Capitolo 6

Le classi in Java

Andrea Gini

Dopo aver introdotto in modo informale l'uso degli oggetti in Java, è giunto il momento di discutere in modo esteso e formale tutti gli aspetti della programmazione a oggetti. La classe è un costrutto che permette di raggruppare, in un pacchetto indivisibile, un gruppo di variabili e un insieme di metodi che hanno accesso esclusivo a tali variabili:

```
public class nome {
  private tipo attributo1;
  private tipo attributo2;
  ….

  public tipo metodo1() {
    // corpo del metodo
  }
  public tipo metodo2(tipo parametro1 , tipo parametro2) {
    // corpo del metodo
  }
}
```

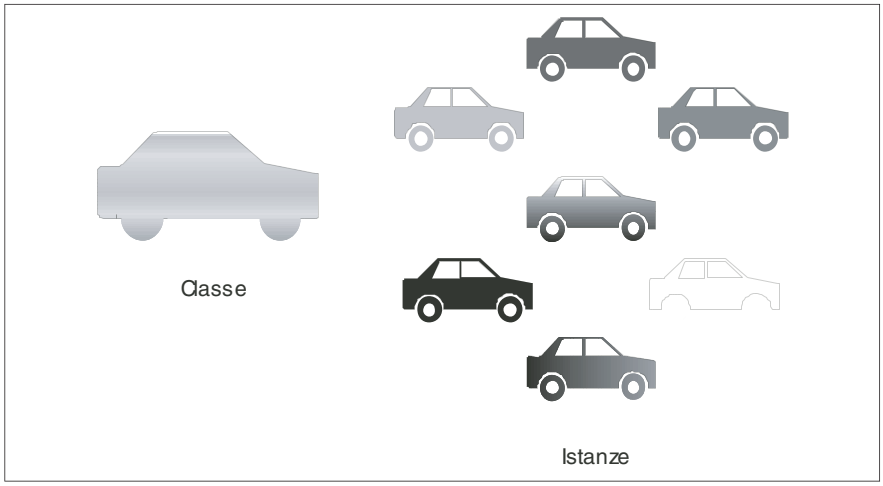
Gli attributi sono variabili accessibili da qualsiasi metodo interno alla classe, ma inaccessibili dall'esterno. I metodi sono procedure che possono operare sia sui dati passati come parametri, sia sugli attributi della classe.

Prima di proseguire oltre, è necessario chiarire la differenza concettuale tra classe, istanza e reference. La classe è la matrice sulla quale vengono prodotti gli oggetti. Essa è come uno stampo, che permette di plasmare un materiale informe per produrre una molteplicità di oggetti

simili tra loro. Per distinguere la matrice dal prodotto, il lessico della programmazione orientata agli oggetti ricorre a due termini: classe e istanza.

La classe, corrispondente al codice sorgente scritto dal programmatore, è presente in singola copia nella memoria del computer. Ogni volta che si ricorre all'operatore `new`, viene creata una nuova istanza della classe, ossia un oggetto di memoria conforme alle specifiche della classe stessa.

Figura 6.1 – *La classe è come uno stampo, capace di generare un'infinità di oggetti simili tra di loro ma dotati nel contempo di attributi univoci.*



La variabile a cui viene associato l'oggetto, d'altra parte, è soltanto un reference: essa è simile a un telecomando con il quale è possibile inviare delle direttive all'oggetto vero e proprio. Ogni volta che si invoca un metodo su una variabile, la variabile in sé non subisce nessun cambiamento: la chiamata a metodo viene inoltrata all'oggetto vero e proprio, che reagisce all'evento nelle modalità previste dal codice presente nella classe. Questa architettura permette di avere più variabili che puntano allo stesso oggetto: l'oggetto è unico, indipendentemente dal numero di reference usati per inoltrare le chiamate a metodo. Il reference può anche non puntare alcun oggetto; in questo caso, esso assume il valore speciale `null`. Al momento della dichiarazione, se non viene specificato diversamente, ogni reference ha valore `null`.

Incapsulamento

L'incapsulamento è un principio di progettazione che prevede che il contenuto informativo di una classe rimanga nascosto all'utente, in modo tale che i metodi siano l'unica via per interagire con un determinato oggetto. Questo approccio presenta almeno due grossi vantaggi: innanzitutto,

esso permette al programmatore di disciplinare l'accesso agli attributi di una classe, in modo da impedire che ne venga fatto un uso sbagliato; in secondo luogo, l'incapsulamento consente a chi utilizza una classe di concentrarsi esclusivamente sull'interfaccia di programmazione, tralasciando ogni aspetto legato all'implementazione.

Quasi tutti gli oggetti del mondo reale presentano questa stessa proprietà. Si pensi a un telefono cellulare: per capirne il funzionamento interno è necessaria una laurea in ingegneria elettronica, mentre bastano pochi minuti per imparare a comporre un numero e mettersi in contatto con qualcuno. Questa filosofia di progettazione ha due importanti implicazioni: per prima cosa, imparare a *usare* un oggetto è di norma più facile che capire come funziona; inoltre, una volta appreso il funzionamento comune a una classe di oggetti (automobili, telefoni, forbici ecc.), diventa facile utilizzare qualsiasi altro oggetto di quella particolare classe (una Fiat Punto, una Renault Clio e così via).

Dichiarazione di metodo

La struttura standard di un metodo senza parametri è la seguente:

```
public tipo nome() {
  istruzione1;
  istruzione2;
  ....
  instruzioneN;

  return returnVal;
}
```

La sequenza public *tipo nome*() viene detta *firma* del metodo (in inglese, *signature*). Come tipo è possibile specificare uno qualsiasi dei tipi primitivi visti in precedenza (int, long, short, byte, char, boolean, float e double) o il tipo di un oggetto. Esiste anche il tipo speciale void, da usare quando il metodo non restituisce nessun valore. Tutti i metodi con tipo diverso da void devono terminare con l'istruzione return, seguita da un valore o da una variabile dello stesso tipo presente nella firma del metodo.

```
public class Mela {
    private int larghezza;
    private int lunghezza;
    private Color colore;

    public int getLarghezza() {
      return larghezza;                    // restituisce un intero
    }
    public int getLunghezza() {
      return lunghezza;                    // restituisce un intero
    }
    public Color getColore() {
```

```
    return colore;                          // restituisce un oggetto di tipo Color
  }
}
```

Dichiarazione di metodo con parametri

L'uso dei metodi può essere potenziato notevolmente grazie ai parametri. I parametri permettono di generalizzare i metodi, in modo da estenderne l'uso a diversi contesti a seconda del valore dei parametri stessi. Per dichiarare un metodo parametrizzato, è necessario seguire una sintassi un po' diversa da quella vista in precedenza: dopo il nome del metodo, tra una coppia di parentesi tonde, bisogna specificare una serie di coppie tipo-nome, in maniera simile a come si fa con le dichiarazioni di variabile:

```
public tipo nome(tipo1 nome1,tipo2 nome2, .... , tipoN nomeN) {
  istruzione1;
  istruzione2;
  ....
  instruzioneN;
}
```

All'interno del metodo i parametri possono essere trattati come normali variabili. Nell'esempio seguente viene definito un metodo min, che prende in input una coppia di interi e restituisce il minore tra i due:

```
public int min(int n1 , int n2) {
  if ( n1 < n2 )
    return n1
  else
    return n2;
}
```

Chiamata a metodo: la dot notation

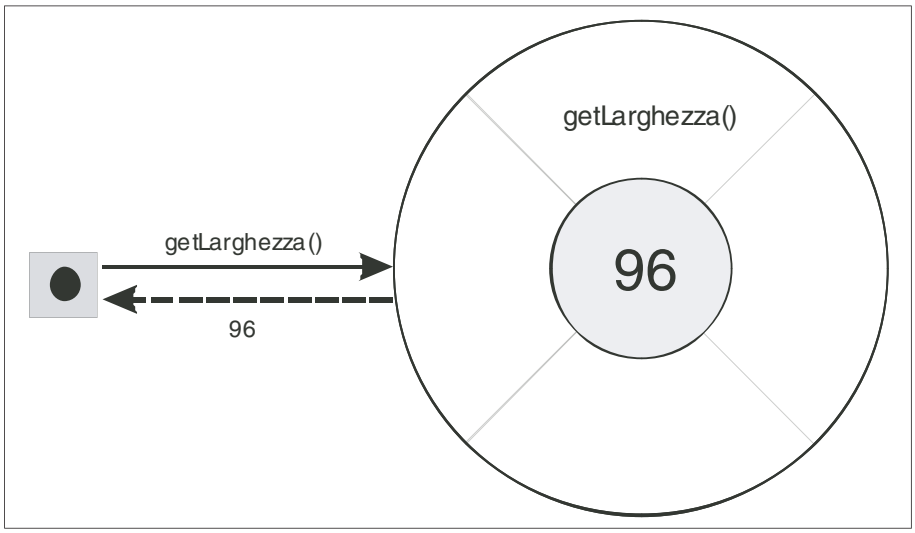
Per invocare un metodo su un oggetto, è necessario applicare l'operatore . su un reference che punti all'oggetto, specificando il nome del metodo da chiamare e gli eventuali parametri:

```
Mela m = new Mela();
Int larghezza = m.getLarghezza();          // Chiamata a metodo
```

Se il metodo invocato fa parte della classe chiamante, la chiamata non deve essere preceduta da alcun identificatore.

La chiamata a metodo denota un protocollo a scambio di messaggi: chi detiene il reference invia un messaggio di chiamata a metodo, specificando il nome del metodo da invocare e l'elenco dei parametri; l'oggetto chiamato, come risposta, restituisce un valore di ritorno.

Figura 6.2 – *Scambio di messaggi durante una chiamata a metodo.*



L'operatore . può essere usato anche per accedere agli attributi public di un oggetto:

```
public class Point {                          // classe con attributi pubblici
  public int x;
  public int y;
}
Point p = new Point();
p.x = 10;                                     // assegnamento sull'attributo x della classe Point
```

La notazione che fa uso dell'operatore . prende comunemente il nome di dot notation, o notazione puntata. Nei prossimi paragrafi verranno illustrate le regole di utilizzo della dot notation per accedere alle classi contenute all'interno di package, e a metodi e ad attributi statici. È possibile applicare l'operatore . a cascata sui valori restituiti da un metodo o da un attributo, ottenendo in tal modo l'inoltro della chiamata all'oggetto restituito dalla chiamata precedente. Per esempio, la chiamata:

```
// chiama il metodo darker() sul valore restituito dal
// metodo getColor() dell'oggetto riferito dalla variabile m
Color c = m.getColor().darker();
```

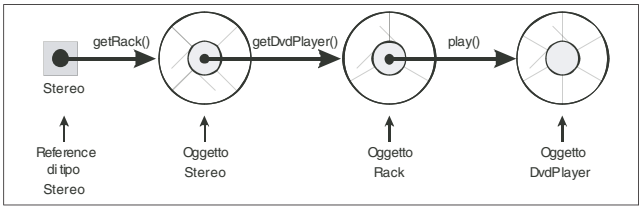
è equivalente alla seguente serie di istruzioni:

```
Color coloreMela = m.getColor();
Color c = coloreMela.darker();
```

Simili catene possono avere una lunghezza indefinita, e permettono di operare su qualsiasi oggetto accessibile per via diretta o indiretta a partire da un reference. Per esempio, per far partire un ipotetico lettore DVD contenuto nel rack di un impianto stereo, si utilizzerà una catena di chiamate di questo tipo:

```
stereo.getRack().getDvdPlayer().play();
```

Figura 6.3 – *Un esempio di chiamata di metodo a cascata.*



Parametro attuale e parametro formale

L'uso di metodi con parametri introduce una problematica concettuale piuttosto sottile: cosa succede quando, nel corso di una chiamata a metodo, si passa come parametro una variabile? Si osservi il seguente programma e si cerchi di immaginare come procede l'esecuzione a partire dal metodo `m1()`:

```
public class ProvaParametri {

  public void m1() {
    int a = 5;
    m2(a);  // chiama m2 usando la variabile a come parametro
    System.out.println(a);
  }
  public void m2(int i) {
    i = 10;
  }
}
```

Cosa succede alla variabile a nel momento in cui viene passata come parametro al metodo `m2()`? Per poter chiarire come vengano trattati casi come questo è necessario illustrare la differenza tra parametro formale e parametro attuale. I parametri definiti nella firma di un metodo vengono detti *parametri formali*. Tali parametri, all'interno del metodo, vengono trattati come delle variabili, la cui visibilità termina alla fine del metodo stesso. La variabile che viene passata come parametro di un metodo viene detta invece *parametro attuale*: esse hanno significato nel contesto del metodo chiamante, e conservano inalterato il loro valore nonostante la chiamata.

Come verrà spiegato nel prossimo paragrafo, nel caso di passaggio di vettori o di oggetti si verifica un fenomeno che può sembrare un'eccezione alla regola secondo cui il parametro attuale non possa essere modificato dalle istruzioni di un metodo. Si raccomanda di leggere con attenzione il paragrafo al fine di capire questa importante distinzione.

```
public class Esempio {

  public void m1() {
    int p1 = 10;
    float p2 = 10.5F;
    Boolean p3 = true;
    m2(p1,p2,p3);                                  // p1, p2 e p3 sono parametri attuali
  }
  public void m2(int i, float f ,  boolean b ) {   // i, f e b sono parametri formali
     ....        // corpo del metodo
  }
}
```

Il compilatore richiede che le variabili fornite come parametri attuali siano dello stesso tipo di quelle richieste dai parametri formali presenti nella firma del metodo, nel rispetto delle regole di casting. Pertanto, se un metodo richiede un parametro long, non verranno segnalati errori se si effettua una chiamata con un int. Nel caso opposto (parametro formale int, parametro attuale long) il compilatore segnalerà un errore, che potrà essere evitato solo ricorrendo al casting.

Passaggio di parametri by value e by ref

All'atto di chiamare un metodo, l'interprete Java copia il valore dei parametri attuali nelle variabili corrispondenti ai rispettivi parametri formali. Tale comportamento viene detto "passaggio di parametri by value", o "per valore". Mediante il passaggio di parametri by value, si può essere sicuri che qualsiasi modifica ai parametri formali non andrà ad alterare il contenuto delle variabili usate come parametri attuali. Si provi a riconsiderare il programma visto nel paragrafo precedente:

```
public class ProvaParametri {

  public void m1() {
    int a = 5;
    m2(a);  // chiama m2 usando la variabile a come parametro
    System.out.println(a);
  }
  public void m2(int i) {
    i = 10;
  }
}
```

Ora è possibile affermare con sicurezza che la variabile a, utilizzata come parametro attuale nella chiamata a m2(), non verrà alterata in alcun modo dalle istruzioni presenti all'interno del metodo m2() stesso.

Cosa succede invece quando, come parametro, viene utilizzato un array o un altro oggetto invece di un tipo primitivo? Si provi a osservare il seguente esempio:

```
public class Parametri2 {
  public void m1() {
    int[] i = new int[10];
    i[0] = 5;
     m2(i);
     System.out.println(i[0]);
  }
  public void m2(nt[] i) {                    // Il parametro è un oggetto
     i[0] = 10;                               // Il metodo modifica l'oggetto
  }
}
```

Diversamente dal programma precedente, questo esempio restituisce in output il valore 10, quello assegnato al primo elemento dell'array all'interno del metodo. Cosa è successo? Al contrario di quanto avviene con le variabili dei tipi primitivi (int, long ecc.), quando si passa a un metodo un array (o qualsiasi altro oggetto), esso viene passato "by ref", o "per riferimento": in altre parole, il parametro formale i presente nel metodo non punta a una copia dell'array definito nel metodo main, ma punta allo stesso identico array, e qualsiasi modifica effettuata su di esso all'interno del metodo andrà ad agire sull'array stesso. Si noti che questo comportamento non è in contraddizione con la regola descritta nel paragrafo precedente, secondo la quale il parametro attuale non può essere modificato dalle istruzioni presenti all'interno di un metodo. Il parametro attuale, che in questo caso è un reference, non può essere modificato in alcun modo dal codice del metodo: a cambiare in questi casi è l'oggetto puntato dal reference, non il reference stesso.

Alcuni linguaggi, come C, C++, Pascal e Visual Basic, permettono di specificare per ogni singolo parametro se si desidera che venga passato per riferimento o per valore. Questa possibilità, sebbene utile in teoria, finisce inevitabilmente per provocare confusione. Per questo, durante la progettazione di Java si è deciso di imporre per default il passaggio by value per tutti i tipi primitivi e il passaggio by ref di array e oggetti.

Visibilità delle variabili: variabili locali

Durante la realizzazione di un metodo occorre fare grande attenzione alla visibilità delle variabili, ossia quello che nei testi in lingua inglese viene normalmente definito *scope*. Quando si dichiara una variabile all'interno di un metodo, essa risulta visibile solamente all'interno del metodo stesso: per questo, le variabili dichiarate all'interno di un metodo prendono il nome di *variabili locali*. Ecco un esempio:

```
public class Esempio1 {

  public void metodo1() {
    int a = 10;
    System.out.println(a);
  }
  public void metodo2() {
    a = a++; //ERRORE!
    System.out.println(a);
  }
}
```

Il primo dei due metodi è corretto: esso dichiara la variabile intera a, la inizializza a 10 e la stampa a schermo. Nel secondo metodo c'è invece un errore: si cerca di far riferimento alla variabile a che in questo contesto non è stata definita. Se si corregge il secondo metodo nel modo seguente:

```
  static void metodo2() {
    int a = 10;
    a++;
    System.out.println(a);
  }
```

si ottiene un metodo corretto, che dichiara una nuova variabile a (diversa, si faccia attenzione, dalla omonima variabile a presente nel metodo1), la inizializza a 10, la incrementa a 11 e la stampa.

Ricorsione

Il codice di un metodo può contenere chiamate a qualsiasi altro metodo: cosa succede nel caso limite in cui un metodo contiene una chiamata a sé stesso? In casi come questo si ottiene un'esecuzione ricorsiva. Per capire l'uso e l'utilità della ricorsione, verrà illustrato un esempio divenuto ormai classico: la funzione fattoriale. La funzione fattoriale viene tipicamente definita per ricorsione, dicendo che il fattoriale di 0 è 1 e che il fattoriale di un qualsiasi numero n intero è pari a n moltiplicato per il fattoriale del numero n - 1. Pertanto il fattoriale di 0 è 1, il fattoriale di 3 è 6 (3*2*1), il fattoriale di 4 è 24 (1*2*3*4) e così via. Grazie alla ricorsione è possibile realizzare un metodo che calcola il fattoriale applicando esattamente la definizione standard:

```
public int fattoriale(int n) {
  if(n=0)
    return 1;
  else
    return n * fattoriale(n - 1);
}
```

Per eseguire correttamente le procedure ricorsive, l'interprete Java ricorre a una speciale memoria a pila (stack, in inglese), che ha la particolarità di comportarsi come la pila di piatti di un ristorante: il primo piatto che viene preso (tipicamente quello in altro) è anche l'ultimo che vi era stato posto. A causa di questo particolare comportamento, si dice che una pila è una struttura dati di tipo LIFO, acronimo inglese il cui significato è "Last In First Out", ossia "l'ultimo a entrare è il primo a uscire". La ricorsione permette di fornire una soluzione elegante a una serie di problemi, ma proprio a causa del ricorso allo stack, questa eleganza viene pagata con una minore efficienza di esecuzione.

Costruttore

Il costruttore è uno speciale metodo che ha lo stesso nome della classe e che è privo di valore di ritorno. Il costruttore svolge un compito fondamentale: esso permette infatti di inizializzare i principali attributi di un oggetto durante la fase di creazione, ricorrendo a un'unica operazione:

```
public class Mela {
  private int larghezza;
  private int lunghezza;
  private Color colore;

  public Mela(int lungh , int largh , Color c) {          // Costruttore della classe Mela
    lunghezza = lung;
    larghezza = largh;
    colore = c;
  }
   }
```

Il costruttore svolge un ruolo cruciale nella programmazione a oggetti: grazie a esso, infatti, è possibile mettere in atto le politiche di incapsulamento più rigide. Per esempio, se si crea all'interno di una classe un attributo privato dotato di metodi get ma privo di metodi set, si crea una situazione in cui il costruttore rappresenta l'unica via per inizializzare un simile attributo. Questa tecnica permette di creare attributi immutabili a sola lettura, una scelta che può rivelarsi utile in numerose circostanze.

Il costruttore viene invocato in fase di creazione tramite l'operatore `new`; la chiamata deve includere tutti i parametri richiesti:

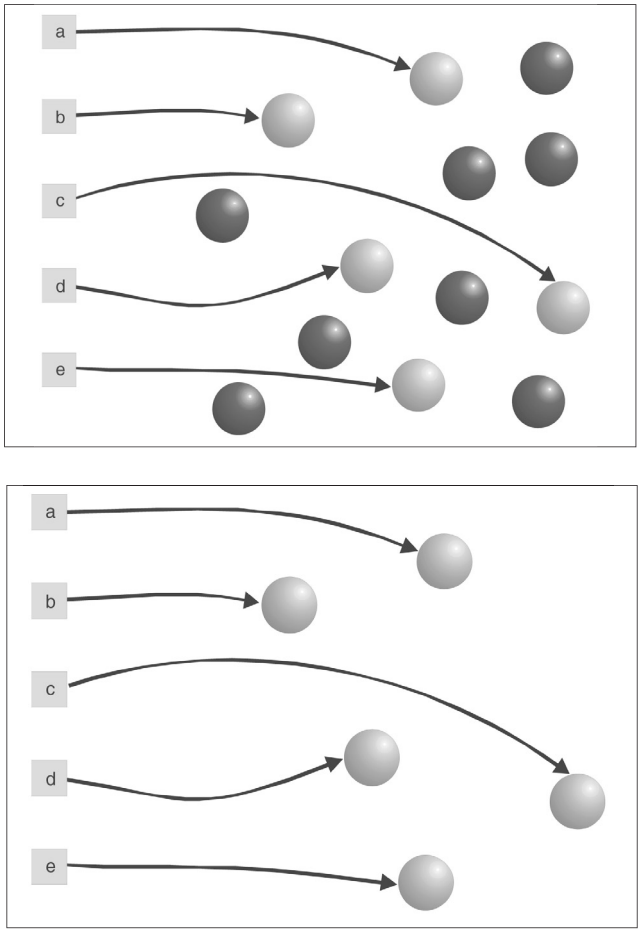
```
Mela m = new Mela(96,100,new Color("Red");
```

Ovviamente, non esiste un modo per chiamare il costruttore di una classe in un momento diverso dalla sua creazione. Il costruttore è una componente indispensabile della classe: se il programmatore non ne definisce uno esplicitamente, il compilatore aggiunge automaticamente il costruttore privo di parametri, detto costruttore di default.

Finalizzatori e garbage collection

Ogni oggetto occupa fisicamente una determinata porzione nella memoria del calcolatore. Dal momento che la memoria è una risorsa finita, è importante capire dove vanno a finire gli oggetti dopo aver svolto il compito per il quale erano stati creati. La gestione della memoria è uno dei punti di forza di Java: la pulizia della memoria dagli oggetti non più utilizzati viene svolta automaticamente durante l'esecuzione da un apposito strumento, detto Garbage Collector (raccoglitore di rifiuti). Non appena la memoria disponibile scende al di sotto di una certa soglia, il Garbage Collector si attiva automaticamente, va in cerca di tutti gli oggetti non più referenziati e li distrugge, liberando la memoria che essi occupavano.

Figure 6.4 e 6.5 – *Rappresentazione della memoria prima e dopo l'intervento del Garbage Collector.*



Prima di procedere alla distruzione, il Garbage Collector invoca il metodo finalize() sull'oggetto da rimuovere. Il programmatore può dichiarare tale metodo nelle proprie classi, e inserirvi delle istruzioni da eseguire subito prima della distruzione dell'oggetto:

```
public void finalize() {
  att1 = null;
  att2 = null;
  file.close();
}
```

I finalizzatori sono utili nei contesti in cui la distruzione di una classe comporta operazioni che non vengono compiute automaticamente dal Garbage Collector, come la chiusura di file e di connessioni a database o più in generale il rilascio di risorse di sistema. In tutti gli altri contesti, esso risulta praticamente inutile, e pertanto si sconsiglia di dichiararlo nelle proprie classi.

Convenzioni di naming

Esistono convenzioni sulle modalità di attribuzione di nomi a variabili, metodi e attributi. Tali convenzioni sono state formalizzate da Sun Microsystems in un documento denominato Code Conventions for the Java Programming Language (http://java.sun.com/docs/codeconv/), che specifica tra l'altro le regole per la disposizione di classi e metodi all'interno di un file, le regole di indentazione e quelle per la stesura dei commenti. Le principali regole di naming sono tre:

- I nomi delle classi devono iniziare con una lettera maiuscola.
- Metodi, variabili e attributi iniziano con una lettera minuscola.
- I nomi composti vengono dichiarati secondo la convenzione CamelCase: le parole vengono riportate per esteso in minuscolo, una di seguito all'altra senza caratteri di separazione, utilizzando un carattere maiuscolo come lettera iniziale di ogni parola.

Queste semplici regole, universalmente utilizzate nel mondo Java, favoriscono una certa uniformità al codice scritto a più mani, e garantiscono un'ottima leggibilità. Ecco di seguito tre identificatori che seguono le convenzioni di naming appena elencate:

NomeClasse nomeMetodo() nomeVariabile

Ereditarietà

L'ereditarietà è una proprietà fondamentale dei linguaggi orientati agli oggetti. Grazie all'ereditarietà è possibile definire una classe come figlia (o sottoclasse) di una classe già esistente, in modo da estenderne il comportamento. La classe figlia "eredita" tutti i metodi e gli attributi

della superclasse (detta anche classe padre), e in tal modo ne acquisisce il comportamento.

Si provi a immaginare una classe Bicicletta, descritta di seguito in pseudo codice:

```
public class Bicicletta {

   public Color colore;

   public Color getColore() {
     return colore;
   }
   public void muoviti() {
     sollevaCavalletto();
     muoviPedali();
   }
   public void frena() {
     premiGanasceSuRuota();
   }
   public void curvaDestra() {
     giraManubrioVersoDestra();
   }
   public void curvaSinistra() {
     giraManubrioVersoSinistra();
   }
}
```

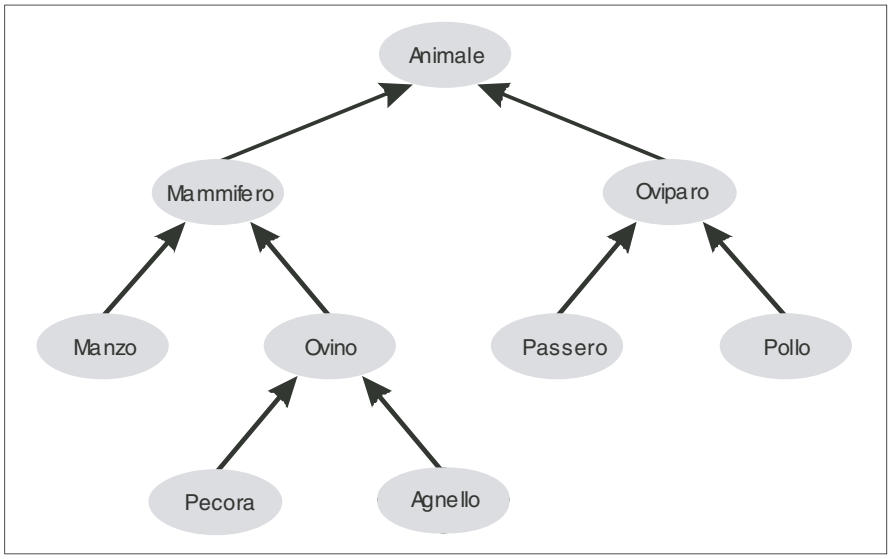
Essa è caratterizzata dall'attributo Colore e dai metodi muoviti(), frena(), curvaDestra() e curvaSinistra(). Un motorino è un mezzo simile a una bicicletta (si pensi a un motorino tipo Ciao, dotato di pedali). Esso, tuttavia, è dotato di alcuni attributi aggiuntivi, come la cilindrata e il numero di targa, e di alcuni metodi non presenti nella bicicletta, come per esempio accendiMotore():

```
public class Motorino extends Bicicletta {

   private String targa;
   private int cilindrata;

   public int getCilindrata() {
      return cilindrata;
   }
   public String getTarga() {
     return targa;
   }
   public void accendiMotore() {
     inserisciMiscelaNelCilindro();
     accendiCandela();
   }
}
```

La classe Motorino viene definita, grazie alla direttiva extends, come sottoclasse di Bicicletta. Di conseguenza, la classe Motorino eredita in modo automatico tutti gli attributi e i metodi di Bicicletta, senza la necessità di riscriverli.

L'ereditarietà permette di creare gerarchie di classi di profondità arbitraria, simili ad alberi genealogici, in cui il comportamento della classe in cima all'albero viene gradualmente specializzato dalle sottoclassi. Ogni classe può discendere da un'unica superclasse, mentre non c'è limite al numero di sottoclassi o alla profondità della derivazione.

Figura 6.6 – *Un esempio di gerarchia di classi.*



Dal codice di una classe è possibile accedere a metodi e attributi pubblici di qualsiasi superclasse, ma non a quelli privati. Il modificatore protected, che verrà studiato meglio in seguito, permette di definire metodi e attributi accessibili solo dalle sottoclassi.

In Java ogni classe in cui non sia definito esplicitamente un padre, viene automaticamente considerato sottoclasse di Object, che è pertanto il capostipite di tutte le classi Java.

Overloading

All'interno di una classe è possibile definire più volte un metodo, in modo da adeguarlo a contesti di utilizzo differenti. Due metodi con lo stesso nome possono coesistere in una classe a due condizioni: devono avere lo stesso tipo di ritorno e devono presentare differenze nel numero e nel tipo dei parametri

All'interno di un metodo è sempre possibile invocare un metodo omonimo: spesso, infatti, una famiglia di metodi con lo stesso nome si limita a fornire differenti vie di accesso programmatiche a un'unica logica di base. In un'ipotetica classe Quadrilatero, dotata di un attributo dimensione, è possibile definire tre metodi setter che permettano di impostare l'attributo, specificando un apposito oggetto Dimension, una coppia di interi o nessun parametro per impostare valori di default:

```
Public class Quadrilatero {
  private Dimension dimensione;

  // metodo di base
  public void setSize(Dimension d) {
    dimensione = d;
  }
  // accesso con interi
  public void setSize(int width,int height) {
    SetSize(new Dimension(width,height));
  }
  // impostazione di default
  public void setSize() {
    setSize(new Dimension(100,100));
  }
}
```

In questo esempio si può notare che solamente il primo dei tre metodi setSize() effettua la modifica diretta dell'attributo dimensione; le altre due versioni del metodo si limitano a riformulare la chiamata in modo da renderla adatta al metodo di base.

È possibile effettuare anche l'overloading dei costruttori:

```
public class MyClass {
  private int att1;
  private int att2;

  public MyClass() {
    att1 = 0;
     att2 = 0;
  }
  public MyClass(int a1 , int a2) {
    att1 = a1;
     att2 = a2;
  }
}
```

Per raggiungere il pieno controllo in scenari che prevedano l'overloading di metodi e costruttori, è necessario comprendere l'uso degli identificatori this e super, che verranno illustrati nei prossimi paragrafi.

Overriding

Grazie all'ereditarietà è possibile estendere il comportamento di una classe sia aggiungendo nuovi metodi sia ridefinendo metodi già dichiarati nella superclasse. Quest'ultima possibilità prende il nome di overriding, ed è un'ulteriore proprietà dei linguaggi a oggetti. Per mettere in atto l'overriding, è sufficiente dichiarare un metodo di cui esiste già un'implementazione in una superclasse: la nuova implementazione acquisirà automaticamente la precedente, sovrascrivendone il comportamento. Nell'esempio del motorino, è naturale pensare alla necessità di fornire una nuova implementazione del metodo muoviti(), in modo tale da adeguarlo allo scenario di un mezzo motorizzato:

```
public class Motorino extends Bicicletta {

   private String targa;
   private int cilindrata;

   public int getCilindrata() {
      return cilindrata;
   }
   public String getTarga() {
     return targa;
   }
   public void accendiMotore() {
     inserisciMiscelaNelCilindro();
     accendiCandela();
   }
   public void muoviti() {
     accendiMotore();
     premiFrizione();
     innestaMarcia(1);
     rilasciaFrizione();
     ....
   }
}
```

La classe Motorino, presente in quest'ultimo esempio, possiede gli stessi metodi e attributi della classe Bicicletta, più alcuni metodi definiti ex novo e una versione nuova del metodo muoviti(), già dichiarato nella superclasse.

Identificatori speciali this e super

L'identificatore this è un puntatore speciale alla classe che costituisce l'attuale contesto di programmazione. Grazie a this è possibile accedere a qualsiasi metodo o attributo della classe stessa mediante un'espressione del tipo:

```
this.methodo();
```

L'uso di this è indispensabile quando ci si trova a dover distinguere tra un attributo e una variabile con lo stesso nome, come avviene spesso nei metodi setter e nei costruttori:

```
public setAtt1(int att1) {
  this.att1 = att1;  // assegna il valore della variabile locale att1 all'attributo omonimo
}
```

L'identificatore this può essere usato anche per richiamare un costruttore. In questo caso, la parola this deve essere seguita dai parametri richiesti dal costruttore in questione racchiusi tra parentesi, e deve per forza comparire come prima istruzione di un altro costruttore:

```
public class MyClass {
  private int att1;
  private int att2;

  public MyClass() {
    this(0,0);  // chiama il secondo costruttore con i parametri di default
  }
  public MyClass(int a1 , int a2) {
    att1 = a1;
     att2 = a2;
  }
}
```

L'identificatore super ha un uso simile a this ma, a differenza di quest'ultimo, invece di far riferimento alla classe di lavoro si riferisce alla superclasse. Tramite super è possibile invocare la versione originale di un metodo sovrascritto, altrimenti inaccessibile:

```
metodo();                    // chiamata a un metodo della classe
super.metodo();              // chiamata al metodo omonimo presente nella superclasse
```

Spesso i metodi sovrascritti sono estensioni dei metodi equivalenti della superclasse; in questi casi, si può utilizzare super in modo da richiamare la versione precedente del metodo, e quindi aggiungere di seguito le istruzioni nuove:

```
public void metodo1() {
  super.metodo1();  // chiamata a metodo1 della superclasse
  // nuove istruzioni che estendono
  // il comportamento del metodo
  // omonimo della superclasse
}
```

L'uso principale di super è nei costruttori, che devono necessariamente estendere un costruttore della superclasse. Se non viene specificato diversamente, il costruttore di una classe viene considerato estensione del costruttore di default della superclasse (quello privo di parametri). Se si desidera un comportamento differente, o se addirittura la superclasse

non dispone di un costruttore di default, occorre invocare in modo esplicito il supercostruttore desiderato.

```
public class Motorino extends Bicicletta {

  private String targa;
  private int cilindrata;

  public Motorino(Color colore, String targa, int cilindrata) {
                super(colore);
                this.targa = targa;
                this.cilindrata = cilindrata;
  }
  ....
}
```

Anche in quest'ultimo caso, la chiamata al supercostruttore deve precedere qualsiasi altra istruzione.

Binding dinamico

Ogni classe denota un tipo. Tuttavia, come conseguenza dell'ereditarietà, ogni classe ha come tipo sia il proprio sia quello di tutte le sue superclassi. Grazie a questa proprietà, una classe può essere utilizzata in qualsiasi contesto valido per una qualunque delle sue superclassi. Per esempio, il metodo:

```
public void parcheggia(Bicicletta b) {
  b.muovi();
  b.giraSinistra();
  ....
  b.frena();
}
```

può lavorare sia su oggetti di tipo Bicicletta sia su quelli di tipo Motorino, una cosa abbastanza intuitiva anche nel mondo reale (un parcheggio per automobili può andare bene anche per i taxi, che in fin dei conti sono pur sempre automobili).

Per questo stesso motivo, è possibile formulare una dichiarazione del tipo:

```
Bicicletta b = new Motorino();
```

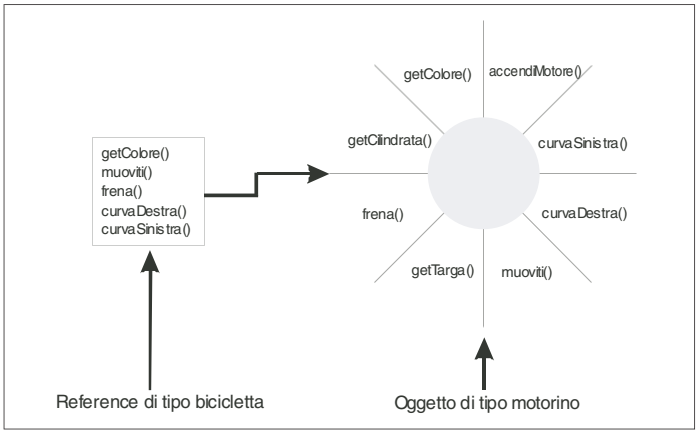
In questo esempio viene creato un oggetto di tipo Motorino, abbinato a un reference di tipo Bicicletta. Quale effetto produce la chiamata sulla variabile b di un metodo cui sia stato applicato l'overriding? Verrà invocato il metodo definito nella classe padre o quello presente nella sottoclasse?

In Java i metodi sono legati al tipo dell'istanza, non a quello del reference: in altre parole, nonostante il reference sia di tipo Bicicletta, le chiamate avranno sempre l'effetto di invocare il metodo valido per il tipo dell'istanza in questione. Il binding è l'associazione di un metodo alla rispettiva classe: in Java, il binding viene effettuato durante l'esecuzione, in modo tale da garantire che su ogni oggetto venga invocato il metodo corrispondente al tipo. Contrariamente a quanto avviene con linguaggi come il C++, non esiste alcun modo per richiamare su un oggetto un metodo appartenente alla superclasse se questo è stato sovrascritto. Per questa ragione, la chiamata:

```
b.muovi();
```

chiamerà sull'oggetto referenziato dalla variabile b il metodo muovi() di Motorino, dal momento che l'oggetto in questione è di tipo Motorino.

Figura 6.7 – *Un reference permette di accedere esclusivamente ai metodi dell'oggetto denotati dal proprio tipo.*



Upcasting, downcasting e operatore instanceof

Come nei tipi primitivi, l'upcasting, o promozione, è automatico. È sempre possibile referenziare un oggetto con una variabile il cui tipo è quello di una delle sue superclassi:

```
Motorino m = new Motorino();
Bicicletta b = m; // downcasting
```

Quando invece si dispone di un oggetto di un certo tipo referenziato da una variabile di un supertipo, e si desidera passare il reference a un'altra variabile di un tipo più specializzato, è necessario ricorrere all'operatore di casting:

```
 Bicicletta b = new Motorino();
Motorino m = (Motorino)b;
```

Il casting è sempre sconsigliato, dal momento che viola il principio del polimorfismo. Negli scenari in cui si desidera operare un casting, è possibile ricorrere all'operatore instanceof, che permette di verificare il reale tipo di un oggetto:

```
If (b instanceof Motorino)
  m  = (Motorino)b;
```

Equals e operatore ==

Come si è già visto nel caso delle stringhe, quando si lavora con gli oggetti è necessario prestare una grande attenzione a come si usa l'operatore di uguaglianza ==. L'operatore di uguaglianza permette di verificare l'identità tra due reference, che si verifica nel caso in cui essi facciano riferimento allo stesso oggetto in memoria. Qualora si desideri testare la somiglianza tra due oggetti, ossia la circostanza in cui due oggetti distinti presentano lo stesso stato, è necessario ricorrere al metodo equals(). Per esempio, in un caso del tipo:

```
Integer i1 = new Integer(12);
Integer i2 = new Integer(12);
```

Il test i1 == i2 darà esito false, in quanto le due variabili fanno riferimento a due oggetti distinti; al contrario, l'espressione i1.equals(i2) risulterà vera, dal momento che i due oggetti hanno lo stesso identico stato.

Figura 6.8 – *Esempi di somiglianza e di identità tra oggetti.*