impiego in applicazioni *server* di tipo *enterprise* per le quali è indispensabile seguire rigorose convenzioni di codifica e implementazione che riguardano in particolare i cosidetti componenti software EJB (*Enterprise Java Bean*).
In un mondo di acronimi Martin Flower ha un po' ironicamente coniato il termine POJO (*Plain Old Java Object*) per identificare le normali classi Java come quelle che presentiamo in questo corso.

```
32   public String toString() { return "("+x+";"+y+")"; }
33
34   public static void main(String[] args) {
35     Punto p1 = new Punto(1.,1.);
36     Punto p2 = new Punto(2.,2.);
37     Punto p3 = new Punto(p1);
38     System.out.println("P1="+p1.toString());
39     System.out.println("P2="+p2.toString());
40     System.out.println("P3="+p3.toString());
41     System.out.println("Distanza P1-P2: "+p1.distanza(p2));
42     System.out.println("Distanza P1-P3: "+p1.distanza(p3));
43     if (p1.equals(p3))
42       System.out.println("P1 e P3 coincidono");
43     else
44       System.out.println("P1 e P3 non coincidono");
45   }
46 }
```

che produce in output:

```
P1=(1.0;1.0)
P2=(2.0;2.0)
P3=(2.0;2.0)
Distanza P1-P2: 1.4142135623730951
Distanza P1-P3: 0.0
P1 e P3 coincidono
```

Analizziamo i dettagli del codice (tra parentesi sono indicati i numeri di riga a cui si fa riferimento).

- Gli attributi della classe sono due (4, 5): la coordinata *x* e la coordinata *y* del punto nel piano cartesiano, per entrambi esistono i metodi di accesso (*getter*) e modificatori *(setter*).
- Esistono due distinti costruttori (7-10 e 12-15): il primo istanzia un punto a partire dalle coordinate *x* e *y* fornite come parametri, il secondo istanzia un punto a partire dalle coordinate di un altro punto fornito come parametro.

**OSSERVAZIONE** Il secondo costruttore è un costruttore di copia. Il compilatore Java seleziona il costruttore corretto in funzione dei parametri passati al momento dell'invocazione: il primo viene scelto se sono forniti due parametri di tipo `double`, mentre il secondo se è fornito come argomento un oggetto di tipo *Punto*.

> Il costruttore di copia istanzia un nuovo oggetto punto i cui attributi hanno lo stesso valore di quelli dell'oggetto passato come parametro.

- Per determinare la distanza tra due punti si è fatto ricorso al metodo statico *sqrt* della classe di libreria *Math* per calcolare la radice quadrata (25).

- È stato definito il metodo *toString* che trasforma in una stringa di caratteri opportunamente formattata il valore degli attributi della classe (32).

**OSSERVAZIONE** Questo metodo sarebbe stato comunque disponibile per gli oggetti di classe *Punto*, come per qualsiasi altro oggetto.

Tutte le classi dichiarate in Java infatti ereditano dalla classe predefinita *Object* che definisce un formato standard per alcuni metodi di utilità generale tra cui *toString*. Il comportamento del metodo originale deve però essere sempre ridefinito in base agli attributi specifici della classe.

- È stato definito il metodo *equals* che verifica se un punto coincide con un diverso punto fornito come parametro (28-30).

**OSSERVAZIONE**

Anche il metodo di utilità *equals* viene ereditato dalla classe *Object*, ma è necessario ridefinirlo in base alle caratteristiche della classe, come nel caso specifico. Infatti nel caso in cui esso non sia ridefinito il suo esito coincide con quello dell'operatore di confronto «==» che risulta vero solo se due oggetti sono in realtà due riferimenti allo stesso oggetto nello *heap*: non è questo che il programmatore di solito intende per uguaglianza di due oggetti!

- È stato definito previsto il metodo *main* che consente di verificare la funzionalità dei metodi della classe indipendentemente da altre classi (34-45).

**OSSERVAZIONE** La pratica di prevedere un metodo *main* per effettuare un test dei metodi della classe è valida solo per classi piuttosto semplici. Avendo a che fare con classi più complesse è preferibile definire una classe distinta di test con un proprio metodo *main* che istanzi uno o più oggetti della classe da testare.

Proseguiamo l'esempio definendo la classe *Triangolo* che rappresenta un triangolo nel piano cartesiano utilizzando oggetti di classe *Punto* per definirne i vertici. Il diagramma UML delle due classi esplicita un'associazione di tipo compositivo in quanto un triangolo **ha** tre punti come vertici (FIGURA **4**).
Il codice che segue è una possibile implementazione della classe *Triangolo* in Java:

```
1  package geometria;
2
3  public class Triangolo {
4    private Punto a;                                    ▶
```

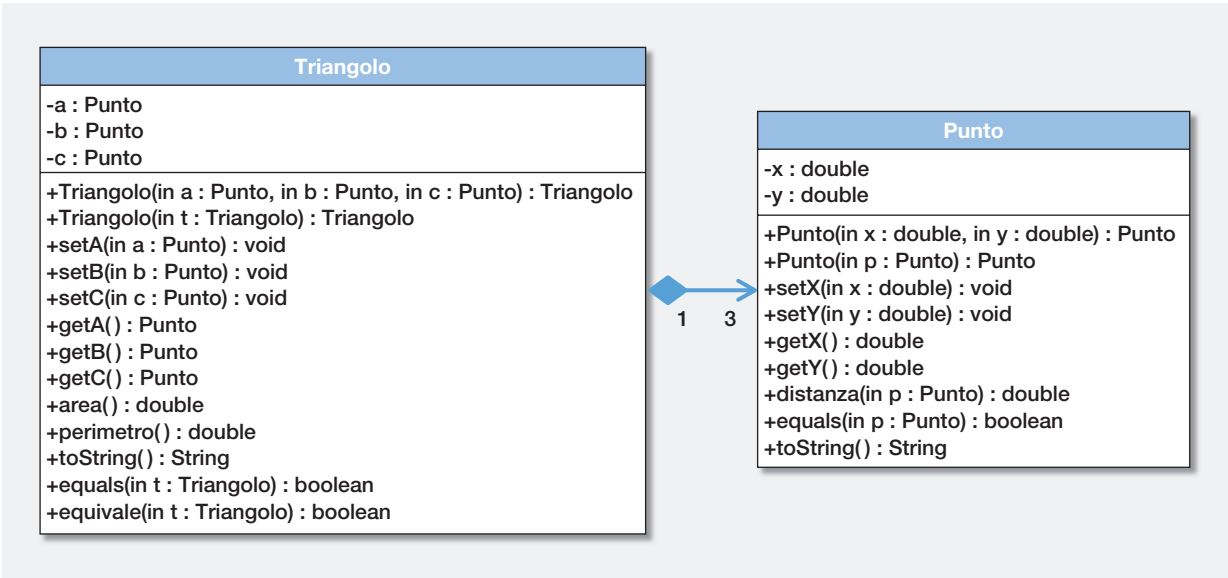



FIGURA **4**

```
5     private Punto b;
6     private Punto c;
7
8     private final double ERR_MAX = 0.00000001;
9
10    public Triangolo (Punto a, Punto b, Punto c) {
11      this.a = a;
12      this.b = b;
13      this.c = c;
14    }
15
16    public Triangolo (Triangolo t) {
17      a = t.a;
18      b = t.b;
19      c = t.c;
20    }
21
22    public void setA(Punto a){this.a = a;}
23    public void setB(Punto b){this.b = b;}
24    public void setC(Punto c){this.c = c;}
25
26    public Punto getA(){return a;}
27    public Punto getB(){return b;}
28    public Punto getC(){return c;}
29
30    public double area() {
31      double p = perimetro()/2;
32      return Math.sqrt(p*(p-a.distanza(b))*(p-b.distanza(c))*
33        (p-c.distanza(a)));
34    }
```

```
35  public double perimetro() {
36    return a.distanza(b)+b.distanza(c)+c.distanza(a);
37  }
38
39  public String toString() {
40    String msg = "";
41    if (area() <= ERR_MAX) msg = " NON È UN TRIANGOLO!";
42    return " A"+a.toString()+" B"+b.toString()+
43      " C"+c.toString()+msg;
44  }
45
46  public boolean equals(Triangolo t) {
47    return ((a.equals(t.a))&&(b.equals(t.b))&&
            (c.equals(t.c)));
48  }
49
50  public boolean equivale(Triangolo t){
51    return (Math.abs(this.area()-t.area()) <= ERR_MAX);
52  }
53
54  public static void main (String[] args) {
55    Punto p1 = new Punto(1.,1.);
56    Punto p2 = new Punto(2.,2.);
57    Punto p3 = new Punto(2.,1.);
58    Punto p4 = new Punto(1.,4.);
59    Punto p5 = new Punto(3.,3.);
60  Triangolo t1 = new Triangolo(p1,p2,p3);
61  Triangolo t2 = new Triangolo(p1,p3,p4);
62  Triangolo t3 = new Triangolo(p1,p2,p2);
63  Triangolo t4 = new Triangolo(t1);
64
65  System.out.println("Triangolo T1: "+t1.toString());
66  System.out.println("Perimetro triangolo T1:
    "+t1.perimetro());
67  System.out.println("Area triangolo T1: "+t1.area());
68  System.out.println("Triangolo T2: "+t2.toString());
69  System.out.println("Perimetro triangolo T2:
    "+t2.perimetro());
70  System.out.println("Area triangolo T2: "+t2.area());
71  System.out.println("Triangolo T3: "+t3.toString());
72  System.out.println("Perimetro triangolo T3:
    "+t3.perimetro());
73  System.out.println("Area triangolo T3: "+t3.area());
74  System.out.println("Triangolo T4: "+t4.toString());
75  System.out.println("Perimetro triangolo T4:
    "+t4.perimetro());
76  System.out.println("Area triangolo T4: "+t4.area());
77
```

```
78    if (t1.equals(t4))
79      System.out.println("I triangoli T1 e T4 sono uguali");
80    else
81      System.out.println("I triangoli T1 e T4 non sono uguali");
82    if (t1.equivale(t4))
83      System.out.println("I triangoli T1 e T4 sono equivalenti");
84    else
85      System.out.println("I triangoli T1 e T4 non sono
        equivalenti");
86
87    if (t1.equals(t2))
88      System.out.println("I triangoli T1 e T2 sono uguali");
89    else
90      System.out.println("I triangoli T1 e T2 non sono uguali");
91    if (t1.equivale(t2))
92      System.out.println("I triangoli T1 e T2 sono
        equivalenti");
93    else
94      System.out.println("I triangoli T1 e T2 non sono
        equivalenti");
95    }
96  }
```

che produce il seguente output:

```
Triangolo T1: A(1.0;1.0) B(2.0;2.0) C(2.0;1.0)
Perimetro triangolo T1: 3.414213562373095
Area triangolo T1: 0.4999999999999998
Triangolo T2: A(1.0;1.0) B(2.0;1.0) C(1.0;4.0)
Perimetro triangolo T2: 7.16227766016838
Area triangolo T2: 1.5000000000000007
Triangolo T3: A(1.0;1.0) B(2.0;2.0) C(2.0;2.0) NON E' UN TRIANGOLO!
Perimetro triangolo T3: 2.8284271247461903
Area triangolo T3: 0.0
Triangolo T4: A(1.0;1.0) B(2.0;2.0) C(2.0;1.0)
Perimetro triangolo T4: 3.414213562373095
Area triangolo T4: 0.4999999999999998
I triangoli T1 e T4 sono uguali
I triangoli T1 e T4 sono equivalenti
I triangoli T1 e T2 non sono uguali
I triangoli T1 e T2 non sono equivalenti
```

Analizziamo i dettagli del codice (tra parentesi sono indicati i numeri di riga a cui si fa riferimento).

- Viene utilizzata la classe *Punto* prima definita per rappresentare per ogni oggetto di classe *Triangolo* i tre punti che rappresentano i suoi vertici (4-5).

- Esistono due diversi costruttori in *overloading* (10-14, 16-20): il primo istanzia un triangolo a partire dai tre vertici forniti come argomenti, il secondo è un costruttore di copia che istanzia un triangolo a partire da un diverso triangolo fornito come argomento di cui copia i vertici.
- Sono definiti metodi *getter* e *setter* per i tre attributi dichiarati di tipo privato (22-24, 26-28).
- Si è fatto ricorso al metodo statico `sqrt()` della classe `Math` (32) per calcolare la radice quadrata necessaria nella formula di Erone per la determinazione dell'area del triangolo (se l'area è nulla i punti dei vertici di fatto coincidono e il triangolo è degenere).
- Il metodo *equivale* (50-52) deve semplicemente verificare se le aree di due triangoli sono o meno uguali.

**OSSERVAZIONE** I valori delle aree a causa degli inevitabili errori di approssimazione numerica presenti nei risultati dei calcoli con numeri non interi non sono confrontati direttamente, cosa che porterebbe a considerare diversi valori di fatto uguali ma affetti da piccoli errori di approssimazione; si verifica che la loro differenza in valore assoluto – determinato invocando il metodo statico `abs()` della classe `Math` – sia minore di un errore massimo prefissato, in questo caso dato dall'attributo costante ERR_MAX.

- È stato definito un metodo per il calcolo del perimetro del triangolo (35-37) e ridefinito il metodo `equals()` per la verifica dell'uguaglianza di due triangoli (46-48).
- È stato ridefinito il metodo `toString()` per la restituzione in formato testuale del contenuto informativo di un oggetto di tipo `Triangolo` (39-44).

**OSSERVAZIONE** L'operatore «+» è stato usato per concatenare più stringhe di caratteri in una sola stringa.

- È stato definito il metodo `main()` per la verifica delle funzionalità dei metodi della classe (54-95).

# 4 Convenzioni di codifica del linguaggio Java

Nella codifica di programmi in linguaggio Java ci sono convenzioni da seguire per lo stile del codice sorgente. Il linguaggio Java – così come il linguaggio C/C++ – è *case sensitive* (distingue cioè i caratteri maiuscoli da quelli minuscoli) per cui è necessario fare attenzione a come sono scritte sia le parole chiave che gli identificatori.
Per quanto riguarda gli identificatori vengono utilizzate convenzioni che benché irrilevanti ai fini della compilazione, sono invece importanti per la leggibilità del codice:

- i nomi di classi devono iniziare con una lettera maiuscola;
- i nomi di metodi, attributi e variabili iniziano con una lettera minuscola;
- i nomi composti usano la convenzione «*CamelCase*»: le parole vengono riportate in minuscolo, una di seguito all'altra senza caratteri di separazione, usando un carattere maiuscolo come lettera iniziale di ogni parola, esclusa la prima se non si tratta di un nome di classe.

Inoltre tali nomi dovrebbero essere sempre scelti in modo da:

- descrivere adeguatamente ciò che identificano (esempio: *FiguraGeometrica* come nome di una classe, o *getDescrizione* come nome di un metodo che restituisce la descrizione di un oggetto);
- essere pronunciabili e formati da parole complete piuttosto che da abbreviazioni (esempio: *valoreMedioRelativo* piuttosto che *valMedRel*);
- essere di lunghezza contenuta, ma sufficientemente descrittivi (esempio: *ElencoArticoli* e non *ElencoArticoliAcquistabiliSuWeb*);
- essere omogenei in tutto il codice oggetto di sviluppo (esempio: se si è utilizzato il termine *Prodotto* come nome di classe è preferibile definire nella classe *Ordini* il metodo *aggiungiProdotto* piuttosto che *aggiungiArticolo*);
- esprimere affermatività (esempio: è preferibile per il nome di una variabile booleana *trovato* piuttosto che *nonTrovato*).

Infine ogni riga non dovrebbe essere troppo lunga (normalmente non più di 80 caratteri) per evitare lo scorrimento orizzontale del testo che richiede tempo e affatica la lettura: è spesso possibile suddividere un'istruzione su più righe di codice.

**OSSERVAZIONE** Le convenzioni per i componenti Java riusabili (*Javabeans*) prevedono che per ogni attributo vi siano due specifici metodi che consentono di acquisirne o impostarne il valore avendo come nome rispettivamente il prefisso *get* o *set* seguito dal nome dell'attributo:

```
public String getDatiUtente() {…};
public void setDatiUtente(String datiUtente) {…};
```

Nel caso di un valore da restituire di tipo booleano i prefisso da utilizzare non è *get*, ma *is*:

```
public boolean isFunzionante() {…};
```

oppure *has* se più indicato come verbo:

```
public boolean hasDipendenti() {…};
```

Per quanto riguarda il codice dei metodi si dovrebbero infine osservare le seguenti regole.

- Per i blocchi di codice le parentesi si aprono al termine della riga dell'istruzione e si chiudono in corrispondenza della colonna di inizio dell'istruzione stessa.

ESEMPIO

```
if (n == 0) {
   …
   …
   …
}

while ( n > 0 ) {
   …
   …
   …
}

do {
   …
   …
   …
} while( !fine );
```

- L'indentazione del codice nel corpo delle istruzioni `if-else`, `switch-case`, `while`, `do-while` e `for` deve essere fissata in un numero costante di spazi (normalmente gli 8 spazi di una tabulazione) in modo da evidenziare chiaramente la struttura del codice.
- Ogni riga non dovrebbe contenere più istruzioni separate dal simbolo «;».

## 5 Tipi di dato primitivi e classi *wrapper*

Java è un linguaggio staticamente tipizzato: questo significa che ogni volta che dichiariamo una variabile dobbiamo specificare il tipo della variabile che può essere uno dei tipi primitivi del linguaggio, oppure una classe predefinita, della libreria, o definita dall'utente. La TABELLA **2** elenca tutti i tipi primitivi del linguaggio Java.

TABELLA **2**

| | |
|---|---|
| `int` | Il tipo `int` rappresenta un numero intero a 32 bit che può assumere un valore compreso tra<br>–2.147.483.648 e +2.147.483.647<br>Il valore di default che assume una variabile non inizializzata è 0. |
| `long` | Il tipo `long` rappresenta un numero intero a 64 bit e può assumere un valore compreso tra<br>–9.223.372.036.854.775.808 e +9.223.372.036.854.775.807<br>Il valore di default che assume una variabile non inizializzata è 0l, il suffisso «l» indica il tipo `long`. ▶ |

▶ TABELLA 2

| | |
|---|---|
| `short` | Il tipo `short` rappresenta un numero intero a 16 bit e può assumere un valore compreso tra<br>–32,768 e +32,767<br>Il valore di default che assume una variabile non inizializzata è 0. |
| `float` | Il tipo `float` rappresenta un numero in virgola mobile (*floating-point*) a 32 bit secondo lo standard IEEE-754 e può assumere un valore reale compreso tra<br>$+/-2^{-149}$ (circa 1.04239846E–45) e $+/- (2.2^{-23}) \times 2^{127}$ (circa 3.40282347E+38)<br>e ha 7 cifre decimali significative; il valore di default che assume una variabile non inizializzata è 0.0f, il suffisso «f» indica il tipo `float`. |
| `double` | Il tipo `double` rappresenta un numero in virgola mobile (*floating-point*) a 64 bit secondo lo standard IEEE-754 e può assumere un valore compreso tra<br>$+/-2^{-1074}$ (circa 4.94065645841246544E–324) e $+/-(2.2^{-52}) \times 2^{1023}$ (circa 1.79769313486231570E+38)<br>e ha 15 cifre decimali significative; il valore di default che assume una variabile non inizializzata è 0.0. |
| `byte` | Il tipo `byte` rappresenta un numero intero compreso tra –128 e +127. Il valore di default che assume una variabile non inizializzata è 0. |
| `boolean` | Il tipo `boolean` può assumere i soli valori `true` e `false`. Il valore di default che assume una variabile non inizializzata è `false`. |
| `char` | Il tipo `char` rappresenta un singolo carattere Unicode a 16 bit: i valori rappresentati si estendono da<br>'\u0000' (0) a '\uFFFF' (65535)<br>Il valore di default che assume una variabile non inizializzata è '\u0000'. |

In molte situazioni può tornare utile trattare i dati primitivi come oggetti: a questo proposito il linguaggio Java rende disponibili le classi denominate *wrapper* (o *adapter*) nel *package java.lang* che non richiede di essere importato. Una classe *wrapper* incapsula una variabile di tipo primitivo, ovvero trasforma un tipo primitivo di cui mantiene il valore in un oggetto corrispondente che integra alcune utili funzionalità. Nella maggior parte dei casi la classe *wrapper* ha lo stesso nome del tipo primitivo corrispondente come mostra la TABELLA 3.

TABELLA 3

| Tipo primitivo | Classe *wrapper* |
|---|---|
| `byte` | `Byte` |
| `boolean` | `Boolean` |
| `char` | `Character` |
| `int` | `Integer` |
| `long` | `Long` |
| `short` | `Short` |
| `float` | `Float` |
| `double` | `Double` |

Ogni classe *wrapper* ha un unico attributo del tipo primitivo che essa incapsula (la classe *Integer* ha un attributo di tipo `int`, la classe *Long* ha un attributo di tipo `long`, …): esse sono state progettate e realizzate in modo da essere «immutabili», gli oggetti istanza della classi *wrapper* i loro oggetti assumono il valore dell'attributo incapsulato al momento della creazione tramite il costruttore e questo valore non può essere successivamente modificato, ma solo acquisito invocando uno specifico metodo. Inoltre le classi *wrapper* espongono molti metodi statici di utilità, ad esempio per convertire stringhe di caratteri in valori numerici e viceversa.

**ESEMPIO**

L'esecuzione del seguente codice Java

```
public class EsempioInteger {
  public static void main(String args[]) {
    int n = 100;
    Integer numero = new Integer("10");
    n = n*numero.intvalue();
    System.out.println("N="+n);
  }
}
```

visualizza come risultato

```
N=1000
```

Uno dei costruttori della classe `Integer` infatti consente di valutare numericamente il contenuto di una stringa di caratteri (in questo caso «10» che fornisce il valore numerico 10), mentre il metodo `intvalue()` restituisce il valore di tipo `int` dell'attributo interno dell'oggetto `numero`.
Tra i metodi statici di utilità della classe `Integer` vi è `valueOf()` che restituisce un oggetto di classe `Integer` inizializzato con un valore di tipo `int`:

```
Integer numero = Integer.valueOf(123);
```

e `parseInt()` che restituisce un valore di tipo `int` a partire da una stringa di caratteri:

```
int n = Integer.parseInt("123");
```

Quest'ultima tecnica è la più utilizzata per trasformare una stringa di caratteri in un valore numerico ed è sostanzialmente diversa da quella presente nel codice del metodo di esempio in quanto non viene istanziato nessuno oggetto di tipo `Integer`, ma viene semplicemente utilizzato un metodo statico della classe `Integer`.
Infine il metodo statico `toString()` della classe `Integer` consente di trasformare in una stringa di caratteri un valore numerico di tipo `int`:

```
String string = Integer.toString(123);
```

**OSSERVAZIONE** Il codice dell'esempio precedente è relativo alla classe *Integer*, ma funzionalità analoghe sono offerte da tutte le classi *wrapper* dei tipi primitivi.

**OSSERVAZIONE** Nel codice dell'esempio precedente il valore numerico della variabile *n* viene implicitamente convertito in una stringa concatenata con il prefisso «N»: questo comportamento poteva essere esplicitato come nella seguente riga di codice:

```
System.out.println("N="+Integer.toString(n));
```

Ma l'operatore di concatenazione di stringhe «+» effettua questa trasformazione in modo automatico.

A solo titolo di esempio sono riepilogati nelle TABELLE **4** e **5** alcuni dei metodi più utilizzati delle classi *wrapper* dei tipi primitivi.

TABELLA **4**

| Classe | Metodo per acquisizione valore incapsulato | Esempio di codice per la dichiarazione e la costruzione di un oggetto *wrapper* da valore di tipo primitivo |
|---|---|---|
| Integer | `int intValue()` | `Integer i = new Integer(123);` |
| Double | `double doubleValue()` | `Double d = new Double(3.1416);` |
| Boolean | `boolean booleanValue()` | `Boolean b = new Boolean(true);` |
| Character | `char charValue()` | `Character d = new Character('F');` |

TABELLA **5**

| Classe | Metodo per trasformazione da stringa di caratteri a valore numerico | Metodo per trasformazione da valore numerico a stringa di caratteri |
|---|---|---|
| Integer | `int parseInt(String s)` | `String toString(int value)` |
| Long | `long parseLong(String s)` | `String toString(long value)` |
| Float | `float parseFloat(String s)` | `String toString(float value)` |
| Double | `double parseDouble(String s)` | `String toString(double value)` |

Tradizionalmente l'operazione di incapsulare un valore di tipo primitivo in un oggetto della corrispondente classe *wrapper* e, viceversa, l'operazione di estrarre il valore di un tipo di dato primitivo da un oggetto della corrispondente classe *wrapper* perndono rispettivamente il nome di ***boxing*** e ***unboxing*** (da *box*, «scatola»). Nelle più recenti versioni del linguaggio Java è stato introdotto l'***autoboxing***, cioè la conversione automatica di un tipo di dato primitivo nel corrispondente oggetto della classe *wrapper* e viceversa la conversione automatica di un oggetto di una classe *wrapper* nel corrispondente tipo di dato primitivo.

**ESEMPIO**

Con la funzionalità di *autoboxing* è possibile scrivere istruzioni come le seguenti:

```
…
Integer x = 123, z;
int y = x;
z = y;
…
```

**OSSERVAZIONE** La tecnica dell'*autoboxing* porta alla creazione di oggetti senza invocazione esplicita dell'operatore `new`.

Lo scopo principale delle classi è quello di istanziare oggetti, ma nel caso delle classi *wrapper* sono importanti anche i servizi di utilità forniti dai metodi statici che non richiedono di essere applicati a oggetti. Esistono classi progettate esclusivamente per fornire servizi di utilità, prive di costruttori e con solo metodi statici: un esempio notevole è la classe Math del *package java.lang* che non necessita di essere importato. Dato che il linguaggio Java non prevede la dichiarazione di funzioni, ma esclusivamente di metodi di classe e che i valori numerici non sono oggetti, ma tipi primitivi si ha la necessità di metodi per i calcoli delle funzioni matematiche (radici, logaritmi, esponenziali, funzioni trigonometriche, …). Una possibile soluzione avrebbe potuto prevedere l'inserimento nella classi *wrapper Double e Float* di un metodo per ogni funzione matematica, ma questo approccio sarebbe stato poco pratico in quanto avrebbe richiesto la creazione di un oggetto per il calcolo di una funzione; la soluzione seguita dai progettisti del linguaggio Java è stata quella di realizzare la classe *Math* come raccolta di metodi pubblici statici ognuno dei quali implementa una specifica funzione matematica di cui nella TABELLA **6** si riportano le principali.

TABELLA **6**

| Tipologia di funzioni matematiche | Funzioni della classe *Math* |
|---|---|
| Potenze e radici | `pow()`, `sqrt()`, … |
| Trigonometriche | `sin()`, `cos()`, `tan()`, … |
| Logaritmiche ed esponenziali | `log()`, `exp()`, … |
| Altre | `abs()`, `floor()`, `round()`, … |

**ESEMPIO**

Il codice del metodo *main* della seguente classe esemplifica l'uso dei metodi della classe *Math*:

```
public class EsempioMath{
  public static void main(String args[]) {
    double x, y;
    x = 2.0;
    y = Math.sqrt(x);
    System.out.println(«La radice quadrata di «+x+» e' «+y);
    x = Math.PI; // PI e' un attributo statico costante di Math
    y = Math.cos(x);
    System.out.println("Il coseno di "+x+" e' "+y);
  }
}
```

# 6 Stringhe e codifica Unicode

Il linguaggio Java non ha un tipo primitivo per le stringhe di caratteri, tuttavia rende disponibili una classe denominata *String* e varie funzionalità che ne permettono l'uso con facilità. Come in tutti i linguaggi che derivano la propria sintassi dal C/C++ una costante di tipo stringa è una sequenza di caratteri delimitata dal simbolo «"» come ad esempio `"Hello, world!"`. Una stringa è quindi una sequenza di caratteri considerati come un unico elemento e un oggetto di tipo *String* assume come valore tale sequenza.

**ESEMPIO**

La seguente dichiarazione Java definisce l'oggetto *messaggio* cui viene successivamente assegnata la stringa «Hello, world!»:

```
String messaggio;
messaggio = "Hello, world!";
```

Lo stesso risultato si sarebbe ottenuto con la seguente riga di codice:

```
String messaggio = "Hello, world!";
```

**OSSERVAZIONE** La sintassi utilizzata per dichiarare gli oggetti di tipo *String* nell'esempio precedente pare quella utilizzata per i tipi primitivi, ma in realtà trattandosi di un oggetto a tutti gli effetti una variabile di tipo *String* come *messaggio* è un riferimento a un oggetto istanza della classe *String* creato nello *heap* e il cui valore è «Hello, world!». In effetti è possibile usare una sintassi più tradizionale per dichiarare e creare l'oggetto *messaggio*:

```
String messaggio = new String("Hello, world!");
```

Le sintassi utilizzate nell'esempio consentono al programmatore di scrivere codice in modo più naturale e veloce rendendo implicita l'invocazione dell'operatore `new`.

Una stringa può contenere un numero qualsiasi di caratteri; la stringa vuota non ne contiene nessuno ed è rappresentata dal simbolo «""».

Diversamente da altri linguaggi di programmazione come C/C++ la codifica dei caratteri in Java non è basata sullo standard ASCII, la cui codifica numerica è limitata a un byte e quindi ai soli 256 simboli alfanumerici che si possono codificare con 8 bit, ma utilizza il set di caratteri Unicode la cui codifica numerica impiega 16 bit con i quali è possibile codificare fino a 65536 simboli diversi. I progettisti del linguaggio Java hanno adottato il set di caratteri Unicode per supportare in modo adeguato i vari alfabeti internazionali. Dato che i primo 256 simboli del set Unicode sono i simboli dello standard ASCII l'adozione di questa codifica non ha alcuna conseguenza per le applicazioni che utilizzano l'alfabeto occidentale.

L'operatore «+» usato con variabili di tipo stringa consente di concatenare tra loro due stringhe per crearne una nuova il cui contenuto è dato dalla sequenza di caratteri della seconda stringa giustapposti alla sequenza di caratteri della prima stringa (l'operatore «+» può essere utilizzato associativamente per concatenare più di due stringhe).

**ESEMPIO**

Il seguente frammento di codice posto in un eventuale metodo *main* di una classe Java:

```
…
String saluto = "Hello, ";
String mondo = "world!";
saluto = saluto+mondo;
System.out.println(saluto);
…
```

visualizzerebbe il seguente messaggio

```
Hello, world!
```

**OSSERVAZIONE** In Java gli oggetti di tipo *String* sono «immutabili», cioè costanti: assumono un valore al momento della creazione che non può successivamente più essere modificato. Di conseguenza «modificare» una stringa equivale a distruggere l'oggetto esistente e crearne uno nuovo. L'esecuzione del codice dell'esempio precedente comporta i seguenti passi:

- creazione della variabile *saluto* che riferisce nello *heap* un oggetto di tipo stringa con valore «Hello»;
- creazione della variabile *mondo* che riferisce nello *heap* un oggetto di tipo stringa con valore «world»;
- creazione di un oggetto nello *heap* di tipo stringa il cui valore è «Hello, world!»;
- cambiamento del riferimento della variabile *saluto* al nuovo oggetto di tipo stringa creato;
- distruzione dell'oggetto di tipo stringa con valore «Hello» precedentemente riferito dalla variabile *saluto* e ora non più riferito da parte del *garbage collector* che ne recupera la memoria occupata.

In questo lungo procedimento nessuna stringa ha cambiato valore!

**OSSERVAZIONE** L'uso dell'operatore «+» richiede un po' di attenzione perché essendo polimorfico in alcuni casi può portare a risultati non voluti. Le seguenti istruzioni Java:

```
System.out.println("numero di telefono "+1+5+":"+339+ 1116798);

System.out.println (339+1116798+" numero di telefono "+1+5);
```

producono rispettivamente il seguente output:

```
numero di telefono 15: 3391116798
```

```
1117137 numero di telefono 15
```

Nel primo caso l'operatore «+» viene interpretato come operatore di concatenazione tra stringhe essendo il primo operando una stringa (`"numero di telefono"`): gli operandi numerici che seguono sono convertiti automaticamente in una stringa prima di effettuare la concatenazione. Nel secondo caso i primi due operandi sono numerici per cui il simbolo «+» viene interpretato come operatore algebrico e prima di eseguire la concatenazione con il terzo operando che richiede la conversione in stringa viene effettuata la somma; gli operandi successivi pur essendo valori numerici sono ormai interpretati come stringhe perché uno dei due operandi è una stringa. Per ottenere il risultato voluto anche nel secondo caso si sarebbe potuto scrivere così:

```
System.out.println(""+339+1116798+" numero di telefono "+1+5);
```

Essendo il primo operando una stringa vuota l'operatore sarebbe stato interpretato come operatore di concatenazione e, convertendo i valori numerici in stringhe, avrebbe prodotto il seguente output:

```
3391116798 numero di telefono 15
```

La classe *String* rende disponibili molti metodi per la gestione delle stringhe di caratteri di cui la TABELLA **7** riporta i più significativi.

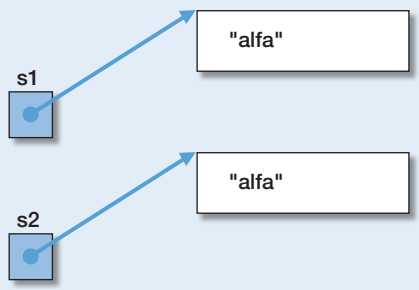
TABELLA **7**

| Metodo | Descrizione |
|---|---|
| `s.length()` | Restituisce la lunghezza in caratteri della stringa *s*. |
| `s.charAt(ind)` | Restituisce il carattere che si torva nella posizione *ind* della stringa *s* tenendo conto che il primo carattere occupa la posizione 0; non è possibile utilizzare la notazione C/C++ `s[indice]` perché le stringhe in Java non sono array di caratteri e l'operatore di indicizzazione non può essere ridefinito. |
| `s.indexOf('x')` | Ritorna la posizione della prima occorrenza del carattere `'x'` nella stringa *s*, o il valore –1 se non è presente. |
| `s1.equals(s2)` | Restituisce **`true`** se le stringhe *s1* e *s2* hanno lo stesso contenuto, **`false`** altrimenti; non è possibile utilizzare l'operatore «==» per effettuare questo confronto perché verificherebbe se quanto riferito da *s1* e *s2* è lo stesso oggetto restituendo **`false`** in caso contrario anche se il contenuto degli oggetti riferiti è esattamente lo stesso. |
| `s2 = s1.substring(2,8)` | La stringa *s2* assume il valore della sottostringa di *s1* compresa tra i caratteri di indice 2 e 8. |
| `s2 = s1.replace('A','a')` | Restituisce in *s2* il valore della stringa *s1* in cui tutte i caratteri «A» sono stati sostituiti con caratteri «a». |

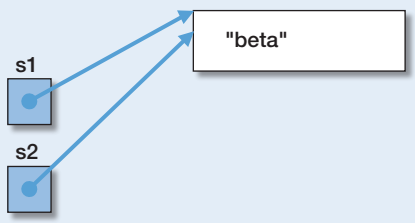
**OSSERVAZIONE** Per confrontare l'uguaglianza di due stringhe di caratteri in Java – o, in generale, di due oggetti – è necessario utilizzare il metodo *equals* invece dell'operatore di confronto «==». Prendiamo in esame i due seguenti frammenti di codice:

```
String s1 = new String("alfa");
String s2 = new String("alfa");
System.out.println((s1==s2));

String s1 = "beta";
String s2 = "beta";
System.out.println((s1==s2));
```

Inaspettatamente nel primo caso sarà prodotto il risultato `false`, mentre nel secondo il risultato `true`. Il primo risultato è corretto perché il confronto avviene tra due riferimenti a stringhe diverse che hanno lo stesso valore:




Come mai il secondo caso fornisce un risultato diverso? Nel primo caso il ricorso esplicito all'operatore `new` crea due distinti oggetti di tipo stringa nello *heap* che l'operatore «==» considera diversi. Nel secondo caso, dato che le due costanti di tipo stringa sono uguali, il compilatore Java effettua un'ottimizzazione creando una sola stringa nello *heap*:




In pratica è come se il secondo frammento di codice fosse

```
String s1 = new String("beta");
String s2 = s1;
```

per cui i due riferimenti *s1* e *s2* risultano uguali in quanto riferiscono lo stesso oggetto.

# 7 Gli array in Java

Gli array sono dichiarati nel linguaggio di programmazione Java con la seguente sintassi:

```
tipo[] identificatore;
```

Questa dichiarazione definisce un vettore di elementi di tipo *tipo* (un tipo primitivo o una classe); la coppia di simboli giustapposti «[» e «]» che identifica il vettore può essere posta anche dopo il nome dell'identificatore possono anche essere disposte dopo l'identificatore:

```
tipo identificatore[];
```

**ESEMPIO**

Le seguenti dichiarazioni definiscono entrambe un vettore di valori interi (tipo primitivo `int`) denominato *unVettore*:

```
int[] unVettore;
int unVettore[];
```

**OSSERVAZIONE** Diversamente da altri linguaggi – come C++ ad esempio – la **dichiarazione** di un array in Java comporta la sola allocazione della variabile che contiene il riferimento a un array, ma non l'allocazione dell'array vero e proprio. Per questo motivo essa non prevede la specifica delle dimensioni, cioè del numero di elementi, dell'array. Nel caso dell'esempio precedente *unVettore* è solo una variabile dichiarata in modo tale da poter contenere il riferimento a un array.

Dal momento che in Java gli array sono oggetti, l'array vero e proprio viene creato – normalmente nel costruttore di una classe – utilizzando l'operatore `new`.

**ESEMPIO**

L'istruzione

```
unVettore = new int[10];
```

a seguito della dichiarazione dell'esempio precedente crea un array di 10 elementi di tipo intero all'interno dell'area di memoria *heap* che nella JVM è riservata all'allocazione degli oggetti.

L'esempio precedente è relativo a un array di elementi di un tipo di dato primitivo (specificatamente `int`) in cui i singoli elementi dell'array contengono direttamente i valori.
Dovendo utilizzare un array a più dimensioni la sintassi è la seguente:

```
tipo[][]…[] identificatore;
```

**ESEMPIO**

Per definire una matrice di valori interi è possibile usare la seguente dichiarazione

```
int[][] tavolaPitagorica;
```

che dovrà essere seguita dalla seguente istruzione per istanziare l'oggetto matrice

```
tavolaPitagorica = new int [10][10];
```

Nel caso di dichiarazione di array di oggetti di una determinata classe i singoli elementi dell'array contengono i **riferimenti** agli oggetti.

**ESEMPIO**

In precedenza è stata introdotta come esempio la classe *Libro*; supponiamo di dovere modellare una mensola sopra alla quale collocare dei libri. Un possibile diagramma UML potrebbe essere quello di FIGURA **5**.

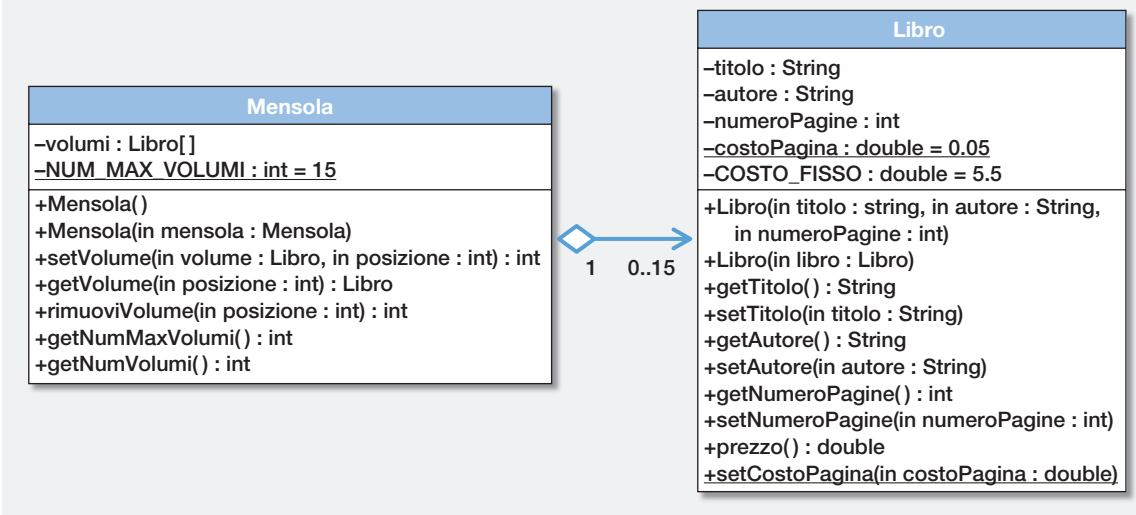



FIGURA **5**

La mensola è in questo caso un «contenitore» di volumi disposti uno di seguito all'altro: ogni mensola prevede un numero fisso di posizioni per i volumi – nell'esempio è 15 – ognuna delle quali può essere vuota o contenere un libro. L'associazione tra le due classi modella un'aggregazione perché un oggetto di classe *Mensola* può esistere indipendentemente dal fatto che contenga o meno dei libri.
Facendo riferimento alla classe *Libro* vista in precedenza, avremo la seguente classe Java *Mensola*:

```
public class Mensola {
  // Attributi
  private static final int MAX_NUM_VOLUMI=15;
  private Libro volumi[];

  // Costruttori
  public Mensola() {
    volumi=new Libro[MAX_NUM_VOLUMI];
  }
```

```
  public Mensola(Mensola mensola) {
    int posizione;

    volumi=new Libro[MAX_NUM_VOLUMI];
    for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
      if (mensola.getVolume(posizione)!=null)
        volumi[posizione] = mensola.getVolume(posizione);
    }
  }

  // Metodi
  public int setVolume(Libro libro, int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]!= null)
      return -2; // posizione occupata
    volumi[posizione]=libro; // inserimento libro in posizione
    return posizione; // restituisce la posizione di inserimento
  }

  public Libro getVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return null; // posizione non valida: nessun libro restituito
    return volumi[posizione]; // restituisce il libro in posizione
  }

  public int rimuoviVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]==null)
      return -2; // posizione vuota
    volumi[posizione]=null; // rimozione libro in posizione
    return posizione; // restituisce la posizione liberata
  }

  public int getNumMaxVolumi() {
    return MAX_NUM_VOLUMI;
  }

  public int getNumVolumi() {
    int posizione, n=0;
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
      if (volumi[posizione]!=null)
        n++;
    return n;
  }
}
```

**OSSERVAZIONI**

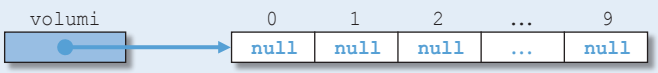
- Una *mensola* incapsula un array di oggetti Java: ogni singola posizione dell'array *volumi* può contenere un oggetto di tipo *Libro*, o essere vuota (`null`).

- Esistono due costruttori: il primo istanzia un oggetto di classe *Mensola* privo di libri, mentre il secondo istanzia un oggetto di tipo *Mensola* contenente gli stessi libri di un oggetto della stessa classe *Mensola* preesistente e fornito come parametro (costruttore «di copia»).
- Il metodo *setVolume* inserisce un volume in un posizione della mensola specificata mediante un indice se questa è valida (indice compreso tra 0 e il numero massimo di volumi previsto per la mensola) e vuota (`null`); se la posizione espressa dall'indice non è valida il metodo restituisce il valore negativo –1, se la posizione espressa dall'indice è valida, ma non è vuota restituisce il valore negativo –2, se l'operazione è eseguita con successo restituisce l'indice della posizione (un valore sempre positivo o uguale a 0).
- Il metodo *getVolume* restituisce un oggetto di tipo *Libro* da una determinata posizione della mensola specificata mediante un indice se questa è valida (indice compreso tra 0 e il numero massimo di volumi previsto per la mensola), in caso contrario – o se la posizione è vuota (`null`) – il metodo restituisceil valore `null`.
- Il metodo *rimuoviVolume* libera una determinata posizione della mensola specificata mediante un indice se questa è valida (secondo gli stessi criteri dei metodi *setVolume* e *getVolume*) e non vuota (`null`); se la posizione non è valida il metodo restituisce il valore negativo –1, se la posizione è valida, ma vuota restituisce il valore negativo –2, se l'operazione è eseguita con successo (viene posto a `null` il valore contenuto nella posizione indicata) restituisce l'indice della posizione resa disponibile.
- I metodi *getNumMaxVolumi* e *getNumVolumi* restituiscono rispettivamente il numero massimo di volumi che la mensola può contenere (nel codice si fa sempre riferimento al valore dell'attributo costante `NUM_MAX_VOLUMI` in modo che volendo adattare la classe a rappresentare mensole di dimensione diversa è sufficiente modificare solo l'inizializzazione di questo attributo) e il numero dei volumi effettivamente presenti su una mensola nel momento dell'invocazione.

**OSSERVAZIONE** Nell'area di memoria *heap* della JVM dopo l'esecuzione dell'istruzione

```
Libro volumi=new Libro[10];
```

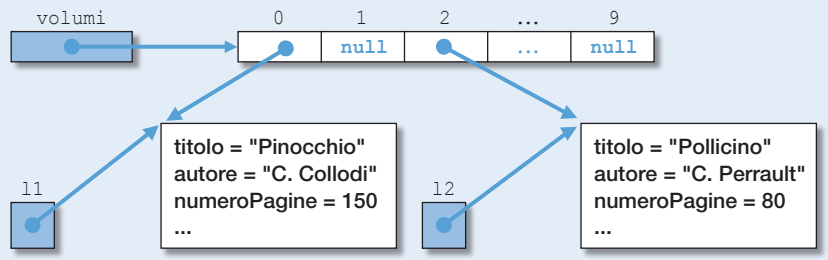
si incontrerà una situazione come quella che segue:




Supponendo di eseguire le seguenti istruzioni

```
Libro l1=new Libro("Pinocchio", "C. Collodi", 150);
Libro l2=new Libro("Pollicino", "C. Perrault", 80);
volumi[0]=l1;
volumi[2]=l2;
```

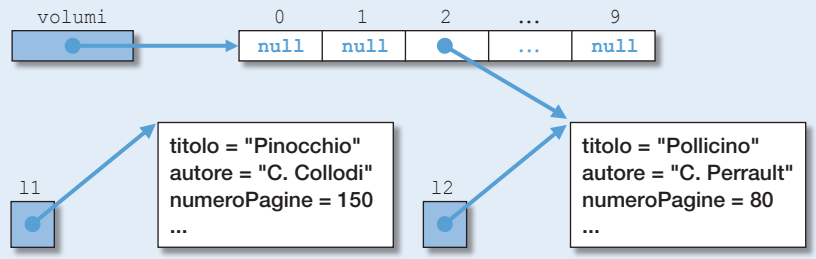
la situazione diviene la seguente:




Se successivamente viene eseguita la seguente istruzione:

```
volumi[0] = null;
```

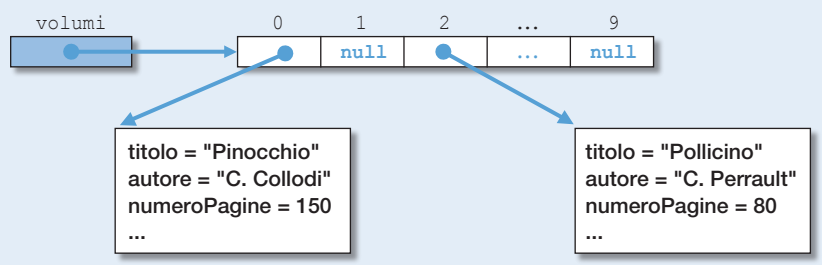
si avrà:




Se invece l'inizializzazione dei vettori fosse stata effettuata mediante le seguenti istruzioni:

```
volumi[0] = new Libro("Pinocchio", "C. Collodi", 150);
volumi[2] = new Libro("Pollicino", "C. Perrault", 80);
```

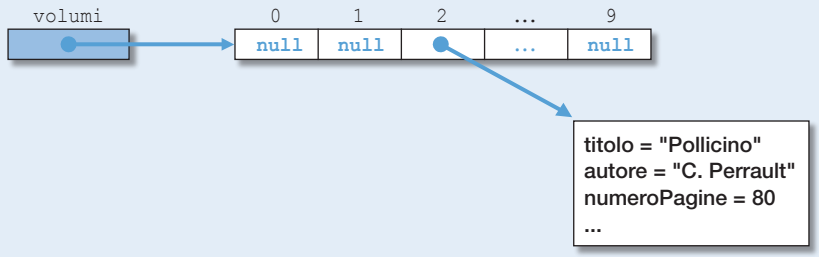
la situazione nello *heap* sarebbe stata la seguente:




e quindi, dopo l'esecuzione dell'istruzione

```
volumi[0] = null;
```

sarebbe divenuta la seguente:




dove l'oggetto di classe *Libro* a cui veniva fatto riferimento dall'elemento di indice 0, non essendo più riferito da alcuna variabile è stato rimosso dal *garbage collector* della JVM.

Per quanto riguarda le modalità di riferimento dei singoli elementi di un array, sia esso monodimensionale che multidimensionale, esse adottano la stessa sintassi di altri linguaggi di programmazione, come ad esempio C/C++, utilizzando i simboli «[» e «]» per specificare l'indice.

**ESEMPIO**

Il seguente metodo *cercaTitolo* aggiunto alla classe *Mensola* consente di determinare se su una mensola è presente o meno un volume avente il titolo specificato:

```
public boolean cercaTitolo (String titolo) {
  int posizione;

  for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
    if (volumi[posizione]!=null)
      if (volumi[posizione].getTitolo().equals(titolo))
        return true;
  return false;
}
```

**OSSERVAZIONE** Dato che un array può essere un array di oggetti e che in Java un array è esso stesso un oggetto, è possibile creare strutture dati estremamente complesse.

**ESEMPIO**

A partire dalle classi *Libro* e *Mensola* possiamo pensare di sviluppare una classe *Scaffale* strutturata come un insieme ordinato di 5 mensole; in sintesi uno scaffale è un oggetto composto da 5 mensole ciascuna delle quali può contenere fino a 15 libri (FIGURA **6**).
Uno scaffale non può esistere senza le mensole che lo costituiscono: per questa ragione l'associazione tra le classi *Scaffale* e *Mensola* è di tipo compositivo.
Una possibile implementazione Java della classe *Scaffale* è la seguente:

```
public class Scaffale {
  // Attributi
  private static final int NUM_RIPIANI=5;
  private Mensola ripiani[];

  // Costruttori
  public Scaffale() {
    int ripiano;
```

▶

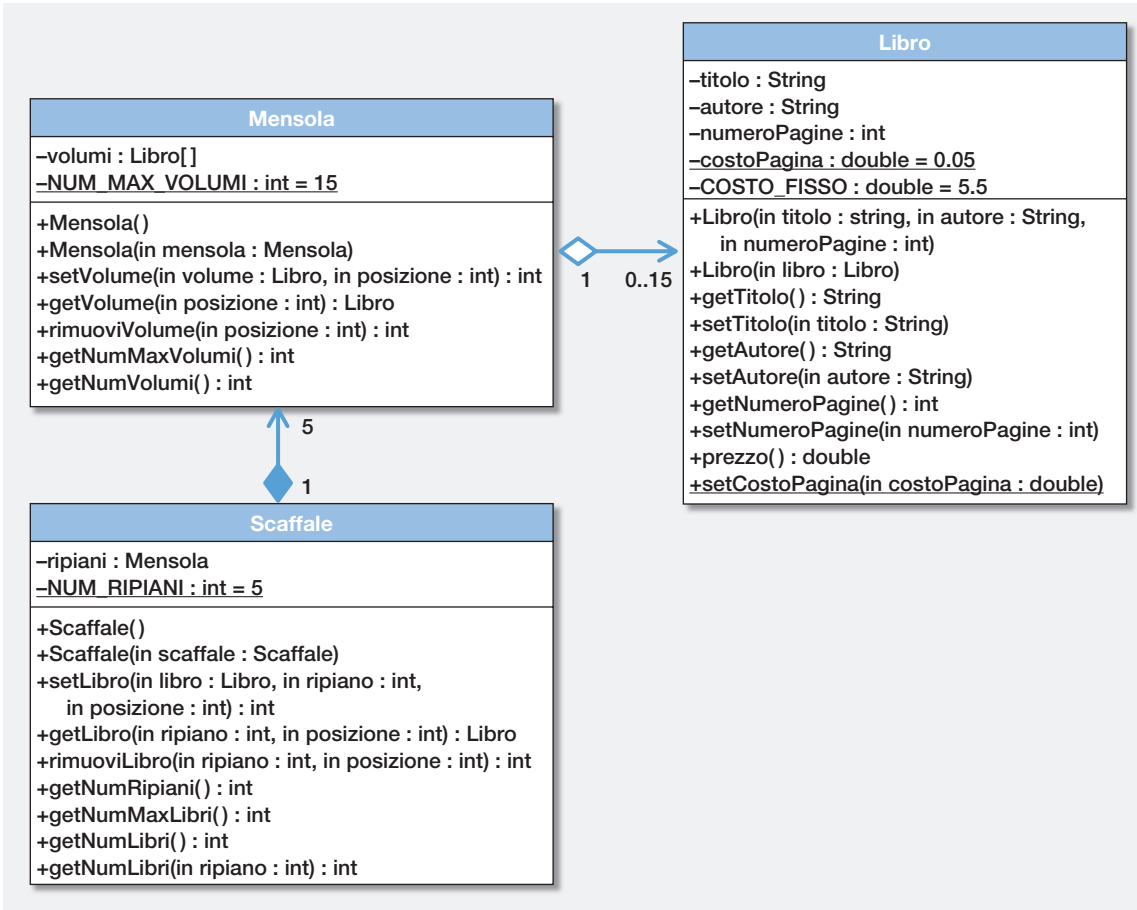



FIGURA **6**

```
    ripiani=new Mensola[NUM_RIPIANI];
    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      ripiani[ripiano]=new Mensola();
  }

  public Scaffale(Scaffale scaffale) {
    int ripiano, posizione;
    Libro libro;

    ripiani=new Mensola[NUM_RIPIANI];
    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      ripiani[ripiano]=new Mensola();
      for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi(); ripiano++) {
        libro=scaffale.getLibro(ripiano, posizione);
        if (libro!=null)
          ripiani[ripiano].setVolume(libro, posizione);
      }
    }
  }

  // Metodi
  public Libro getLibro(int ripiano, int posizione) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return null; // ripiano non valido
```

```
    return ripiani[ripiano].getVolume(posizione);
  }

  public int setLibro(Libro libro, int ripiano, int posizione) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return -1; // ripiano non valido

    if (ripiani[ripiano].setVolume(libro, posizione)<0)
      return -2; // posizione nel ripiano non valida o non vuota

    return 1; // inserimento effettuato
  }

  public int rimuoviLibro(int ripiano, int posizione) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return -1; // ripiano non valido

    if (ripiani[ripiano].rimuoviVolume(posizione)<0)
      return -2; // posizione nel ripiano non valida o vuota

    return 1; // eliminazione effettuata
  }

  public int getNumRipiani() {
    return NUM_RIPIANI;
  }

  public int getNumMaxLibri() {
    int ripiano, n=0;

    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      n += ripiani[ripiano].getNumMaxVolumi();
    }
    return n;
  }

  public int getNumLibri() {
    int ripiano, n=0;

    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      n += ripiani[ripiano].getNumVolumi();
    }
    return n;
  }

  public int getNumLibri(int ripiano) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return -1; // ripiano non valido

    return ripiani[ripiano].getNumVolumi();
  }
}
```

**OSSERVAZIONE** Nell'esempio precedente il codice dei costruttori, oltre a inizializzare il vettore *ripiani*, provvede a invocare esplicitamente il costruttore di ogni singola mensola che costituisce un ripiano per evitare che il vettore *ripiani* abbia elementi nulli. Questo è uno dei motivi per cui le specifiche *Javabeans* prevedono che ogni classe abbia un costruttore di default privo di parametri.

In Java gli array hanno un attributo specifico, denominato *length*, che indica il numero di elementi; di conseguenza uno qualsiasi dei cicli dell'esempio precedente potrebbe essere così riformulato:

```
for (ripiano=0; ripiano<ripiani.length; ripiano++) {
  …
  …
  …
}
```

**OSSERVAZIONE** L'esistenza dell'attributo *length* per gli oggetti di tipo array consente il passaggio di un array come parametro di un metodo senza la necessità di specificarne la dimensione mediante un diverso parametro come è solito fare in C/C++.

**ESEMPIO**

Un test parziale della classe *Scaffale* può essere eseguito mediante il seguente metodo *main*:

```
public static void main (String[] args){
  Scaffale scaffale = new Scaffale();
  Libro libro;

  // creazione di tre oggetti di tipo Libro
  Libro l1=new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2=new Libro("Pollicino", "C. Perrault", 80);
  Libro l3=new Libro("La bella addormentata nel bosco", "C. Perrault", 50);

  // inserimento mensola #0
  scaffale.setLibro(l1, 0, 10);
  scaffale.setLibro(l2, 0, 0);
  // test inserimento mensola #1
  libro=new Libro("Cappuccetto Rosso", "F.lli Grimm", 150);
  scaffale.setLibro(libro, 1, 1);

  //test errori inserimento
  if (scaffale.setLibro(l3, 10, 0) == -1)
    System.out.println("mensola non valida");
  if (scaffale.setLibro(l3, 0, 20) == -2)
    System.out.println("posizione non valida o non libera");
  if (scaffale.setLibro(l3, 0, 10) == -2)
    System.out.println("posizione non valida o non libera");
```

▶

```
    // inserimento mensola #1
    scaffale.setLibro(l3, 1, 0);

    // visualizzazione contenuto mensole
    for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
      for (int posizione=0; posizione<scaffale.getNumLibri(ripiano); posizione++) {
        libro = scaffale.getLibro(ripiano, posizione);
        if (libro!=null)
          System.out.println("ripiano: "+ripiano+" posizione: "+ posizione+" -> "+
                             libro.getTitolo()+" "+ libro.prezzo()+"€");
      }
    }

    // modifica titolo e autore libro estratto da scaffale
    libro=scaffale.getLibro(0,0);
    if (libro!=null) {
      libro.setTitolo("Sussi e Biribissi");
      libro.setAutore("Collodi nipote");
    }

    // visualizzazione contenuto mensole
    for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
      for (int posizione=0; posizione<scaffale.getNumLibri(ripiano); posizione++) {
        libro = scaffale.getLibro(ripiano, posizione);
        if (libro!=null)
          System.out.println("ripiano: "+ripiano+" posizione: "+ posizione+" -> "+
                             libro.getTitolo()+" "+ libro.prezzo()+"€");
      }
    }
}
```

il cui output è:

```
mensola non valida
posizione non valida o non libera
posizione non valida o non libera
ripiano: 0 posizione: 0 -> Pollicino 9.5€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
ripiano: 0 posizione: 0 -> Sussi e Biribissi 9.5€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
```

**OSSERVAZIONE** Nell'esempio precedente gli attributi dei libri contenuti in un oggetto di tipo *Scaffale* possono essere alterati modificando gli attributi di un oggetto restituito dal metodo *getLibro*. Questo comportamento è conseguenza del fatto che il metodo *getLibro* restituisce un riferimento allo stesso oggetto di classe *Libro* riferito dall'oggetto *Scaffale* stesso e potrebbe non rappresentare il risultato atteso dal programmatore.

# 8 Oggetti e riferimenti: implementazione e uso del costruttore di copia

Le variabili di tipi di dato non primitivi del linguaggio Java contengano, anziché un valore, un riferimento a un oggetto nell'area di memoria riservata (*heap*); un **costruttore di copia** è un metodo costruttore che consente di istanziare un nuovo oggetto come copia di un oggetto creato in precedenza e fornito come parametro. In pratica il codice di un costruttore di copia non fa altro che assegnare agli attributi dell'oggetto in fase di creazione il valore degli omonimi attributi dell'oggetto passato come parametro. In questo modo opera ad esempio il costruttore di copia della classe *Libro* del paragrafo precedente:

```
public Libro(Libro libro){
  this.titolo=libro.getTitolo();
  this.autore=libro.getAutore();
  this.numeroPagine=libro.getNumeroPagine();
}
```

Quando gli attributi di una classe non sono di un tipo di dato primitivo o di tipo *String*[1], ma sono riferimenti a oggetti creati istanziando una classe la situazione richiede un po' attenzione.

1. Il fatto che gli oggetti di classe *String* sono immutabili li rende, da questo punto di vista, utilizzabili come variabili di un tipo di dato primitivo.

**ESEMPIO**

Il costruttore di copia della classe *Mensola* introdotta nei paragrafi precedenti

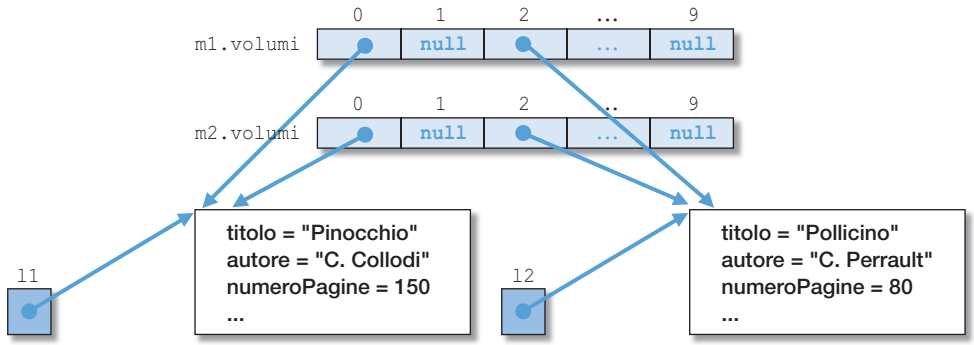
```
public Mensola(Mensola mensola) {
  int posizione;

  volumi=new Libro[MAX_NUM_VOLUMI];
  for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
    if (mensola.getVolume(posizione)!=null)
      volumi[posizione] = mensola.getVolume(posizione);
  }
```

presenta un codice formalmente corretto che si limita ad assegnare ai vari elementi dell'attributo *volumi* dell'oggetto *m* di classe *Mensola* in fase di costruzione gli stessi valori dell'oggetto *mensola* passato come parametro al costruttore. Quello che il costruttore copia sono però dei **riferimenti** e non dei valori: i riferimenti presenti nell'attributo *volumi* dell'oggetto in fase di costruzione saranno gli stessi riferimenti dell'oggetto *mensola*.
Eseguendo il seguente frammento di codice

```
Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
Mensola m1 = new Mensola();
m1.setVolume(l1, 0);
m1.setVolume(l2, 2);
Mensola m2 = new Mensola(m1);
```

▶

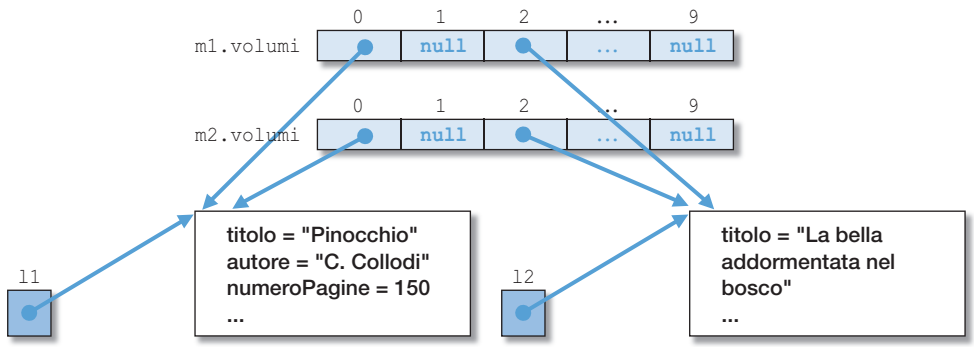
la situazione che si ottiene è quella presentata nella figura seguente:




Come conseguenza si ha che apportando delle modifiche agli attributi di un libro contenuto nell'oggetto di classe *Mensola m1*, si modifica anche il corrispondente libro contenuto nell'oggetto di classe *Mensola m2* (e viceversa) perché di fatto sono un solo oggetto e, a meno che la situazione non sia esplicitamente voluta dal programmatore, potrebbero ingenerarsi ambiguità. L'esecuzione della seguente istruzione:

```
m1.getVolume(2).setTitolo("La bella addormentata nel bosco");
```

modifica l'oggetto *l2* e di conseguenza il contento informativo degli oggetti riferiti da entrambe le mensole *m1* e *m2*:




La versione corretta del costruttore di copia crea invece un nuovo oggetto istanza della classe *Mensola* con l'attributo *volumi* che riferisce nuovi oggetti di tipo *Libro* «clonati» a partire dagli oggetti di tipo *Libro* riferiti dall'attributo *volumi* dell'oggetto di tipo *Mensola* fornito come parametro:

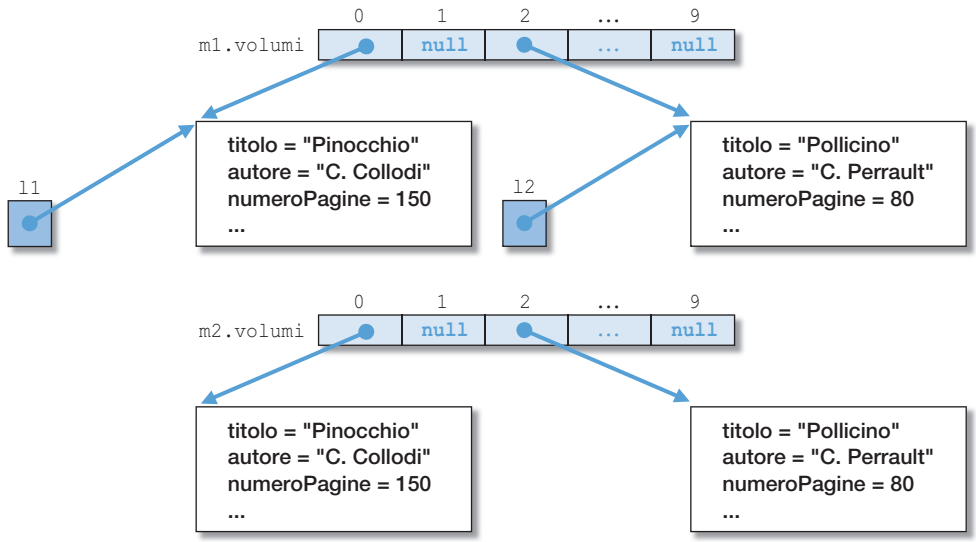
```
public Mensola(Mensola mensola) {
  int posizione;

  volumi=new Libro[MAX_NUM_VOLUMI];
  for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
    if (mensola.getVolume(posizione)!=null)
      volumi[posizione] = new Libro(mensola.getVolume(posizione));
}
```

per cui l'esecuzione del frammento di codice

```
Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
Mensola m1 = new Mensola();
m1.setVolume(l1, 0);
m1.setVolume(l2, 2);
Mensola m2 = new Mensola(m1);
```

▶

da origine ora alla seguente situazione:




Gli oggetti *m1* e *m2* hanno ora copie separate dei propri attributi ed eventuali modifiche apportate a uno dei due oggetti sono ininfluenti per l'altro.

In realtà il problema illustrato nell'esempio precedente è propagato anche da altri metodi della classe *Mensola*. Il metodo *getVolume* ad esempio

```
public Libro getVolume(int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return null; // posizione non valida: nessun libro restituito
  return volumi[posizione]; // restituisce il libro in posizione
}
```

restituisce lo stesso riferimento a un oggetto di tipo *Libro* presente nell'array *volumi* attributo dell'oggetto di classe *Mensola* su cui viene invocato: anche in questo caso una modifica dell'oggetto restituito esternamente alla classe si riflette sul contenuto dell'oggetto interno alla classe, probabilmente contro l'intenzione del programmatore. Un modo più sicuro di implementare il metodo *getVolume* è il seguente che sfrutta il costruttore di copia della classe *Libro* per restituire un oggetto di tipo *Libro* clonato da quello riferito dall'oggetto di classe *Mensola*:

```
public Libro getVolume(int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return null; // posizione non valida: nessun libro restituito
  // restituisce una copia del libro in posizione
  if (volumi[posizione]!=null)
    return new Libro(volumi[posizione]);
  else
    return null;
}
```

Non potendo clonare un oggetto `null` diviene indispensabile trattare separatamente questo caso.
In modo meno evidente anche il metodo *setVolume* presenta lo stesso problema:

```
public int setVolume(Libro libro, int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return -1; // posizione non valida
  if (volumi[posizione]!= null)
    return -2; // posizione occupata
  volumi[posizione]=libro; // inserimento libro in posizione
  return posizione; // restituisce la posizione di inserimento
}
```

Infatti il codice esterno alla classe *Mensola* che lo invoca mantiene un riferimento all'oggetto di tipo *Libro* fornito come parametro e, trattandosi dello stesso riferimento che viene copiato nell'elemento dell'array *volumi* una sua eventuale successiva modifica si rifletterebbe sul contenuto informativo dell'oggetto istanza della classe *Mensola*; anche in questo caso è semplice porre rimedio utilizzando opportunamente il costruttore di copia della classe *Libro* per clonare l'oggetto prima di riferirlo:

```
public int setVolume(Libro libro, int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return -1; // posizione non valida
  if (volumi[posizione]!= null)
    return -2; // posizione occupata
  // inserimento copia di libro in posizione
  volumi[posizione]=new Libro(libro);
  return posizione; // restituisce la posizione di inserimento
}
```

**ESEMPIO**

Tenendo conto delle considerazioni fatte il seguente codice Java implementa una versione corretta della classe *Mensola*:

```
public class Mensola {
  // Attributi
  private static final int MAX_NUM_VOLUMI=15;
  private Libro volumi[];

  // Costruttori
  public Mensola() {
    volumi=new Libro[MAX_NUM_VOLUMI];
  }

  public Mensola(Mensola mensola) {
    int posizione;
```

▶

```
    volumi=new Libro[MAX_NUM_VOLUMI];
    for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
      if (mensola.getVolume(posizione)!=null)
        volumi[posizione]=new Libro(mensola.getVolume(posizione));
    }
  }

  // Metodi
  public int setVolume(Libro libro, int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]!= null)
      return -2; // posizione occupata
    // inserimento copia di libro in posizione
    volumi[posizione]=new Libro(libro);
    return posizione; // restituisce la posizione di inserimento
  }

  public Libro getVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return null; // posizione non valida: nessun libro restituito
    // restituisce una copia del libro in posizione
    if (volumi[posizione]!=null)
      return new Libro(volumi[posizione]);
    else
      return null;
  }

  public int rimuoviVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]==null)
      return -2; // posizione vuota
    volumi[posizione]=null; // rimozione libro in posizione
    return posizione; // restituisce la posizione liberata
  }

  public int getNumMaxVolumi() {
    return MAX_NUM_VOLUMI;
  }

  public int getNumVolumi() {
    int posizione, n=0;
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
      if (volumi[posizione]!=null)
        n++;
    return n;
  }
}
```

# 9 Array come parametri e valori di ritorno dei metodi di una classe

Il linguaggio di programmazione Java consente sia di passare un array come parametro a un metodo che di ottenerlo come valore di ritorno.

ESEMPIO

Per ottenere la lista dei titoli dei libri di un certo autore presenti in uno scaffale in forma di array di stringhe si può predisporre un metodo che svolge le seguenti operazoni:

- conta quanti sono i libri dell'autore analizzando tutte le posizioni di tutte le mensole dello scaffale;
- dimensiona di conseguenza un vettore di stringhe per memorizzare i titoli dei libri dell'autore;
- restituisce il vettore creato, oppure null se non è stato trovato alcun libro dell'autore specificato.

Nel linguaggio di programmazione Java il codice del metodo potrebbe essere il seguente:

```
public String[] elencoTitoliAutore(String autore) {
  int ripiano, posizione;
  int libriAutore=0;
  Libro libro;

  for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++)
    for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi(); posizione++)
      if (ripiani[ripiano].getVolume(posizione) != null) {
        libro = ripiani[ripiano].getVolume(posizione);
        if (libro.getAutore().equalsIgnoreCase(autore))
          libriAutore++;
        }
  if (libriAutore==0)
    return null;
  // dichiarazione e creazione array risultato
  String elencoLibri[] = new String[libriAutore];
  libriAutore = 0;
  for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++)
    for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi(); posizione++)
      if (ripiani[ripiano].getVolume(posizione) != null) {
        libro = ripiani[ripiano].getVolume(posizione);
      if (libro.getAutore().equalsIgnoreCase(autore)) {
        elencoLibri[libriAutore] = libro.getTitolo();
        libriAutore++;
      }
  return elencoLibri;
}
```

Questo metodo può essere verificato nel metodo *main* con le seguenti istruzioni:

```
…
String titoli[]=scaffale.elencoTitoliAutore("C. Perrault");
if (titoli.length>0) {
  System.out.println("Libri di C. Perrault sullo scaffale:");
```

▶

```
  for (int indice=0; indice<titoli.length; indice++)
    System.out.println(titoli[indice]);
}
else
  System.out.println("Nessun libro di C. Perrault sullo scaffale");
…
```

che produce un output del tipo:

```
Libri di C. Perrault sullo scaffale:
Pollicino
La bella addormentata
```

**ESEMPIO**

Per creare una mensola su cui disporre già inizialmente alcuni libri è possibile passare al costruttore come parametro un array di libri:

- se l'array passato come argomento contiene più libri della dimensione della mensola i libri in eccesso sono ignorati;
- se l'array non contiene alcun libro viene creata una mensola vuota.

```
public Mensola(Libro[] elencoLibri) {
  int posizione, libri;

  volumi=new Libro[MAX_NUM_VOLUMI];
  if (MAX_NUM_VOLUMI>elencoLibri.length)
    libri = elencoLibri.length;
  else
    libri = MAX_NUM_VOLUMI;
  for (posizione=0; posizione<libri; posizione++)
    volumi[posizione] = new Libro(elencoLibri[posizione]);
  }
```

Questo metodo può essere verificato nel metodo *main* con le seguenti istruzioni:

```
…
Libro elencoLibri[] = new Libro[3];
elencoLibri[0] = new Libro("Pinocchio","C. Collodi", 150);
elencoLibri[1]=new Libro("Pollicino","C. Perrault",80);
elencoLibri[2]=new Libro("La bella addormentata","C. Perrault", 50);
Mensola mensola = new Mensola(elencoLibri);
```

Se nel *main* inseriamo le seguenti istruzioni:

```
…
Libro elencoLibri[]=new Libro[3];
elencoLibri[0]=new Libro("Pinocchio","C.Collodi",150);
elencoLibri[1]=new Libro("Pollicino","C.Perrault",80);
elencoLibri[2]=new Libro("La bella addormentata","C.Perrault",50);
```

▶

```
// visualizzazione elenco volumi
for (int posizione=0; posizione<mensola.getNumMaxVolumi(); posizione++) {
  Libro libro = mensola.getVolume(posizione);
  if (libro!= null)
    System.out.println("posizione "+posizione+" -> "+ libro.getTitolo()+" "+libro.prezzo()+"€");…
…
```

si ottiene in output:

```
posizione 0 -> Pinocchio 13.0€
posizione 1 -> Pollicino 9.5€
posizione 2 -> La bella addormentata 8.0€
```

# 10 Eccezioni

Le eccezioni sono eventi che si presentano in fase di esecuzione (*run-time*) di un programma e sono generalmente collegate al verificarsi di situazioni anomale.

Tra le eccezioni più comuni ricordiamo: la divisione per zero, l'uso di un indice fuori dal *range* di un array e l'accesso a un riferimento nullo perché non inizializzato. Al verificarsi di un'eccezione si hanno due possibilità: intercettarla e gestirla mediante del codice specifico, oppure lasciare che il programma termini la sua esecuzione in maniera anomala e con conseguenze imprevedibili rispetto all'elaborazione interrotta.
Le cause che possono portare alla generazione di un'eccezione sono molteplici, classifichiamo nel seguito le più comuni:

- **errori software**: errori imputabili allo sviluppatore;
- **input errato da parte dell'utente**: l'immissione di dati in un formato non consentito è una delle più comuni cause di eccezione nell'esecuzione di un programma, la responsabilità è comunque dello sviluppatore che avrebbe dovuto prevedere nel codice controlli per prevenire la situazione di errore;
- **violazioni della sicurezza**: un tipico esempio è il tentativo di modificare un file senza avere i necessari diritti per farlo;
- **indisponibilità delle risorse**: errore dovuto alla non disponibilità temporanea o permanente di una risorsa hardware (ad esempio una connessione di rete) o software (ad esempio un file).

## 10.1 Eccezioni predefinite non controllate

Quando si verifica un'eccezione non gestita dal codice del programma la JVM interrompe l'esecuzione del programma e produce un messaggio noto come *stack-trace* che riporta, oltre al tipo di eccezione generata e il metodo che l'ha causata, la catena di invocazione dei metodi che hanno determinato l'eccezione stessa.

**ESEMPIO**

Il metodo *main* della seguente classe Java ha il solo scopo di generare un'eccezione di divisione per zero:

```
1  public class Eccezione {
2
3     static int divisione(int a, int b) {
4       return a/b;
5     }
6
7     static int quoziente10(int d){
8       return divisione(10, d);
9     }
10
11    public static void main (String[] args) {
12      for (int n=10; n>=0; n--)
13        System.out.println(quoziente10(n));
14    }
15 }
```

Il messaggio di output prodotto al verificarsi dell'eccezione (quando la variabile *n* assume valore 0) specifica la catena di invocazione del metodo che ha causato l'eccezione stessa, in questo caso il metodo *divisione* invocato dal metodo *quoziente10* a sua volta invocato dal metodo *main*:

```
1
1
1
1
1
2
2
3
5
10
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Eccezione.divisione(Eccezione.java:4)
    at Eccezione.quoziente10(Eccezione.java:8)
    at Eccezione.main(Eccezione.java:13)
```

Naturalmente l'interruzione dell'esecuzione di un programma a causa di un'eccezione non è un evento auspicato: per questa ragione Java – come altri linguaggi di programmazione – prevede dei meccanismi per la gestione di tali eventi finalizzati alla costruzione di programmi robusti e affidabili.

**OSSERVAZIONE** Anche se il tipo di errore che ha generato un'eccezione non è recuperabile, nel senso che il programma dovrà essere in ogni caso terminato, potrebbe essere comunque importante intercettare l'eccezione e gestire correttamente la terminazione del programma per rilasciare in modo controllato le eventuali risorse acquisite (ad esempio per chiudere correttamente un file aperto in scrittura evitando una possibile perdita di dati).

### Eccezioni in Java e C++

In Java la gestione delle eccezioni (*exception-handling*) consiste in più costrutti del linguaggio finalizzati a rendere semplice, chiara e sicura la gestione delle situazioni anomale che si possono verificare durante l'esecuzione di un programma.
L'*exception-handling* del Java deriva direttamente da quello del linguaggio C++, è forse più oneroso, ma certamente più sicuro. Realizza infatti la regola «*handle or declare*» («gestisci o dichiara»), che obbliga il programmatore a prevedere esplicite contromisure per ogni situazione anomala potenziale.

Nel linguaggio di programmazione Java le eccezioni sono esse stesse delle classi (contenute nel *package* predefinito *java.lang*) che derivano della classe *Exception* che deriva a sua volta dalla classe *Throwable*.

Mediante il meccanismo dell'ereditarietà il linguaggio Java raggruppa in categorie le eccezioni predefinite di cui si elencano le più comuni:

- *ArithmeticException*: errori aritmetici;
- *NullPointerException*: uso di riferimenti nulli non inizializzati;
- *ArrayIndexOutOfBoundsException*: indicizzazione di un array fuori dai limiti;
- *IOException*: errori di I/O.

**OSSERVAZIONE** Le eccezioni predefinite del linguaggio Java sono di tipo *unchecked* («non controllato»): lo sviluppatore ha la facoltà di intercettarle per gestirle, o di non intercettarle affatto lasciando che la loro eventuale generazione porti a una terminazione anomala del programma.

L'intercettazione delle eccezioni viene realizzata in Java mediante un costrutto di programmazione ereditato dal linguaggio C++ che costituisce una «trappola» per gli errori:

```
try {
  // istruzioni che potenzialmente generano eccezioni
  …;
  …;
}
catch (TipoEccezione1 identificatore){
  // istruzioni da eseguire se si verifica l'eccezione
  // TipoEccezione1
  …;
  …;
}
catch (TipoEccezione2 identificatore){
  // istruzioni da eseguire se si verifica l'eccezione
  // TipoEccezione2
  …;
  …;
}
finally {
  // istruzioni da eseguire per un qualsiasi tipo di eccezione
  …;
  …;
}
```

Se nel corso dell'esecuzione delle istruzioni contenute nel blocco `try` si verificano delle eccezioni il controllo dell'esecuzione passa alle istruzioni del blocco `catch` corrispondente al tipo di eccezione rilevata: il codice del

blocco `catch` per quanto possibile dovrebbe ripristinare uno stato corretto. Il programmatore specificherà i blocchi `catch` necessari a catturare i tipi di eccezione che ritiene necessario intercettare. La clausola `finally` è opzionale: essa è obbligatoria solo nel caso in cui non esista alcuna clausola `catch`; se presente il blocco `finally` viene sempre eseguito dopo il verificarsi di un'eccezione anche nel caso che il blocco `catch` eseguito contenga un'istruzione `return`, o nel caso che nessun blocco `catch` corrisponda al tipo di eccezione generata.

**ESEMPIO**

La seguente classe modifica quella dell'esempio precedente prevedendo l'intercettazione e la gestione dell'eccezione dovuta alla divisione per zero, a questo scopo i metodi *divisione* e *quoziente10* restituiscono una stringa di caratteri invece che un valore numerico.

```
1  public class Eccezione {
2
3    static String divisione(int a, int b) {
4      try{
5        return Integer.toString(a/b);
6      }
7      catch(ArithmeticException exception) {
8        return "impossibile calcolare "+a+"/"+b;
9        }
10     }
11
12   static String quoziente10(int d){
13     return divisione(10, d);
14   }
15
16   public static void main (String[] args) {
17     for (int n=10; n>=0; n--)
18       System.out.println(quoziente10(n));
19   }
20 }
```

L'output che si ottiene è il seguente e il programma non termina in questo caso l'esecuzione in modo anomalo:

```
1
1
1
1
1
2
2
3
5
10
impossibile calcolare 10/0
```

Infatti al verificarsi dell'eccezione di classe *ArithmeticException* (*exception* è un oggetto istanza di questa classe specifico per l'errore rilevato) questa viene intercettata e il codice del blocco `catch` costruisce in questo caso il messaggio di risposta appropriato.

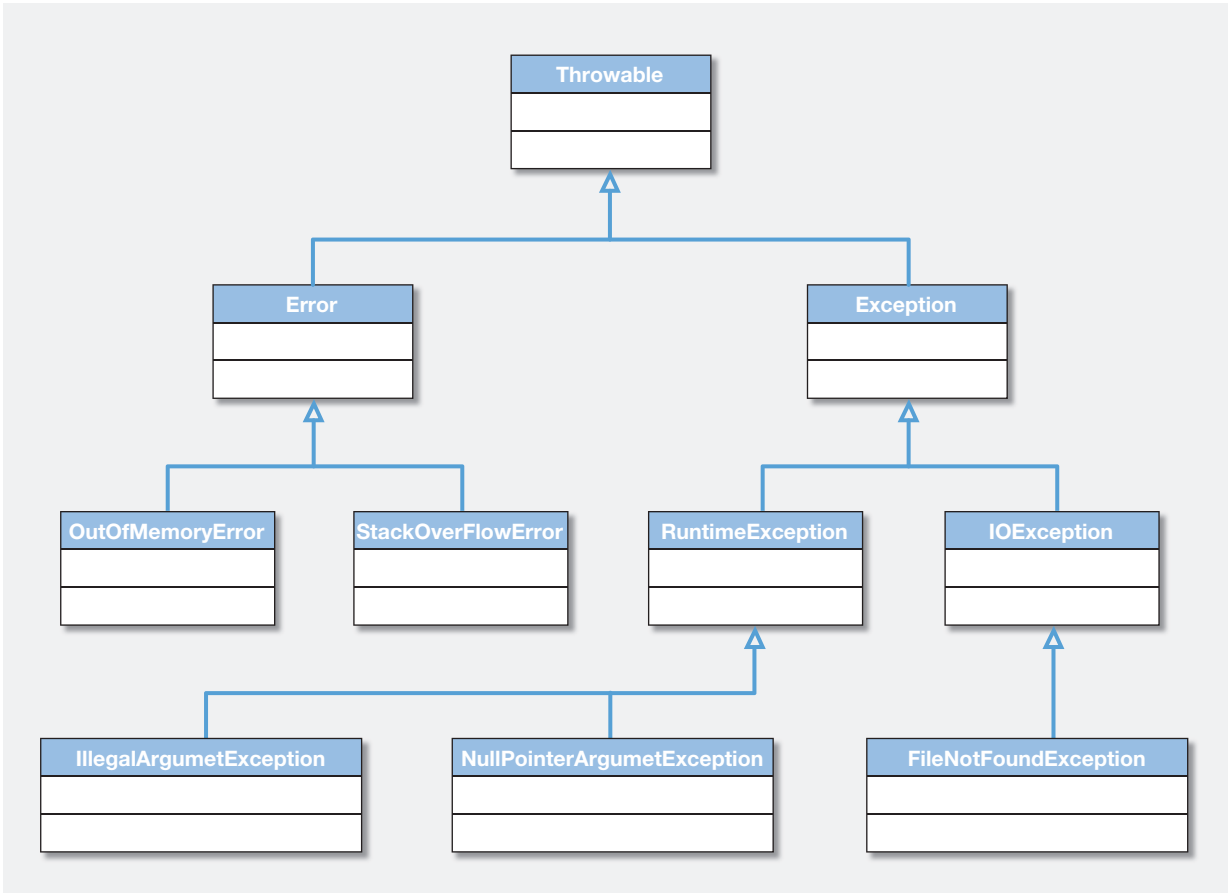



FIGURA **7**

I possibili tipi di eccezioni generate in un programma Java sono strutturati in una gerarchia di classi rappresentata schematicamente e in maniera non esaustiva nel diagramma UML di FIGURA **7**.

**OSSERVAZIONE** Le eccezioni che derivano dalla classe *Error* (tipicamente relative al funzionamento della JVM) pur essendo intercettabili sono in ogni caso irrecuperabili. Le eccezioni che derivano dalla classe *RuntimeException* sono eccezioni *unchecked* («non controllato») che il programmatore può decidere di intercettare o meno; le altre eccezioni che derivano dalla classe *Exception* sono di tipo *checked* e il programmatore deve necessariamente intercettarle e gestirle.

**ESEMPIO**

La classe *Mensola* introdotta in precedenza può essere riscritta utilizzando opportunamente l'intercettazione delle eccezioni per evitare i tediosi controlli sulla correttezza degli indici dei vettori e sulla nullità o meno dei riferimenti:

```
public class Mensola {
  // Attributi
  private static final int MAX_NUM_VOLUMI=15;
  private Libro volumi[];
```

```
// Costruttori
public Mensola() {
  volumi=new Libro[MAX_NUM_VOLUMI];
}

public Mensola(Mensola mensola) {
  int posizione;

  volumi=new Libro[MAX_NUM_VOLUMI];
  for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
    if (mensola.getVolume(posizione)!=null)
      volumi[posizione]=new Libro(mensola.getVolume(posizione));
  }
}

// Metodi
public int setVolume(Libro libro, int posizione) {
  try {
        // inserimento copia di libro in posizione
        if (volumi[posizione]!=null)
          return -2; // posizione occupata
        volumi[posizione]=new Libro(libro);
        return posizione; // restituisce la posizione di inserimento
      }
  catch (ArrayIndexOutOfBoundsException exception) {
    return -1; // posizione non valida
  }
}

public Libro getVolume(int posizione) {
  try {
        // restituisce una copia del libro in posizione
        return new Libro(volumi[posizione]);
      }
  catch (ArrayIndexOutOfBoundsException exception) {
    return null; // posizione non valida: nessun libro restituito
  }
  catch (NullPointerException exception) {
    return null; // posizione vuota: nessun libro restituito
  }
}

public int rimuoviVolume(int posizione) {
  try {
        volumi[posizione]=null; // rimozione libro in posizione
        return posizione; // restituisce la posizione liberata
      }
  catch (ArrayIndexOutOfBoundsException exception) {
    return -1; // posizione non valida
  }
}
```

```
  public int getNumMaxVolumi() {
    return MAX_NUM_VOLUMI;
  }

  public int getNumVolumi() {
    int posizione, n=0;
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
      if (volumi[posizione]!=null)
        n++;
    return n;
  }
}
```

**OSSERVAZIONE** Essendo un'eccezione Java un normale oggetto (istanza di una classe che deriva dalla classe *Exception*) è possibile da parte del programmatore definire nuovi tipi di eccezione definendo nuove classi che derivano dalla classe *Exception*.

## 10.2 Definizione e generazione delle eccezioni

Ogni singolo metodo di una classe viene progettato e implementato per effettuare una specifica operazione: in presenza di situazioni anomale è possibile che un metodo fallisca senza portare a termine l'operazione. Questa evenienza deve essere segnalata al metodo invocante che potrà, a seconda dei casi, prevedere o meno una contromisura per concludere il proprio compito nonostante il fallimento del metodo chiamato, oppure, se questo risulta impossibile, dichiarare segnalare a sua volta il proprio fallimento nei confronti del proprio chiamante. Per segnalare il proprio insuccesso un metodo Java può generare (si utilizzano comunemente anche i verbi «sollevare», o «lanciare») un'eccezione. La generazione di un'eccezione da parte di un metodo è un comportamento analogo alla restituzione di un valore, ma in Java i due concetti sono volutamente distinti: un metodo può ad esempio restituire un valore di tipo `int`, **oppure** generare un'eccezione che è un oggetto istanza di una classe che deriva dalla classe *Exception*.

Un metodo che può generare un'eccezione deve specificare questa eventualità nella propria firma utilizzando la parola chiave `throws` seguita dal tipo di eccezione, o da un elenco di tipi di eccezioni, che possono essere sollevate dal metodo; nel codice del metodo la parola chiave `throw` consente di interrompere l'esecuzione e di sollevare l'eccezione specificata come un nuovo oggetto.

**ESEMPIO**

La dichiarazione del metodo *differenzaOrari* di una ipotetica classe *GestioneOrari* specifica che possono essere sollevate eccezioni di tipo *OrarioNonValido* la cui effettiva generazione avviene se i valori delle ore (0–23), dei minuti (0–59) e dei secondi (0–59) forniti come parametri non rispettano i limiti verificati dal metodo privato *orarioValido*: ▶

```
/* Verifica la validità di un orario espresso nel formato
   h:m:s (ore, minuti e secondi)
*/
private static boolean orarioValido(int h, int m, int s) {
  if (h>=0 && h<24 && m>=0 && m<60 && s>=0 && s<60)
    return true;
  else
    return false;
}

/* Calcola la differenza in secondi fra 2 orari espressi nel formato
   h:m:s (ore, minuti e secondi)
*/
public static int differenzaOrari(int h1, int m1, int s1, int h2, int m2, int s2)
                                    throws OrarioNonValido {
if (!orarioValido(h1, m1, s1) || !orarioValido(h2, m2, s2))
  throw new OrarioNonValido();
else {
       int sec1, sec2;
       sec1 = h1*3600 + m1*60 + s1;
       sec2 = h2*3600 + m2*60 + s2;
       return (sec2-sec1);
     }
}
```

Se non si verificano errori il metodo *differenzaOrari* restituisce un valore intero che rappresenta il numero di secondi che intercorrono tra i due orari. Nel caso in cui uno dei due orari non risulti valido (ad esempio 12h, 62m e 54s), invece di restituire un valore intero il metodo esegue l'istruzione

```
throw new OrarioNonValido();
```

che istanzia e solleva un'eccezione di tipo *OrarioNonValido*. Il tipo di eccezione *OrarioNonValido* deve essere stato in precedenza definito come una classe, eventualmente nella seguente forma minimale:

```
public class OrarioNonValido extends Exception {
}
```

La semantica dell'istruzione `throw` ha alcuni aspetti in comune con quella dell'istruzione `return`: in particolare l'esecuzione di `throw` implica la terminazione immediata del metodo e il passaggio del controllo al codice chiamante del metodo stesso.

**OSSERVAZIONE** È importante non confondere la *keyword* `throw` (un'istruzione che genera un'eccezione) con la *keyword* `throws` (usata nella firma di un metodo per elencare i tipi di eccezione che esso può generare).

Il codice che invoca un metodo che dichiara la potenziale generazione di eccezioni utilizzando la parola chiave `throws` nella firma deve gestirle mediante un blocco `try`/`catch`.

**ESEMPIO**

Il metodo *main* della classe *GestioneOrari* dell'esempio precedente deve verificare il corretto funzionamento del metodo *differenzaOrari* anche nel caso di orari non validi:

```
public static void main(String[] args) {
  …
  try  {
        int ss = GestioneOrari.differenzaOrari(0, 0, 0, 23, 59, 59);
        System.out.println("Secondi di differenza: "+ss);
  }
  catch(OrarioNonValido exception) {
    System.out.println("Errore nell'orario specificato!");
  }
  try  {
        int ss = GestioneOrari.differenzaOrari(0, 0, 0, 24, 0, 0);
        System.out.println("Secondi di differenza: "+ss);
  }
  catch(OrarioNonValido exception) {
    System.out.println("Errore nell'orario specificato!");
  }
  …
}
```

Il blocco **try** controlla la corretta esecuzione delle istruzioni che lo costituiscono, in particolare in questo caso dell'invocazione del metodo *differenzaOrari* che può generare due situazioni distinte:

- il metodo ha successo e ritorna un valore intero (è quello che accade nel primo caso): l'esecuzione delle istruzioni del blocco prosegue normalmente e al termine il controllo passa alla prima istruzione successiva al blocco **catch**;
- il metodo fallisce e solleva un'eccezione di tipo *OrarioNonValido* (è quello che accade nel secondo caso): l'esecuzione delle istruzioni si interrompe e sono eseguite le istruzioni del blocco **catch** prima che il controllo passi alla prima istruzione successiva al blocco stesso.

Il blocco **try**/**catch** consente di separare il funzionamento del metodo nel caso «normale» e nel caso in cui sia necessario gestire eventuali situazioni anomale; il metodo *main* precedente visualizza il seguente output:

```
Secondi di differenza: 86399
Errore nell'orario specificato!
```

Nel caso in cui un metodo non sia in grado di effettuare azioni di recupero di un'eccezione sollevata da un metodo invocato non può che risollevare l'eccezione al metodo che a suo volta lo ha invocato.

**ESEMPIO**

Se la classe *GestioneOrari* prevede un metodo statico per confrontare due orari tra loro è ragionevole delegare la gestione di eventuali errori dei parametri forniti al metodo che li ha forniti:

```
public static boolean confrontaOrari(int h1, int m1, int s1, int h2, int m2, int s2)
                                     throws OrarioNonValido {
if (differenzaOrari(h1, m1, s1, h2, m2, s2) == 0)
  return true;
else
  return false;
}
```

▶

Il metodo *confrontaOrari* non prevede un blocco `try`/`catch` e di conseguenza non può intercettare l'eccezione eventualmente generata dall'invocazione del metodo *differenzaOrari*: in questo caso l'esecuzione del codice viene interrotta e l'eccezione generata da *differenzaOrari* viene propagata al metodo che ha invocato il metodo *confrontaOrari* esattamente come se questo avesse eseguito un'istruzione `throw`. Per questo motivo è obbligatorio inserire la dichiarazione

```
throws OrarioNonValido
```

anche nella firma del metodo *confrontaOrari*, segnalando così il fatto che anche questo metodo può indirettamente generare un'eccezione di tipo *OrarioNonValido*.

Nel linguaggio Java in relazione alla gestione delle eccezioni vale la cosiddetta regola «*handle or declare*» («gestisci o dichiara»): a fronte di una possibile eccezione un metodo deve gestirla utilizzando un blocco `try`/`catch` oppure esplicitare a sua volta di sollevarla specificando la clausola `throws`. Una potenziale eccezione non deve in definitiva mai passare inosservata: se non intercettata e gestita deve essere esplicitamente rimandato al chiamante l'obbligo di gestirla.

**OSSERVAZIONE** Le eccezioni dichiarate dal programmatore in linguaggio Java sono normalmente di tipo *checked* («controllato») per le quali si applica la regola «*handle or declare*».

**OSSERVAZIONE** Nei linguaggi che non prevedono la gestione esplicita delle eccezioni un metodo, o una funzione, segnala di norma il proprio fallimento ritornando un valore particolare a cui il programmatore attribuisce convenzionalmente il significato di segnalazione di un errore. In questo contesto ad esempio il metodo *differenzaOrari* potrebbe restituire il valore negativo –1 in caso di fallimento. Questa tecnica di gestione delle «eccezioni» presenta però diverse controindicazioni:

- la segnalazione non è imposta dal linguaggio, ma segue una convenzione che deve essere documentata accuratamente affinché il metodo chiamante possa interpretare il valore restituito dal metodo invocato;
- non sempre è possibile identificare un valore particolare da usare come segnalazione di errore; se ad esempio il metodo *differenzaOrari* è pensato per fornire la differenza non i valore assoluto allora il valore negativo –1 è un risultato lecito e non può rappresentare una condizione di errore anomala.

In ogni caso senza gestione delle eccezioni non esiste alcun vincolo che imponga al metodo chiamante di verificare se il metodo invocato ha fallito: rendendo obbligatoria la gestione delle eccezioni sollevate il modello proposto dal linguaggio Java impedisce alle anomalie di esecuzione di passare inosservate. Questo comporta un onere per il programmatore, ma è un importante contributo alla robustezza e affidabilità del programma.

Le eccezioni generate negli esempi precedenti sono istanze della classe *OrarioNonValido*: il linguaggio di programmazione Java impone che le eccezioni siano oggetti istanza di una classe che deriva da *Exception*, o dalla superclasse *Throwable*.

Se si esclude questo vincolo la definizione di una classe di eccezione è lasciata allo sviluppatore: in particolare è possibile usare attributi e metodi per dotare l'oggetto che rappresenta un'eccezione di informazioni specifiche sul tipo di errore verificatosi.

**ESEMPIO**

La seguente classe che consente di rappresentare un'eccezione relativa all'orario memorizzando l'orario invalido che ha generato l'eccezione stessa:

```
public class OrarioNonValido extends Exception {
  private int h;
  private int m;
  private int s;

  public OrarioNonValido(int h, int m, int s){
    this.h = h;
    this.m = m;
    this.s = s;
  }

  public int getOre() { return h; }
  public int getMinuti() { return m; }
  public int getSecondi() { return s; }
  public String toString() { return (""+h+":"+m+":"+s); }
}
```

Comporta una revisione del metodo statico *differenzaOrari* della classe *GestioneOrari*:

```
public static int differenzaOrari(int h1, int m1, int s1, int h2, int m2, int s2)
                                  throws OrarioNonValido {
  if(!orarioValido(h1, m1, s1))
    throw new OrarioNonValido(h1, m1, s1);
  else if(!orarioValido(h2, m2, s2))
    throw new OrarioNonValido(h2, m2, s2);
  else {
        int sec1, sec2;
        sec1 = h1*3600 + m1*60 + s1;
        sec2 = h2*3600 + m2*60 + s2;
        return (sec2-sec1);
      }
  }
```

In questo modo il metodo che intercetta l'anomalia, diversamente dai casi precedenti, dispone di un oggetto che può fornire informazioni sulla natura dell'errore che ha causato l'anomalia stessa e il metodo *main* della classe *GestioneOrari* potrebbe essere il seguente:

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

▶

```
    try {
          int ss = GestioneOrari.differenzaOrari(0, 0, 0, 23, 59, 59);
          System.out.println("Secondi di differenza: "+ss);
    }
    catch(OrarioNonValido exception) {
      System.out.println("Errore nell'orario specificato ("+ exception.getOre()+":"+
                         exception.getMinuti()+":"+ exception.getSecondi()+")!");
    }
    try {
          int ss = GestioneOrari.differenzaOrari(0, 0, 0, 24, 0, 0);
          System.out.println("Secondi di differenza: "+ss);
    }
    catch(OrarioNonValido exception) {
      System.out.println("Errore nell'orario specificato ("+ exception.getOre()+":"+
                         exception.getMinuti()+":"+ exception.getSecondi()+")!");
    }
    …
}
```

Il parametro che compare nella clausola **catch** è simile al parametro di un metodo: identifica un riferimento a cui viene associato l'oggetto di tipo eccezione istanziato e generato dall'istruzione **throw**. Tale oggetto può essere manipolato come un qualsiasi altro oggetto.

# 11 Gestione dell'input/output

## 11.1 Gestione dell'input/output predefinito

Nel corso degli esempi presentati si sono utilizzate più volte istruzioni del tipo

```
System.out.println("…");
```

che visualizzano i risultati di un programma sullo standard output di sistema (normalmente una finestra testuale sullo schermo). *System.out* è un oggetto predefinito di classe *PrintStream* associato allo standard output, così come *System.err* normalmente utilizzato per la visualizzazione dei messaggi di errore. L'oggetto *System.out* dispone di metodi – come *println* – che accettano come parametro una stringa, o un tipo di dato primitivo da visualizzare. Il metodo *print* – a differenza del metodo *println* – non effettua il ritorno a capo dopo aver visualizzato il parametro passato come argomento.

**OSSERVAZIONE** Entrambi i metodi *println* e *print* accettano un unico argomento, ma utilizzando l'operatore di concatenazione tra stringhe «+» che ha la proprietà di convertire eventuali valori di tipi di dato predefiniti, o di oggetti di classi che implementano il metodo standard *toString* è possibile effettuare visualizzazioni composite.

Per le operazioni di input esiste un oggetto predefinito analogo associato allo standard input (normalmente la tastiera): *System.in* di classe *Input-*

*Stream*. La maggiore complessità delle operazioni di input rende necessario clonare l'oggetto e incapsularlo in un oggetto di classe *BufferedReader* che ne semplifica le funzionalità:

```
InputStreamReader input = new InputStreamReader(System.in);
BufferedReader keyboard = new BufferedReader(input);
```

In questo modo viene definito un oggetto *keyboard* di classe *BufferedReader* che rende disponibile il metodo *readLine* che consente di leggere dallo standard input un'intera riga di testo.

**ESEMPIO**

Le seguenti istruzioni effettuano la lettura di una stringa di caratteri digitata sulla tastiera:

```
String s;
s = keyboard.readLine();
```

L'operazione deve essere effettuata gestendo le eccezioni di classe *IOException* (del *package java.io*) che possono essere generate dal metodo *readLine*:

```
String s;
try  {
       s = keyboard.readLine();
}
catch (java.io.IOException exception){
     …
     …
     …
}
```

Nel caso in cui si debbano leggere dei valori numeri è necessario convertire la stringa comunque restituita dal metodo *readLine* in un valore numero. I metodi della seguente classe di utilità *ConsoleInput* consentono di acquisire da tastiera in modo controllato vari tipi dai dato:

```
import java.io.*;

//Classe acquisire stringhe e numeri dallo standard input.
public class ConsoleInput {
  BufferedReader reader;

  /* Istanzia un oggetto BufferedReader sullo standard input
     (System.in) */
  public ConsoleInput() {
    reader = new BufferedReader(new InputStreamReader(System.in));
  }

  // Legge una riga in input e la converte in un valore intero
  public int readInt() throws IOException {
    return Integer.parseInt(reader.readLine());
  }
```

▶

```
  //Legge una riga in input e la converte in un valore double
  public double readDouble() throws IOException {
    return Double.parseDouble(reader.readLine());
  }

  // Legge una riga di testo in input.
  public String readLine() throws IOException {
    return reader.readLine();
  }
}
```

**OSSERVAZIONE** I metodi statici di conversione delle classi *Integer* (*parseInt*) e *Double* (*parseDouble*) generano un'eccezione di tipo *NumberFormatException* che, essendo sottoclasse di *RuntimeException*, è di tipo *unchecked* e quindi non deve necessariamente essere intercettata, o dichiarata, lasciandone l'eventuale gestione al codice chiamante.

**ESEMPIO**

La classe *InputCoin* ha un unico metodo *main* che esemplifica l'uso delle funzionalità rese disponibili dalla classe *ConsoleInput*:

```
public class InputCoin {
  public static void main(String[] args) {
    int centesimi = 0;
    double euro = 0.0, totale;
    boolean ok;
    String chi = "";

    // crea un oggetto ConsoleInput
    ConsoleInput keyboard = new ConsoleInput();

    System.out.println("Quanti centesimi hai?");
    ok = false;
    while(!ok) {
      try {
            centesimi = keyboard.readInt();
            ok = true;
          }
      catch(java.io.IOException exception) {
        System.out.println("Valore non corretto: "+ "reinseriscilo!");
      }
      catch(NumberFormatException exception) {
        System.out.println("Valore non corretto: "+ "reinseriscilo!");
      }
    }

    System.out.println("Supponiamo di avere 75 centesimi "+ "di Euro,
                       quanti Euro sono?");
    ok=false;
```

▶

```
    while(!ok) {
      try  {
             euro = keyboard.readDouble();
             ok = true;
           }
      catch(java.io.IOException exception) {
        System.out.println("Valore non corretto: "+ "reinseriscilo!");
      }
      catch(NumberFormatException exception) {
        System.out.println("Valore non corretto: "+ "reinseriscilo!");
      }
      if (ok)
        if (euro != 0.75) {
          System.out.println("I conti non tornano! Quanti Euro "+
                             "sono 75 centesimi?");
          ok=false;
        }
      }

    System.out.println("A chi devi questi soldi?");
    try  {
           chi = keyboard.readLine();
         }
    catch(java.io.IOException exception) {
    }

    totale = centesimi*0.01 + euro;
    System.out.println("Devi "+totale+"€ a "+chi);
  }
}
```

**OSSERVAZIONE** Nell'esempio precedente l'invocazione del metodo *readLine* dell'oggetto *keyboard* istanza della classe *ConsoleInput* deve necessariamente essere inserita in un blocco `try`/`catch` perché è dichiarata la possibile generazione dell'eccezione di tipo *IOException*, ma dato che in pratica questa eccezione non può essere sollevata da nessun possibile input da parte dell'utente il blocco `catch` è stato lasciato vuoto.

## 11.2 Gestione dell'input/output da file di testo

Con tecniche di programmazione simili a quelle utilizzate nel paragrafo precedente per la gestione dell'input/output predefinito è possibile gestire l'input/output da/in un file di testo. I metodi della seguente classe di utilità *TextFile* consentono di aprire, in lettura o scrittura, e chiudere un file di testo e di leggere o scrivere da/in esso stringhe corrispondenti a linee di testo:

```
import java.io.*;

/**
  * Classe per la gestione sequenziale di un file di testo.
  */
public class TextFile {
  private char mode; // R=read, W=write
  private BufferedReader reader;
  private BufferedWriter writer;

  /* Costruisce un oggetto di tipo BufferedReader/BufferedWriter sopra
     il file specificato dal nome indicato */
  public TextFile(String filename, char mode) throws IOException
  {
    this.mode = 'R';
    if (mode == 'W' || mode == 'w') {
      this.mode = 'W';
      writer = new BufferedWriter(new FileWriter(filename));
    }
    else {
      reader = new BufferedReader(new FileReader(filename));
    }
  }

  // Scrive una riga di testo in un file aperto in scrittura.
  public void toFile(String line) throws FileException, IOException
  {
    if (this.mode == 'R') throw new FileException("Read-only "+ "file!");
    writer.write(line);
    writer.newLine();
  }

  // Legge una riga di testo da un file aperto in lettura.
  public String fromFile() throws FileException, IOException
  {
    String tmp;
    if (this.mode == 'W') throw new FileException("Write-only"+"file!");
      tmp = reader.readLine();
    if (tmp == null) throw new FileException("End of file!");
    return tmp;
  }

  // Chiude il file aperto dal costruttore.
  public void closeFile() throws IOException
  {
    if (this.mode == 'W')
      writer.close();
```

```
      else // this.mode == 'R'
        reader.close();
    }

    public static void main(String args[]) throws IOException
    {
      TextFile out = new TextFile("file.txt",'W');
      try {
            out.toFile("Riga 1");
            out.toFile("Riga 2");
            out.toFile("Riga 3");
          }
      catch (FileException exception) {
        System.out.println(exception.getMatter());
      }
      out.closeFile();

      TextFile in = new TextFile("file.txt",'R');
      String line;
      try {
            while (true) {
              line = in.fromFile();
              System.out.println(line);
            }
          }
      catch (FileException exception) {
        System.out.println(exception.getMatter());
      }
      out.closeFile();
    }
}
```

La classe di utilità *TextFile* genera eccezioni specifiche di tipo *FileException* definite dalla seguente classe che consente, nel costruttore, di specificare un messaggio descrittivo dell'errore che ha causato l'anomalia:

```
public class FileException extends Exception {
  private String matter="";

  public FileException(String matter) {
    this.matter = matter;
  }

  public String getMatter() {
    return this.matter;
  }
}
```

**OSSERVAZIONE** Il metodo *main* della classe *TextFile* oltre a costituire un test dei metodi della classe rappresenta un esempio di uso della stessa: in particolare il metodo *fromFile* genera un'eccezione di tipo *FileException* quando viene invocato dopo che è stata raggiunta la fine del file per cui un normale ciclo di lettura di tutte le righe è un ciclo potenzialmente infinito da cui si esce intercettando l'eccezione. L'esecuzione del metodo *main*, oltre a creare un file denominato «file.txt» costituito da 3 righe di testo – rispettivamente «Riga 1», «Riga 2» e «Riga 3» – produce il seguente output:

```
Riga 1
Riga 2
Riga 3
End of file!
```

dove l'ultima riga è prodotta dalla visualizzazione del messaggio dell'eccezione sollevata dall'invocazione del metodo *fromFile* dopo la lettura dell'ultima riga del file.

**ESEMPIO**

Volendo dotare la classe *Mensola* della possibilità di salvare/ripristinare su/da file l'elenco dei libri che essa contiene è possibile aggiungere i seguenti metodi:

```
public void salvaVolumi() throws java.io.IOException {
  TextFile out = new TextFile("volumi.txt", 'W');

  try {
        for (int posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
          if (volumi[posizione]!=null) {
            String line = Integer.toString(posizione);
            line += ";"+volumi[posizione].getTitolo();
            line += ";"+volumi[posizione].getAutore();
            line += ";"+volumi[posizione].getNumeroPagine();
            out.toFile(line);
          }
      }
  catch (FileException exception) {
  }
  out.closeFile();
}

public void caricaVolumi() throws java.io.IOException {
  TextFile in = new TextFile("volumi.txt", 'R');
  int posizione, pagine;
  String linea, autore, titolo;
  String[] elementi;
  Libro libro;

  try {
        while(true) {
          linea = in.fromFile();
          elementi = linea.split(";");
          if (elementi.length == 4) {
            posizione = Integer.parseInt(elementi[0]);
            titolo = elementi[1];
```

▶

```
                autore = elementi[2];
                pagine = Integer.parseInt(elementi[3]);
                libro = new Libro(titolo, autore, pagine);
                setVolume(libro, posizione);
              }
          }
      }
      catch (FileException exception) {
    }
}
```

Il metodo *salvaVolumi* crea un file di testo in formato CSV (*Comma Separated Values*) in cui ogni riga corrisponde a un volume e i valori della posizione, del titolo, dell'autore e del numero di pagine sono separati dal carattere «;», come nel seguente esempio:

```
0;Pinocchio;C. Collodi;150
2;Pollicino;C. Perrault;80
10;La bella addormentata nel bosco;C. Perrault;50
```

Il metodo *caricaVolumi* legge ogni singola riga del file di testo e utilizza il metodo *split* della classe *String* per separare in un array di stringhe (il vettore *elementi*) gli elementi della riga del file separati dal carattere fornito come parametro (in questo caso il carattere «;»); a partire da queste sottostringhe della riga del file che in alcuni casi devono essere convertire in valori numerici viene costruito un oggetto di classe *Libro* che viene fornito come argomento al metodo *setVolume* della classe *Mensola* per inserire il volume caricato dal file nella posizione indicata. Il seguente metodo *main* serve a testare i metodi illustrati

```
public static void main (String[] args) throws java.io.IOException {
  Mensola mensola = new Mensola();

  Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
  Libro l3 = new Libro("La bella addormentata nel bosco", "C. Perrault", 50);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // inserimento volumi
  mensola.setVolume(l1, 0);
  mensola.setVolume(l2, 2);
  mensola.setVolume(l3, 10);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // salvataggio volumi su file
  mensola.salvaVolumi();
  // creazione nuova mensola vuota
  mensola = new Mensola();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // caricamento volumi da file
  mensola.caricaVolumi();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
}
```

e produce il seguente output

```
Numero volumi: 0
Numero volumi: 3
Numero volumi: 0
Numero volumi: 3
```

che dimostra il successo del caricamento dei volumi dal file dove erano stati in precedenza salvati.

# 12 Serializzazione e persistenza degli oggetti su file

La serializzazione di un oggetto è un processo che permette di salvarlo in un supporto di memorizzazione sequenziale (come un file), o di trasmetterlo su un canale di comunicazione sequenziale (come una connessione di rete). La serializzazione può essere effettuata in forma binaria, oppure può impiegare codifiche testuali come il formato XML (*eXtensible Markup Language*). Lo scopo della serializzazione è quello di salvare e/o trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, spesso denominato «deserializzazione».

**ESEMPIO**

Volendo salvare lo stato di un oggetto istanza della classe *Mensola* introdotta in precedenza possiamo aggiungere il seguente metodo che serializza e memorizza in un file denominato «volumi.bin» l'array *volumi* unico attributo della classe:

```
public void salvaMensola() throws java.io.IOException {
   ObjectOutputStream stream =
   new ObjectOutputStream(new FileOutputStream("volumi.bin"));

   stream.writeObject(this.volumi);
   stream.close();
}
```

La deserializzazione dell'array *volumi* può essere effettuata a partire dal contenuto del file mediante il seguente metodo:

```
public void caricaMensola() throws java.io.IOException {
   ObjectInputStream stream =
   new ObjectInputStream(new FileInputStream("volumi.bin"));

   try  {
          this.volumi = (Libro[])stream.readObject();
        }
   catch (ClassNotFoundException exception) {
   }
   stream.close();
}
```

È necessario in questo caso effettuare un *casting* di quanto restituito dal metodo *readObject* al tipo dell'attributo cui lo si assegna (in questo caso un array di oggetti di classe *Libro*): questa operazione, in caso di discordanza tra il tipo specificato e quanto contenuto nel file, può generare un'eccezione di tipo *ClassNotFoundException* che deve essere di conseguenza intercettata.

**OSSERVAZIONE** Nel caso che una classe abbia più attributi è possibile serializzarli all'interno dello stesso file, è però necessario che l'ordine di invocazione dei metodi *readObject* sia rigorosamente lo stesso con cui sono stati invocati i metodi *writeObject* per evitare di assegnare un oggetto a un riferimento incoerente generando un'eccezione di tipo *ClassNotFoundException*.

Le classi degli oggetti che devono essere serializzati devono implementare l'interfaccia *Serializable* del *package java.io*: se hanno come attributi istanze di altre classi ciascuna di esse deve implementare la stessa interfaccia. Dato che *Serializable* è un'interfaccia vuota non è necessario definire alcun metodo particolare.

Nel caso specifico della classe *Libro* cui si riferisce l'esempio precedente è sufficiente ridefinirne la dichiarazione nel seguente modo:

```
public class Libro implements java.io.Serializable {
  …
  …
}
```

**ESEMPIO**

Per testare la funzionalità del codice visto è possibile modificare il metodo *main* della classe *Mensola* come segue:

```
public static void main (String[] args) throws java.io.IOException {
  Mensola mensola = new Mensola();

  Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
  Libro l3 = new Libro("La bella addormentata nel bosco", "C. Perrault", 50);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // inserimento volumi
  mensola.setVolume(l1, 0);
  mensola.setVolume(l2, 2);
  mensola.setVolume(l3, 10);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // visualizzazione elenco volumi
  for (int posizione=0; posizione<mensola.getNumMaxVolumi(); posizione++) {
    Libro libro = mensola.getVolume(posizione);
    if (libro != null)
      System.out.println("posizione: "+posizione+" -> "+ libro.getTitolo()+"
                         "+ libro.prezzo()+"€");
  }
  // salvataggio volumi su file
  mensola.salvaMensola();
  // creazione nuova mensola vuota
  mensola = new Mensola();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // caricamento volumi da file
  mensola.caricaMensola();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // visualizzazione elenco volumi
  for (int posizione=0; posizione<mensola.getNumMaxVolumi(); posizione++) {
    Libro libro = mensola.getVolume(posizione);
    if (libro != null)
      System.out.println("posizione: "+posizione+" -> "+ libro.getTitolo()+"
                         "+ libro.prezzo()+"€");
  }
}
```

L'output prodotto dimostra il successo del caricamento del vettore *volumi* dal file dove era stato in precedenza salvato:

```
Numero volumi: 0
Numero volumi: 3
posizione: 0 -> Pinocchio 13.0€
posizione: 2 -> Pollicino 9.5€
posizione: 10 -> La bella addormentata nel bosco 8.0€
Numero volumi: 0
Numero volumi: 3
posizione: 0 -> Pinocchio 13.0€
posizione: 2 -> Pollicino 9.5€
posizione: 10 -> La bella addormentata nel bosco 8.0€
```

La serializzazione in un file degli attributi di un oggetto consente di renderlo «persistente»: è infatti sufficiente invocare il metodo che serializza gli attributi su file prima della sua distruzione e richiamare il metodo che deserializza gli attributi da file nel costruttore (intercettando l'eventuale eccezione generata nel caso che il file non esista) per fare in modo che un oggetto sopravviva all'esecuzione del programm in cui viene creato.

**OSSERVAZIONE** I file in cui sono serializzati gli oggetti Java hanno un formato binario per cui non è possibile ispezionarli utilizzando un normale editor. La serializzazione di un oggetto può esporre i dettagli implementativi della classe di cui è istanza: spesso i produttori software non documentano il formato di serializzazione e in alcuni casi cifrano i dati serializzati.

# 13 Ereditarietà

Prendiamo ancora una volta in considerazione la classe *Punto* introdotta precedentemente (FIGURA 8).

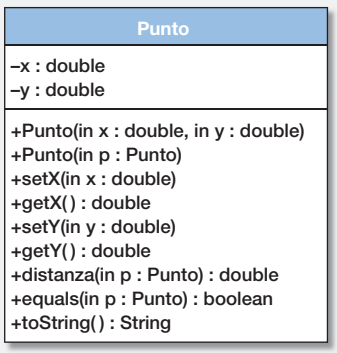



FIGURA 8

Supponiamo adesso di voler definire la classe *PuntoOrientato* che rispetto alla precedente presenti le seguenti componenti aggiuntive:

- un attributo in più, ovvero la direzione verso cui è orientato il punto (Alto, Basso, Destra, Sinistra con riferimento all'usuale rappresentazione grafica del piano cartesiano);
- un costruttore privo di parametri, uno con parametri per tutti gli attributi e un costruttore di copia;
- due metodi che ruotano il punto rispettivamente in senso orario (verso destra) o antiorario (verso sinistra): quando invocati modificano l'orientamento verso il punto cardinale adiacente a quello corrente;
- un metodo che sposta il punto di una certa distanza in direzione dell'orientamento corrente;

e inoltre riformuli i seguenti metodi per tenere conto dell'attributo aggiuntivo:

- *equals* per stabilire se un punto orientato è uguale a un altro;
- *toString* per formattare in una stringa il contenuto informativo di un oggetto di classe *PuntoOrientato*.

In questi casi, piuttosto che definire una nuova classe, è possibile sfruttare la classe esistente (*Punto*) e definire la nuova classe (*PuntoOrientato*) come «estensione» della classe originale. In questo modo *PuntoOrientato* eredita tutti i membri di *Punto* e vi sarà solo la necessità di introdurre quelli aggiuntivi. La rappresentazione UML di questa situazione è mostrata in FIGURA **9** dove la generalizzazione, come avevamo visto nel primo capitolo, è rappresentata dalla freccia orientata che unisce la classe derivata alla classe originale.

> Nel linguaggio di programmazione Java la derivazione di una classe a partire da una classe preesistente si ottiene con l'uso della parola chiave `extends`.

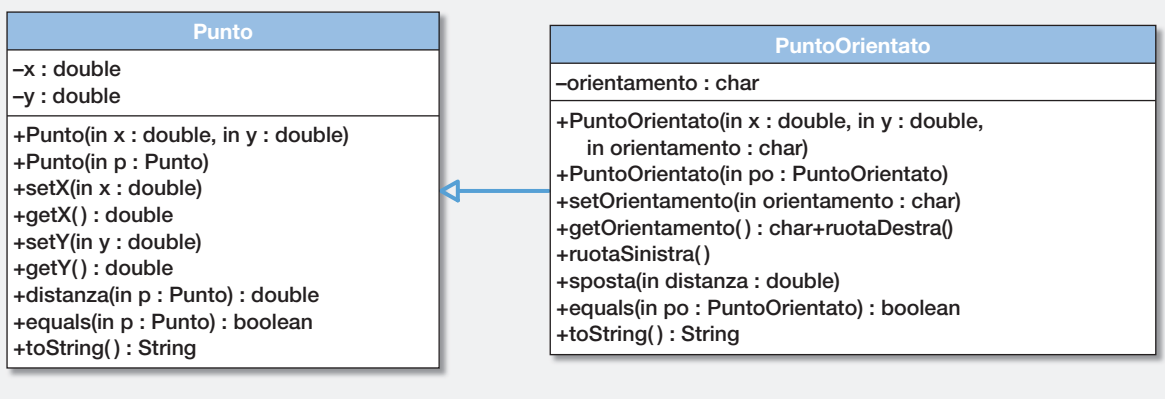



FIGURA **9**

ESEMPIO

Senza ridefinire la classe *Punto* introdotta nel secondo capitolo, la classe *PuntoOrientato* in Java potrebbe avere la seguente implementazione:

```
public class EccezionePuntoOrientato extends Exception {}

public class PuntoOrientato extends Punto {
  private char orientamento; // nuovo attributo

  // costruttori
  public PuntoOrientato() {
    super(0., 0.);
    this.orientamento = 'A';
  }

  public PuntoOrientato(double x, double y, char orientamento)
         throws EccezionePuntoOrientato {
    super(x, y);
    setOrientamento(orientamento);
  }

  public PuntoOrientato(PuntoOrientato po)
         throws EccezionePuntoOrientato {
    super(po.getX(), po.getY());
    setOrientamento(po.getOrientamento());
  }

  // set/get del nuovo attributo
  public void setOrientamento(char orientamento)
              throws EccezionePuntoOrientato {
    if ((orientamento!='A') && (orientamento!='B') &&
        (orientamento!='D') && (orientamento!='S'))
      throw new EccezionePuntoOrientato();
    this.orientamento = orientamento;
  }

  public char getOrientamento() {
    return orientamento;
  }

  // altri metodi
  public void ruotaDestra() {
    switch (orientamento) {
      case 'A': orientamento = 'D'; break;
      case 'D': orientamento = 'B'; break;
      case 'B': orientamento = 'S'; break;
      case 'S': orientamento = 'A'; break;
    }
  }

  public void ruotaSinistra() {
    switch (orientamento) {
      case 'A': orientamento = 'S'; break;
```

▶

```
        case 'D': orientamento = 'A'; break;
        case 'B': orientamento = 'D'; break;
        case 'S': orientamento = 'B'; break;
      }
    }

    public void sposta(double distanza){
      switch (orientamento) {
        case 'A': setY(getY() + distanza); break;
        case 'B': setY(getY() - distanza); break;
        case 'S': setX(getX() - distanza); break;
        case 'D': setX(getX() + distanza); break;
      }
    }

    public boolean equals(PuntoOrientato po) {
      return (super.getX() == po.getX() && super.getY() == po.getY() &&
              this.getOrientamento() == po.getOrientamento());
    }

    public String toString() {
      return super.toString()+orientamento;
    }
}
```

Il seguente metodo *main* consente di testare le funzionalità della classe *PuntoOrientato*

```
public static void main(String[] args) {
    PuntoOrientato po1 = new PuntoOrientato();
    PuntoOrientato po2 = new PuntoOrientato();
    PuntoOrientato po3 = new PuntoOrientato();
    PuntoOrientato po4 = new PuntoOrientato();

    try {
      po1 = new PuntoOrientato(1.,1.,'D');
      po2 = new PuntoOrientato(2.,2.,'A');
      po3 = new PuntoOrientato(po1);
      po4 = new PuntoOrientato(0.,0.,'X');
    }
    catch (EccezionePuntoOrientato eccezione) {
      System.out.println("Generata eccezione");
    }

    System.out.println("PO1 = "+po1.toString());
    System.out.println("PO2 = "+po2.toString());
    System.out.println("PO3 = "+po3.toString());
    System.out.println("Distanza PO1-PO2: "+po1.distanza(po2));
    System.out.println("Distanza PO1-PO3: "+po1.distanza(po3));

    if (po1.equals(po3))
      System.out.println("P01 e P03 coincidono");
```

```
    else
      System.out.println("P01 e P03 NON coincidono");
    po1.ruotaSinistra();
    System.out.println("PO1 = "+po1.toString());
    po1.sposta(10.0);
    System.out.println("PO1 = "+po1.toString());
}
```

e produce il seguente output:

```
Generata eccezione
PO1 = (1.0;1.0)D
PO2 = (2.0;2.0)A
PO3 = (1.0;1.0)D
Distanza PO1-PO2: 1.4142135623730951
Distanza P01-P03: 0.0
P01 e P03 coincidono
PO1 = (1.0;1.0)A
PO1 = (1.0;11.0)A
```

**OSSERVAZIONE** La parola chiave `super` nell'esempio precedente riferisce gli attributi e i metodi – compreso il costruttore – della classe *Punto* da cui la classe *PuntoOrientato* deriva.

## 13.1 Classi derivate; *overriding* e *overloading* dei metodi

L'ereditarietà è un meccanismo di astrazione finalizzato alla creazione di gerarchie di classi: con essa si ha la possibilità di riutilizzare codice comune a più classi della stessa gerarchia.
Nell'esempio di apertura abbiamo visto come, tramite la parola chiave `extends`, sia possibile dichiarare una nuova classe come estensione di una classe già esistente da cui la nuova classe eredita i membri (attributi e metodi). La classe che viene estesa è denominata «superclasse» e rappresenta una generalizzazione della «sottoclasse» derivata che è invece una specializzazione della classe originale. La classe derivata può aggiungere aggiungere nuovi membri a quelli ereditati dalla superclasse, o ridefinirne alcuni.
La parola chiave `final` del linguaggio Java applicata a una classe impedisce di estenderla per eredità.

**OSSERVAZIONE** I costruttori di una classe `non` sono ereditati dalle classi derivate che deve necessariamente ridefinire i propri costruttori.

Analizzando la classe *PuntoOrientato* dell'esempio di apertura si può osservare che:

### Il riuso del software

Il meccanismo di derivazione delle classi e la conseguente ereditarietà di metodi e attributi è la modalità con cui i linguaggi di programmazione OO consentono di realizzare la pratica del riuso del software.
Il riuso di software esistente – già progettato, implementato e testato – nella realizzazione di nuovo software è considerata una pratica fondamentale per il rispetto dei tempi e dei costi di sviluppo da parte di molte aziende e organizzazioni.
La realizzazione di «librerie» di classi estendibili e quindi facilmente adattabili a nuovi contesti utilizzando il meccanismo dell'ereditarietà del codice ha rivoluzionato la pratica dello sviluppo software.

- definisce il nuovo attributo (nuovo rispetto agli attributi della classe originale *Punto*) *orientamento* di tipo carattere che consente di codificare la direzione di orientamento del punto; gli attributi *x* e *y* non devono essere nuovamente definiti perché sono ereditati dalla classe *Punto*;
- il metodo *setOrientamento* accetta come valori validi i caratteri «A» (Alto), «B» (Basso), «D» (Destra) e «S» (Sinistra): nel caso sia fornito un carattere diverso viene generata una specifica eccezione;
- i metodi *equals* e *toString* sono ridefiniti rispetto ai metodi omonimi della classe *Punto* per includere la gestione del nuovo attributo *orientamento*.

**OSSERVAZIONE** Sia nella classe *Punto* che nella classe derivata *PuntoOrientato* vi sono più costruttori: tutti hanno lo stesso nome e, come già abbiamo visto nei capitoli precedenti, sarà il compilatore a selezionare quale utilizzare in funzione degli argomenti specificati al momento dell'invocazione. Questa tecnica è denominata *overloading* e consente di definire più metodi (non esclusivamente i costruttori) nell'ambito della stessa classe aventi lo stesso nome a patto che differiscano nella firma per numero e tipo dei parametri specificati.

La sola modifica del tipo del valore di ritorno non è sufficiente a realizzare un *overloading* e non è accettata dal compilatore.

La tecnica di ridefinizione in una classe derivata di metodi già presenti nella superclasse è denominata *overridding*: un metodo sovrascritto nasconde – negli oggetti istanza della classe derivata – la definizione data nella superclasse sostituendola con quella data nella classe derivata. La parola chiave `final` del linguaggio Java applicata a un metodi di una classe ne impedisce la ridefinizione in una classe derivata.

**OSSERVAZIONE** Un metodo ridefinito in una classe derivata nasconde un metodo omonimo della superclasse solo se la sua firma è esattamente la stessa del metodo originale; nell'esempio della classe *PuntoOrientato* il metodo *toString* effettua l'*overriding* del metodo omonimo della classe *Punto*, mentre il metodo *equals* originale rimane disponibile anche per gli oggetti di classe *PuntoOrientato* in *overloading* con il metodo originale ereditato.

Il linguaggio Java definisce una classe radice da cui derivano implicitamente tutte le altre: la classe *Object*.

Tale classe prevede tra i suoi metodi anche i metodi *toString* ed *equals*. Quando il compilatore Java deve convertire un oggetto in una stringa, invoca il metodo *toString* della classe di cui l'oggetto è istanza; se il programmatore non ha ridefinito nella propria classe il metodo *toString* viene invocato il metodo originale definito nella classe *Object* e automaticamente ereditato da qualsiasi classe Java: esso genera una stringa che è spesso di scarsa utilità, per ottenere un risultato che abbia un senso è necessario ridefinire nella propria classe un appropriato metodo *toString* effettuandone l'*overriding*.

**ESEMPIO**

Se nella classe *Punto* non fosse stato definito un metodo *toString* il seguente frammento di codice

```
…
Punto p = new Punto(0.,0.);
System.out.println(p);
…
```

produrrebbe un output del tipo

```
Punto@3e25a5
```

invece di

```
(0.0;0.0)
```

che è la stringa formattata dal metodo ridefinito. Infatti il risultato prodotto dal metodo *toString* della classe *Object* è una stringa che, oltre al nome della classe di cui l'oggetto è istanza, comprende la codifica esadecimale del codice *hash* dell'oggetto stesso, così come fornita dal metodo *hashCode* della stessa classe *Object*.

**OSSERVAZIONE** Il metodo *hashCode* della classe *Object* restituisce la codifica *hash* dell'indirizzo di memorizzazione nello *heap* dell'oggetto su cui è invocato. Dato che anche il metodo *equals* definito nella classe *Object* opera sui risultati del metodo *hashCode* esso considera uguali due oggetti solo se sono lo stesso oggetto: due oggetti distinti infatti, anche se con tutti gli attributi tra loro identici, hanno necessariamente indirizzi diversi e il risultato del loro confronto sarà sempre falso!

In generale l'ereditarietà viene utilizzata per fattorizzare (raggruppare) attributi comuni a più classi evitando inutili ridondanze.

**ESEMPIO**

Per sviluppare un'applicazione per la gestione del personale della scuola è necessario definire le classi *Impiegato* e *Docente*. Tra di esse è possibile individuare attributi comuni a tutti i dipendenti come il nominativo, il sesso e l'indirizzo e attributi specifici come l'ufficio per gli impiegati, o come il ruolo (supplente, insegnante diplomato, insegnante laureato, …) e la disciplina di insegnamento per i docenti. Per evitare inutili ridondanze si può introdurre una classe *Dipendente* per la gestione degli attributi comuni: le classi *Impiegato* e *Docente* sono definite come estensioni di *Dipendente* e comprendono gli attributi specifici.
Il diagramma UML delle classi di FIGURA **10** rappresenta questa soluzione, che in linguaggio Java può essere così implementata:

```
public class Dipendente {
  private String nominativo;
  private char sesso;
  private String indirizzo;

  public Dipendente(String nominativo, char sesso, String indirizzo) {
    this.nominativo = nominativo;
    this.sesso = sesso;
    this.indirizzo = indirizzo;
  }
```

▶

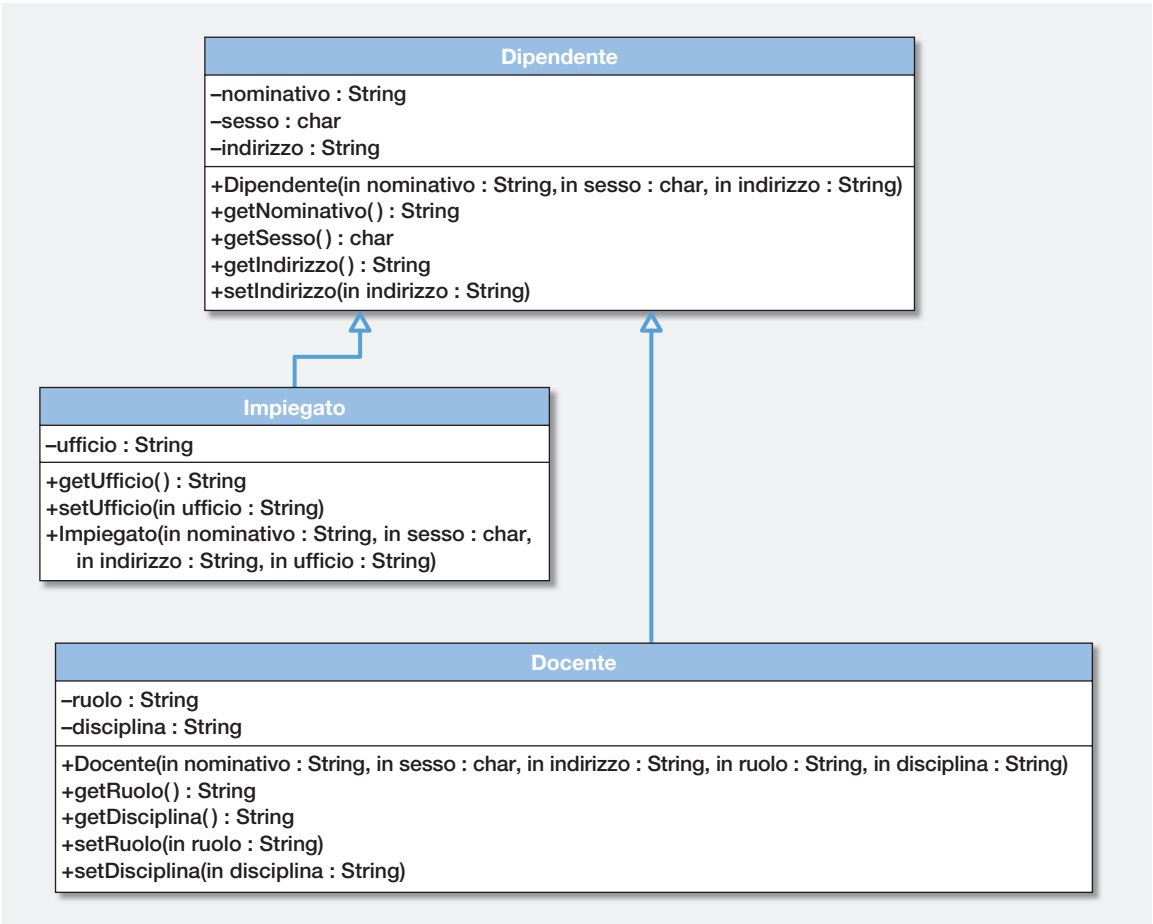



FIGURA **10**

```
  public String getIndirizzo() {
    return indirizzo;
  }

  public void setIndirizzo(String indirizzo) {
    this.indirizzo = indirizzo;
  }

  public String getNominativo() {
    return nominativo;
  }

  public char getSesso() {
    return sesso;
  }
}

public class Impiegato extends Dipendente {
  private String ufficio;

  public Impiegato(String nominativo, char sesso, String indirizzo, String ufficio) {
    super(nominativo, sesso, indirizzo);
    this.ufficio = ufficio;
  }
```

```
  public String getUfficio() {
    return ufficio;
  }

  public void setUfficio(String ufficio) {
    this.ufficio = ufficio;
  }
}

public class Docente extends Dipendente {
  private String ruolo;
  private String disciplina;

  public Docente(String nominativo, char sesso, String indirizzo, String ruolo,
                 String disciplina) {
    super(nominativo, sesso, indirizzo);
    this.ruolo = ruolo;
    this.disciplina = disciplina;
  }

  public String getDisciplina() {
    return disciplina;
  }

  public void setDisciplina(String disciplina) {
    this.disciplina = disciplina;
  }

  public String getRuolo() {
    return ruolo;
  }

  public void setRuolo(String ruolo) {
    this.ruolo = ruolo;
  }
}
```

**OSSERVAZIONE** Il costruttore delle classi derivate (*Impiegato*, *Docente*) deve inizializzare tutti gli attributi di un oggetto istanza della classe: ha quindi un parametro per ogni attributo ereditato dalla superclasse (*Dipendente*) più un parametro per ogni nuovo attributo definito nella sottoclasse. Dato che la parola chiave `super` nel linguaggio Java riferisce la superclasse e che i costruttori hanno per definizione il nome della classe in cui sono definiti, il costruttore della superclasse viene normalmente invocato nel costruttore della sottoclasse mediante la seguente sintassi:

```
super(...);
```

Questa invocazione – se presente – deve essere la prima istruzione del costruttore di una classe derivata.

Riprendendo l'analisi del codice della classe *PuntoOrientato* si può ancora osservare quanto segue.

- I primi due costruttori hanno come prima istruzione l'invocazione del costruttore della superclasse `super(...,...);` a cui sono forniti come parametri i valori delle coordinate che il codice del costruttore della classe *Punto* assegna agli attributi *x* e *y*; il codice dei costruttori provvede di seguito ad assegnare il valore all'attributo *orientamento* invocando il metodo *setOrientamento*.
- Il terzo costruttore è un costruttore di copia e anche esso ricorre all'uso della parola chiave `super` nella prima istruzione per invocare il costruttore della superclasse, ma in questo caso – dovendo copiare i valori degli attributi privati della classe *Punto* ereditati dalla classe *PuntoOrientato* dell'oggetto *po* passato come argomento – utilizza i metodi pubblici *getX* e *getY* per accedervi.

**OSSERVAZIONE**

Anche se ereditati gli attributi di livello `private` non sono mai accessibili dal codice dei metodi di una classe diversa da quella in cui sono dichiarati: come è stato illustrato nel secondo capitolo per risultare accessibili dal codice dei metodi di una classe derivata gli attributi devono essere dichiarati di livello `protected`.

- Nel metodo ridefinito *toString* l'espressione `super.toString()` riferisce il risultato prodotto dall'omonimo metodo della classe *Punto* per ottenere una stringa formattata con i valori numerici delle due coordinate cartesiane (ad esempio `(1.0;1.0)`) a cui concatenare il valore dell'attributo *orientamento* che indica la direzione verso cui un oggetto di tipo *PuntoOrientato* è rivolto (ad esempio `(1.0;1.0)A`).
- Nel metodo *equals* le espressioni `super.getX()` e `super.getY()` riferiscono i valori restituiti dagli omonimi metodi della classe *Punto* per essere confrontati con i valori restituiti dai corrispondenti metodi ereditati dalla classe *PuntoOrientato* applicati all'oggetto *po* fornito come parametro.

**OSSERVAZIONE** In realtà i metodi `getX()` e `getY()` ereditati dalla classe *PuntoOrientato* sono di livello `public` e, non essendo ridefiniti, possono essere invocati dal codice dei metodi di una classe qualsiasi: non è in questo caso necessario premettere la parola chiave `super` (potrebbe infatti essere indifferentemente usata la parola chiave `this`).

L'*overridding* fa in modo che un metodo sovrascritto nasconda nella classe derivata la definizione data nella superclasse per un metodo avente la stessa firma, tuttavia la forma originale risulta accessibile nel codice dei metodi della classe derivata utilizzando la parola chiave `super` mediante la quale risultano invocabili anche i costruttori della superclasse. La possibilità di accedere ai costruttori e ai metodi della classe originale

utilizzando la parola chiave `super` garantisce il rispetto del principio di incapsulamento rispetto alle classi derivate: eventuali modifiche del codice dei metodi della superclasse che non ne modifichino la firma non si rifletteranno sul codice dei metodi della sottoclasse.

## 13.2 Gerarchie di classi: *up-casting* e *down-casting* di oggetti

In una gerarchia di classi derivate è possibile assegnare un oggetto istanza di una classe della gerarchia a un riferimento avente come tipo una qualsiasi superclasse, ma non viceversa.

**ESEMPIO**

Riferendosi alla gerarchia di classi *Dipendente*, *Impiegato* e *Docente* definite nel paragrafo precedente, a un riferimento di tipo *Dipendente* può essere assegnato un oggetto istanza della classe *Impiegato* (o *Docente*), ma a un riferimento di tipo *Docente* non può essere assegnato un oggetto istanza della classe *Dipendente* (o *Impiegato*).

**OSSERVAZIONE** La regola precedente stabilisce semplicemente che l'assegnamento di un oggetto a un riferimento deve garantire la corretta invocazione di tutti i metodi definiti dalla classe del riferimento, cosa che è sempre vera nel caso delle sottoclassi, ma che non lo è nel caso delle superclassi.

**ESEMPIO**

Supponendo di avere un oggetto *po* istanza della classe *PuntoOrientato* un assegnamento come il seguente è corretto:

```
Punto p = po;
```

Infatti l'oggetto *po* riferito da *p* ha tutti i requisiti per poter rispondere ai metodi della classe *Punto* che potrebbero essere invocati. Viceversa disponendo di un oggetto *p* istanza della classe *Punto* il seguente assegnamento

```
PuntoOrientato po = p;
```

risulta errato in quanto l'oggetto riferito da *po* non sarebbe in grado di rispodere a metodi quali *setOrientamento*, *getOrientamento*, *ruotaDestra*, *ruotaSinistra* e *sposta* invocabili su un qualsiasi riferimento di classe *PuntoOrientato*. Il compilatore del linguaggio Java genera in questo caso un errore rilevando l'incompatibilità dei tipi.

Rispettando la regola enunciata in precedenza il linguaggio Java consente di trasformare il tipo degli oggetti mediante operazioni di *casting*. L'operatore di *casting* viene premesso al riferimento dell'oggetto e ha il seguente formato

```
(tipo) oggetto
```

dove *tipo* è il nuovo tipo di *oggetto*.

### Il principio di sostituzione di Liskov e la relazione «*is a*»

La relazione di specializzazione di una classe derivata rispetto alla superclasse da cui deriva è spesso denominata relazione «*is a*» per sottolineare come un'istanza di una classe è anche un'istanza della superclasse.

Nel 1987 Barbara Liskov enunciò il «principio di sostituzione» che si applica ai linguaggi di programmazione che realizzano il polimorfismo e che è oggi noto con il suo nome; esso afferma che un oggetto il cui tipo è un sottotipo di *T* deve poter essere utilizzato in un qualsiasi contesto in cui è legale utilizzare un oggetto di tipo *T*.

La definizione di «sottotipo» data da Liskov è piuttosto complessa, ma coincide con quella di classe derivata della maggior parte dei linguaggi di programmazione OO.

**ESEMPIO**

Il metodo *equals* della classe *PuntoOrientato* potrebbe essere così codificato:

```
public boolean equals(PuntoOrientato po) {
  return (super.equals((Punto)po) &&
          getOrientamento() == po.getOrientamento());
}
```

L'invocazione del metodo *equals* della superclasse *Punto* richiede il passaggio di un argomento di tipo *Punto* cosa che si ottiene effettuando un *casting* dell'oggetto *po* fornito come parametro. Essendo *Punto* una superclasse di *PuntoOrientato* questa operazione è lecita: il metodo confronta gli attributi *x* e *y* che la classe *PuntoOrientato* eredita dalla classe *Punto*.

Nel rispetto della regola che impedisce che un oggetto possa essere assegnato a un riferimento di una propria sottoclasse, le operazioni di *casting* di un riferimento possono essere utilizzate per trasformare il riferimento a un oggetto in un riferimento avente come tipo una superclasse – si tratta in questo caso di *up-casting* – oppure in un riferimento a una sottoclasse – si tratta allora di un *down-casting*.

**ESEMPIO**

Con riferimento alla gerarchia delle classi *Dipendente*, *Impiegato* e *Docente* introdotta nel paragrafo precedente il seguente frammento di codice esemplifica alcune operazioni di *up-casting* e *down-casting*:

```
…
Dipendente dipendente1, dipendente2;
Impiegato impiegato1, impiegato2;
impiegato1 = new Impiegato("Rossi Mario", 'M', "Via del mare,
                           1 - Livorno", "Segreteria");
Docente docente1, docente2;
docente1 = new Docente("Neri Maria", 'F', "Via del monte, 99 -
                       Livorno", "Supplente", "Informatica");
…
dipendente1 = (Dipendente)impiegato1; // up-casting
dipendente2 = (Dipendente)docente1;   // up-casting
impiegato2 = (Impiegato)dipendente1;  // down-casting legale
docente2 = (Docente)dipendente2;      // down-casting legale
impiegato2 = (Impiegato)dipendente2;  // down-casting illegale
docente2 = (Docente)dipendente1;      // down-casting illegale
…
```

Il *casting* dei riferimenti agli oggetti dipende quindi dalle relazioni di ereditarietà che intercorrono tra le classi di una gerarchia.

**OSSERVAZIONE** Qualsiasi riferimento a un oggetto può sempre essere assegnato a un riferimento di tipo *Object* perché la classe *Object* è la superclasse di ogni classe definita nel linguaggio Java.

Quando il *casting* di un riferimento viene effettuato da una sottoclasse a una sua superclasse non è necessario esplicitarlo perché viene sempre effettuato implicitamente dal compilatore del linguaggio Java.

**ESEMPIO**

Il metodo *equals* della classe *PuntoOrientato* può essere in realtà codificato in questo modo:

```
public boolean equals(PuntoOrientato po) {
  return (super.equals(po) &&
          getOrientamento() == po.getOrientamento());
}
```

Il *casting* dell'oggetto *po* al tipo *PuntoOrientato* necessario per l'invocazione del metodo *equals* della classe *Punto* viene effettuato implicitamente.

Più in generale si ha un *casting* implicito quando il riferimento a un oggetto è assegnato a un riferimento il cui tipo:

- è lo stesso della classe di cui l'oggetto è istanza;
- è la classe *Object*;
- è una superclasse della classe di cui l'oggetto è istanza.

Le regole di compilazione non permettono di operare in tutti quei casi in cui il *casting* non è possibile (quando, ad esempio, le classi dei riferimenti non hanno tra loro alcuna relazione gerarchica di ereditarietà), ma in ogni caso in fase di esecuzione del programma (*runtime*) si può verificare un'eccezione di tipo *ClassCastException* se il l'oggetto del *casting* non è compatibile col il tipo del riferimento a cui viene assegnato.

**ESEMPIO**

Riprendendo il precedente esempio relativo alla gerarchia delle classi *Dipendente*, *Impiegato* e *Docente* il seguente frammento di codice esemplifica alcune operazioni di *casting* e il loro esito (per semplicità sono stati omessi i parametri dei costruttori):

```
…
// compilazione corretta: nessun casting
Dipendente dipendente1 = new Dipendente(…);
Docente docente1 = new Docente(…);
Impiegato impiegato1 = new Impiegato(…);
// compilazione corretta: up-casting implicito
Dipendente dipendente2 = new Docente(…);
Dipendente dipendente3 = new Impiegato(…);
Object oggetto1 = docente1;
// compilazione errata: tipi NON compatibili per il casting implicito
Docente docente2 = new Dipendente(…);
Docente docente3 = new Impiegato(…);
Impiegato impiegato2 = new Docente(…);
Impiegato impiegato3 = new Dipendente(…);
```

```
// compilazione corretta con errore in fase di esecuzione:
// casting esplicito di tipi incompatibili
Docente docente4 = (Docente)dipendente1; // è un dipendente
Impiegato impiegato4 = (Impiegato)dipendente1; // è un dipendente
Docente docente5 = (Docente)impiegato1; // è un docente
Impiegato impiegato5 = (Impiegato)docente1; // è un docente
Docente docente6 = (Docente)dipendente3; // è un impiegato
Impiegato impiegato6 = (Impiegato)dipendente2; // è un docente
Object oggetto2 = (Impiegato)oggetto1; // è un docente
// compilazione corretta ed esecuzione priva di errori:
// casting esplicito di tipi compatibili
Docente docente7 = (Docente)dipendente2; // è un docente
Impiegato impiegato7 = (Impiegato)dipendente3; // è un impiegato
…
```

Dalla trattazione svolta risulta chiaro che l'ambiente di esecuzione dei programmi in linguaggio Java deve «ricordare» per ogni singolo oggetto la classe da cui è stato istanziato. Senza anticipare il tema del polimorfismo che è centrale nella programmazione OO, uno dei vantaggi derivanti dall'*up-casting* dei tipi è la possibilità di memorizzare in unica struttura dati – come, ad esempio, un vettore – oggetti eterogenei istanze di classi diverse della stessa gerarchia di ereditarietà.

**ESEMPIO**

Dovendo costruire un elenco di dipendenti includendo sia gli impiegati che i docenti è sufficiente dichiarare un vettore come il seguente

```
Dipendente elenco[] = new Dipendente[100];
```

ai cui elementi è possibile assegnare sia oggetti di tipo *Impiegato* che oggetti di tipo *Docente* (oltre che oggetti di tipo *Dipendente* ovviamente):

```
elenco[0] = new Docente(…);
elenco[1] = new Impiegato(…);
```

**OSSERVAZIONE** Nella situazione dell'esempio precedente è importante poter distinguere il tipo dei singoli elementi del vettore per poter effettuare un corretto *down-casting*: questa problematica prende il nome di RTTI (*Run-Time Type Identification*) ed è trattata nel seguito del capitolo.

Anche se non è una tecnica molto praticata dai programmatori è possibile operare l'*overridding* degli attributi di una classe modificandone eventualmente anche il tipo nella classe derivata; in realtà in questo caso più che di *overridding* si dovrebbe parlare di *hiding* in quanto la classe derivata eredita anche gli eventuali attributi sovrascritti che esistono, ma non risultano direttamente accessibili se non ricorrendo alla parola chiave `super` o utilizzando un riferimento del tipo della superclasse (eventualmente ottenuto mediante un *casting*).

**ESEMPIO**

L'output prodotto dal metodo *main* della classe *Test* esemplifica la situazione di *hiding* dell'attributo *val*:

```
public class Uno {
   public String val = "uno";

   public void show() {
      System.out.println(val);
   }
}

public class Due extends Uno {
   public int val = 2;

   public void show() {
      System.out.println(val);
   }

   public void superShow() {
      System.out.println(super.val);
      super.show();
   }
}

public class Test {
   public static void main (String[] args) {
      Due due = new Due();
      Uno uno = due;

      uno.show();
      due.show();

      System.out.println(uno.val);
      System.out.println(due.val);
      System.out.println(((Uno)due).val);
      due.superShow();
   }
}
```

```
2
2
uno
2
uno
uno
uno
```

**OSSERVAZIONE** Il *casting* esplicito nell'istruzione `((Uno)due).val` prevede l'accesso all'attributo *val* dell'oggetto *due* riferito come oggetto di classe *Uno*: l'attributo selezionato in questo caso è *val* della classe *Uno* che in ogni caso l'oggetto *due* eredita anche se nascosto dallo *hiding* operato su tale attributo.

## 13.3 La classe *Object* e l'*overriding* del metodo *clone*

La classe *Object* definisce – oltre ai metodi *equals* e *toString* che sono stati illustrati in precedenza – il metodo *clone* la cui firma è

```
protected Object clone() throws CloneNotSupportedException {…}
```

e che consente di effettuare copie di oggetti con una modalità più adatta al contesto dell'ereditarietà rispetto alla tecnica, precedentemente introdotta col costruttore di copia.

**OSSERVAZIONE** Si ricordi che il costruttore di copia era stato introdotto perché l'operatore di assegnamento «=» del linguaggio Java non effettua una copia dell'oggetto, ma una duplicazione del riferimento allo stesso oggetto.

Due distinti oggetti istanziati in un programma in linguaggio Java possono essere considerati uguali in base a due diversi criteri (coincidenti nel caso di oggetti istanze di classi che abbiano esclusivamente attributi di tipo primitivo):

1) se i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono lo stesso oggetto se sono riferimenti;
2) se i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono oggetti che sono ricorsivamente uguali se sono riferimenti.

Dato che un'operazione di copia ha come scopo la creazione di un nuovo oggetto «uguale» a un oggetto preesistente esistono di conseguenza due diverse tipologie di copia:

- ***shallow copy*** (copia superficiale). Viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 1: gli attributi del nuovo oggetto creato hanno lo stesso valore dell'oggetto originale, sia che si tratti di tipi primitivi che di riferimenti;
- ***deep copy*** (copia profonda). Viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 2: gli attributi del nuovo oggetto, se non sono di tipo primitivo nel qual caso assumono lo stesso valore, sono ricorsivamente copie profonde degli attributi corrispondenti dell'oggetto originale.

**OSSERVAZIONE** La *deep copy* impone di copiare ogni eventuale oggetto riferito da un attributo e ogni singolo elemento degli array: la sua applicazione ricorsiva comporta che clonare in profondità un oggetto significa clonare tutti gli oggetti raggiungibili a partire dal suo riferimento iniziale. Avendo a che fare con oggetti complessi si tratta potenzialmente di una tecnica poco efficiente.

Dal momento che i due tipi di copia coincidono nel caso di oggetti in cui tutti gli attributi sono di tipo primitivo, se si desidera disporre di un meccanismo di copia in cui l'oggetto creato risulti completamente

indipendente in tutte le sue componenti dall'oggetto originale (senza cioè riferimenti comuni) e dato che il metodo *clone* ereditato dalla classe *Object* implementa una *shallow copy* è necessario realizzare la *deep copy* ridefinendo il metodo *clone* stesso.

**ESEMPIO**

Il seguente metodo *main* effettua una *shallow copy* (in questo specifico caso equivalente a una *deep copy* essendo tutti gli attributi di tipo primitivo) di un oggetto di tipo *PuntoOrientato* invocando il metodo *clone* della classe *Object*:

```
public static void main(String args[])
                    throws CloneNotSupportedException {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato(2, 3, 'B');
  po2 = (PuntoOrientato)po1.clone();
}
```

**OSSERVAZIONE** Nel codice dell'esempio precedente:

- è dichiarata la possibile generazione di un'eccezione di classe *CloneNotSupportedException* eventualmente sollevata dall'invocazione del metodo *clone*;
- viene effettuato il *casting* al tipo *PuntoOrientato* del risultato dell'invocazione del metodo *clone* che restituisce un oggetto di tipo *Object*.

Entrambi questi aspetti possono essere evitati ridefinendo il metodo *clone* nella classe *PuntoOrientato*:

```
public PuntoOrientato clone() {
  PuntoOrientato po = null;

  try {
    po = new PuntoOrientato(this.x, this.y,
                            this.orientamento);
  }
  catch (EccezionePuntoOrientato eccezione) {
  }

  return po;
}

public static void main(String args[]) {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato(2, 3, 'B');
  po2 = po1.clone();
}
```

Si noti che l'eccezione catturata e non gestita nel codice del metodo *clone* non viene mai generata in quanto il valore dell'attributo *orientamento* fornito come parametro al costruttore della classe *PuntoOrientato* assume sempre un valore corretto.

**OSSERVAZIONE** Utilizzando il costruttore di copia della classe *PuntoOrientato* il metodo *main* dell'esempio precedente sarebbe stato il seguente:

```
public static void main(String args[]) {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato (2, 3, 'B');
  po2 = new PuntoOrientato (po1);
}
```

In questo caso l'uso del costruttore di copia è più pratico del ricorso al metodo *clone*, ma come avremo modo di scoprire in alcune situazioni ricorrenti il costruttore di copia non può essere invocato.

Affinché gli oggetti istanza di una determinata classe siano clonabili è necessario che:

- la classe implementi l'interfaccia *Cloneable* del package *java.lang*;
- tutti gli oggetti riferiti dagli attributi della classe dell'oggetto da clonare devono essere istanze di classi che implementano a loro volta l'interfaccia *Cloneable*.

**ESEMPIO**

La classe *PuntoOrientato* degli esempi precedenti deve essere così dichiarata:

```
public class PuntoOrientato extends Punto implements Cloneable {…}
```

Poiché che il metodo *clone* della classe *Object* è definito di livello `protected` esso può essere invocato solo dai metodi delle classi dello stesso package e delle classi direttamente derivate. Per questo motivo dovrebbe essere sempre sovrascritto da un metodo di livello `public`.

**ESEMPIO**

Facendo riferimento alla classe *Mensola* introdotta nel secondo capitolo, volendo clonare un oggetto di tipo *Libro* nel codice del metodo *main* il compilatore impedisce l'invocazione del metodo *clone* ereditato dalla classe *Object*. La soluzione è quella di ridefinire il metodo nella classe *Libro*:

```
public Libro clone() throws CloneNotSupportedException {
  Libro l = null;
  // invocazione metodo clone della classe Object
  l = (Libro)super.clone();
  return l;
}
```

Perché non sia generata un'eccezione di tipo *CloneNotSupportedException* la dichiarazione della classe *Libro* deve essere la seguente:

```
public class Libro implements Cloneable {…}
```

**OSSERVAZIONE** Nel codice dell'esempio precedente l'uso della parola chiave `super` è di importanza fondamentale perché specifica l'invocazione del metodo *clone* della classe *Object*; in assenza della parola chiave `super` sarebbe invocato in modo infinitamente ricorsivo il metodo *clone* della classe *Libro*. Il *casting* del risultato al tipo *Libro* è reso necessario dal fatto che il metodo *clone* restituisce un riferimento di tipo *Object*.

**ESEMPIO**

Sempre facendo riferimento alla classe *Mensola* introdotta nel secondo capitolo la *deep copy* degli oggetti istanza della classe richiede, oltre che l'implementazione dell'interfaccia *Cloneable*, la definizione del seguente metodo di clonazione il cui codice scorre l'intero vettore copiandone – mediante invocazione del metodo *clone* della classe *Libro* – i singoli elementi non nulli:

```
public Mensola clone() throws CloneNotSupportedException {
  int posizione;
  Mensola mensola = new Mensola();

  for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
    if (volumi[posizione] != null)
      mensola.setVolume(volumi[posizione].clone(), posizione);
  return mensola;
}
```

**OSSERVAZIONE** Il meccanismo di serializzazione degli oggetti consente di trasformare le strutture complesse degli oggetti in forma di flussi di byte e di crearne copie equivalenti. La serializzazione costituisce quindi un ulteriore modo di ottenere copie di oggetti in Java.

# 14 Classi astratte e interfacce

Ritornando sull'esempio della gerarchia di classi *Dipendente*, *Impiegato* e *Docente* introdotta nei paragrafi precedenti si osserva che probabilmente non vi è nessun interesse a creare oggetti di classe *Dipendente*, ma solo a istanziare impiegati e docenti che sono anche dipendenti (e da questo punto di vista possono essere trattati in modo uniforme).

La parola chiave `abstract` del linguaggio Java premessa alla dichiarazione di una classe fa in modo che non possano essere istanziati oggetti aventi come tipo la classe stessa: è evidente che una classe astratta viene realizzata all'unico scopo di poter derivare da essa una o più classi «concrete». Il diagramma UML di una classe astratta riporta il nome della classe in *carattere corsivo*.

**ESEMPIO**

La classe *Dipendente* dovrebbe essere così dichiarata:

```
abstract public class Dipendente {…}
```

In questo modo dichiarazioni come la seguente risultano illegali e generano un errore a tempo di compilazione:

```
…
Dipendente dipendente = new Dipendente("Rossi Mario", 'M', "Via del mare, 1");
…
```

**OSSERVAZIONE** A partire da una classe astratta non si possono istanziare oggetti, ma è possibile dichiarare riferimenti a cui assegnare oggetti istanza di classi derivate, come nel seguente frammento di codice:

```
…
Dipendente dipendente1, dipendente2;
dipendente1 = new Impiegato("Rossi Mario", 'M',
                            "Via del mare, 1 - Livorno",
                            "Segreteria");
dipendente2 = new Docente("Neri Maria", 'F',
                          "Via del monte, 99 - Livorno",
                          "Supplente", "Informatica");
…
```

## 14.1 Classi astratte e metodi astratti

Spesso lo scopo di una gerarchia di classi è fattorizzare attributi comuni a più classi al livello più alto della gerarchia in modo tale che siano ereditati dalle classi di livello più basso. In questo contesto le classi astratte consentono di usare questo meccanismo di fattorizzazione degli attributi comuni anche nel caso in cui la superclasse risultante non sia «ben definita».
L'uso della classi astratte nel contesto del linguaggio di programmazione Java è disciplinato dalle seguenti regole fondamentali:

- una classe viene definita astratta premettendo alla sua dichiarazione il qualificatore `abstract`;
- una classe astratta può avere costruttori, ma **non** può essere istanziata;
- una classe astratta può avere uno o più «metodi astratti», metodi cioè privi del corpo e qualificati con la parola chiave `abstract` la cui implementazione è vincolante per le classi derivate;
- la dichiarazione di uno o più metodi astratti rende implicitamente la classe astratta;
- se una classe concreta deriva da una classe astratta **deve** fornire una implementazione per tutti i metodi astratti ereditati;
- le classi astratte possono essere superclassi o sottoclassi di altre classi, sia astratte che non.

**ESEMPIO**

Le seguenti classi *Sfera* e *Cubo* hanno alcuni attributi e metodi comuni, anche se il codice dei metodi comuni è diverso:

```
public class Sfera {
  private double raggio;
  private double pesoSpecifico;

  public Sfera(double raggio, double pesoSpecifico) {
    this.raggio = raggio;
    this.pesoSpecifico = pesoSpecifico;
  }

  public double getRaggio() {
    return raggio;
  }

  public double getPesoSpecifico() {
    return pesoSpecifico;
  }

  public double volume() {
    return 4/3 * Math.PI * Math.pow(raggio,3);
  }

  public double superficie() {
    return 4 * Math.PI * Math.pow(raggio,2);
  }

  public double peso() {
    return pesoSpecifico * volume();
  }
}

public class Cubo {
  private double lato;
  private double pesoSpecifico;

  public Cubo(double lato, double pesoSpecifico) {
    this.lato = lato;
    this.pesoSpecifico = pesoSpecifico;
  }

  public double getLato() {
    return lato;
  }

  public double getPesoSpecifico() {
    return pesoSpecifico;
  }

  public double volume() {
    return Math.pow(lato,3);
  }

  public double superficie() {
    return 6* Math.pow(lato,2);
  }
```

▶

```
  public peso() {
    return pesoSpecifico * volume();
  }
}
```

La classe astratta *Solido* fattorizza attributi e metodi comuni alle classi *Sfera* e *Cubo* (FIGURA **11**).

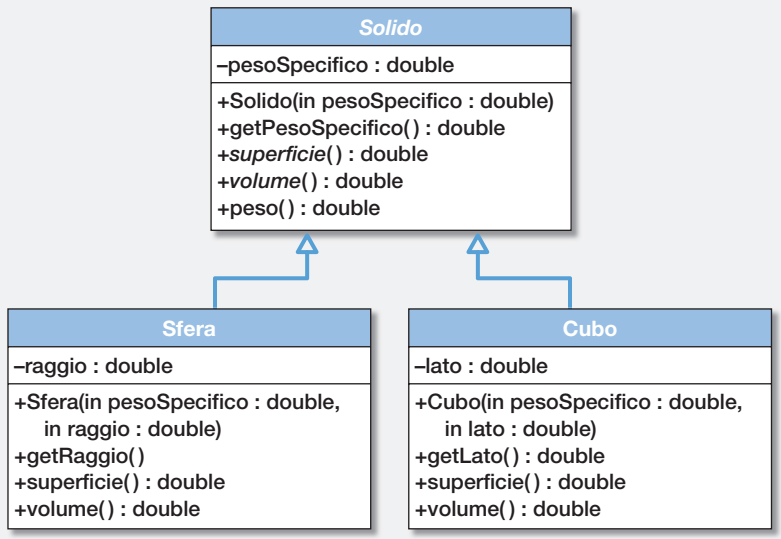



FIGURA **11**

```
abstract public class Solido {
  private double pesoSpecifico;

  public Solido(double pesoSpecifico) {
    this.pesoSpecifico = pesoSpecifico;
  }

  public double getPesoSpecifico() {
    return pesoSpecifico;
  }

  abstract public double volume();

  abstract public double superficie();

  public double peso() {
    return pesoSpecifico * volume();
  }
}

public class Sfera extends Solido {
  private double raggio;

  public Sfera(double raggio, double pesoSpecifico) {
    super(pesoSpecifico);
    this.raggio = raggio;
  }
```

```
    public double getRaggio() {
      return raggio;
    }

    public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
    }

    public double superficie() {
      return 4 * Math.PI * Math.pow(raggio,2);
    }
}

public class Cubo extends Solido {
    private double lato;

    public Cubo(double lato, double pesoSpecifico) {
      super(pesoSpecifico);
      this.lato = lato;
    }

    public double getLato() {
      return lato;
    }

    public double volume() {
      return Math.pow(lato,3);
    }

    public double superficie() {
      return 6* Math.pow(lato,2);
    }
}
```

**OSSERVAZIONE** I metodi *superficie* e *volume* hanno la stessa firma in entrambe le classi *Sfera* e *Cubo*, ma implementazioni diverse: la loro firma è stata fattorizzata nella superclasse *Solido* priva del corpo del metodo premettendo la parola chiave `abstract` e vincolando in questo modo la loro definizione nelle classi derivate.

## 14.2 Interfacce

Nel linguaggio di programmazione Java la parola chiave `interface` consente di dichiarare una «interfaccia» che è analoga a una classe, ma a differenza di questa costituisce una pura specifica di invocazione e come tale si limita a dichiarare i metodi, definendone la firma, senza implementarli (tutti i metodi di un'interfaccia sono implicitamente astratti).

Inoltre, a differenza di una classe astratta, in un'interfaccia è possibile definire attributi costanti, ma non variabili. Nella sua forma più comune un'interfaccia è un elenco di dichiarazioni di metodi privi del corpo.

Una interfaccia viene «implementata» dalle classi mediante l'uso della parola chiave `implements`.

**OSSERVAZIONE** Il linguaggio di programmazione Java definisce delle interfacce vuote (come, ad esempio, *Serializable* e *Cloneable*) al solo scopo di essere utilizzate come «marcatori» per le classi che implementano determinate funzionalità.

Implementare un'interfaccia impone formalmente a una classe di definire i metodi che l'interfaccia stessa dichiara: in questo senso le interfacce costituiscono una specie di «contratto» per le classi che le implementano e il rispetto di tale contratto viene verificato dal compilatore che genera un errore nel caso in cui una classe che dichiara di implementare una determinata interfaccia non ne definisca tutti i metodi dichiarati.

**ESEMPIO**

Con riferimento al precedente esempio relativo alle classi *Solido*, *Sfera* e *Cubo* è ragionevole pensare che la classe astratta *Solido* che integra le caratteristiche di una figura solida geometrica e quelle di un corpo solido fisico implementi le seguenti interfacce:

```
public interface FiguraSolida {
  public double superficie();
  public double volume();
}

public interface CorpoSolido {
  public double peso();
}

abstract public class Solido implements FiguraGeometrica,
                                CorpoSolido {
  private double pesoSpecifico;

  public Solido(double pesoSpecifico) {
    this.pesoSpecifico = pesoSpecifico;
  }

  public double getPesoSpecifico() {
    return pesoSpecifico;
  }

  abstract public double volume();

  abstract public double superficie();

  public double peso() {
    return pesoSpecifico * volume();
  }
}
```

▶

Il diagramma UML delle classi assume l'aspetto di FIGURA **12**, in cui le interfacce sono caratterizzate dal qualificatore `<<interface>>`:

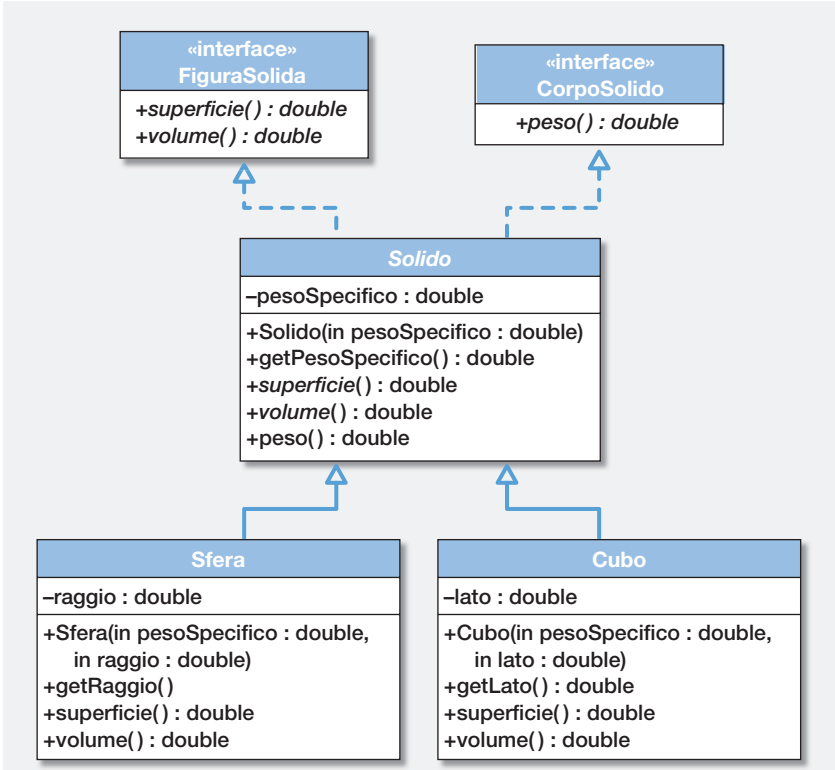



FIGURA **12**

**OSSERVAZIONE**

Nell'esempio precedente si noti come la classe astratta *Solido* implementi due distinte interfacce: il linguaggio Java consente l'implementazione multipla di interfacce anche se proibisce l'eredità multipla di classi.

**ESEMPIO**

Una classe può implementare più interfacce integrando schemi di invocazione diversi; la seguente dichiarazione della classe *Libro*

```
public class Libro implements Serializable, Cloneable {
…
}
```

permette di dotare la stessa sia della possibilità di essere clonabile, sia di quella di essere serializzabile.

Le interfacce possono essere estese per ereditarietà ed è quindi possibile costruire gerarchie di interfacce.

# 15 Polimorfismo e *binding* dinamico

L'operazione di estensione di una classe permette di derivare da essa una o più sottoclassi: è sempre possibile utilizzare un oggetto istanza di una delle sottoclassi derivate in un contesto in cui sia richiesto un oggetto istanza della classe originale. Questa caratteristica dei linguaggi di programmazione OO in generale e del linguaggio Java in particolare viene detta **polimorfismo**, intendendo con questo termine la capacità di un oggetto di assumere «forme» molteplici: istanza della propria classe o istanza di ogni superclasse di essa.

**OSSERVAZIONE** Il polimorfismo degli oggetti risulta particolarmente utile quando in una gerarchia di classi derivate viene ridefinito un metodo e la specifica versione del metodo da eseguire viene scelta automaticamente sulla base del tipo di oggetto effettivamente assegnato a un riferimento in fase di esecuzione piuttosto che al momento della compilazione.

**ESEMPIO**

Nella classe *Punto* introdotta nei paragrafi precedenti è definito il metodo *toString* che viene ridefinito nella classe *PuntoOrientato* sua derivata. Nel codice che segue:

```
…
Punto tmp;

if (Math.random() > 0.5)
   tmp = new Punto(1.0, 0.0);
else
   tmp = new PuntoOrientato(1.0, 0.0, 'A');

tmp.toString();
…
```

Dato che l'invocazione del metodo statico random della classe *Math* genera un numero casuale compreso tra 0 e 1 il compilatore non può determinare se al riferimento *tmp* sarà assegnata un'istanza della classe *Punto* o della classe *PuntoOrientato*. L'ambiente di esecuzione dei programmi Java dovrà determinare il metodo *toString* corretto da invocare (quello definito dalla classe *Punto* piuttosto che quello ridefinito dalla classe *PuntoOrientato*) in fase di esecuzione in base al tipo effettivo di oggetto assegnato al riferimento.

I metodi ridefiniti in una classe derivata sono detti «polimorfi», in quanto lo stesso metodo ha comportamenti diversificati a seconda del tipo di oggetto su cui è invocato.

Il meccanismo che determina quale metodo deve essere invocato in base alla classe di appartenenza dell'oggetto è definito *binding* (letteralmente «legame», «collegamento»). Esistono due tipi fondamentali di *binding*:

### La classificazione del polimorfismo

Nel 1985 Luca Cardelli e Peter Wegner classificarono le forme di polimorfismo di un linguaggio di programmazione in due categorie: «universale» e «ad hoc».
Nella prima categoria rientrano il polimorfismo parametrico, in cui funzioni e operatori sono parametrizzati in base al tipo a cui possono essere applicati (Java realizza questa forma di polimorfismo con le classi «generiche»), e il polimorfismo per inclusione che è esattamente quello implementato dai linguaggi OO: lo stesso metodo può essere applicato a tutti gli oggetti «inclusi» nella classe che lo definisce (cioè istanze di classi derivate).
Nella seconda categoria rientra lo *overloading* delle funzioni e dei metodi in base al tipo dei parametri e del risultato e la cosiddetta *coercion* realizzata dal *casting* implicito dei tipi degli operandi di un operatore, o dei parametri di una funzione o di un metodo.

- ***binding* statico** (*early binding*): il metodo da invocare viene determinato in fase di compilazione del codice (*compile-time binding*);
- ***binding* dinamico** (*late binding*): il metodo da invocare viene determinato durante l'esecuzione del codice (*run-time binding*).

Il linguaggio Java, come la maggior parte dei linguaggi di programmazione a oggetti, adotta il *binding* dinamico a *run-time* quando il *binding* statico a *compile-time* risulta impossibile, come nel caso dei metodi polimorfici.

**ESEMPIO**

Con riferimento alla gerarchia di classi *Solido*, *Sfera* e *Cubo* il seguente metodo *main* di un'ipotetica classe di test

```
public static void main(String[] args) {
  Sfera sfera = new Sfera(1.0,0.5);
  Cubo cubo = new Cubo(3.0,1.1);

  System.out.println("Sfera sfera---> Volume….: " + sfera.volume());
  System.out.println("Sfera sfera---> Peso……: " + sfera.peso());
  System.out.println("Cubo cubo----> Volume….: " + cubo.volume());
  System.out.println("Cubo cubo----> Peso……: " + cubo.peso());
  Solido solido1 = sfera;
  System.out.println("Solido solido1-> Volume….: " + solido1.volume());
  System.out.println("Solido solido1-> Peso……: " + solido1.volume());
  Solido solido2 = cubo;
  System.out.println("Solido solido2-> Volume….: " + solido2.volume());
  System.out.println("Solido solido2-> Peso……: " + solido2.volume());
}
```

produce il seguente output:

```
Sfera sfera---> Volume….: 3.141592653589793
Sfera sfera---> Peso……: 1.5707963267948966
Cubo cubo----> Volume….: 27.0
Cubo cubo----> Peso……: 29.700000000000003
Solido solido1-> Volume….: 3.141592653589793
Solido solido1-> Peso……: 1.5707963267948966
Solido solido2-> Volume….: 27.0
Solido solido2-> Peso……: 29.700000000000003
```

Le prime quattro righe visualizzate sono relative all'invocazione dei metodi della classi *Sfera* e *Cubo* che non sono polimorfici: il metodo *peso* è ereditato dalla classe astratta *Solido* senza essere ridefinito, mentre i metodi *volume* sono implementazioni di un metodo astratto della classe *Solido*. Le ultime quattro righe visualizzate sono il risultato di un classico esempio di invocazione di metodi polimorfici: ai riferimenti *solido1* e *solido2* sono infatti assegnati rispettivamente gli oggetti *sfera* e *cubo* istanze di classi diverse. L'invocazione del metodo *volume* comporta il *binding* dinamico a *run-time* che esegue il codice del metodo definito nella classe *Sfera* nel primo caso e nella classe *Cubo* nel secondo caso; l'invocazione del metodo *peso* invece esegue il codice dell'unica definizione del metodo data nella classe astratta *Solido* da cui *Sfera* e *Cubo* derivano.

Il polimorfismo consente di sviluppare codice a oggetti «estensibile»: l'aggiunta di nuove classi a una gerarchia preesistente non influenza il codice che gestisce oggetti istanza delle classi della gerarchia stessa in modo polimorfico.

**ESEMPIO**

L'aggiunta della seguente classe *Cilindro* alla gerarchia delle classi che ereditano dalla classe astratta *Solido*

```
public class Cilindro extends Solido {
  private double raggio;
  private double altezza;

  public Cilindro(double raggio, double altezza, double pesoSpecifico) {
    super(pesoSpecifico);
    this.raggio = raggio;
    this.altezza = altezza;
  }

  public double getRaggio() {
    return raggio;
  }

  public double getAltezza() {
    return altezza;
  }

  public double volume() {
    return Math.PI * Math.pow(raggio,2) * altezza;
  }

  public double superficie() {
    return Math.PI * Math.pow(raggio,2);
  }
}
```

non comporta alcuna variazione nel seguente frammento di codice che calcola la media dei volumi e dei pesi di tutti i solidi contenuti in un vettore:

```
…
Solido solidi[] = new Solido[…];
double volumeMedio = 0.0;
double pesoMedio = 0.0;
…
for (int i=0; i<solidi.length; i++) {
  volumeMedio += solidi[i].volume();
  pesoMedio += solidi[i].peso();
}
volumeMedio /= solidi.length;
pesoMedio /= solidi.length;
…
```

**OSSERVAZIONE** In assenza di polimorfismo il codice dell'esempio precedente avrebbe dovuto prevedere nel corpo del ciclo un'istruzione di selezione della classe con relativa differenziazione dell'invocazione del metodo *volume*:

```
…
for (int i=0; i<solidi.length; i++) {
  if (solidi[i] instanceof Sfera)
    volumeMedio += (Sfera)solidi[i].volume();
```

```
  if (solidi[i] instanceof Cubo)
    volumeMedio += (Cubo)solidi[i].volume();
  if (solidi[i] instanceof Cilindro)
    volumeMedio += (Cilindro)solidi[i].volume();
  pesoMedio += solidi[i].peso();
}
volumeMedio /= solidi.length;
pesoMedio /= solidi.length;
…
```

L'operatore Java **instanceof**, che sarà introdotto nel prossimo paragrafo, consente di verificare la classe di cui un oggetto è istanza. È evidente che senza polimorfismo il codice che invoca il metodo *volume* dovrebbe essere modificato per ogni nuova classe aggiunta alla gerarchia.

Il polimorfismo – nella forma in cui è concretizzato dal linguaggio Java – realizza un equilibrio fra due opposte esigenze:

- il rigido controllo statico dei tipi degli oggetti che consente di riscontrare il maggior numero possibile di errori di programmazione in fase di compilazione;
- il rilassamento della regola che vuole che ogni oggetto abbia un tipo predefinito in fase di compilazione allo scopo di garantire al programmatore una flessibilità adeguata.

**OSSERVAZIONE** Alcuni linguaggi – come, ad esempio, il linguaggio JavaScript – eliminano alla radice il problema non imponendo che le variabili abbiano un tipo e non effettuandone di conseguenza il controllo statico; in questo caso tutti i controlli sono effettuati a tempo di esecuzione: lo svantaggio di questa soluzione è che eventuali errori dovuti a tipi incompatibili possono comparire anche molto tempo dopo che un programma è stato rilasciato.

Le problematiche derivanti dalla necessità di duplicare gli oggetti nella memoria heap senza avere riferimenti multipli allo stesso oggetto sono ovviamente presenti anche negli oggetti polimorfici; inoltre nel codice di metodi che restituiscono o ricevono come parametri oggetti polimorfici non è possibile invocare il costruttore di copia perché non è nota a priori la classe di cui un oggetto è istanza. La soluzione più adatta per ovviare a questo problema è quella di definire in tutte le classi di una gerarchia uno specifico metodo clone da invocare al posto del costruttore di copia: trattandosi di un metodo polimorfico sarà automaticamente selezionato il metodo definito nella classe di cui l'oggetto su cui lo si applica è istanza anche se il riferimento assume il tipo di una superclasse, eventualmente astratta.

# 16 Run-Time Type Identification e operatore *instanceof*

L'esigenza di meccanismi di *Run-Time Type Identification* (RTTI), ovvero di supporto all'identificazione del tipo effettivo di un oggetto durante l'esecuzione di un programma, nei linguaggi OO in generale e in Java in particolare nasce dall'impossibilità di invocare direttamente il metodo di una sottoclasse su un riferimento una superclasse, cosa che può risultare utile o anche necessaria. Per applicare propriamente il *casting* necessario in questa situazione è indispensabile conoscere il tipo effettivo di cui l'oggetto riferito è istanza:

l'operatore Java `instanceof` permette di verificare la classe di appartenenza di un oggetto.

**ESEMPIO**

Facendo riferimento alla gerarchia di classi *Dipendente*, *Impiegato* e *Docente* può essere necessario invocare il metodo *setUfficio* per un oggetto di classe *Impiegato* assegnato a un riferimento di tipo *Dipendente* mediante un *casting*:

```
…
((Impiegato)dipendente).setUfficio("Personale");
…
```

Naturalmente nel caso in cui l'oggetto assegnato al riferimento *dipendente* non sia un istanza della classe *Impiegato* viene generata un'eccezione; per evitare l'errore si può condizionare l'istruzione a una verifica della classe di appartenenza dell'oggetto assegnato al riferimento:

```
…
if (dipendente instanceof Impiegato)
   ((Impiegato)dipendente).setUfficio("Personale");
…
```

L'espressione logica

*oggetto* **`instanceof`** *Classe*

viene valutata `true` se e solo se *oggetto* è un'istanza di *Classe*: è richiesto che *Classe* sia staticamente il nome del tipo, non è possibile utilizzare riferimenti variabili.

**OSSERVAZIONE**

Una buona pratica di programmazione OO utilizza quanto più possibile il polimorfismo limitando allo stretto indispensabile il ricorso all'operatore `instanceof`.

### La «riflessione» in Java

Il linguaggio Java consente ai programmi di creare, ispezionare e gestire le classi a tempo di esecuzione: questa caratteristica del linguaggio è nota con il nome di «riflessione». In particolare la classe *Class* – le cui possibili istanze sono le classi che costituiscono il programma in esecuzione – rende disponibili metodi per verificare dinamicamente la classe di un oggetto; infatti *Class* eredita da *Object* il metodo *getClass* che restituisce la classe di un oggetto. Il seguente frammento di codice ad esempio visualizza il nome della classe dell'oggetto che lo esegue:

```
…
Class c =
this.getClass();
System.out.println
(c.getName());
…
```

Il metodo *getSuperclass* restituisce il tipo della superclasse; il seguente frammento di codice visualizza il nome della superclasse della classe di appartenenza dell'oggetto che lo esegue:

```
…
Class c =
this.getSuperclass();
System.out.println
(c.getName());
…
```

La classe *Class* definisce inoltre il metodo *isInstance* che consente di verificare dinamicamente a *run-time* la classe di appartenenza di un oggetto.

# 17 Gerarchie di eccezioni e loro gestione

Le eccezioni nel linguaggio Java sono classi che derivano dalla classe *Exception*: è quindi possibile ricorrere all'ereditarietà per costruire gerarchie di eccezioni e sfruttare il polimorfismo per una loro più agevole gestione.

ESEMPIO

Nel paragrafo 11.2 è stata introdotta la classe *TextFile* per la scrittura/lettura sequenziale di file testuali: i metodi della classe generano eccezioni di classe *FileException* fornendo come parametro al costruttore una stringa di descrizione dell'errore che può essere in seguito recuperata mediante un metodo specifico della classe-eccezione. Un modo più razionale consiste nel definire la seguente gerarchia di eccezioni

```
abstract public class FileException extends Exception {
}

public class ReadOnlyFile extends FileException {
  public String toString() {
    return "Read-only file!";
  }
}

public class WriteOnlyFile extends FileException {
  public String toString() {
    return "Write-only file!";
  }
}

public class EndOfFile extends FileException {
  public String toString() {
    return "End of file!";
  }
}
```

che può essere gestita in modo polimorfico indicando nella clausola **throws** dei metodi la sola classe astratta *FileException* da cui derivano le eccezioni concretamente generate

```
// Classe per la gestione sequenziale di un file di testo
public class TextFile {
  private char mode; // R=read, W=write
  private BufferedReader reader;
  private BufferedWriter writer;

  // Costruisce un oggetto di tipo BufferedReader/BufferedWriter
  // sopra il file specificato dal nome indicato
  public TextFile(String filename, char mode) throws IOException
  {
    this.mode = 'R';
    if (mode == 'W' || mode == 'w') {
      this.mode = 'W';
      writer = new BufferedWriter(new FileWriter(filename));
    }
    else {
      reader = new BufferedReader(new FileReader(filename));
    }
  }

  // Scrive una riga di testo in un file aperto in scrittura
  public void toFile(String line) throws FileException, IOException
```

▶

```
    {
      if (this.mode == 'R')
        throw new ReadOnlyFile();
      writer.write(line);
      writer.newLine();
    }

    // Legge una riga di testo da un file aperto in lettura
    public String fromFile() throws FileException, IOException
    {
      String tmp;

      if (this.mode == 'W')
        throw new WriteOnlyFile();
      tmp = reader.readLine();
      if (tmp == null)
        throw new EndOfFile();
      return tmp;
    }

    // Chiude il file aperto dal costruttore
    public void closeFile() throws IOException
    {
      if (this.mode == 'W')
        writer.close();
      else
        reader.close();
    }
}
```

Come è illustrato dal seguente metodo *main* di test della classe *TextFile* è in questo caso sufficiente una sola clausola **catch** del blocco **try** per intercettare tutte le eccezioni che ereditano dalla classe *FileException*, il polimorfismo ne consente la corretta gestione (in questo caso limitata alla visualizzazione della tipologia di errore):

```
public static void main(String args[]) throws IOException
{
  TextFile out = new TextFile("file.txt",'W');
  try {
    out.toFile("Riga 1");
    out.toFile("Riga 2");
    out.toFile("Riga 3");
    String tmp = out.formFile();
  }
  catch (FileException exception) {
      System.out.println(exception);
    }
  out.closeFile();

  TextFile in = new TextFile("file.txt",'R');
  try {
    while (true) {
      String line = in.fromFile();
      System.out.println(line);
    }
  }
  catch (FileException exception) {
    System.out.println(exception);
  }
  out.closeFile();
}
```

**OSSERVAZIONE** Come caso limite si ha che una clausola `catch` con parametro di tipo *Exception* intercetta tutte le eccezioni generate dal codice inserito nel blocco `try`.

Un aspetto non secondario del polimorfismo del linguaggio Java è dato dalla regola relativa alle eccezioni dei metodi ridefiniti in una classe derivata: un metodo che ridefinisce un metodo che dichiara di generare eccezioni non può sollevare tipi di eccezione diversi o aggiuntivi rispetto a quelli del metodo originale. Questa regola assicura che tutte le eccezioni generate siano sempre gestite.

Se, ad esempio, il metodo originale genera eccezioni di classe *Eccezione*, i corrispondente metodo ridefinito non può sollevare eccezioni di una superclasse di *Eccezione*, o di una classe diversa da *Eccezione*, ma può:

- generare eccezioni di classe *Eccezione*;
- generare eccezioni di una sottoclasse di *Eccezione*;
- non generare eccezioni.

Se non esistesse la limitazione esposta si avrebbero situazioni incoerenti per quanto riguarda la gestione delle eccezioni, come esemplificato nel codice che segue:

```
public class Alfa {
  public void esegui() throws Eccezione {
    …
    …
    …
  }
}

public class Beta {
  public void metodo(Alfa a) {
    try {
      a.esegui();
    }
    catch (Eccezione e) {
      …
    }
  }
}
```

Il polimorfismo consente di invocare il metodo *metodo* della classe *Beta* fornendo come parametro un argomento che è una sottoclasse di *Alfa* e, come conseguenza dell'*overriding* e del *binding* dinamico non è certo che il metodo *esegui* invocato dal codice *metodo* sia quello definito nella classe *Alfa*: potrebbe essere stato ridefinito nella sottoclasse, ma la regola enun-

ciata impedisce che esso sollevi eccezioni potenzialmente non previste. Tale situazione potrebbe infatti verificarsi solo se il metodo ridefinito sollevasse eccezioni che non sono istanze della classe *Eccezione*, o di una sua sottoclasse.
Un diverso esempio di uso improprio e di conseguenza illegale delle eccezioni del linguaggio Java è documentato dal seguente codice:

```
public class Alfa {
  public void esegui(){
    …
    …
    …
  }
}

public class Beta extends Alfa {
  public void esegui() throws Eccezione {
    …
    …
    …
  }
}
```

In questo caso il polimorfismo permetterebbe il seguente frammento di codice:

```
…
Alfa a = new Beta();
…
a.esegui();
…
```

Ma l'invocazione del metodo esegui può sollevare un'eccezione di tipo *Eccezione* non dichiarata – e quindi non intercettabile – nella classe *Alfa*.

**OSSERVAZIONE** Le eccezioni generate da parte delle sottoclassi possono solo essere un sottoinsieme delle eccezioni generate dalla superclasse.

# Sintesi

**Java**. Linguaggio di programmazione orientato agli oggetti e progettato per essere portabile tra piattaforme diverse.

***Bytecode***. Il compilatore Java non produce direttamente codice eseguibile, ma un codice intermedio, *bytecode*, che viene eseguito dall'interprete Java.

**JVM: *Java Vitual Machine***. È l'interprete che esegue il *bytecode* dei programmi Java. Ogni piattaforma hardware/software ha una sua propria specifica JVM per cui il *bytecode* è indipendente dalla piattaforma di sviluppo e di esecuzione garantendo la portabilità («*Write Once Run Anywhere*»). L'ambiente della JVM è strutturato in un set di istruzioni per il *bytecode*, un gruppo di registri per lo stato della macchina virtuale, un'area di memoria *stack* per l'esecuzione dei metodi, un'area di memoria *heap* per la creazione degli oggetti e su cui opera il *garbage collector* e un'area per la memorizzazione del codice dei metodi.

**Classe**. È l'elemento di base del linguaggio Java. Una classe è un insieme di dichiarazioni che costituiscono un modello formale per istanziare oggetti. La struttura generale di una classe prevede la dichiarazione del *package* di appartenenza e dei membri della classe (attributi e metodi).

***Package***. Meccanismo per organizzare le classi Java in gruppi ordinati. I *package* possono essere memorizzati in file compressi denominati file JAR.

**Oggetto**. Istanza di una classe il cui valore degli attributi ne determina lo stato; agli oggetti specifici sono applicati i metodi della classe. A ogni oggetto viene assegnata al momento della creazione un'area nella memoria *heap* che viene recuperata dal *garbage collector* della JVM e resa nuovamente disponibile per essere riutilizzata quando l'oggetto risulta non più accessibile. Un oggetto viene istanziato con una invocazione del tipo:

```
NomeClasse nomeOggetto = new NomeClasse(…);
```

**Attributi**. Un attributo (o proprietà o variabile di istanza) è una variabile, o una costante, il cui valore in un determinato istante contribuisce a determinare lo stato di un oggetto.

**Metodi**. I metodi sono funzioni applicate agli oggetti per realizzare l'aspetto operativo di una classe permettendo l'accesso e la gestione dello stato interno determinato dagli attributi. L'invocazione di un metodo avviene con una istruzione del tipo

```
nomeOggetto.nomeMetodo(…) .
```

A differenza degli attributi che assumono valori distinti per i diversi oggetti istanza di una stessa classe, il codice dei metodi è comune a tutti gli oggetti di una classe e viene memorizzato una sola volta in una specifica area di memoria della JVM.

**Accessibilità dei membri di una classe**. Java permette di definire diversi livelli di accessibilità dei membri (attributi e metodi) di una classe: privato, protetto, pubblico e *package*. Mediante i livelli di accessibilità viene implementata l'interfaccia degli oggetti rispettando il principio di *information hiding*.

**Membri statici di una classe**. Java permette di definire i membri di una classe (attributi o metodi) con la qualifica `static`. Membri di questo tipo appartengono alla classe e non al singolo oggetto. Modificare il valore di un attributo statico significa modificare tale valore per tutte le istanze della classe. I metodi statici possono operare solo su attributi statici. Si può far riferimento ai membri statici di una classe con le seguenti notazioni:

```
NomeClasse.nomeAttributo
NomeClasse.nomeMetodo(…)
```

**Costruttore**. Particolare metodo di una classe che viene invocato all'atto della creazione di un oggetto invocando l'operatore `new`. Il costruttore ha sempre lo stesso nome della classe ed è sempre pubblico e non ha un tipo di ritorno in quanto «restituisce» un oggetto della classe stessa. La sua funzione è quella di inizializzare il valore degli attributi di un oggetto al momento della creazione. Se non viene implementato esplicitamente Java crea un oggetto invocando un costruttore di default.

**Costruttore di copia**. È un costruttore che accetta come argomento un oggetto della stessa classe: il suo scopo è quello di creare un nuovo og-

getto il cui stato (valore degli attributi) è uguale a quello dell'oggetto passato come argomento (clonazione).

**Programma Java**. È composto da un insieme di classi i cui oggetti cooperano tra loro scambiandosi messaggi attraverso i metodi.

**Metodo *main***. È il metodo iniziale da cui parte l'esecuzione di un programma. L'intestazione del metodo *main* di una classe è sempre:

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

È bene che ogni classe disponga di un metodo *main* che verifichi la funzionalità dei metodi della classe stessa istanziandone un oggetto. In un programma formato da più classi l'esecuzione si avvia a partire dal metodo *main* della classe principale.

***Javabean***. *Javabean* sono classi scritte in linguaggio Java che seguono particolari convenzioni. È previsto che esista un costruttore privo di argomenti e che l'accesso agli attributi privati risultino accessibili e modificabile tramite metodi il cui nome è dato dal nome dell'attributo stesso e dal prefisso *get* o *set*.

**Dati primitivi e classi *wrapper***. Nel linguaggio di programmazione Java oltre ai tipi definiti dalle classi esistono tipi primitivi per valori numerici (`int`, `long`, `short`, `byte`, `float`, `double`, …), per i caratteri (`char`) e per i valori logici (`boolean`). In determinate situazioni può risultare utile trattare i dati primitivi come oggetti: a questo scopo Java rende disponibili specifiche classi ***wrapper*** nel *package java.lang*. Una classe *wrapper* comprende come attributo una variabile del tipo primitivo che incapsula, ovvero «trasforma» un tipo primitivo in un oggetto equivalente che dispone, attraverso i metodi della classe *wrapper* stessa, di nuove funzionalità.

**Stringhe**. In Java non esiste un tipo primitivo per le stringhe di caratteri, ma una classe predefinita denominata *String*. Un oggetto di tipo *String* è sempre costante: la modifica del suo valore implica la distruzione dell'oggetto stringa originale e la creazione di un nuovo oggetto stringa con il valore aggiornato. L'operatore «+» permette di concatenare tra loro due stringhe per crearne una nuova il cui contenuto è dato dai caratteri della seconda giustapposti a quelli della prima. La classe *String* rende disponibili numerosi metodi per la gestione delle stringhe di **caratteri**.

**Array**. Gli array in Java sono oggetti: è possibile sia definire array i cui elementi contengono tipi di dato primitivi che array i cui elementi contengono riferimenti a oggetti di una determinata classe. Per istanziare un array è possibile utilizzare un'istruzione come la seguente:

```
Tipo identificatore[] =
 new Tipo[dimensione];
```

ad esempio

```
String vettoreDiStringhe[] =
      new String[10];
```

Nel caso di un array di riferimenti a oggetti un elemento che non riferisce alcun oggetto contiene il valore «`null`». Dal momento che un array può contenere oggetti e che un array è esso stesso un oggetto è possibile creare strutture comunque complesse. L'attributo *length* di un array contiene il numero dei suoi elementi. In Java è possibile sia passare array come parametri ai metodi che ricevere da questi array come valori di ritorno.

**Costruttore di copia**. Se gli attributi di una classe non sono di un tipo di dato primitivo o immutabile (come ad esempio gli oggetti di classe *String*), ma riferimenti a oggetti è necessario che il codice del costruttore di copia li cloni senza limitarsi a copiarne i riferimenti, in caso contrario il costruttore crea un nuovo oggetto che condivide con l'oggetto originale i riferimenti agli attributi e ogni varizione dell'uno si rifletterebbe in una automatica e spesso inaspetatta variazione dell'altro. La stessa attenzione deve essere posta ogni volta che un metodo di una classe restituisce un riferimento a un attributo, o ha come parametro un oggetto da riferire da parte di un attributo: è sempre necessario effettuare una clonazione dell'oggetto.

**Eccezioni**. Sono eventi che si presentano in fase di esecuzione (*run-time*) di un programma al verificarsi di situazioni anomale (divisione per zero, uso di un indice fuori dal *range* di un array, accesso a un riferimento nullo, …). Un'eccezione può essere intercettata e gestita, oppure il programma termina la sua esecuzione. L'intercettazione e gestione delle eccezioni si basa sul costrutto `try`/`catch`. La parola chiave `try` permette di definire un

blocco di istruzioni di cui controllare l'esecuzione per intercettare eventuali eccezioni che potrebbero verificarsi; ogni singola clausola `catch` definisce un blocco di istruzioni che devono recuperare la situazione anomala connessa alla specifica eccezione intercettata. In Java le eccezioni sono oggetti istanza di classi che derivano dalla classe predefinita *Exception* che a sua volta deriva da *Throwable*; il linguaggio definisce alcune classi di eccezioni predefinite: *ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerExceptin, IOException, ….* Una classe eccezione può essere minimale come la seguente

```
public class Eccezione extends Exception {};
```

oppure il programmatore può definire attributi, costruttori e metodi per le necessità di gestione degli oggetti che costituiscono le eccezioni concretamente sollevate, in particolare per fornire informazioni specifiche sul tipo di errore che ha causato l'eccezione stessa.

**Generazione di eccezioni**. Per segnalare il proprio insuccesso un metodo di una classe Java può generare un'eccezione. Un metodo che può generare un'eccezione deve specificare questa eventualità nella propria firma utilizzando la parola chiave throws seguita dal tipo di eccezione, o da un elenco di tipi di eccezioni, che possono essere sollevate dal metodo; nel codice del metodo la parola chiave throw consente di interrompere l'esecuzione e di sollevare l'eccezione specificata come un nuovo oggetto. La semantica dell'istruzione `throw` ha alcuni aspetti in comune con quella dell'istruzione `return`: in particolare l'esecuzione di `throw` implica la terminazione immediata del metodo e il passaggio del controllo al codice chiamante del metodo stesso. Un metodo può sollevare più tipi di eccezione che possono essere singolarmente gestite specificando clausole `catch` multiple.

***Handle or declare***. Questa regola («gestisci o dichiara») stabilisce che a fronte di una possibile eccezione di tipo controllato («*checked*») il codice di un metodo deve intercettarla e gestirla, oppure dichiarare a sua volta di sollevarla; un'eccezione non deve mai passare inosservata: se un metodo non la gestisce trasferisce l'obbligo di gestirla al metodo che lo ha invocato. Alcune eccezioni predefinite di classe *RuntimeException* sono di tipo non controllato («*unchecked*») e possono essere non gestite e non dichiarate: in questo caso la loro eventuale generazione causa l'interruzione dell'esecuzione del programma.

**Input/output predefinito**. Gli oggetti predefiniti *System.out* di classe *PrintStream* e *System.in* di classe *InputStream* sono normalmente associati rispettivamente allo standard output (finestra di tipo testuale sullo schermo) e standard input (tastiera). I metodi che rendono disponibili consentono di eseguire operazioni di input/output tradizionali (ad esempio utilizzando i metodi *print* o *println* della classe *PrintStream*): la complessità della classe *InputStream* può essere gestita incapsulandolo in un oggetto di tipo *BufferedReader* che rende disponibile il metodo *readLine* per la lettura di una riga di testo dallo standard input.

**Input/output sequenziale da file di testo**. Le classi *BufferedReader* e *BufferedWriter* che incapsulano a loro volta oggetti di tipo *FileReader* e *FileWriter* rendono disponibili semplici metodi per la lettura/scrittura di singole righe di un file di testo.

**Serializzazione e persistenza degli oggetti**. La serializzazione di un oggetto è un processo che permette di salvarlo in un supporto di memorizzazione sequenziale (come un file), o di trasmetterlo su un canale di comunicazione sequenziale (come una connessione di rete). La serializzazione può essere effettuata in forma binaria, oppure può impiegare codifiche testuali come il formato XML (*eXtensible Markup Language*). Lo scopo della serializzazione è quello di salvare e/o trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, spesso denominato «deserializzazione». Una classe i cui oggetti devono essere serializzati deve implementare l'interfaccia *Serializable* del *package java.io*:

```
public class ClasseSerializzabile
      implements java.io.Serializable { …
}
```

Se tale classe ha come attributi dei tipi di dato non primitivi anche le classi di cui sono istanza devono a loro volta implementare l'interfaccia *Serializable*. Un oggetto serializzabile può essere reso persistente su file invocando il metodo *writeObject* della classe *ObjectOutputStream* che incapsula un oggetto di tipo *FileOutputStream*. L'oggetto può essere ri-

pristinato dal file invocando il metodo *readObject* della classe *ObjectInputStream* che incapsula un oggetto di tipo *FileInputStream*.

**Ereditarietà**. È un meccanismo di astrazione finalizzato alla creazione di gerarchie di classi: nel linguaggio Java è realizzato mediante la parola chiave `extends` utilizzata come nella seguente dichiarazione

```
public class B extends A {…}
```

dove *A* è detta superclasse e *B* classe derivata, o sottoclasse; la classe *A* rappresenta una generalizzazione della classe *B*, mentre questa è una specializzazione della classe *A*. La classe *B* eredita attributi e metodi della classe *A*, può introdurne di nuovi e ridefinirne alcuni. Tutte le classi definite in linguaggio Java derivano implicitamente dalla classe «radice» *Object*. L'ereditarietà viene spesso usata per fattorizzare caratteristiche (attributi e metodi) comuni a più classi.

***Overrridding***. Consiste nella ridefinizione in una classe derivata di metodi già definiti nella superclasse: per gli oggetti istanza di una classe derivata un metodo sovrascritto nasconde la definizione corrispondente data nella superclasse.

***hashCode***. La classe *Object* definisce i metodi *toString* ed *equals* che sono ereditati da qualsiasi classe definita in un programma Java: un uso coerente di questi metodi ne richiede la ridefinizione ad hoc nel contesto di una specifica classe. Essi infatti operano sullo *hashCode* di un oggetto che l'ambiente di esecuzione Java calcola a partire dall'indirizzo di memorizzazione nello *heap* piuttosto che sul contenuto informativo. In particolare il metodo *equals* definito nella classe *Object* considera sempre diversi due oggetti distinti che hanno necessariamente indirizzi di memoria diversi.

**`super`**. Questa parola chiave del linguaggio Java permette di far riferimento ai metodi e agli attributi definiti nella superclasse nel codice dei metodi di una classe derivata. Utilizzando la stessa parola chiave è possibile invocare il costruttore della superclasse nel codice del costruttore di una classe derivata. L'uso corretto della parola chiave `super` garantisce il rispetto del principio di incapsulamento e rende indipendente il codice di una classe derivata da eventuali modifiche del codice della superclasse.

***Up-casting* e *down-casting***. È consentito fare riferire un oggetto istanza di una certa classe da una variabile avente come tipo una qualsiasi superclasse, ma non il viceversa. Il principio da seguire è quello per cui un oggetto riferito da una variabile di diverso tipo debba avere tutti i requisiti affinché l'invocazione dei metodi della classe che definisce il tipo della variabile abbia successo. Nel linguaggio di programmazione Java è possibile operare il *casting* dei tipi degli oggetti in modo che un riferimento a un oggetto possa essere trasformato in un riferimento di diverso tipo. Le operazioni di casting di un riferimento possono essere utilizzate per trasformare il riferimento a un oggetto in un riferimento avente come tipo una superclasse – si tratta in questo caso di *up-casting* – oppure in un riferimento a una sottoclasse – si tratta allora di un *down-casting* (nel caso di *up-casting* non è necessario il ricorso esplicito all'operatore di *cast* perché il *casting* è implicito). Ogni oggetto può essere assegnato a una variabile di classe *Object*. Le regole di compilazione non permettono di operare in tutti quei casi in cui il *casting* non è possibile (quando, ad esempio, le classi delle variabili non hanno tra loro alcuna relazione gerarchica di ereditarietà), ma in ogni caso in fase di esecuzione del programma (*runtime*) si può verificare un'eccezione di tipo *ClassCastException* se l'oggetto del *casting* non è compatibile con il tipo della variabile a cui viene assegnato.

**Metodo *clone***. È un metodo definito dalla classe *Object* che restituisce una copia dell'oggetto su cui viene invocato. Tale metodo deve essere opportunamente sovrascritto per effettuare delle copie corrette in caso di oggetti particolarmente complessi (oggetti che contengono altri oggetti).

**Classi astratte**. In una gerarchia di classi è conveniente fattorizzare attributi e metodi comuni a più classi al livello più alto della gerarchia, in modo che possano essere ereditati dal maggior numero possibile di classi. In questo contesto le classi astratte consentono di usare questo meccanismo di fattorizzazione degli attributi comuni anche nel caso in cui la superclasse risultante non sia «ben definita». Nel linguaggio Java una classe astratta viene definita premettendo al suo nome il qualificatore `abstract` e non può essere istanziata anche se può avere costruttori. Una classe astratta può avere uno o più «metodi astratti», metodi cioè privi del corpo e qualificati con la parola chia-

ve `abstract` la cui implementazione è vincolante per le classi derivate: la dichiarazione di uno o più metodi astratti rende implicitamente una classe astratta; se una classe concreta deriva da una classe astratta deve fornire un'implementazione per tutti i metodi astratti ereditati.

**Interfacce**. Nel linguaggio di programmazione Java la parola chiave `interface` consente di dichiarare una «interfaccia» che è analoga a una classe, ma a differenza di questa costituisce una pura specifica di invocazione e come tale si limita a dichiarare i metodi, definendone la firma, senza implementarli (tutti i metodi di un'interfaccia sono implicitamente astratti). Una interfaccia viene «implementata» dalle classi mediante l'uso della parola chiave `implements`. Implementare un'interfaccia impone formalmente a una classe di definire i metodi che l'interfaccia stessa dichiara: in questo senso le interfacce costituiscono una specie di «contratto» per le classi che le implementano e il rispetto di tale contratto viene verificato dal compilatore che genera un errore nel caso in cui una classe che dichiara di implementare una determinata interfaccia non ne definisca tutti i metodi dichiarati. Dal momento che in Java una classe può implementare più di un'interfaccia, questo meccanismo permette di simulare l'eredità multipla di cui il linguaggio non dispone.

**Polimorfismo**. L'operazione di estensione di una classe permette di derivare da essa una o più sottoclassi: è sempre possibile utilizzare un oggetto istanza di una delle sottoclassi derivate in un contesto in cui sia richiesto un oggetto istanza della classe originale. Questa caratteristica dei linguaggi di programmazione OO in generale e del linguaggio Java in particolare viene detta polimorfismo, intendendo con questo termine la capacità di un oggetto di assumere «forme» molteplici: istanza della propria classe o istanza di ogni superclasse di essa. Il polimorfismo degli oggetti risulta particolarmente utile quando in una gerarchia di classi derivate viene ridefinito un metodo e la specifica versione del metodo da eseguire viene scelta automaticamente sulla base del tipo di oggetto effettivamente assegnato a un riferimento in fase di esecuzione piuttosto che al momento della compilazione.

***Dynamic binding***. Il meccanismo che determina quale metodo deve essere invocato in base alla classe di appartenenza dell'oggetto è definito binding (letteralmente «legame», «collegamento»). Esistono due tipi fondamentali di binding: *binding* statico (*early binding*) in cui il metodo da invocare viene determinato in fase di compilazione del codice (*compile-time binding*), e *binding* dinamico (*late binding*) in cui il metodo da invocare viene determinato durante l'esecuzione del codice (*run-time binding*). Il linguaggio Java, come la maggior parte dei linguaggi di programmazione a oggetti, adotta il *binding* dinamico a *run-time* quando il *binding* statico a *compile-time* risulta impossibile, come nel caso dei metodi polimorfici. Il polimorfismo consente di sviluppare codice a oggetti «estensibile»: l'aggiunta di nuove classi a una gerarchia preesistente non influenza il codice che gestisce oggetti istanza delle classi della gerarchia stessa in modo polimorfico.

**RTTI (*Run-Time Type Identification*)**. L'esigenza di un supporto per l'identificazione del tipo effettivo di un oggetto durante l'esecuzione di un programma in linguaggio Java nasce dall'impossibilità di invocare direttamente il metodo di una sottoclasse su un riferimento una superclasse, cosa che può risultare utile o anche necessaria. Per applicare propriamente il casting necessario in questa situazione è indispensabile conoscere il tipo effettivo di cui l'oggetto riferito è istanza: l'operatore Java instanceof permette di verificare la classe di appartenenza di un oggetto. L'espressione logica oggetto instanceof Classe viene valutata true se e solo se oggetto è un'istanza di Classe: è richiesto che Classe sia staticamente il nome del tipo, non è possibile utilizzare riferimenti variabili.

***Eccezioni e polimorfismo***. Un aspetto non secondario del polimorfismo del linguaggio Java è dato dalla regola relativa alle eccezioni dei metodi ridefiniti in una classe derivata: un metodo che ridefinisce un metodo che dichiara di generare eccezioni non può sollevare tipi di eccezione diversi o aggiuntivi rispetto a quelli del metodo originale: questa regola assicura che tutte le eccezioni generate siano sempre correttamente gestite.

## QUESITI

**1** **Il *bytecode* è …**

A … il codice sorgente di un programma Java.

B … il codice eseguibile generato dal compilatore Java.

C … un codice intermedio generato dal compilatore Java che deve essere interpretato dalla JVM.

D Nessuna delle risposte precedenti.

**2** **Quali delle seguenti affermazioni sono vere a proposito della *Java Virtual Machine* (JVM)?**

A È indipendente dalla piattaforma hardware/software.

B Ogni tipo di piattaforma hardware/software ha una specifica JVM.

C La JVM è di fatto un interprete del *bytecode*.

D La JVM è un componente hardware.

**3** **La portabilità del codice Java è data dal fatto che …**

A … il *bytecode* sviluppato su una qualsiasi piattaforma hardware/software può essere eseguito su una qualsiasi piattaforma hardware/software.

B … un programma sorgente Java può essere compilato con un qualsiasi compilatore, anche di un altro linguaggio di programmazione.

C … che la JVM sia sempre la stessa su qualsiasi piattaforma hardware/software.

D Nessuna delle risposte precedenti.

**4** **Una classe è …**

A … una collezione di oggetti: collegando gli oggetti tra di loro si implementa l'incapsulamento.

B … un modello da cui si creano oggetti simili.

C … l'implementazione di un oggetto nella sintassi Java.

D Nessuna delle risposte precedenti.

**5** **Attraverso quale meccanismo gli oggetti interagiscono tra di loro?**

A Lo scambio di messaggi attraverso l'invocazione dei metodi.

B Lo scambio di metodi attraverso l'inoltro di messaggi.

C La condivisione del valore degli attributi.

D Nessuna delle risposte precedenti.

**6** **Quale è la funzione dell'operatore Java `new`?**

A Creare una classe.

B Creare un metodo.

C Creare un oggetto.

D Creare un *package*.

**7** **È possibile creare due riferimenti distinti a uno stesso oggetto?**

A Sì, sempre.

B No, mai.

C Sì, ma solo se i riferimenti sono entrambi dello stesso tipo dell'oggetto che riferiscono.

D Sì, ma solo se i riferimenti non sono entrambi dello stesso tipo dell'oggetto che riferiscono.

**8** **Nel linguaggio Java lo spazio di memoria assegnato agli attributi di un oggetto istanziato l'operatore `new` viene liberato …**

A … automaticamente dal supporto a tempo di esecuzione del linguaggio.

B … solo al termine dell'esecuzione del programma.

C … automaticamente dal distruttore della classe.

D Nessuna delle risposte precedenti.

**9** **Quali delle seguenti affermazioni a proposito del metodo *main* di una classe sono vere?**

A È il primo metodo da cui si avvia l'esecuzione del codice di una classe.

B Il primo metodo che viene applicato ogni volta che si istanzia un oggetto.

C È un metodo statico.

D Non prevede parametri.

**10** **Quali delle seguenti affermazioni a proposito di un costruttore sono vere?**

A È un metodo che viene eseguito automaticamente all'atto della creazione di un oggetto.

B Può prevedere come parametro un oggetto.

C Può non essere pubblico.

D Non prevede tipo di ritorno perché implicitamente restituisce un oggetto della classe.

**11** **Il valore degli attributi dichiarati nella sezione privata di una classe …**

A … può essere modificato mediante specifici metodi della classe stessa.

B … può essere modificato solo dal costruttore.

C … non può essere modificato.

D … può essere modificato direttamente con un espressione del tipo *oggetto.attributo = espressione;*

**12** **Quali delle seguenti affermazioni circa un attributo con accessibilità di livello `protected` sono vere?**

A La sua accessibilità è equivalente a `public`.

B La sua accessibilità diretta è possibile solo a livello di classe e classe derivata.

C La sua accessibilità diretta è possibile solo a livello di classe e di *package*.

D La sua accessibilità diretta è possibile solo a livello di classe, classe derivata e di *package*.

**13** **I membri statici di una classe sono …**

A … le costanti.

B … gli attributi e i metodi che appartengono non alle singole istanze, ma alla classe.

C … membri che possono essere riferiti con una notazione del tipo `NomeClasse.nomeMembro`.

D Nessuna delle risposte precedenti.

**14** **Le classi *wrapper* dei tipi di dato primitivi sono …**

A … gli stessi tipi di dato primitivi.

B … classi che permettono di gestire valori associati ai tipi di dato primitivi come fossero oggetti.

C … classi che permettono a oggetti di qualunque classe di gestire tipi di dato primitivi.

D … classi che hanno come metodi gli operatori algebrici per definire espressioni con i tipi di dato primitivi.

**15** **Indicare quanto vale in Java l'output generato dalla seguente istruzione**

```
System.out.println(3+4+" = "+3+4);
```

A `7 = 7`

B `7 = 34`

C `34 = 7`

D `34 = 34`

**16** **Una stringa di caratteri in Java è …**

A … un tipo di dato primitivo.

B … un *array* di caratteri.

C … un oggetto costante di classe *String*.

D … un oggetto variabile di classe *String*.

**17** ***Javabean* è …**

A … una classe i cui oggetti permettono di sviluppare documenti.

B … uno strumento Java per la documentazione automatica dei programmi.

C … una classe Java che osserva alcune convenzioni di codifica.

D … la classe da cui derivano tutte le classi del linguaggio Java.

**18** **In Java gli array sono oggetti?**

A No, mai.

B Sì in ogni caso.

C Sì, ma solo se non contengono tipi di dato primitivi.

D Nessuna delle risposte precedenti.

**19** **Un elemento di un array di oggetti è vuoto se …**

A … non contiene nulla.

B … contiene il valore 0.

C … contiene il valore predefinito `null`.

D … contiene il riferimento a un oggetto creato con il costruttore di default.

**20** Con l'istruzione `int[] unVettore;` ...

A … si istanzia un vettore di valori numerici interi.

B … si dichiara un riferimento a un vettore di valori numerici interi.

C … si istanzia un vettore di.

D … si dichiara un vettore di riferimenti a oggetti di tipo `int`.

**21** Ponendo uguale a `null` un elemento di un array di riferimenti a oggetti ...

A … l'oggetto a cui faceva riferimento viene distrutto.

B … l'oggetto a cui faceva riferimento viene distrutto solo se non è riferito da altre variabili.

C … continua a esistere in ogni caso.

D Nessuna delle risposte precedenti.

**22** Le eccezioni sono ...

A … eventi anomali rilevati dal compilatore in fase di generazione del *byte-code*.

B … eventi anomali rilevati dall'ambiente di esecuzione a *run-time*.

C … avvisi normalmente generati dall'ambiente di esecuzione.

D Nessuna delle risposte precedenti.

**23** In Java la gestione delle eccezioni avviene tramite il costrutto ...

A … `if`/`else`

B … `try`/`catch`

C … `switch`/`case`

D … `do`/`while`

**24** Indicare quali delle seguenti affermazioni sono vere rispetto alla generazione delle eccezioni.

A Se un metodo genera un'eccezione deve anche gestirla.

B Un metodo che genera un'eccezione deve contenere la clausola `throws` nella sua firma.

C Un'eccezione può essere generata da un metodo utilizzando l'istruzione `throw`.

D Un'eccezione può essere generata da un metodo utilizzando l'istruzione `try`.

**25** Indicare quali delle seguenti affermazioni sono false rispetto al sollevamento delle eccezioni.

A L'istruzione `throw` sostituisce il costrutto `try`/`catch`.

B Nel codice di un metodo che genera eccezioni non è necessaria l'istruzione `return`.

C Per generare specifici tipi di eccezioni definite dall'utente è necessario definire una classe ad hoc.

D Un metodo non può generare più di un tipo di eccezioni.

**26** La regola «*Handle or declare*» si riferisce al fatto che ...

A … tutti gli oggetti per essere gestiti (*handle*) devono essere preventivamente dichiarati (*declare*).

B … a fronte di una possibile eccezione un metodo deve gestirla (*handle*), oppure dichiarare a sua volta di generarla (*declare*).

C … a fronte di una possibile eccezione un metodo deve gestirla (*handle*) e dichiarare di generarla (*declare*).

D Nessuna delle risposte precedenti.

**27** Serializzare un oggetto significa ...

A … memorizzarlo di seguito ad altri oggetti dello stesso tipo.

B … memorizzarlo di seguito ad altri oggetti non necessariamente dello stesso tipo.

C … salvare un oggetto su un supporto di memorizzazione sequenziale per renderlo persistente, o trasmetterlo su un canale di comunicazione sequenziale per renderlo trasferibile.

D Nessuna delle risposte precedenti.

**28** Quali delle seguenti affermazioni sono vere rispetto al processo di serializzazione/deserializzazione di un oggetto?

A Permette di rendere un oggetto persistente.

B La deserializzazione permette di ripristinare un oggetto precedentemente serializzato.

C Una classe i cui oggetti debbono essere seria-

lizzati deve implementare l'interfaccia Java *Serializable*.

D Il formato della serializzazione può essere sia binario che testuale.

**29** **L'ereditarietà è un …**

A … meccanismo di astrazione finalizzato alla definizione di interfacce.

B … meccanismo di astrazione finalizzato all'implementazione dell'*information hiding*.

C … meccanismo di astrazione finalizzato alla costruzione di gerarchie classi.

D … meccanismo di astrazione finalizzato alla gestione degli attributi complessi.

**30** **Rispetto all'ereditarietà, il linguaggio Java …**

A … non ne permette l'implementazione.

B … permette l'implementazione dell'ereditarietà singola.

C … permette l'implementazione dell'ereditarietà multipla.

D … permette l'implementazione dell'ereditarietà multipla solo per le classi astratte.

**31** **In un contesto di ereditarietà tra due classi …**

A … la classe derivata eredita attributi e metodi della superclasse.

B … la superclasse eredita attributi e metodi della classe derivata.

C … la classe derivata eredita solo i metodi della superclasse.

D … la classe derivata eredita solo gli attributi della superclasse.

**32** **Il meccanismo di *overriding* permette di …**

A … ridefinire nella classe derivata metodi definiti nella superclasse.

B … ridefinire nella superclasse metodi definiti nella classe derivata.

C … definire nell'ambito di una stessa classe più metodi con lo stesso nome.

D … ridefinire in una classe i metodi definiti dalla classe *Object*.

**33** **La classe *Object* Java è …**

A … la classe derivata di una qualsiasi altra classe.

B … una classe di sistema che non può essere derivata.

C … è la classe radice di una qualsiasi gerarchia di classi.

D Nessuna delle risposte precedenti.

**34** **I metodi *toString* ed *equals* …**

A … sono metodi della classe *Object* che per poter essere adeguatamente utilizzati debbono essere sovrascritti nelle classi in cui si è interessati a utilizzarli.

B … sono metodi della classe *Object* che non possono essere sovrascritti in altre classi.

C … non sono metodi della classe *Object*.

D … sono metodi della classe *Class*.

**35** **Con «fattorizzazione delle caratteristiche» si intende la possibilità di …**

A … raggruppare utilizzando l'ereditarietà caratteristiche comuni a più classi evitando così inutili ridondanze.

B … duplicare utilizzando l'ereditarietà caratteristiche comuni a più classi.

C … ridondare le caratteristiche comuni alle varie classi di una gerarchia.

D Nessuna delle risposte precedenti.

**36** **La parola chiave `super` del linguaggio Java può essere usata per …**

A … invocare in una sottoclasse un metodo della superclasse che è stato sovrascritto.

B … invocare da una superclasse un metodo sovrascritto in una sottoclasse.

C … invocare in una classe derivata il costruttore di una qualsiasi delle superclassi.

D … riferire in una classe derivata gli attributi ereditati.

**37** **In quali situazioni è sempre possibile operare il *casting* dei tipi dei riferimenti?**

A Verso la stessa classe dell'oggetto riferito.

B Verso una superclasse o una sottoclasse dell'oggetto riferito.

C Verso una classe che non appartiene alla gerarchia della classe dell'oggetto riferito.

D Verso *Object*.

**38** **Il *casting* implicito si ha quando questo avviene …**

A … verso una superclasse della classe dell'oggetto riferito.

B … verso una sottoclasse della classe dell'oggetto riferito.

C … verso la classe dell'oggetto riferito.

D … verso una qualsiasi classe se non si specifica l'operatore di *casting*.

**39** **Il *down-casting* permette di trasformare …**

A … un riferimento dalla classe radice della gerarchia verso una sottoclasse.

B … un riferimento di una sottoclasse verso classe radice della gerarchia.

C … un riferimento da una qualsiasi classe a una qualsiasi altra classe.

D … un riferimento qualsiasi verso la classe *Object*.

**40** **Indicare quali delle seguenti affermazioni sono vere relativamente a una classe astratta in linguaggio Java.**

A Non può essere estesa.

B In Java viene definita premettendo alla sua dichiarazione il qualificatore `abstract`.

C Non può essere istanziata.

D Non può avere costruttori.

**41** **Indicare quali delle seguenti affermazioni sono vere relativamente a un'interfaccia in linguaggio Java.**

A Tutti i metodi di un'interfaccia sono implicitamente astratti.

B È analoga a una classe in cui tutti i metodi sono astratti.

C Una classe non può implementare più di un'interfaccia.

D Non vi sono limitazioni per la dichiarazioni degli attributi.

**42** **Indicare quali delle seguenti affermazioni sono vere relativamente al polimorfismo in linguaggio Java.**

A È il meccanismo tramite il quale un oggetto ha la capacità di assumere forme molteplici: come oggetto della propria classe o come oggetto di ogni sua superclasse.

B È un meccanismo che interessa esclusivamente la dichiarazione dei metodi.

C È un meccanismo che interessa esclusivamente da dichiarazione degli attributi.

D Nel linguaggio Java il polimorfismo è basato sul *binding* statico.

**43** **Il polimorfismo si applica a …**

A … classi che non sono in relazione gerarchica.

B … classi che sono in relazione gerarchica.

C … esclusivamente classi astratte.

D Nessuna delle risposte precedenti.

**44** **L'acronimo RTTI significa …**

A … Run-Time Type Identification.

B … Run-Time Trouble Identification.

C … Run-Time Type Implementation.

D … Run-Type Time Identification.

**45** **La parola chiave `instanceof` del linguaggio Java rappresenta …**

A … un operatore che permette di istanziare un oggetto di una specifica classe.

B … un operatore che permette l'identificazione del tipo di un oggetto nella fase di esecuzione di un programma.

C … un attributo della classe *Object*.

D … un metodo della classe *Object*.

**46** **Indicare quali delle seguenti affermazioni relative alla generazione delle eccezioni da parte di metodi sovrascritti in una classe derivata sono vere.**

A Possono generare a loro volta eccezioni della stessa classe di quelle generate dai rispettivi metodi della superclasse.

B Non possono generare eccezioni di sottoclassi di quelle generate dai rispettivi metodi della superclasse.

C Possono non generare eccezioni.

D Possono generare eccezioni di superclassi di quelle generate dai rispettivi metodi della superclasse.

## ESERCIZI

**1** Una organizzazione deve catalogare i computer utilizzati dai propri dipendenti; per ogni computer devono essere memorizzate le seguenti informazioni:

- codice;
- marca;
- modello;
- velocità del processore;
- dimensioni della memoria RAM;
- dimensioni del disco;
- dimensioni del monitor;
- anno di acquisto.

Il codice di ogni computer è un numero progressivo generato automaticamente e non ulteriormente modificabile. Definire mediante un diagramma UML e implementare in linguaggio Java una classe per rappresentare gli oggetti di tipo *Computer* che rispetti le specifiche *Javabean* prevedendo un metodo *main* per testarne le funzionalità.

**2** Un vettore nel piano cartesiano è un segmento orientato definito dalle coordinate dei due estremi («origine» e «vertice») e dal loro ordinamento: due vettori con estremi coincidenti diversamente orientati sono sovrapposti, ma diversi! Implementare in linguaggio Java la classe di FIGURA **13**, definita mediante un diagramma UML e relativa a un vettore nel piano cartesiano.
Inserire nella classe un metodo *main* che consenta di verificarne tutte le funzionalità.

**3** Modificare la classe *Mensola* presentata nel corso del capitolo per fare i modo che, ferma restando la capienza massima, i libri siano sempre disposti in modo compatto dalla parte sinistra della mensola senza spazi vuoti tra un libro e l'altro:

- inserire un libro in una determinata posizione comporta spostare tutti i libri a partire da tale posizione di un posto a destra;

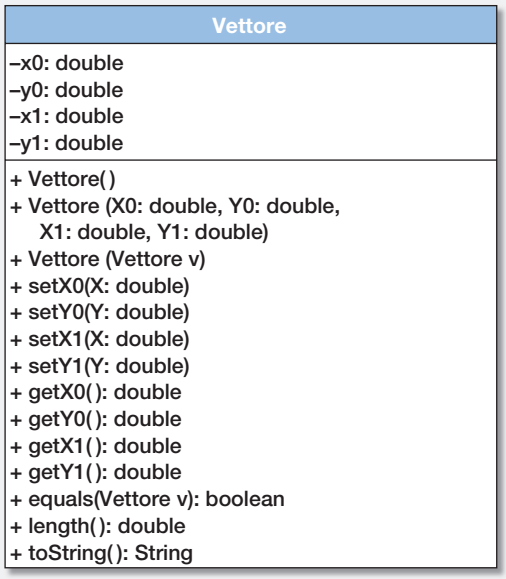



FIGURA **13**

- inserire un libro in una posizione a detsra dell'ultimo libro presente comporta inserire comunque il nuovo libro nella posizione immediatamente a destra dell'ultimo libro;
- eliminare un libro in una determinata posizione significa spostare di una posizione verso sinistra tutti i libri alla sua destra.

**4** Una catena di autonoleggio deve gestire con un sistema informatico i propri veicoli; per ogni veicolo devono essere memorizzate le seguenti informazioni: codice, targa, marca e modello, numero di posti (si veda in proposito l'esercizio 5 del capitolo precedente). Si intende progettare una possibile soluzione per la gestione informatica di quasi 1000 veicoli avente le seguenti funzionalità:

- aggiunta di un nuovo veicolo (il codice deve essere un numero sequenziale incrementato automaticamente ogni volta che si aggiunge un veicolo);
- eliminazione di un veicolo dato il codice o la targa;
- ricerca delle informazioni di un veicolo dato il codice o la targa;
- ricerca di tutti i veicoli aventi un dato numero di posti;
- salvataggio e ripristino su/da file dell'intero insieme di veicoli;

- effettuare l'inventario di quante macchine per ogni marca dispone l'autonoleggio, nella forma *marca*, *numero veicoli*.

1. Definire mediante un diagramma UML le classi che consentono di rappresentare adeguatamente la soluzione del problema.
2. Implementare in linguaggio Java la classi progettate prevedendo e sollevando specifiche eccezioni.
3. Scrivere un metodo *main* che consenta la gestione (aggiunta, eliminazione, ricerca per targa o per codice, elenco dato il numero di posti) dell'intero insieme di veicoli visualizzando messaggi di errore in caso di sollevamento di eccezioni.

**5** Un porto turistico affitta i propri posti-barca (circa un centinaio) alle imbarcazioni che ne fanno richiesta. Per legge è tenuto a registrare per ogni barca ospitata le seguenti informazioni: nome, nazionalità, lunghezza, stazza, tipologia (vela o motore); ma non vi è obbligo di mantenere le informazioni relative alle imbarcazioni dopo che hanno lasciato il porto. I posti-barca sono numerati: i posti da 1 a 20 non possono ospitare barche più lunghe di 10 m e le barche a vela devono essere piazzate in via prioritaria nei posti successivi al 50. Il costo dell'affitto per le barche a vela è di 10 € per metro di lunghezza al giorno, mentre per le barche a motore è di 20 € per tonnellata di stazza al giorno. È richiesta la progettazione di una possibile soluzione per la gestione informatica dei posti-barca che implementi le seguenti funzionalità:

- assegnazione di un posto a una barca in arrivo;
- liberazione di un posto occupato con calcolo dell'importo dell'affitto (in input viene fornito il numero dei giorni di sosta);
- ricerca delle informazioni relative alla barca che occupa un dato posto;
- salvataggio su file dello stato del porto in un certo istante in modo da renderlo persistente;
- produrre una struttura dati (array) dei nomi delle barche di una certa nazionalità specificata dall'utente.

1. Definire mediante un diagramma UML le classi che consentono di rappresentare adeguatamente la soluzione del problema.
2. Implementare in linguaggio Java le classi progettate prevedendo e sollevando specifiche eccezioni.
3. Dotare la classe principale di un metodo *main* che permetta all'utente di esercitare le funzionalità elencate visualizzando messaggi di errore in caso di sollevamento di eccezioni.

**6** Si consideri la situazione esposta nell'esercizio 1 e la si riprogetti facendo in modo che:

- il codice di ogni computer sia un numero progressivo generato automaticamente e non ulteriormente modificabile;
- siano previsti metodi che prevedano per il parco computer operazioni per aggiunta, eliminazione, ricerca per codice, ricerca di tutti i computer con caratteristiche migliori di valori di velocità e dimensione di memoria/disco specificate, salvataggio e ripristino su/da file dell'intero catalogo.

Della classe descritta si fornisca il diagramma UML e la si implementi prevedendo e generando specifiche eccezioni.

**7** Si intende realizzare una gerarchia di classi per rappresentare e gestire i seguenti tipi di oggetti: PC fissi, suddivisi in desktop e *server*, e PC portatili, suddivisi in notebook e palmari. Le caratteristiche generali di interesse sono: il tipo di processore, la dimensione della memoria RAM, la dimensione della memoria di massa, la marca, il modello e il sistema operativo. I PC fissi sono caratterizzati dal tipo di case (grande, medio, piccolo); per i PC fissi di tipo desktop è necessario registrare il tipo di scheda video e di scheda audio, mentre per i PC fissi di tipo *server* è necessario sapere il numero dei processori e se hanno o meno dischi di tipo RAID. I PC portatili sono caratterizzati dal peso e dalle dimensioni fisiche (altezza, larghezza e profondità), dalle dimensioni del video, dal fatto di avere o meno l'interfaccia di rete wireless; per i PC portatili di tipo notebook è necessario conoscere se dispongono o meno di webcam e, in caso affermativo, la relativa risoluzione; per i PC portatili di tipo palmare infine deve essere memorizzata la presenza o meno dell'interfaccia Bluetooth e il tipo di espansione della memoria di massa (SD, mini-SD, micro-SD, …). Definire mediante un diagramma UML le classi

che realizzano la gerarchia descritta valutando l'opportunità di utilizzare una o più classi astratte. Implementare in linguaggio Java le classi progettate specificando costruttori e metodi di accesso agli attributi e sovrascrivendo opportunamente i metodi *toString* ed *equals*. Codificare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di PC e invochi ciascuno dei metodi definiti almeno una volta.

**8** Il Ministero dell'Istruzione deve commissionare un software per il calcolo dei contributi statali dovuti alle scuole. Un professionista viene incaricato di progettare e implementare la gerarchia di classi che rappresenta le scuole. Durante un'intervista col direttore generale del Ministero emerge quanto segue:

- le scuole possono essere: elementari, medie o superiori;
- per ogni scuola è necessario memorizzare il codice alfanumerico, la denominazione, l'indirizzo e la città, il numero di studenti, il numero di classi, il numero di sedi aggiuntive e il numero complessivo di laboratori;
- le scuole elementari hanno diritto a un contributo annuale per ogni studente e per ogni sede aggiuntiva: i contributi valgono oggi 125 € per ogni studente e 9000 € per ogni sede aggiuntiva (i valori potrebbero essere modificato in futuro);
- le scuole medie hanno diritto a un contributo annuale per ogni studente, per ogni laboratorio e per ogni sede aggiuntiva: i contributi valgono oggi 150 € per ogni studente, 1100 € per ogni laboratorio e 9000 € per ogni sede aggiuntiva (i valori potrebbero essere modificati in futuro);
- le scuole superiori sono di tre tipi diversi: licei, tecnici e professionali;
- i licei hanno diritto a un contributo annuale uguale a quello delle scuole medie, escluso il contributo per eventuali sedi aggiuntive;
- i tecnici hanno diritto a un contributo annuale per ogni classe e per ogni indirizzo: i contributi valgono oggi 3500 € per ogni classe e 6000 € per ogni laboratorio (i valori potrebbero essere modificati in futuro);
- i professionali – che hanno diritto anche a contributi regionali – hanno diritto a un contributo statale di 2400 € per ogni classe e di 3000 € per ogni laboratorio (i valori potrebbero essere modificati in futuro).

L'incarico del professionista consiste nel progettare mediante un diagramma UML le classi che rappresentano la situazione descritta dal direttore generale e di implementarle in linguaggio Java. Per verificare il lavoro svolto è necessario realizzare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di scuole calcolando e visualizzando per ciascuno una sintetica descrizione e il contributo da pagare.

**9** Un'agenzia di pratiche automobilistiche deve commissionare un software per il pagamento delle tasse annuali sui veicoli. Un professionista viene incaricato di progettare e implementare la gerarchia di classi che rappresentano i veicoli. Durante un'intervista al proprietario dell'agenzia emerge quanto segue:

- i veicoli possono essere motoveicoli o autoveicoli;
- per ogni veicolo è necessario memorizzare la targa, la marca, il modello, l'anno di immatricolazione e il numero di passeggeri consentito oltre al conducente;
- i motoveicoli sono sempre alimentati a benzina e sono caratterizzati dalla potenza espressa in HP: la tassa viene calcolata moltiplicando per la potenza un valore che a oggi vale 1,5 €/HP;
- gli autoveicoli tradizionali possono essere alimentati a benzina o a gasolio e sono caratterizzati dalla potenza espressa in HP: la tassa viene calcolata moltiplicando per la potenza un valore che a oggi vale 2,5 €/HP;
- per gli autoveicoli alimentati a gas, oltre alla potenza è necessario memorizzare il tipo gas (GPL o metano): questi autoveicoli non pagano nessuna tassa per i primi 5 anni dall'immatricolazione, trascorso questo periodo la tassa viene calcolata moltiplicando per la potenza un valore che a oggi vale 0,5 €/HP per il metano e 0,75 €/HP per il GPL;
- gli autoveicoli alimentati a gas idrogeno pagano una tassa che aumenta di 0,1 €/HP per ogni anno di vita del veicolo a partire da una tassa iniziale pari a 0 €/HP il primo anno di immatricolazione;
- come incentivo governativo gli autoveicoli elettrici non pagano alcuna tassa.

L'incarico consiste nel progettare mediante un diagramma UML le classi che rappresentano la situazione descritta dal proprietario dell'agenzia e di implementarle in linguaggio Java. Per verificare il lavoro svolto è necessario realizzare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di veicoli visualizzando per ciascuno una breve descrizione delle caratteristiche e la tassa da pagare.

**10** Si intende gestire mediante un programma Java i vagoni che compongono un treno. Per ogni vagone si hanno alcuni attributi fondamentali:

- codice;
- peso a vuoto;
- azienda costruttrice;
- anno di costruzione.

Per i vagoni passeggeri si devono inoltre memorizzare:

- classe;
- numero di posti disponibili;
- numero di posti occupati;

mentre per i vagoni merci si devono memorizzare:

- volume di carico;
- peso massimo di carico;
- peso effettivo di carico.

Per la composizione di un treno è fondamentale la gestione del peso dei vagoni che nel caso dei carri merci è di immediata determinazione, mentre per le carrozze passeggeri deve essere stimato considerando un peso medio per occupante di 65 kg (valore che potrebbe essere necessario modificare).
Dopo avere disegnato il diagramma UML delle classi della soluzione proposta e averlo implementato in linguaggio Java codificare una classe Java *Treno* con uno o più metodi per l'aggiunta di vagoni: la classe dovrà prevedere un metodo che restituisca il peso complessivo del treno esclusa/e la/e motrice/i.

**11** Di tutti i membri del personale di un'organizzazione di consulenza sono memorizzati nel sistema informatico della stessa i seguenti dati:

- codice;
- cognome;
- nome;
- anno di assunzione o inizio collaborazione.

I membri del personale dell'organizzazione si suddividono in:

- dirigenti;
- funzionari;
- tecnici;

di cui solo i tecnici sono specializzati in una specifica area di competenza (informatica-telecomunicazioni o elettronica-automazione) e possono essere sia interni che esterni (il personale dipendente dell'organizzazione è classificato come «interno», i professionisti collaboratori come «esterni»).
Si intende realizzare un programma per la stima del costo complessivo di partecipazione a un progetto di alcuni membri del personale a partire dal numero di ore di attività previsto per ciascuno di loro sapendo che i costi orari sono valutati come segue:

- tecnico dell'area informatica/telecomunicazioni: 40 €/ora più – ma solo se interno – 1 €/ora per ogni anno trascorso dall'anno di assunzione;
- tecnico dell'area elettronica/automazione: 50 €/ora più – ma solo se interno – 1 €/ora per ogni anno trascorso dall'anno di assunzione;
- funzionario junior (meno di 10 anni di esperienza a partire dall'anno di assunzione o inizio collaborazione): 70 €/ora;
- funzionario senior (più di 10 anni di esperienza a partire dall'anno di assunzione o inizio collaborazione): 80 €/ora;
- dirigente: sempre 100 €/ora.

Ferme restando le regole di calcolo, gli importi orari devono poter essere modificati. Dopo avere disegnato il diagramma UML delle classi della soluzione proposta e averlo implementato in linguaggio Java codificare una classe Java denominata *Progetto* con uno o più metodi per l'aggiunta di membri del personale al progetto: la classe dovrà prevedere un metodo che restituisca il costo complessivo relativo al personale per l'intero progetto.

**12** Una grande catena di autonoleggio deve gestire con un sistema informatico i propri veicoli (vet-

ture e furgoni); per ogni veicolo devono essere memorizzate le seguenti informazioni:

- targa;
- numero di matricola;
- marca;
- modello;
- cilindrata;
- anno di acquisto;
- capacità serbatoio in litri.

Per le vetture è inoltre necessario memorizzare il numero di posti, mentre per i furgoni deve essere memorizzata la capacità di carico.
Progettare mediante un diagramma UML una gerarchia di classi che consenta di rappresentare adeguatamente la situazione descritta; implementare in linguaggio Java la gerarchia di classi progettata. Sapendo inoltre che i veicoli vengono forniti col pieno di carburante e che il costo del noleggio è così calcolato:

- autovetture: 50 euro al giorno, più 1 euro ogni 25 km percorsi, più 2 euro per ogni litro di carburante che manca dal pieno al momento della restituzione;
- furgoni: 70 euro al giorno, più 1 euro ogni 30 km percorsi dopo i primi 100 km, più 2 euro per ogni litro di carburante che manca al pieno al momento della restituzione;

implementare una classe Java che consenta la gestione del noleggio dei mezzi e il calcolo dell'importo da far pagare al noleggiatore in funzione dei parametri descritti.

**13** Nel videogioco «La battaglia della terra di mezzo» le forze del Bene sono composte dalle razze degli Uomini, degli Elfi, dei Nani e degli Hobbit, mentre le forze del Male dalle razze degli Orchi, degli Urukhai e dei Sudroni.
Nel gioco ogni singolo personaggio appartiene a una delle razze e, di conseguenza, all'uno o all'altro schieramento e ha un attributo esperienza di combattimento codificato mediante un numero intero compreso tra 1 e 10 variabile in base all'esito dei combattimenti a cui partecipa; dispone inoltre di una forza di combattimento calcolabile mediante le seguenti regole:

| Razza | Calcolo forza in base all'esperienza |
|---|---|
| Uomini | 30 più 6 volte l'esperienza |
| Elfi | Se l'esperienza è inferiore a 5 la forza è 20 più 3 volte l'esperienza, altrimenti è 80 + 2 volte l'esperienza |
| Nani | 20 più 4 volte l'esperienza |
| Hobbit | 10 più 3 volte l'esperienza |
| Orchi | Se l'esperienza è inferiore a 5 la forza è 30 + 2 volte l'esperienza, altrimenti è 70 + 3 volte l'esperienza |
| Urukhai | 50 più 5 volte l'esperienza |
| Sudroni | 40 più 5 volte l'esperienza |

Oltre ai personaggi delle varie razze il gioco prevede gli eroi: ogni eroe appartiene all'uno, o all'altro schieramento, ma non a una razza specifica ed è identificato da un nome, da un livello di energia vitale che varia (da 1 a 10) in base alle vicissitudini dell'eroe e da una forza di combattimento che è sempre 50 volte l'esperienza di combattimento più 50 volte l'energia vitale.
Il gioco prevede di gestire i singoli personaggi e i singoli Eroi che si aggiungono/eliminano nelle varie fasi: quando il gioco ha termine lo schieramento vincente è quello che totalizza la maggiore forza di combattimento. Si richiede:

- un diagramma UML delle classi che modellano lo scenario descritto;
- il codice Java che implementa le classi del diagramma UML progettato;
- il codice di test che inserisca nel gioco un certo numero di personaggi e di eroi con valori di esperienza (ed energia per gli eroi) casuali e che determini lo schieramento vincitore.

*Opzionale*. Il gioco si svolge su un territorio suddiviso in caselle appartenenti a una griglia righe/colonne: due personaggi (sono esclusi gli Eroi che possono intraprendere combattimenti multipli e subire danni che diminuiscono la propria energia vitale) possono combattere tra di loro solo se si trovano in caselle adiacenti; vince chi ha una forza di combattimento maggiore aumentando la propria esperienza di combattimento di un punto se non è già massima, mentre il personaggio sconfitto viene eliminato dal gioco. Integrare le classi già progettate e implementate

con gli attributi e i metodi ritenuti necessari per eseguire i combattimenti.

**14** È richiesto un programma di allenamento per gli allievi di un corso di scacchi: ogni pezzo ha un tipo specifico (pedone, torre, cavallo, alfiere, regina e re) e appartiene a uno dei due schieramenti (bianco o nero) e inoltre occupa una posizione nella scacchiera individuata da un valore numerico compreso tra 1 e 8 inclusi e un carattere alfabetico compreso tra A e H inclusi (FIGURA **14**). Ai fini dell'allenamento i pezzi possono essere posti sulla scacchiera in un modo qualunque, ma ovviamente due pezzi non possono occupare la stessa casella (non è necessaria una matrice per rappresentare la posizione dei pezzi sulla scacchiera ☺).

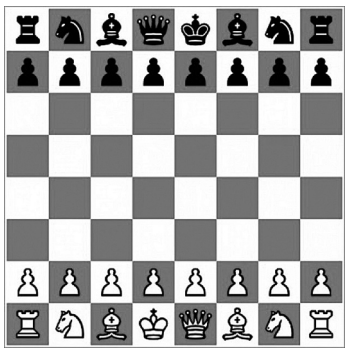


FIGURA **14**

Una stima della situazione di gioco può essere effettuata considerando i seguenti punteggi associati ai vari tipi di pezzo (il re deve essere sempre presente, quindi il suo punteggio è inutile a questo scopo):

| | |
|---|---|
| pedone | 1 |
| torre | 5 |
| cavallo | 3 |
| alfiere | 3 |
| regina | 10 |

Il programma di allenamento deve consentire di stimare la situazione di gioco fornendo il punteggio per il bianco e per il nero e di aggiornare la posizione dei pezzi (senza controllarne la correttezza, se non limitatamente al fatto che due pezzi non devono occupare la stessa casella della scacchiera), oltre che aggiungere/rimuovere i pezzi dal gioco. Si richiede:

- un diagramma UML delle classi che modelli lo scenario descritto in modo coerente con un design OO;
- il codice Java che implementi le classi del diagramma UML progettato gestendo le opportune eccezioni;
- il codice di test che inserisca nel gioco un certo numero di pezzi e che simuli vari spostamenti ed eliminazioni visualizzando l'elenco finale dei pezzi sulla scacchiera e il punteggio relativo al bianco e al nero.

*Opzionale*. Integrare la progettazione e il codice in modo che non sia possibile inserire nella scacchiera un numero di pezzi superiore a quello consentito per ciascun tipo e schieramento (8 pedoni, 2 torri, 2 cavalli, 2 alfieri, 1 regina, 1 re). Per una soluzione OO può risultare utile sapere che ogni oggetto Java eredità dalla classe *Object* il metodo *getClass* che restituisce la classe di cui è istanza.

## LABORATORIO

**1** Si vuole simulare un cronometro che effettua la misura del tempo in secondi. Realizzare un diagramma UML e la relativa implementazione in linguaggio Java di una classe Cronometro assumendo che le operazioni possibili siano le seguenti:

| | |
|---|---|
| `init()` | Inizializza un oggetto cronometro azzerando il suo accumulatore di secondi |
| `start()` | Avvia il cronometro che inizia a contare i secondi |
| `stop()` | Ferma il cronometro interrompendo il conteggio dei secondi |
| `show()` | Prevede un parametro che può assumere i valori «s», «m» o «h» in funzione del quale visualizza il tempo conteggiato in secondi dal cronometro rispettivamente in: secondi, minuti:secondi, ore:minuti:secondi |

Per realizzare la classe *Cronometro* si utilizzi il metodo statico *System.currentTimeMillis* che re-

stituisce un valore di tipo `long` che rappresenta il numero di millisecondi trascorsi dalle ore 00:00 del 1/1/1970.
Dotare inoltre la classe di un costruttore che azzera il cronometro e di un metodo *main* che istanzi due oggetti di tipo *Cronometro c1* e *c2* e che visualizzi i risultati forniti dalla seguente sequenza di operazioni:

- *start* di *c1*
- *attesa 5 secondi*
- *show* in secondi del valore di *c1*
- *start* di *c2*
- *attesa 61 secondi*
- *stop* di *c1*
- *show* in minuti:secondi di *c1*
- *attesa 4 secondi*
- *start* di *c1*
- *attesa 2 secondi*
- *stop* di *c2*
- *show* di *c2* in secondi
- *stop* di *c1*
- *show* di *c1* in ore:minuti:secondi

Per realizzare il metodo *main* si implementi il seguente metodo statico avente lo scopo di sospendere l'esecuzione del codice di *n* secondi:

```
public static void attendiNsecondi
   (long n) {
     try  {
            Thread.currentThread().
            sleep(n * 1000);
          }
     catch (InterruptedException
           exception) {
     }
}
```

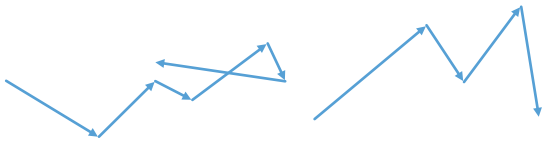
**2** Facendo riferimento alla classe *Vettore* implementata nell'esercizio 2 aggiungere un metodo booleano *interseca* avente come parametro un altro oggetto *Vettore* e che restituisca `true` se i due vettori si intersecano, `false` altrimenti. Per implementare il metodo *interseca* si prenda in considerazione il seguente codice C/C++:

```
struct POINT
       {
         float x ;
         float y;
       };
```

```
struct SEGMENT
       {
         POINT p1;
         POINT p2;
       };

int ccw (POINT p0, POINT p1, POINT p2)
     {
       int dx1, dx2, dy1, dy2;

       dx1 = p1.x - p0.x;
       dy1 = p1.y - p0.y;
       dx2 = p2.x - p0.x;
       dy2 = p2.y - p0.y;

       if (dx1*dy2 > dy1*dx2)
         return +1;
       if (dx1*dy2 < dy1*dx2)
         return -1;
       if ((dx1*dx2 < 0) || (dy1*dy2 < 0))
         return -1;
       if ((dx1*dx1+dy1*dy1) <
           (dx2*dx2+dy2*dy2))
         return +1;
       return 0;
     }

bool intersect(SEGMENT s1, SEGMENT s2)
       {
         return ((ccw(s1.p1,s1.p2,s2.
                  p1)*ccw(s1.p1,s1.
                  p2,s2.p2)) <=0) &&
                ((ccw(s2.p1,s2.p2,s1.
                  p1)*ccw(s2.p1,s2.
                  p2,s1.p2)) <=0);
       }
```

[R. Sedgewick, Algoritmi in C++, Addison-Wesley, 1993]

Nei sistemi CAD (*Computer Aided Design*) e GIS (*Geogrphic Informatin Systems*) una *polyline* è una sequenza ordinata di *N* vettori *v1*, *v2*, *v3*, … *vN* nel piano cartesiano concatenati in modo che l'origine di ciascuno (escluso il primo) coincida con il vertice del precedente, come illustrato negli esempi seguenti che riproducono rispettivanmente una *polyline* intrecciata e una semplice:



Progettare – mediante un diagramma di classe UML – e implementare in linguaggio Java una classe *Polyline* che consenta di definire in sequenza tutti i vettori che costituiscono la *polyline* stessa e inoltre di eseguire le seguenti operazioni:

- calcolare la lunghezza totale della *polyline*;
- determinare se la *polyline* è semplice o intrecciata;
- salvare e ripristinare tutti i vettori che costituiscono la *polyline* in/da un file di tipo testuale.

**3** Si intende realizzare un programma Java che consenta di gestire la navigazione in barca. Il programma deve permettere all'operatore di inserire manualmente le successive posizioni rilevate mediante un dispositivo GPS che restituisce latitudine e longitudine in gradi, primi e secondi e di calcolare in ogni momento la distanza percorsa dall'ultimo punto rilevato e dal punto di partenza. Il programma deve permettere inoltre di inserire una destinazione espressa mediante i valori della latitudine e della longitudine e di calcolare la rotta (cioè l'angolo rispetto al Nord) e la lunghezza del percorso rettilineo che congiunge l'ultimo punto rilevato con il punto di destinazione. Sono richiesti:

- un diagramma UML delle classi;
- un programma in linguaggio Java con interfaccia utente testuale.

Per adattare le coordinate geografiche (latitudine/longitudine) in formato decimale in coordinate cartesiane Nord/Est espresse in metri modificare il codice C/C++ riportato in basso:

```
#define PI                          3.141592653589793

#define WGS84_E2                    0.006694379990197
#define WGS84_E4                    WGS84_E2*WGS84_E2
#define WGS84_E6                    WGS84_E4*WGS84_E2
#define WGS84_SEMI_MAJOR_AXIS       6378137.0
#define WGS84_SEMI_MINOR_AXIS       6356752.314245
#define UTM_LONGITUDE_OF_ORIGIN     3.0/180.0*PI
#define UTM_LATITUDE_OF_ORIGIN      0.0
#define UTM_FALSE_EASTING           500000.0
#define UTM_FALSE_NORTHING_N        0.0
#define UTM_FALSE_NORTHING_S        10000000.0
#define UTM_SCALE_FACTOR            0.9996

double m_calc(double latitude)
{
  return (1.0 - WGS84_E2/4.0 -
          3.0*WGS84_E4/64.0 -
          5.0*WGS84_E6/256.0) * latitude -
          (3.0*WGS84_E2/8.0 +
           3.0*WGS84_E4/32.0 +
           45.0*WGS84_E6/1024.0) * sin(2.0*latitude) +
          (15.0*WGS84_E4/256.0 +
           45.0*WGS84_E6/1024.0) * sin(4.0*latitude) -
          (35.0*WGS84_E6/3072.0) * sin(6.0*latitude);
}
```

```
// INPUT: position in latitude/longitude (WGS84)
// OUTPUT: position in UTM easting/northing (meters)
void GPS2UTM(double latitude, double longitude,
             double* easting, double* northing)
{
  int int_zone;
  double M, M_origin, A, A2, e2_prim, C, T, v;

  int_zone = (int)(longitude/6.0);
  if (longitude < 0)
    int_zone--;
  longitude -= (double)(int_zone)*6.0;
  longitude *= PI/180.0;
  latitude *= PI/180.0;
  M = WGS84_SEMI_MAJOR_AXIS*m_calc(latitude);
  M_origin = WGS84_SEMI_MAJOR_AXIS *
             m_calc(UTM_LATITUDE_OF_ORIGIN);
  A = (longitude - UTM_LONGITUDE_OF_ORIGIN) * cos(latitude);
  A2 = A*A;
  e2_prim = WGS84_E2/(1.0 - WGS84_E2);
  C = e2_prim*pow(cos(latitude),2.0);
  T = tan(latitude);
  T *= T;
  v = WGS84_SEMI_MAJOR_AXIS /
      sqrt(1.0 - WGS84_E2 * pow(sin(latitude),2.0));
  *northing = UTM_SCALE_FACTOR*(M - M_origin + v*tan(latitude) *
              (A2/2.0 + (5.0 - T + 9.0*C + 4.0*C*C)*
              A2*A2/24.0 + (61.0 - 58.0*T + T*T + 600.0*C -
              330.0*e2_prim)*A2*A2*A2/720.0));
  if (latitude < 0)
    *northing += UTM_FALSE_NORTHING_S;
  *easting = UTM_FALSE_EASTING + UTM_SCALE_FACTOR*v *
             (A + (1.0 - T + C)*A2*A/6.0 +
                  (5.0 - 18.0*T + T*T + 72.0*C - 58.0*e2_prim) *
             A2*A2*A/120.0);
  return;
}
```

**4** **Realizzare un'applicazione Java per la gestione di un garage pubblico secondo le specifiche illustrate nella bozza di diagramma UML di** FIGURA **15, da completare.**
**Il garage ha al massimo 25 posti, ognuno dei quali è identificato da un numero di posizione: per motivi di capienza possono entrare solo automobili, furgoni e motociclette. Dopo avere progettato e implementato la gerarchia delle classi *Veicolo*, *Furgone*, *Autovettura* e *Motocicletta* implementare prevedendo le opportune eccezioni la classe *Garage* i cui metodi devono:**

- **gestire l'arrivo di un nuovo veicolo nel garage assegnando un posto di cui restituire il numero di posizione e registrando ora e minuti di arrivo;**
- **gestire l'uscita di un nuovo veicolo dal garage liberando il posto occupato fornito come para-**

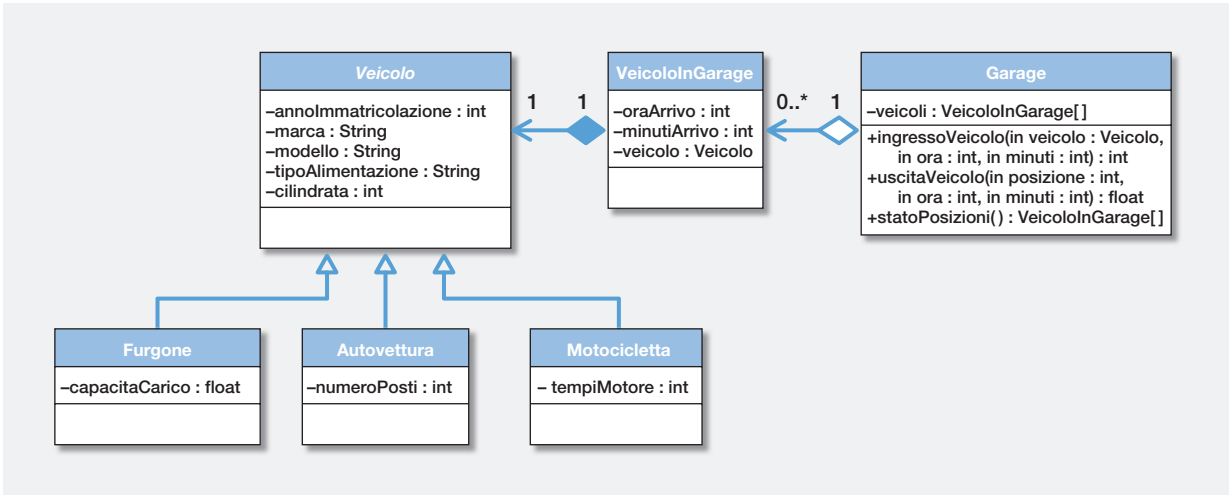



FIGURA **15**

metro e restituendo l'importo da pagare calcolato in base alla seguente tabella (ora e minuti di uscita sono forniti come parametro):

| | |
|---|---|
| Furgone | 2 € per ora o frazione |
| Autovettura | 1,5 € per ora o frazione |
| Motocicletta | 1 € per ora o frazione |

- esportare ai fini della visualizzazione la situazione corrente dei posti del garage indicando i posti liberi e le informazioni relative ai veicoli per quelli occupati.

La classe *Garage* deve essere dotata di un metodo *main* che – mediante un menù interattivo – deve consentire di simulare tutte le operazioni di gestione del garage sopra descritte.