

## Capitolo 9

# Astrazione ed ereditarietà

9.1	Uso delle classi	289
9.2	Astrazione mediante interfacce	291
9.3	Ereditarietà	300
9.4	Metodi astratti e classi astratte	306
9.5	Case Study 4: Etichette veterinarie	307

### 9.1 Uso delle classi

---

La programmazione orientata agli oggetti fornisce molti meccanismi che rendono più potente l'uso delle classi. Fino a questo punto abbiamo ampiamente usato le classi per quella che si chiama **composizione**, cioè per creare in una classe oggetti che siano istanze di un'altra classe. Un esempio di composizione si trova nell'array di partite che viene tenuto dalla classe `Coffee` dell'Esempio 8.2. Altre due tecniche che stiamo per descrivere sono l'**astrazione** e l'**ereditarietà**.

#### Astrazione

L'**astrazione** è un meccanismo che permette di concentrarsi sugli aspetti essenziali di una classe o un di metodo (cioè, sul suo comportamento e sul modo in cui si interfaccia con il resto del sistema), rinviando l'analisi dei dettagli ad una fase successiva. Java ha due costrutti per ottenere l'astrazione, le interfacce e le classi astratte, che verranno trattati in seguito.

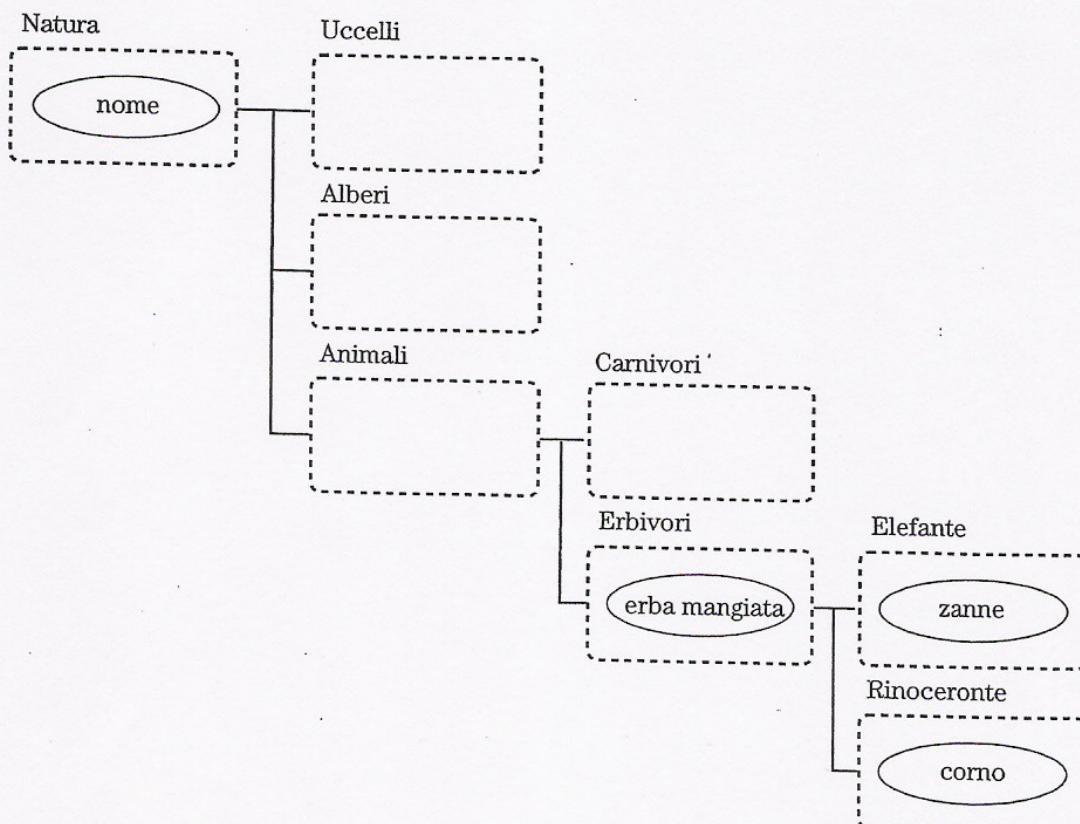
Negli esempi precedenti, ciascuna classe rappresentava la descrizione concreta di oggetti reali, come biglietti, voti, varietà di caffè, eccetera. Se volessimo effettuare un'operazione su tutte queste classi, come ad esempio l'ordinamento di un array di biglietti, o di un array di voti, o di un array di caffè, avremmo bisogno in realtà di scrivere tre diversi metodi di ordinamento, uno per ogni classe. L'astrazione permette di ridurre questo tipo di

ridondanze e di consentire il riutilizzo del codice, due obiettivi che ci eravamo prefissi fin dall'inizio.

## Ereditarietà

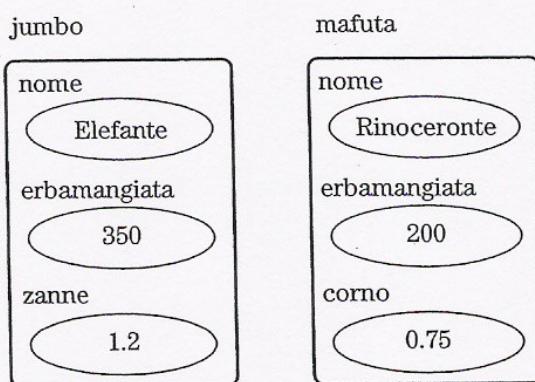
Mediante l'**ereditarietà** possiamo concentrarci sulle definizioni delle classi che conosciamo, lasciando aperta la strada alla possibilità di definirne in seguito nuove versioni. Queste nuove versioni potranno ereditare le proprietà e le caratteristiche delle classi originali, aggiungendone nuove e, come conseguenza, potranno essere molto più piccole e concise. Così facendo, sarà possibile costruire una gerarchia di classi in cui le modifiche e le estensioni siano concentrate nei punti in cui sono realmente necessarie. Siccome tutte le classi della gerarchia appartengono alla stessa famiglia, sono tutte dello stesso tipo rispetto al meccanismo di tipizzazione di Java, e ciascuna potrebbe essere usata dove ne sia richiesta un'altra, come spiegheremo in dettaglio fra poco.

Consideriamo come esempio il diagramma della Figura 9.1. Nel Capitolo 1, abbiamo visto che la classe Natura è costituita da animali, alberi e uccelli. Gli animali possono essere erbivori o carnivori; fra i possibili erbivori ci sono elefanti e rinoceronti<sup>1</sup>. Il campo che contiene il nome



**Figura 9.1** Gerarchia Natura.

<sup>1</sup> I termini, che nel Capitolo 1 erano stati mantenuti nella dizione inglese per coerenza con il sito Web, vengono invece in questo contesto tradotti per favorire la comprensibilità.



Elefante jumbo ("Elefante"), 350, 1.2);  
 Rinoceronte mafuta ("Rinoceronte", 200, 0.75);

**Figura 9.2** Dichiarazione di oggetti nella gerarchia Natura.

degli animali potrebbe stare nella classe Natura, perché comunque non solo gli animali, ma anche gli alberi e gli uccelli avranno un nome. Analogamente il metodo che visualizza il nome potrebbe essere messo a questo livello. Se disponiamo di un oggetto "elefante" e vogliamo visualizzarne il nome, il metodo che potremo invocare sarebbe proprio quello definito nella classe Natura.

Però ci sono alcune proprietà che sono caratteristiche solo degli elefanti, come ad esempio la lunghezza delle zanne. Questo campo e il metodo per visualizzarlo devono necessariamente essere inclusi nella classe Elefante. Nella gerarchia, da qualche parte, ci sarà anche un campo che dice quanta erba un certo erbivoro è in grado di mangiare al giorno. Dato che questa caratteristica riguarda tutti gli erbivori, l'informazione può essere messa nella classe Erbivori. La Figura 9.1 illustra queste decisioni.

Nella Figura 9.2 si vede l'effetto della dichiarazione di due oggetti delle ultime due classi della gerarchia, un Elefante e un Rinoceronte. Entrambi gli oggetti (jumbo e mafuta) hanno tre variabili, ma l'ultima è diversa nei due casi. Le altre due sono invece ereditate dalle classi di livello superiore. L'ereditarietà verrà discussa in dettaglio nel Paragrafo 9.3.

## 9.2 Astrazione mediante interfacce

Abbiamo già visto che l'astrazione ha come scopo quello di ridurre le ripetizioni e favorire la generalità e il riutilizzo delle classi. Java fornisce due modi per ottenere l'astrazione: le **interfacce** e le **classi astratte**. In questa parte del libro consideriamo le prime, mentre le seconde saranno oggetto di analisi nel Paragrafo 9.4.

Un'**interfaccia** è una classe di tipo speciale che serve per specificare un insieme di metodi senza implementarli. Lo scopo di encapsulare i metodi fornisce una garanzia: qualsiasi classe affermi di implementare questa

interfaccia dovrà necessariamente fornire questi metodi. Possiamo affermare che un'interfaccia è un modo per definire un insieme di standard, e una classe che implementa un'interfaccia ottiene un "timbro di approvazione" che sta ad indicare il fatto che si è adeguata a quegli standard. Gli oggetti della classe potranno dunque essere utilizzati ovunque sia richiesto un oggetto conforme a quegli stessi standard.

Le sintassi per dichiarare un'interfaccia e per implementarla sono:

### Dichiarazione di un'interfaccia

```
interface nome_interfaccia {
    specifica metodi
}
```

### Implementazione di un'interfaccia

```
class nome_classe implements nome_interfaccia {
    corpi dei metodi d'interfaccia
    altri dati e metodi
}
```

Ad esempio, si considerino le tipiche operazioni riguardanti macchine in grado di muoversi:

```
interface Movable {
    boolean start ();
    void stop ();
    boolean turn (int degrees);
    double fuelRemaining ();
    boolean changeSpeed (double kmperhour);
}
```

`Movable` definisce l'insieme delle cose che qualunque macchina che si muove deve saper fare: deve saper partire, fermarsi, curvare di un certo numero di gradi, modificare la propria velocità, e deve essere in grado di dire quanto carburante è rimasto. Naturalmente una macchina mobile può anche saper fare altre cose, ma questi metodi rappresentano il minimo comun denominatore. Ora se vogliamo dichiarare una classe per gli aerei (`Planes`) indicando che un aereo è una macchina mobile, opereremo così:

```
class Planes implements Movable {
    boolean start () {
        // fai cio' che serve per far partire l'aereo
        // e restituisci true in caso di successo.
    }
}
```

```

void stop () {
    // fai cio' che serve per fermare l'aereo
    // e restituisci true in caso di successo.
}

boolean turn (int degrees) {
    // fai cio' che serve per virare di quel numero di gradi
}

double fuelRemaining () {
    // restituisci la quantita' di carburante rimasta
}

boolean changeSpeed (double kmperhour) {
    // accelera di kmperhour' (decelera, se negativa)
}
}
}

```

Un'auto è anch'essa una macchina che si muove, come lo è un treno o una nave. Pensandoci bene, persino una falciatrice è una macchina che si muove, e potrebbe dunque implementare la nostra interfaccia Movable.

In quale modo l'uso dell'interfaccia può esserci di aiuto? L'interfaccia Movable, ad esempio, può servire ogni volta che vogliamo utilizzare il concetto astratto di macchina mobile. Supponete che vogliamo progettare un telecomando manuale per guidare dei modellini; questo telecomando sarà dotato di pulsanti per impartire i comandi e di un visore per ottenere informazioni, e dovrà permetterci di implementare tutti i metodi dell'interfaccia Movable (cioè, ci sarà un pulsante per far partire il modellino, uno per farlo girare ecc.). Se progettiamo il telecomando basandoci solamente sull'interfaccia Movable, lo stesso telecomando potrà in seguito essere utilizzato su qualsiasi tipo di modellino che implementi tale interfaccia, sia esso un aeroplano, un'automobilina, o una piccola falciatrice giocattolo. La classe RemoteControl che realizza il telecomando utilizzerà la composizione, dichiarando un oggetto Movable:

```

class RemoteControl {

    private Movable machine;

    RemoteControl (Movable m) {
        machine = m;
    }

    ...
    // quando si preme il pulsante "START" sul telecomando
    Boolean okay = machine.start();
    if (!okay) display ("Il modellino non e' partito!");
    ...
}

```

Al costruttore della classe RemoteControl è fornito un oggetto di classe Movable (ovvero, di una classe che implementa l'interfaccia Movable).

Potrebbe trattarsi di un aereo, di un'auto, di una barchetta, o di qualunque altra implementazione valida di Movable: tutti questi oggetti avranno per definizione le proprietà di un Movable. Questo oggetto è memorizzato come parte dei dati privati del telecomando, e ogni volta che quest'ultimo invoca uno dei metodi di Movable viene in realtà eseguita l'implementazione adatta a quella classe.

Per esempio, potremmo dichiarare:

```
Planes plane = new Plane();
Ships ship = new Ship();

RemoteControl planeRemote = new RemoteControl (plane);
RemoteControl shipRemote = new RemoteControl (ship);
```

e i due telecomandi opereranno correttamente, ciascuno dei due invocando ogni volta l'implementazione corretta dei metodi specificati dall'interfaccia Movable (il primo invocherà le implementazioni tipiche di un aeroplano, il secondo quelle tipiche di una barchetta).

### ESEMPIO 9.1 Ordinamento

**Problema.** L'algoritmo di ordinamento presentato nel Capitolo 6 è utile in svariate situazioni, e vorremmo modificarlo in modo da renderlo indipendente dal tipo di oggetti da ordinare.

**Soluzione.** Iniziamo ad affrontare il problema chiedendoci quali sono le

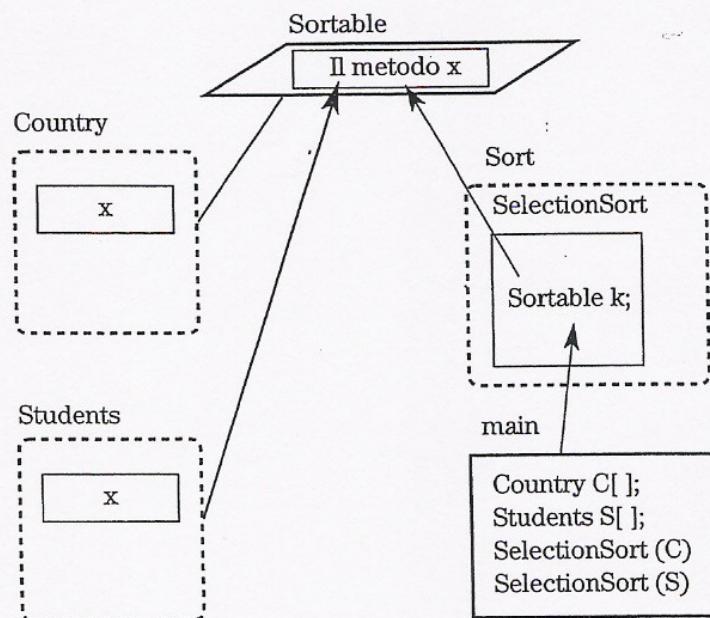


Figura 9.3 Classi coinvolte in un ordinamento astratto.

## Esempio Interfacce

### Movable.java

```
package it.alberghetti.esempioInterfacce;

public interface Movable {
    boolean start();
    boolean stop();
    boolean turn(int degrees);
    double fuelRemaining();
    boolean changeSpeed(double kmperhour);
}
```

### RemoteControl.java

```
package it.alberghetti.esempioInterfacce;

public class RemoteControl {
    private Movable machine;
    RemoteControl (Movable m){
        machine=m;
    }

    //quando si preme il pulsante start
    boolean tastoStart(){
        System.out.println("Sono nel metodo tastoStart di RemoteControl.");
        boolean okay=machine.start();
        if(!okay){
            System.out.println("Il modellino non è partito!");
        }
        return okay;
    }
}
```

**Planes.java**

```
package it.alberghetti.esempioInterfacce;

public class Planes implements Movable{

    public boolean start() {
        //fai ciò che serve per far partire l'aereo
        //e restituisci true in caso di successo
        System.out.println("Sono nel metodo start di Planes");
        return false;
    }

    public boolean stop() {
        //fai ciò che serve per far fermare l'aereo
        //e restituisci true in caso di successo
        return false;
    }

    public boolean turn(int degrees) {
        // fai ciò che serve per far virare l'aereo
        //e restituisci true in caso di successo
        return false;
    }

    public double fuelRemaining() {
        //restituisci la quantità di carburante
        return 0;
    }

    public boolean changeSpeed(double kmperhour) {
        //accelera in km/h (decelera se negativa)
        return false;
    }
}
```

### **Ships.java**

```
package it.alberghetti.esempioInterfacce;

public class Ships implements Movable {

    public boolean start() {
        //fai ciò che serve per far partire la nave l'aereo
        //e restituisci true in caso di successo
        System.out.println("Sono nel metodo start di Ships");
        return false;
    }

    public boolean stop() {
        //fai ciò che serve per far fermare la nave
        //e restituisci true in caso di successo
        return false;
    }

    public boolean turn(int degrees) {
        // fai ciò che serve per far virare la nave
        //e restituisci true in caso di successo
        return false;
    }

    public double fuelRemaining() {
        //restituisci la quantità di carburante
        return 0;
    }

    public boolean changeSpeed(double kmperhour) {
        //accelera in km/h (decelera se negativa)
        return false;
    }
}
```

### **TestInterfacce.java**

```
package it.alberghetti.esempioInterfacce;

public class TestInterfacce {
    public static void main(String args[]){
        Planes plane=new Planes();
        Ships ship=new Ships();
        RemoteControl shipRemote=new RemoteControl(ship);
        RemoteControl planeRemote=new RemoteControl(plane);
        System.out.println("Ho creato un oggetto telecomando per l'aereo");
        System.out.println("Premo il tasto start del telecomando per l'aereo");
        planeRemote.tastoStart();
        System.out.println("\n\nHo creato un oggetto telecomando per la nave");
        System.out.println("Premo il tasto start del telecomando per la nave");
        shipRemote.tastoStart();
    }
}
```

**Machine.java**

```
package it.alberghetti.esempioInterfacce;

public abstract class Machine implements Movable {
    //essendo una classe astratta posso implementare solo alcuni metodi e
    //demandare l'implementazione degli altri ad una classe derivata

    @Override
    public boolean start() {
        //fai ciò che serve per far partire l'aereo
        //e restituisci true in caso di successo
        System.out.println("Sono nel metodo start di Machine (Classe Astratta). Ho "+"+
            "implementato solo il metodo start");
        return false;
    }
}
```

**MacchininaGiocattolo.java**

```
package it.alberghetti.esempioInterfacce;

public class MacchininaGiocattolo extends Machine{
    // questa classe deriva da Machine, che è una classe astratta.
    // Machine ha implementato solo il metodo start, gli altri saranno implementati qui
    @Override
    public boolean stop() {
        System.out.println("Sono nel metodo stop di MacchininaGiocattolo");
        return false;
    }

    @Override
    public boolean turn(int degrees) {
        return false;
    }

    @Override
    public double fuelRemaining() {
        return 0;
    }

    @Override
    public boolean changeSpeed(double kmperhour) {
        return false;
    }
}
```

```

package it.alberghetti.esempioInterfacce;

public class TestInterfacce {
    public static void main(String args[]){
        Planes plane=new Planes();
        Ships ship=new Ships();
        RemoteControl shipRemote=new RemoteControl(ship);
        RemoteControl planeRemote=new RemoteControl(plane);
        System.out.println("Ho creato un oggetto telecomando per l'aereo");
        System.out.println("Premo il tasto start del telecomando per l'aereo");
        planeRemote.tastoStart();
        System.out.println("\n\nHo creato un oggetto telecomando per la nave");
        System.out.println("Premo il tasto start del telecomando per la nave");
        shipRemote.tastoStart();

        Machine m;
        // non posso creare un'istanza di tipo Machine (è una classe astratta)
        //m=new Machine(); //cannot instantiate the type Machine
        //invece posso dichiarare una variabile di tipo Movable (che è un'interfaccia)
        Movable m2;
        //posso assegnare a m2 un oggetto di tipo Planes (si tratta di un upcasting)
        System.out.println("\n\nDichiaro una variabile di tipo Movable e le assegno il valore
        //di plane facendo un upcasting implicito");
        m2=plane;
        m2.start();
        //m=m2; // ERRORE: cannot convert from Movable to Machine
        // non posso creare un'istanza di tipo Movable (è un'interfaccia)
        //m2=new Movable(); //cannot instantiate the type Movable
        //istanzio MacchininaGiocattolo, posso farlo perchè è una classe concreta che estende
        //la classe astratta Machine
        System.out.println("\n\nCreo una istanza della classe MacchininaGiocattolo, la "+
            "faccio partire e la fermo.");
        MacchininaGiocattolo mg=new MacchininaGiocattolo();
        mg.start();
        mg.stop();
        System.out.println("\nMacchininaGiocattolo è una classe concreta derivata dalla "+
            "classe astratta Machine.\n" +
            "Machine è una classe atratta che implementa Movable. "+
            "+Dell'interfaccia Movable implementa solo il metodo start");

    }
}

```