2

Introduzione a classi e oggetti

Obiettivi del capitolo

- Capire i concetti di classe e oggetto
- Apprendere la differenza tra oggetti e riferimenti a oggetti
- Acquisire familiarità con il procedimento di realizzazione delle classi
- Riuscire a scrivere semplici metodi
- Capire lo scopo e l'utilizzo dei costruttori
- Capire come accedere a variabili istanza e a variabili locali
- Apprezzare l'importanza dei commenti per la documentazione

La maggior parte dei programmi utili non manipola soltanto numeri e stringhe, ma gestisce dati più complessi e più aderenti alla realtà. Esempi di tali dati sono conti bancari, informazioni sugli impiegati e forme grafiche.

Il linguaggio Java è perfettamente appropriato per progettare e per gestire questi tipi di dati, o *oggetti*. In Java, definiremo *classi* per descrivere il comportamento di questi oggetti. In questo capitolo imparerete come definire classi per descrivere oggetti dal comportamento molto semplice. Dopo aver appreso ulteriori elementi di programmazione Java nei capitoli seguenti, sarete in grado di realizzare classi i cui oggetti possano eseguire azioni più sofisticate.

2.1 Usare e costruire oggetti

Oggetti e classi sono concetti fondamentali per la programmazione in Java. Per padroneggiarli completamente occorrerà un po' di tempo, ma, dal momento che ciascun programma Java utilizza almeno un paio di oggetti e di classi, conviene averne una conoscenza di base fin dall'inizio.

> Gli oggetti sono entità di un programma che si possono manipolare invocando metodi.

Un *oggetto* è un'entità che potete manipolare nel vostro programma, generalmente mediante l'invocazione di *metodi*. Per esempio, `System.out` si riferisce a un oggetto e abbiamo già visto nel Capitolo 1 come utilizzarlo, mediante la chiamata del metodo `println`. (In realtà, sono disponibili numerosi metodi diversi, tutti chiamati `println`: uno serve per stampare stringhe, uno per i numeri interi, uno per i numeri in virgola mobile e così via; il motivo per l'esistenza di tanti metodi diversi verrà discusso nel Paragrafo 2.7). Quando chiamate il metodo `println`, all'interno dell'oggetto si svolgono alcune attività, il cui effetto finale è che l'oggetto fa comparire il testo nella finestra della console.

> L'interfaccia pubblica di una classe specifica cosa si può fare con gli oggetti che si creano da essa. L'implementazione nascosta descrive come si svolgono tali azioni.

Per il momento, dovreste considerare gli oggetti quali "scatole nere" (*black box*) fornite di un'*interfaccia* pubblica (ovvero, i metodi che potete chiamare) e di un'*implementazione* nascosta, ovvero il codice e i dati che sono necessari per fare funzionare i metodi.

Oggetti differenti forniscono serie diverse di metodi. Per esempio, il metodo `println` si può applicare all'oggetto `System.out`, ma non all'oggetto stringa `"Hello, World!"`. Pertanto, questa invocazione sarebbe un errore:

```
"Hello, World!".println(); // Questa chiamata di metodo è un errore
```

La ragione è semplice: gli oggetti `System.out` e `"Hello, World!"` appartengono a *classi* diverse. Infatti, `System.out` è un oggetto della classe `PrintStream`, mentre `"Hello, World!"` è un oggetto della classe `String`. Si può applicare il metodo `println` a *qualsiasi* oggetto della classe `PrintStream`, ma la classe `String` non fornisce il metodo `println`. La classe `String` prevede un buon numero di altri metodi, molti dei quali vedrete nel Capitolo 3. Per esempio, il metodo `length` conta il numero di caratteri in una stringa e si può applicare a qualsiasi oggetto del tipo `String`. Pertanto,

```
"Hello, World!".length(); // Questa chiamata di metodo è corretta
```

è una chiamata di metodo corretta, che calcola il numero di caratteri nell'oggetto stringa `"Hello, World!"` e restituisce il risultato, 13 (le virgolette non vengono considerate).

Potete verificare che il metodo `length` restituisce la lunghezza di un oggetto `String` scrivendo un breve programma di prova

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

Ogni oggetto appartiene a una classe, e la classe definisce i metodi per gli oggetti. Quindi, la classe `PrintStream` definisce i metodi `print` e `println`; la classe `String` definisce il metodo `length` e molti altri metodi.

> Le classi sono fabbriche di oggetti. Un nuovo oggetto di una classe si costruisce con l'operatore `new`.

L'oggetto `System.out` viene creato automaticamente quando un programma Java carica la classe `System`. Gli oggetti stringa vengono creati quando si specifica una stringa racchiusa tra virgolette. Tuttavia, nella maggior parte dei programmi Java volete creare più oggetti.

Per vedere come una classe possa creare nuovi oggetti, esaminiamo un'altra classe, `Rectangle`, nella libreria delle classi di Java. Gli oggetti di tipo `Rectangle` descrivono forme rettangolari, come quelle illustrate nella Figura 1.

Notate che un oggetto `Rectangle` non è una forma rettangolare, ma un insieme di numeri che descrivono il rettangolo (osservare la Figura 2). Ciascun rettangolo è descritto dalle coordinate x e y del suo angolo superiore sinistro, dalla sua larghezza e dalla sua altezza. Per creare un nuovo rettangolo, è necessario specificare questi quattro valori. Per esempio, potete costruire un rettangolo con le coordinate dell'angolo superiore sinistro uguali a (5, 10), con larghezza 20 e altezza 30, nel modo seguente:

```
new Rectangle(5, 10, 20, 30)
```

L'operatore `new` produce la creazione di un oggetto di tipo `Rectangle`. Il processo che crea un nuovo oggetto è detto *costruzione*. I quattro valori 5, 10, 20 e 30 rappresentano

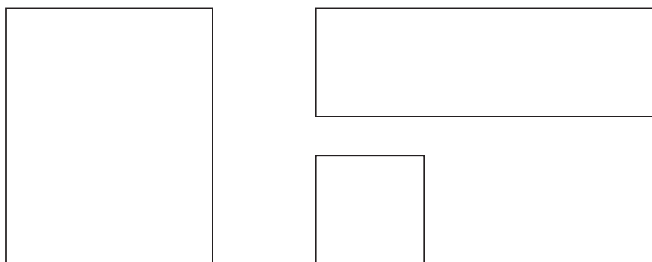


Figura 1
Forme rettangolari

Rectangle	
x	5
y	10
width	20
height	30

Figura 2
Un oggetto `Rectangle`

i *parametri di costruzione*. Classi diverse richiederanno parametri di costruzione differenti. Per esempio, per costruire un oggetto `Rectangle`, fornite quattro numeri che descrivono la posizione e le dimensioni del rettangolo. Per costruire un oggetto `Car`, potreste fornire il nome e l'anno del modello di automobile.

In realtà, alcune classi permettono di costruire oggetti in più maniere. Per esempio, potete ottenere un oggetto rettangolo anche senza fornire alcun parametro di costruzione (ma dovete sempre inserire le parentesi):

```
new Rectangle()
```

Questo enunciato costruisce un rettangolo (piuttosto inutile) con l'angolo superiore sinistro posizionato nell'origine (0, 0), con larghezza 0 e altezza 0.

Per costruire qualsiasi oggetto, seguite questi passaggi:

1. Usate l'operatore `new`.
2. Date il nome della classe.
3. Fornite i parametri di costruzione (se esistono), all'interno delle parentesi.

Che cosa si può fare con un oggetto `Rectangle`? Non molto, per adesso. Nel Capitolo 4, imparerete come visualizzare rettangoli e altre figure in una finestra. Potete passare un oggetto `Rectangle` al metodo `System.out.print` o `println` che visualizza una descrizione dell'oggetto rettangolo nella finestra della console:

```
public class RectangleTest
{
   public static void main(String[] args)
   {
      System.out.println(new Rectangle(5, 10, 20, 30));
   }
}
```

Questo programma stampa la riga:

```
java.awt.Rectangle[x=5,y=10,width=20,height=30]
```

Più specificatamente, questo programma crea un oggetto di tipo `Rectangle`, poi passa l'oggetto al metodo `println` e, infine, non lo utilizza più.

Sintassi di Java
2.1 Costruzione di oggetti

```
new NomeClasse(parametri)
```

Esempio:

```
new Rectangle(5, 10, 20, 30);
new Car("BMW 540ti", 2004);
```

Obiettivo:

Costruire un nuovo oggetto, inizializzarlo tramite i parametri di costruzione e restituire un riferimento per l'oggetto costruito.

2.2 Variabili oggetto

Nelle variabili oggetto vengono memorizzati gli indirizzi degli oggetti.

Naturalmente, in genere, con un oggetto dovreste fare qualcosa di più che semplicemente crearlo, stamparlo e scordarvene. Per conservare traccia di un oggetto, avete bisogno di inserirlo in una *variabile oggetto*. Come abbiamo accennato nel Capitolo 1, una variabile è un'informazione in memoria la cui posizione è identificata da un nome simbolico.

In Java, ogni variabile ha un particolare *tipo* che identifica il tipo di informazione che essa può contenere. Per creare una variabile, occorre fornire il suo tipo, seguito da un nome per la variabile, come in questo esempio:

```
Rectangle cerealBox;
```

Questo enunciato definisce una variabile oggetto, `cerealBox`. Il *tipo* di questa variabile è `Rectangle:` ciò significa che, una volta definita la variabile `cerealBox` con l'enunciato precedente, da allora in poi nel programma la variabile dovrà sempre contenere la posizione di un oggetto di tipo `Rectangle` e mai di un oggetto di tipo `Car` o di tipo `String`.

Il nome della variabile può essere scelto arbitrariamente, rispettando soltanto poche semplici regole.

- I nomi possono essere composti di lettere, cifre e segni di sottolineatura (_), ma non possono iniziare con una cifra.
- Nei nomi di variabile non si possono usare altri simboli, come ad esempio ? o %.
- Nemmeno gli spazi sono ammessi all'interno dei nomi.
- Inoltre, le *parole riservate*, come `public`, non possono essere usate come nomi; tali parole sono riservate in modo esclusivo allo speciale significato che hanno nel linguaggio Java. L'Appendice A4 elenca tutte le parole riservate.
- Anche i nomi delle variabili sono sensibili alla differenza tra lettere maiuscole e minuscole, cioè `cerealBox` e `Cerealbox` sono nomi *diversi*.

Tutte le variabili oggetto devono essere inizializzate prima di accedervi.

Torniamo alla dichiarazione della variabile `cerealBox`. Finora la variabile `cerealBox` non fa riferimento ad alcun oggetto, è una variabile *non inizializzata* (osservare la Figura 3). Occorre impostare `cerealBox` in modo che si riferisca a un oggetto. Come ottenere un riferimento a un ogget-

Figura 3
Una variabile oggetto non inizializzata

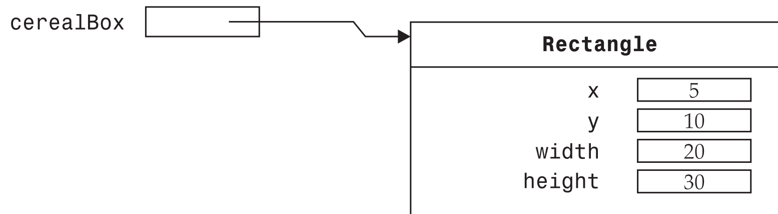


Figura 4
Una variabile oggetto che contiene un riferimento a oggetto

to? L'operatore `new` costruisce un nuovo oggetto e ne restituisce la posizione, basta quindi usare tale valore per inizializzare la variabile:

```
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
```

Un riferimento a un oggetto descrive la posizione dell'oggetto in memoria.

Ci si potrebbe chiedere cosa accada se si lascia la variabile `cerealBox` non inizializzata: si veda Errori Comuni 2.1 per una risposta. La Figura 4 mostra l'effetto dell'enunciato precedente.

La posizione di un oggetto in memoria (locazione) viene spesso chiamata *riferimento* all'oggetto. Quando una variabile contiene la locazione di un oggetto, si dice che *si riferisce* all'oggetto. Ad esempio, `cerealBox` si riferisce all'oggetto `Rectangle` costruito dall'operatore `new`.

Più variabili oggetto possono contenere riferimenti allo stesso oggetto.

È essenziale ricordarsi che la variabile `cerealBox` *non contiene* l'oggetto, ma *si riferisce* all'oggetto. Infatti, è possibile avere due variabili oggetto che si riferiscono allo stesso oggetto:

```
Rectangle r = cerealBox;
```

Ora potete accedere allo stesso oggetto `Rectangle` attraverso entrambe le variabili `cerealBox` e `r`, come illustrato nella Figura 5.

Solitamente i programmi usano gli oggetti in questo modo:

1. Costruiscono un oggetto con l'operatore `new`.
2. Memorizzano il riferimento all'oggetto in una variabile oggetto.
3. Invocano metodi tramite la variabile oggetto.

La classe `Rectangle` ha più di 50 metodi, alcuni utili e altri un po' meno. Per avere un'idea di che cosa significhi manipolare oggetti `Rectangle`, esaminiamo un metodo della classe `Rectangle`. Il metodo `translate` *sposta* un rettangolo per una certa distanza, nelle direzioni x e y. Per esempio:

```
cerealBox.translate(15, 25);
```

sposta il rettangolo di 15 unità nella direzione x e di 25 unità nella direzione y. Spostare un rettangolo non modifica la sua larghezza o la sua altezza, ma cambia la posizione del suo angolo superiore sinistro. La porzione di codice seguente:

```
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
cerealBox.translate(15, 25);
System.out.println(cerealBox);
```

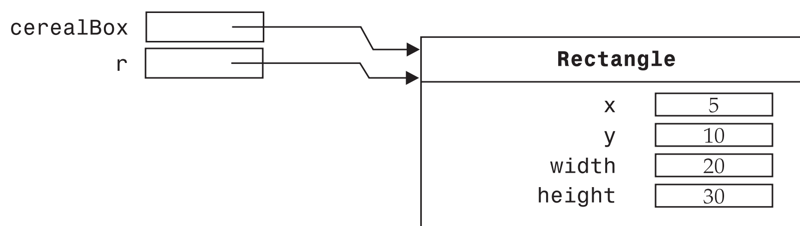


Figura 5
Due variabili oggetto che si riferiscono allo stesso oggetto

stampa questa riga:

```
java.awt.Rectangle[x=20,y=35,width=20,height=30]
```

Trasformiamo questo frammento di codice in un programma completo. Come nel caso del programma `Hello`, dovete seguire questi tre passaggi:

1. Inventate un nuova classe, per esempio `MoveTest`.
2. Fornite un metodo `main`.
3. Inserite le istruzioni all'interno del metodo `main`.

> Le classi Java sono raggruppate in pacchetti. Se si usa una classe di un pacchetto diverso da `java.lang` bisogna importarla.

Per questo programma dovete eseguire un passaggio ulteriore: *importare* la classe `Rectangle` da un *pacchetto* ("package"), vale a dire da una raccolta di classi che hanno finalità simili. Tutte le classi della libreria standard sono contenute in pacchetti. La classe `Rectangle` appartiene al pacchetto `java.awt` (l'abbreviazione `awt` significa "**A**bstract **W**indowing **T**oolkit"), che contiene molte classi utili per disegnare finestre e forme grafiche.

Per usare la classe `Rectangle` del pacchetto `java.awt`, inserite semplicemente la riga seguente all'inizio del vostro programma:

```
import java.awt.Rectangle;
```

Perché non abbiamo dovuto importare le classi `System` e `String` che abbiamo usato nel programma `Hello`? Il motivo è che le classi `System` e `String` si trovano nel pacchetto `java.lang`, e tutte le classi di questo pacchetto sono importate automaticamente, per cui non avete mai bisogno di importarle.

Pertanto, il programma completo è il seguente:

File MoveTest.java

```
import java.awt.Rectangle;

public class MoveTest
{
   public static void main(String[] args)
   {
      Rectangle cerealBox = new Rectangle(5, 10, 20, 30);

      // sposta il rettangolo
      cerealBox.translate(15, 25);

      // stampa il rettangolo spostato
      System.out.println(cerealBox);
   }
}
```

Sintassi di Java
2.2 Definizione di variabile

NomeTipo nomeVariabile;
NomeTipo nomeVariabile = *espressione*;

Esempio:

```
Rectangle cerealBox;
String name = "Dave";
```

Obiettivo:

Definire una nuova variabile di tipo *NomeTipo* e fornirne eventualmente un valore iniziale.

Sintassi di Java
2.3 Importare una classe da un pacchetto

```
import nomePacchetto.NomeClasse;
```

Esempio:

```
import java.awt.Rectangle;
```

Obiettivo:

Importare una classe da un pacchetto, per utilizzarla in un programma.

Errori comuni 2.1

Dimenticarsi di inizializzare le variabili

Avete appena imparato come memorizzare il riferimento a un oggetto in una variabile in modo da poter manipolare l'oggetto in un programma. Questa è un'azione molto comune e può portare a uno degli errori di programmazione più frequenti, l'utilizzo di una variabile che non è stata inizializzata.

Supponiamo che il programma contenga le seguenti linee

```
Rectangle cerealBox;
cerealBox.translate(15, 25);
```

Ora esiste la variabile `cerealBox`. Una variabile oggetto è un contenitore per un riferimento a un oggetto, ma nella variabile non è stato inserito nulla, non è inizializzata. Quindi, non c'è alcun rettangolo da spostare.

Il compilatore segnala questi problemi: se fate questo errore, il compilatore protesterà perché state tentando di usare una variabile non inizializzata.

Il rimedio consiste nell'inizializzare la variabile. Si può inizializzare una variabile con un riferimento a un qualsiasi oggetto, sia un riferimento a un nuovo oggetto, sia a un oggetto esistente.

```
// inizializza con un riferimento a un nuovo oggetto
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
// inizializza con un riferimento a un oggetto esistente
Rectangle cerealBox = anotherRectangle;
```

Argomenti avanzati 2.1

Importare le classi

Abbiamo visto qual è il metodo più chiaro e più semplice per importare classi dai pacchetti. Adoperate semplicemente un enunciato `import`, che indichi il pacchetto e la classe per ciascuna classe che volete importare, come in questo esempio:

```
import java.awt.Rectangle;
import java.awt.Point;
```

Esiste una scorciatoia che molti programmatori trovano conveniente: si possono importare *tutte* le classi di un pacchetto, mediante il seguente costrutto:

```
import nomePacchetto.*;
```

Per esempio, l'enunciato seguente importa tutte le classi del pacchetto `java.awt`:

```
import java.awt.*;
```

Si tratta di una forma meno complicata da digitare, ma non vogliamo usare questa forma nel libro per una semplice ragione. Se un programma importa più pacchetti e vi imbattete in un nome di una classe sconosciuta, poi dovete cercare in tutti i pacchetti per ritrovarla. Per esempio, supponiamo di vedere un programma con queste importazioni:

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

In aggiunta, ipotizziamo di leggere un nome di classe, `Image`. Non potete sapere se la classe `Image` appartenga al pacchetto `java.awt` o a `java.io`. Perché preoccuparsi di quale pacchetto si tratta? Occorre saperlo se volete usare la classe nel vostro programma.

Notate che *non si possono* importare più pacchetti mediante un unico enunciato `import`. Per esempio, questo è un errore di sintassi:

```
import java.*.*; // Errore
```

Potete evitare tutti gli enunciati `import` se utilizzate il nome *completo* (costituito da entrambi i nomi, del pacchetto e della classe) ogni volta che adoperate una classe, come in questo esempio:

```
java.awt.Rectangle cerealBox =
    new java.awt.Rectangle(5, 10, 20, 30);
```

È un sistema abbastanza noioso e non troverete molti programmatori che lo usano.

2.3 Definire una classe

In questo paragrafo imparerete come definire le vostre classi, ricordando che una classe definisce i metodi che si possono applicare ai suoi oggetti. Inizieremo con una classe molto semplice che contiene un unico metodo:

```
public class Greeter
{
   public String sayHello()
   {
      String message = "Hello, World!";
      return message;
   }
}
```

La definizione di un metodo contiene le seguenti parti:

- Uno *specificatore di accesso* (come `public`)
- Il *tipo di dati restituito* dal metodo (come `String`)
- Il nome del metodo (come `sayHello`)
- Un elenco di *parametri* del metodo, racchiusi fra parentesi (il metodo `sayHello` non ha parametri)
- Il *corpo* del metodo: una sequenza di enunciati racchiusi fra parentesi graffe

Lo specificatore di accesso determina quali altri metodi possono chiamare questo metodo. La maggior parte dei metodi dovrebbe essere dichiarata `public`: in questo modo, tutti gli altri metodi nei vostri programmi possono chiamarli. (Talvolta è meglio evitare di avere metodi che possano essere invocati così indiscriminatamente: consultate il Capitolo 10 per maggiori informazioni su questo argomento.)

> La definizione di un metodo specifica il nome del metodo, i suoi parametri e gli enunciati che servono per svolgere l'azione del metodo.

Il tipo restituito indica il tipo di dati del valore che il metodo restituisce a chi l'ha invocato. Il metodo `sayHello` restituisce un oggetto di tipo `String` (più precisamente, la stringa `"Hello, World!"`).

Alcuni metodi eseguono semplicemente alcuni enunciati, senza restituire alcun valore. Tali metodi vengono identificati con un tipo del valore restituito uguale a `void`.

Molti metodi dipendono da altre informazioni: ad esempio, il metodo `translate` della classe `Rectangle` deve sapere di quanto volete spostare il rettangolo orizzontalmente e verticalmente. Questi dati vengono detti *parametri* del metodo: ogni parametro è una variabile, con un tipo e un nome. Le variabili parametro sono separate da virgole. Ad esempio, chi ha realizzato la libreria di Java ha definito il metodo `translate` come segue:

```
public class Rectangle
{  ...
   public void translate(int x, int y)
   {
      corpo del metodo
   }
   ...
}
```

Sintassi di Java
2.4 Implementazione di metodo

```
public class NomeClasse
{  ...
   specificatoreDiAccesso tipoRestituito nomeMetodo
      (tipoParametro nomeParametro, ...)
   {
      corpo del metodo
   }
   ...
}
```

Esempio:

```
public class Greeter
{
   public String sayHello()
   {
      String message = "Hello, World!";
      return message;
   }
}
```

Obiettivo:

Definire il comportamento di un metodo

Il corpo del metodo contiene gli enunciati che vengono eseguiti dal metodo; il corpo del metodo `sayHello`, ad esempio, contiene due enunciati. Il primo enunciato inizializza una variabile di tipo `String` con un oggetto di tipo `String`:

```
String message = "Hello, World!";
```

Per specificare il valore restituito da un metodo al suo chiamante si usa l'enunciato `return`.

Il secondo enunciato è un enunciato speciale che pone termine al metodo: quando viene eseguito l'enunciato `return`, il metodo termina. Se il metodo ha un tipo di dati da restituire diverso da `void`, allora l'enunciato `return` deve contenere un *valore da restituire*, cioè il valore che il metodo invia a chi l'ha invocato. Il metodo `sayHello` restituisce il riferimento all'oggetto memorizzato in `message`, cioè un riferimento all'oggetto stringa `"Hello, World!"`.

Avete quindi visto come definire una classe che contiene un metodo; nel prossimo paragrafo vedrete cosa si può fare con tale classe.

Sintassi di Java
2.5 L'enunciato `return`

```
return espressione;
```

oppure

```
return;
```

Esempio:

```
return message;
```

Obiettivo:

Specificare il valore restituito da un metodo e terminare il metodo immediatamente. Il valore restituito diventa il valore dell'espressione di invocazione del metodo.

2.4 Collaudare una classe

Per collaudare una classe, usate un ambiente per il collaudo interattivo o scrivete una seconda classe che esegua istruzioni di collaudo.

Nel paragrafo precedente avete visto la definizione di una semplice classe, `Greeter`. Come potete usarla? Potete sicuramente compilare il file `Greeter.java`, ma non potete eseguire il file `Greeter.class` che viene prodotto, perché non contiene un metodo `main`. Questo è molto comune: la maggior parte delle classi non contiene un metodo `main`.

Per fare qualcosa di utile con questa classe, ci sono due possibilità. Alcuni ambienti di sviluppo, come l'eccellente programma BlueJ, consentono di creare oggetti di una classe e di invocare metodi con tali oggetti. La Figura 6 mostra il risultato della creazione di un oggetto della classe `Greeter` e dell'invocazione del metodo `sayHello`. La finestra di dialogo contiene il valore restituito dal metodo.

In alternativa, se non avete a disposizione un ambiente di sviluppo che consenta il collaudo interattivo di una classe, potete scrivere una classe per il collaudo (*classe di test*). Una classe di test è una classe con un metodo `main` che contiene gli enunciati che servono al collaudo di un'altra classe, eseguendo solitamente i passi seguenti:

1. Costruzione di uno o più oggetti della classe che si sta collaudando.
2. Invocazione di uno o più metodi.
3. Visualizzazione di uno o più valori restituiti.

La classe `RectangleTest` nel Paragrafo 2.1 è un valido esempio di classe di test: tale classe collauda la classe `Rectangle`, una classe della libreria Java.

Ecco, invece, una classe per collaudare la classe `Greeter`. Il metodo `main` costruisce un oggetto di tipo `Greeter`, invoca il metodo `sayHello` e visualizza sulla console il valore restituito.

```
public class GreeterTest
{
   public static void main(String[] args)
   {
      Greeter worldGreeter = new Greeter();
      System.out.println(worldGreeter.sayHello());
   }
}
```

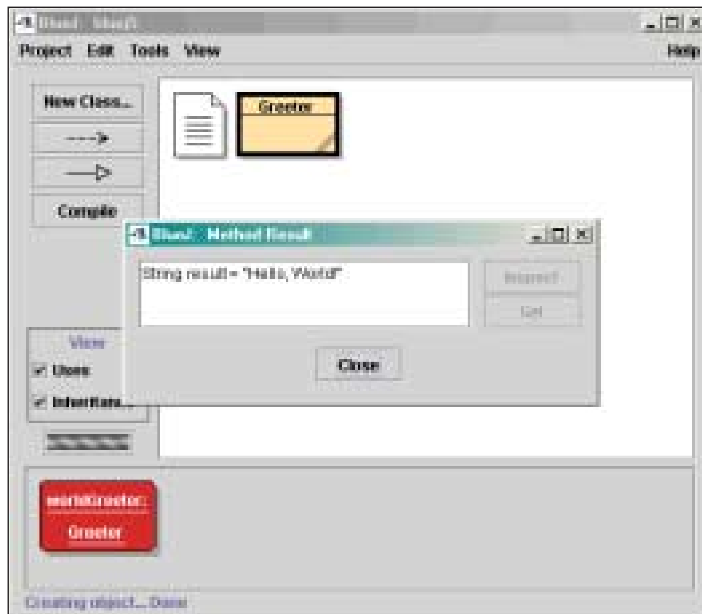


Figura 6
Il collaudo di una classe nell'ambiente BlueJ

Per ottenere un programma, occorre mettere insieme queste due classi. I dettagli per costruire il programma dipendono dal vostro compilatore e dall'ambiente di sviluppo; nella maggior parte degli ambienti, dovrete eseguire questi passi:

1. Creare una nuova cartella per il vostro programma.
2. Creare due file, uno per ciascuna classe.
3. Compilare entrambi i file.
4. Eseguire il programma di collaudo.

Ad esempio, se usate gli strumenti del Java SDK sulla linea di comando, i passi sono:

```
mkdir greeter
cd greeter
edit Greeter.java
edit GreeterTest.java
javac Greeter.java
javac GreeterTest.java
java GreeterTest
```

Molti studenti si sorprendono del fatto che un programma così semplice contenga due classi, tuttavia ciò è normale, perché le due classi hanno obiettivi radicalmente distinti. La classe `Greeter` (che renderemo più interessante nel prossimo paragrafo) descrive oggetti che possono produrre saluti. La classe `GreeterTest` esegue un collaudo che mette alla prova un oggetto `Greeter` sul campo. Il programma `GreeterTest` è necessario soltanto se il vostro ambiente di sviluppo non mette a disposizione strumenti per il collaudo interattivo.

Consigli per la produttività 2.1

Usare efficacemente la riga dei comandi

Se il vostro ambiente di programmazione permette di svolgere tutte le attività di routine tramite menu e finestre di dialogo, potete saltare questa nota. Tuttavia, se siete costretti a chiamare manualmente l'editor, il compilatore e l'interprete per collaudare il programma, allora vale proprio la pena di imparare a *gestire la riga dei comandi.*

La maggior parte dei sistemi operativi (UNIX, DOS, OS/2) ha un'*interfaccia per la riga dei comandi*, per interagire con il computer. (In Windows, potete usare l'interfaccia per la riga dei comandi DOS, selezionando l'icona "Prompt di MS-DOS", oppure, se tale icona non compare nel vostro menu "Programmi", selezionando "Esegui…" e scrivendo `command.com`). In questo modo, potete lanciare comandi attraverso un *prompt*: il comando viene eseguito e, al termine, avrete un altro prompt.

Quando sviluppate un programma, vi trovate a eseguire gli stessi comandi più e più volte, e sarebbe meglio non essere costretti a digitare ripetutamente comandi terribili come questo:

```
javac MyProg.java
```

Sarebbe bello non essere costretti a ridigitare il comando interamente per correggere un errore. Molte interfacce per la riga dei comandi presentano un'opzione proprio per questo, ma non sempre ne fanno una cosa ovvia. Se usate Windows, dovete installare un programma che si chiama `doskey`. Se usate UNIX, alcune shell vi consentono di accedere ai comandi precedenti. Se la vostra configurazione standard non ha questa possibilità, chiedete come poter passare a una shell migliore, come `bash` o `tcsh`.

Una volta configurata correttamente la shell, potrete usare le frecce verso l'alto e verso il basso per scorrere l'elenco dei comandi immessi, e le frecce verso sinistra e verso destra per modificare il comando richiamato. Potete utilizzare anche il *completamento del comando*. Per esempio, per inserire nuovamente lo stesso comando `javac`, digitate `javac` e premete F8 (Windows), oppure scrivete `!javac` (UNIX).

2.5 Variabili istanza

Finora la nostra classe `Greeter` non è molto interessante, perché tutti i suoi oggetti si comportano allo stesso modo. Supponiamo di costruire due oggetti:

```
Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();
```

> Un oggetto usa variabili istanza per memorizzare il suo stato, cioè i dati di cui ha bisogno per eseguire i suoi metodi.

I due oggetti `greeter1` e `greeter2` forniscono esattamente lo stesso risultato, quando viene invocato il metodo `sayHello`. Cerchiamo ora di modificare la classe `Greeter` in modo che un oggetto restituisca il messaggio `"Hello, World!"`, mentre un altro oggetto possa restituire `"Hello, Dave!"`.

Per raggiungere l'obiettivo, ogni oggetto di tipo `Greeter` deve memorizzare uno *stato*. Lo stato di un oggetto è l'insieme dei valori che determina come l'oggetto reagisce alle invocazioni dei suoi metodi. Nel caso del nostro oggetto `Greeter` migliorato, lo stato è il nome che vogliamo usare nel saluto, come `"World"` o `"Dave"`.

Ciascun oggetto conserva il proprio stato in una o più *variabili istanza* (o *campi*), che vengono dichiarate nella classe.

```
public class Greeter
{  ...
   private String name;
}
```

La dichiarazione di una variabile istanza è formata dalle parti seguenti:

- uno *specificatore di accesso* (come `private`)
- il *tipo* della variabile (come `String`)
- il *nome* della variabile (come `name`)

Ogni oggetto di una classe ha il proprio insieme di variabili istanza.

Ciascun oggetto di una certa classe ha la propria copia delle variabili istanza. Per esempio, se `worldGreeter` e `daveGreeter` sono due oggetti della classe `Greeter`, allora ciascun oggetto ha il proprio campo `name`, ovvero `worldGreeter.name` e `daveGreeter.name` (osservare la Figura 7).

L'incapsulamento è il processo che nasconde i dati dell'oggetto, fornendo metodi per accedervi.

Generalmente, le variabili istanza si dichiarano con lo specificatore di accesso `private`. Questo significa che solo i metodi della *stessa classe* possono accedervi, non qualsiasi altro metodo. In particolare, si può accedere alla variabile `name` solo attraverso il metodo `sayHello`.

In altre parole, se le variabili istanza si dichiarano private, tutti gli accessi ai dati devono avvenire tramite metodi pubblici. Pertanto, le variabili istanza relative a un oggetto in realtà sono nascoste per il programmatore che utilizza una classe, e riguardano solamente il programmatore che implementa la classe.

Tutte le variabili istanza dovrebbero essere dichiarate private.

Il processo di protezione dei dati è detto *incapsulamento*. Sebbene in Java sia possibile, teoricamente, lasciare le variabili istanza non incapsulate (definendole `public`), in pratica ciò avviene molto di rado. In questo

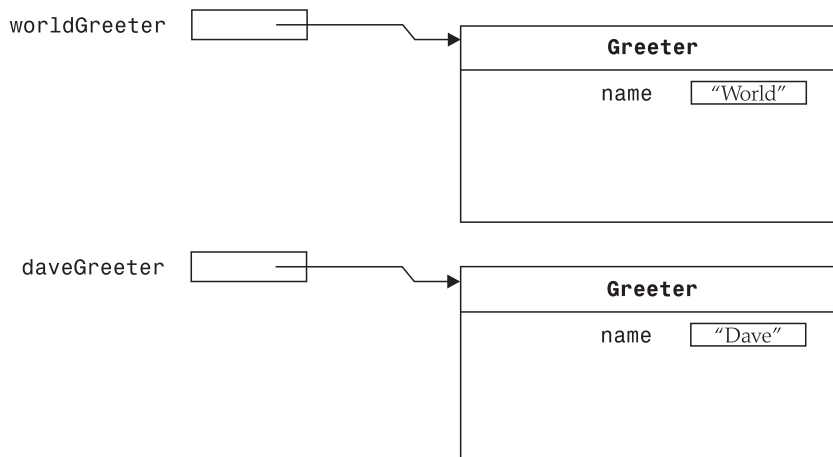


Figura 7
Variabili istanza

libro, definiremo sempre `private` le variabili istanza. Per esempio, poiché la variabile istanza `name` è privata, non potete accedervi da metodi di un'altra classe:

```
public class GreeterTest
{
   public static void main(String[] args)
   {
      ...
      System.out.println(daveGreeter.name); // ERRORE
   }
}
```

Soltanto il metodo `sayHello` può accedere alla variabile privata `name`. Se successivamente aggiungiamo altri metodi alla classe `Greeter`, come ad esempio un metodo `sayGoodbye`, allora anche tali metodi potranno accedere alla variabile istanza privata.

Ecco l'implementazione del metodo `sayHello` della classe `Greeter` migliorata.

```
public String sayHello()
{
   String message = "Hello, " + name + "!";
   return message;
}
```

Il simbolo + indica la *concatenazione di stringhe*, un'operazione che genera una nuova stringa mettendo una dopo l'altra stringhe più brevi.

Questo metodo calcola la stringa `message` combinando tre stringhe: `"Hello, "`, la stringa memorizzata nella variabile istanza `name` e la stringa composta da un punto esclamativo, `"!"`. Se la variabile `name` si riferisce alla stringa `"Dave"`, la stringa risultante è `"Hello, Dave!"`.

Notate che questo metodo usa due diverse variabili oggetto: la *variabile locale* `message` e la variabile istanza `name`. Una variabile locale appartiene a un solo metodo e può essere utilizzata soltanto all'interno del metodo in cui viene dichiarata. Una variabile istanza appartiene invece a un oggetto e può essere usata in tutti i metodi della sua classe.

Sintassi di Java
2.6 Dichiarazione delle variabili istanza

```
specificatoreDiAccesso class NomeClasse
{
   ...
   specificatoreDiAccesso tipoVariabile nomeVariabile;
   ...
}
```

Esempio:

```
public class Greeter
{
   ...
```

```
   private String name;
   ...
}
```

Obiettivo:

Definire una variabile che sia presente in ciascun oggetto di una classe.

2.6 Costruttori

I costruttori contengono istruzioni per inizializzare gli oggetti. Il nome del costruttore è sempre uguale al nome della classe.

Per completare la classe `Greeter` migliorata dobbiamo essere in grado di costruire oggetti aventi diversi valori della variabile istanza `name`. Vogliamo specificare il nome quando viene costruito l'oggetto:

```
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
```

Per realizzare ciò abbiamo bisogno di inserire un *costruttore* nella definizione della classe. Un costruttore specifica come un oggetto deve essere inizializzato. Nel nostro esempio, abbiamo un solo parametro per la costruzione di un oggetto, una stringa che specifica il nome. Ecco quindi il codice per il costruttore.

```
public Greeter(String aName)
{
   name = aName;
}
```

Un costruttore ha sempre lo stesso nome della classe degli oggetti che costruisce. Analogamente ai metodi, generalmente i costruttori sono dichiarati `public`, per consentire a qualsiasi codice in un programma di costruire nuovi oggetti della classe. Tuttavia, diversamente dai metodi, i costruttori non prevedono tipi di dati restituiti.

L'operatore `new` invoca il costruttore:

```
new Greeter("Dave")
```

L'operatore `new` invoca il costruttore.

Questa espressione costruisce un nuovo oggetto la cui variabile istanza `name` assume un valore che si riferisce alla stringa `"Dave"`.

I costruttori non sono metodi e, quindi, non potete invocare un costruttore su un oggetto esistente. Per esempio, questa chiamata non è ammessa:

```
worldGreeter.Greeter("Harry"); // Errore
```

Potete usare un costruttore solo combinandolo con l'operatore `new`.

Ecco il codice completo per la classe `Greeter` migliorata.

File Greeter.java

```
public class Greeter
{
   public Greeter(String aName)
   {
      name = aName;
   }

   public String sayHello()
   {
      String message = "Hello, " + name + "!";
      return message;
   }

   private String name;
}
```

Ecco, infine, una classe di prova che potete usare per avere conferma che la classe `Greeter` funzioni correttamente.

File GreeterTest.java

```
public class GreeterTest
{
   public static void main(String[] args)
   {
      Greeter worldGreeter = new Greeter("World");
      System.out.println(worldGreeter.sayHello());

      Greeter daveGreeter = new Greeter("Dave");
      System.out.println(daveGreeter.sayHello());
   }
}
```

Sintassi di Java
2.7 Implementazione di costruttori

```
specificatoreDiAccesso class NomeClasse
{
   ...
   specificatoreDiAccesso NomeClasse(TipoParametro NomeParametro, ...)
   {
      implementazione costruttore
   }
   ...
}
```

Esempio:

```
public class Greeter
{
   ...
```

```
    public Greeter(String aName)
    {
        name = aName;
    }
    ...
}
```

Obiettivo:

Definire il comportamento di un costruttore, che viene utilizzato per inizializzare le variabili istanza di oggetti nuovi appena creati.

2.7 Progettare l'interfaccia pubblica di una classe

Lo scopo della classe `Greeter` è stato quello di mostrarvi il meccanismo di definizione delle classi, dei metodi, delle variabili istanza e dei costruttori. Sinceramente, quella classe non è poi tanto utile. In questo paragrafo progetterete una classe più interessante, che descrive il comportamento di un *conto bancario*. Cosa ancor più importante, percorrete tutti i passi necessari per progettare una nuova classe.

Prima di iniziare a programmare, dovete capire come si comportano gli oggetti della vostra classe. Considerate che tipo di operazioni potete svolgere con un conto bancario:

- depositare denaro
- prelevare denaro
- chiedere il saldo attuale

In Java, queste operazioni si effettuano grazie a *chiamate di metodi*. Supponiamo di avere una variabile `harrysChecking` che contiene un riferimento a un oggetto di tipo `BankAccount`. Vorrete poter chiamare metodi di questo genere:

```
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```

In pratica, la classe `BankAccount` dovrebbe definire tre metodi:

- `deposit`
- `withdraw`
- `getBalance`

Successivamente occorre identificare il tipo dei parametri e dei valori di ritorno di questi metodi. Come potete vedere dagli esempi riportati, i metodi `deposit` e `withdraw` ricevono un numero (l'importo, in dollari) e non restituiscono alcun valore. Il metodo `getBalance` non ha parametri e restituisce un numero.

Java ha parecchi tipi di numeri, che imparerete nel prossimo capitolo. Il tipo di numero più flessibile si chiama `double`, che significa "numero in virgola mobile in *doppia* precisione". Pensate a un numero nel formato `double` come a un qualsiasi numero che viene visualizzato sul quadro di una calcolatrice, quale `250`, `6.75` oppure `-0.333333333`.

Ora che sapete di poter usare il tipo `double` per i numeri, potete anche scrivere i metodi della classe `BankAccount`:

```
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
```

E ora facciamo la stessa cosa per i costruttori della classe. Come vogliamo costruire un conto bancario? Sembra ragionevole che un'invocazione come

```
BankAccount harrysChecking = new BankAccount();
```

debba costruire un nuovo conto bancario con saldo zero. Come potremmo iniziare con un saldo diverso? Sarebbe utile un secondo costruttore che impostasse il saldo a un valore iniziale:

```
BankAccount harrysChecking = new BankAccount(5000);
```

In questo modo otteniamo due costruttori:

```
public BankAccount()
public BankAccount(double initialBalance)
```

Il compilatore è in grado di individuare quale sia il costruttore da invocare guardando i parametri. Ad esempio, se invocate

```
new BankAccount()
```

allora il compilatore sceglie il primo costruttore. Se invocate

```
new BankAccount(5000)
```

allora il compilatore sceglie il secondo costruttore. Ma se invocate

```
new BankAccount("un sacco di soldi")
```

il compilatore genera un messaggio d'errore, perché questa classe non ha alcun costruttore che riceve un parametro di tipo `String`.

> Metodi sovraccarichi hanno lo stesso nome ma diversi tipi di parametri.

Potreste pensare che sia strano avere due costruttori che hanno lo stesso nome e che differiscono soltanto nel tipo dei parametri (il primo costruttore non ha parametri; il secondo ha un solo parametro, un numero). Se un nome viene usato per indicare più di un costruttore o metodo, si dice che tale nome è *sovraccarico* (*overloaded*). Troverete più informazioni sul sovraccarico dei nomi in Argomenti avanzati 2.2. Il sovraccarico dei nomi è un fenomeno comune in Java, soprattutto per i costruttori, dato che in fondo non ci sono

alternative sul nome dei costruttori: il nome di un costruttore deve essere identico al nome della classe.

I costruttori e i metodi di una classe costituiscono l'*interfaccia pubblica* della classe, l'insieme delle operazioni alle quali qualsiasi parte del codice del vostro programma può accedere per creare e manipolare oggetti di tipo `BankAccount`. Ecco quindi l'elenco completo che costituisce l'interfaccia pubblica della classe `BankAccount`:

```
public BankAccount()
public BankAccount(double initialBalance)
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
```

Il comportamento della classe `BankAccount` è elementare, ma ci permette di svolgere tutte le operazioni principali che si richiedono comunemente per conti bancari. Per esempio, ecco un esempio per trasferire un importo da un conto a un altro:

```
// trasferisci da un conto a un altro
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

Ecco come potete sommare gli interessi a un conto di risparmio:

```
double interestRate = 5; // interessi del 5%
double interestAmount =
   momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

L'astrazione è l'identificazione dell'insieme delle caratteristiche essenziali di una classe.

Come potete vedere, potete utilizzare oggetti della classe `BankAccount` per eseguire attività significative, senza sapere come gli oggetti `BankAccount` conservino i loro dati o come i metodi di `BankAccount` svolgano il loro lavoro. Questo è un aspetto importante della programmazione orientata agli oggetti. Il processo mediante il quale si determinano le caratteristiche di una classe viene chiamato *astrazione*. Pensate a come un dipinto astratto elimina i dettagli non significativi e cerca di rappresentare soltanto le caratteristiche essenziali di un oggetto. Quando progettate l'interfaccia pubblica di una classe, dovete anche identificare quali operazioni siano fondamentali per manipolare gli oggetti nel vostro programma.

Argomenti avanzati 2.2

Sovraccarico

Quando si usa lo stesso nome per più metodi o costruttori, il nome diventa *sovraccarico*. In particolare, accade di frequente per i costruttori, perché tutti i costruttori devono avere lo stesso nome, quello della classe. In Java, potete sovraccaricare metodi e

costruttori, a patto che i tipi dei rispettivi parametri siano diversi. Per esempio, la classe `PrintStream` definisce molti metodi, tutti chiamati `println`, per stampare vari tipi di numeri e per stampare oggetti:

```
public class PrintStream
{
   public void println(String s) { ... }
   public void println(double a) { ... }
   ...
}
```

Quando si chiama il metodo `println`, in questo modo:

```
System.out.println(x);
```

il compilatore esamina il tipo dati di `x`. Se `x` è un valore `String`, chiama il primo metodo; se il valore è `double`, chiama il secondo. Se `x` non corrisponde ad alcun tipo di parametro dei metodi, il compilatore genera un errore.

Ai fini del sovraccarico, il tipo del *valore restituito* non conta. Quindi, non serve a nulla avere due metodi con nomi e tipi di parametro identici, ma con valori restituiti differenti.

2.8 Commentare l'interfaccia pubblica

Usate i commenti di documentazione per descrivere le classi e i metodi pubblici dei vostri programmi.

Mentre realizzate classi e metodi più complessi, dovreste prendere l'abitudine di *commentare* esaurientemente il loro comportamento. In Java, esiste una forma standard molto utile per i *commenti di documentazione*. Se usate tale forma nelle vostre classi, potrete usare il programma `javadoc` per generare automaticamente un insieme chiaro ed elegante di pagine HTML che le descrivono (si vedano i Consigli per la produttività 2.2 per una descrizione di questo programma di utilità).

Un commento di documentazione inizia con i caratteri `/**`, che rappresentano un delimitatore speciale per i commenti che viene usato dal programma di utilità `javadoc`. Quindi descriverete lo *scopo* del metodo. Poi, per ciascun parametro del metodo, inserirete una riga che inizia con `@param`, seguito dal nome del parametro e da una sua breve spiegazione. Infine, inserirete una riga che inizia con `@return`, per descrivere il valore restituito. Non si indica il marcatore `@param` nei metodi che non hanno parametri, né il marcatore `@return` nei metodi il cui valore restituito è `void`.

Il programma di utilità `javadoc` copia la *prima* frase di ciascun commento in una tabella riassuntiva, per cui è bene scrivere tale frase con cura, possibilmente iniziando con una lettera maiuscola e terminando con un punto. Non è necessario che sia una frase grammaticalmente completa, ma dovrebbe mantenere un significato compiuto quando viene estratta dal commento e visualizzata in un riassunto.

Ecco due tipici esempi:

```
/**
   Preleva denaro dal conto bancario.
   @param amount l'importo da prelevare
```

```
*/
public void withdraw(double amount)
{
   realizzazione (completata in seguito)
}
/**
   Ispeziona il saldo attuale del conto corrente.
   @return il saldo attuale
*/
public double getBalance()
{
   realizzazione (completata in seguito)
}
```

I commenti che avete appena visto descrivono *metodi* singoli. Dovreste prendere anche l'abitudine di fornire un breve commento per ciascuna *classe*, per spiegarne lo scopo. La sintassi per i commenti delle classi è molto semplice: basta inserire il commento di documentazione all'inizio della classe:

```
/**
   Un conto bancario ha un saldo che può essere modificato da depositi
   e prelievi.
*/
public class BankAccount
{
   ...
}
```

La vostra prima reazione potrebbe essere: "Ma devo proprio scrivere tutte queste cose?" Questi commenti sembrano molto ripetitivi, ma dovreste comunque trovare il tempo per scriverli, anche se a volte sembra una cosa noiosa. Ci sono tre ragioni per fare ciò.

Prima di tutto, il programma `javadoc` organizzerà i vostri commenti in un insieme chiaro ed elegante di documenti che si possono visualizzare con un *browser* Web, facendo buon uso di quelle frasi apparentemente ripetitive. La prima frase di ogni commento viene inserita in una *tabella riassuntiva* di tutti i metodi della classe (osservare la Figura 8), mentre i commenti di `@param` e di `@return` vanno a formare la descrizione dettagliata di ciascun metodo (osservare la Figura 9). Se qualcuno di tali commenti viene omesso, il programma `javadoc` genera documenti con degli strani spazi vuoti.

In aggiunta, capita spesso di perdere più tempo a pensare se un commento sia banale oppure no, piuttosto che scriverlo e basta. Nella programmazione realistica, i metodi molto semplici sono rari: avere un metodo banale con un commento inutile non è pericoloso, mentre un metodo complicato senza commento può veramente creare problemi nella futura manutenzione del programma. Secondo lo stile standard per la documentazione Java, *ogni* classe, *ogni* metodo, *ogni* parametro e *ogni* valore restituito dovrebbero essere commentati.

Inserite i commenti di documentazione in ogni classe, ogni metodo, ogni parametro e ogni valore restituito.

Infine, scrivere il commento di un metodo *prima* di scriverne il codice è sempre una buona idea, una eccellente verifica della reale comprensione di ciò che si sta programmando. Se non siete in grado di spiegare cosa faccia un metodo o una classe, non siete pronti per scriverne il codice.

Figura 8
Un riassunto dei metodi generato da `javadoc`

Figura 9
Dettagli di metodi generati da `javadoc`

Consigli per la produttività 2.2

Il programma di utilità javadoc

Dovreste sempre inserire i commenti di documentazione nel vostro codice, indipendentemente dal fatto che usiate `javadoc` per produrre la documentazione HTML. Dato, poi, che molte persone ritengono efficace la documentazione HTML, è utile imparare come si esegue `javadoc`.

Da una shell di comandi, si invoca il programma di utilità `javadoc` mediante il comando:

```
javadoc MyClass.java
```

oppure

```
javadoc *.java
```

In seguito, il programma `javadoc` produce un file in formato HTML, chiamato `MyClass.html`, che potete esaminare mediante un browser. Se conoscete l'HTML (consultate il Capitolo 4), potete incorporare marcatori HTML nei commenti, per specificare font o per aggiungere immagini. Forse ancora più importante, `javadoc` fornisce automaticamente *collegamenti ipertestuali* ad altre classi e metodi.

Potete addirittura eseguire `javadoc` prima di realizzare i metodi, lasciando semplicemente vuoti tutti i corpi dei metodi stessi. Non eseguite il compilatore, che segnalerebbe la mancanza degli eventuali valori da restituire: eseguite soltanto `javadoc` sul vostro file per generare la documentazione dell'interfaccia pubblica che state per realizzare.

Lo strumento `javadoc` è magnifico, perché fa una cosa giusta: vi permette di scrivere la documentazione *insieme con il codice*. In questo modo, quando aggiornate il programma, potete vedere immediatamente quale documentazione bisogna aggiornare. Quindi, si spera che la aggiorniate senza esitare. Al termine, eseguite nuovamente `javadoc` per ottenere una nuova pagina HTML, perfettamente impaginata e aggiornata.

Consigli per la produttività 2.3

Combinazioni di tasti per la selezione rapida delle operazioni del mouse

I programmatori passano un mucchio di tempo sulla tastiera e sul mouse. Programmi e documentazione occupano molte pagine e richiedono un sacco di digitazione. I passaggi continui fra editor, compilatore e debugger richiedono un grande uso del mouse. I progettisti di programmi, quali gli ambienti di sviluppo integrati per Java, hanno

aggiunto alcune caratteristiche per facilitare il lavoro, ma tocca a voi scoprirle.

Quasi tutti i programmi presentano un'interfaccia utente con menu e finestre di dialogo. Per scegliere un'attività, selezionate un menu e un sottomenu, oppure selezionate un campo di una finestra di dialogo, inserite i dati richiesti e premete il pulsante OK. Queste sono ottime interfacce utente per i principianti, perché sono facili da padroneggiare, ma sono tremende per un utente abituale. Il passaggio continuo fra la tastiera e il mouse rallenta il lavoro: occorre spostare una mano dalla tastiera, trovare il mouse, muoverlo, premere il pulsante e riportare la mano sulla tastiera. Per questo motivo, la maggior parte delle interfacce utente ha *combinazioni di tasti per la selezione rapida* (*keyboard shortcuts*), ovvero combinazioni di pressioni sui tasti che permettono di svolgere le stesse attività, senza dover affatto passare al mouse.

Tutte le applicazioni di Microsoft Windows usano le seguenti convenzioni:

- Il tasto Alt, più la lettera sottolineata nel nome del menu (come la F in "File"), apre il menu. All'interno del menu, basta digitare il carattere sottolineato nel nome del sottomenu per attivarlo. Per esempio, Alt + F A seleziona "File" e "Apri". Quando le vostre dita conosceranno questa combinazione, potrete aprire file più rapidamente del più veloce virtuoso del mouse.
- All'interno delle finestre di dialogo, il tasto di tabulazione (Tab) è fondamentale, perché sposta il cursore da un'opzione alla successiva. Le frecce di direzione permettono di spostarsi all'interno di un'opzione. Il tasto Invio accetta tutta la finestra di dialogo, mentre il tasto Esc la cancella.
- In un programma con più finestre, generalmente Ctrl + Tab scorre fra le finestre gestite dal programma, per esempio fra quella del codice sorgente e quella degli errori.
- Alt + Tab scorre fra le applicazioni, permettendo di passare rapidamente, per esempio, fra il compilatore e una cartella nel programma di Gestione risorse.
- Per evidenziare il testo, tenete premuto il tasto del maiuscolo e agite sulle frecce di direzione. Poi, usate Ctrl + X, per tagliare il testo, Ctrl + C, per copiarlo, e Ctrl + V, per incollarlo. Sono tasti facili da ricordare: la V assomiglia al segno di inserimento che un redattore userebbe per inserire del testo. La X dovrebbe ricordare la cancellazione del testo, mentre la C è proprio la prima lettera di "copia". (Nondimeno, è anche l'iniziale di "Cut", tagliare in inglese, ma nessuna regola mnemonica è perfetta.) Troverete un promemoria di queste abbreviazioni nel menu Modifica.

Il mouse, naturalmente, conserva la propria funzione nell'elaborazione del testo, per individuare o per selezionare le parole che si trovano nella stessa schermata, ma lontano dal cursore.

Riservatevi un pizzico di tempo per imparare le combinazioni di tasti che i progettisti dei programmi hanno predisposto per voi, e il tempo investito verrà ricompensato molte volte durante la vostra carriera di programmatori. Quando, nel laboratorio di informatica, lavorerete a raffica con le combinazioni di tasti, vi troverete circondati da astanti sbalorditi che si chiederanno come riuscite a farlo.

2.9 Realizzare una classe

Ora che conosciamo l'interfaccia pubblica della classe `BankAccount`, siamo in grado di realizzarla. Come già sapete, occorre scrivere una classe con questi componenti:

```
public class BankAccount
{
   costruttori
   metodi
   campi
}
```

Abbiamo già visto quali costruttori e quali metodi ci servono, vediamo ora di analizzare le variabili istanza, che hanno il compito di conservare lo stato dell'oggetto. Nel caso dei nostri semplici oggetti di tipo conto bancario, lo stato è rappresentato dal saldo attuale del conto (un conto bancario più completo potrebbe avere uno stato più complesso, forse il saldo attuale, il tasso di interesse, la data in cui inviare il successivo estratto conto, e così via). Per ora ci basta quindi una sola variabile istanza:

```
public class BankAccount
{
   ...
   private double balance;
}
```

Si noti che il campo è stato dichiarato con lo specificatore di accesso `private`. Ciò significa che al saldo bancario possono accedere soltanto i costruttori e i metodi *della stessa classe*, cioè `deposit`, `withdraw` e `getBalance`, e non i costruttori o i metodi di altre classi. Come venga gestito lo stato di un conto bancario è un dettaglio privato della classe. Ricordate che la strategia di nascondere i dettagli realizzativi fornendo metodi per accedere ai dati viene chiamata incapsulamento.

La classe `BankAccount` è così semplice che i vantaggi derivanti dall'incapsulamento non sono molto evidenti. In fin dei conti, è sempre possibile conoscere il saldo attuale invocando il metodo `getBalance`, così come è possibile impostare il saldo a un qualsiasi valore invocando `deposit` con un importo opportuno.

Il vantaggio principale dell'incapsulamento consiste nella garanzia che un oggetto non venga posto accidentalmente in uno stato non corretto. Ad esempio, supponiamo di voler garantire che da un conto bancario non vengano mai effettuati prelievi in misura superiore alla disponibilità di denaro. Possiamo semplicemente realizzare il metodo `withdraw` in modo che rifiuti l'esecuzione di un prelievo che porterebbe il saldo a un valore negativo (dovrete attendere fino al Capitolo 5 per vedere come si realizza una tale protezione). D'altra parte, se qualsiasi parte di codice potesse modificare liberamente il campo `balance` di un oggetto `BankAccount`, sarebbe molto semplice memorizzarvi un numero negativo.

Ora che sappiamo di quali metodi abbiamo bisogno e come rappresentare lo stato dell'oggetto, è facile implementare i metodi. Ad esempio, ecco il metodo `deposit`:

```
public void deposit(double amount)
{
   double newBalance = balance + amount;
   balance = newBalance;
}
```

Ecco invece il costruttore senza parametri:

```
public BankAccount()
{
   balance = 0;
}
```

Alla fine del paragrafo trovate la classe `BankAccount` completa, con la realizzazione di tutti i metodi.

Se il vostro ambiente di sviluppo vi consente di costruire oggetti interattivamente, potete collaudare immediatamente questa classe: le Figure 10 e 11 mostrano il collaudo della classe in BlueJ. Altrimenti dovete fornire una classe di test. La classe `BankAccountTest` alla fine di questo paragrafo costruisce un conto bancario, vi deposita e preleva un po' di denaro, e visualizza il saldo finale.

File BankAccount.java

```
/**
   Un conto bancario ha un saldo che può essere modificato da depositi
   e prelievi
*/
public class BankAccount
{
   /**
      Costruisce un conto bancario con saldo uguale a zero
   */
   public BankAccount()
```

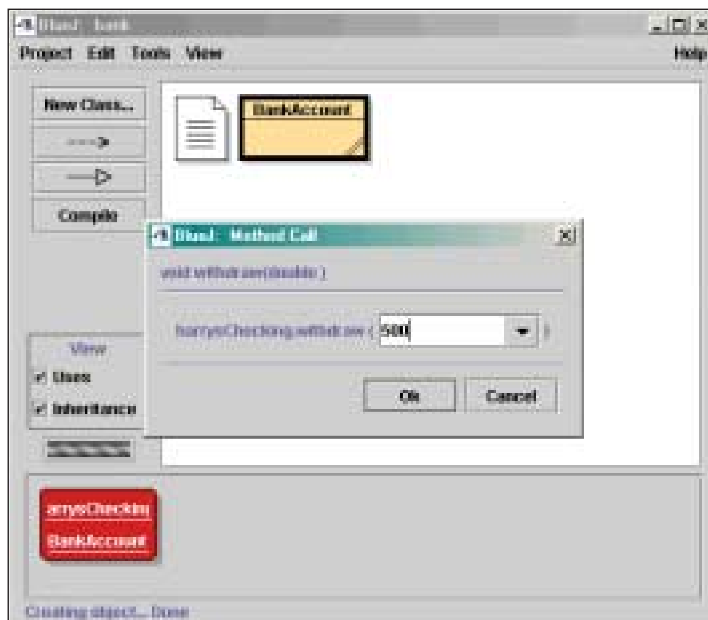


Figura 10
Invocazione del metodo `withdraw` in BlueJ

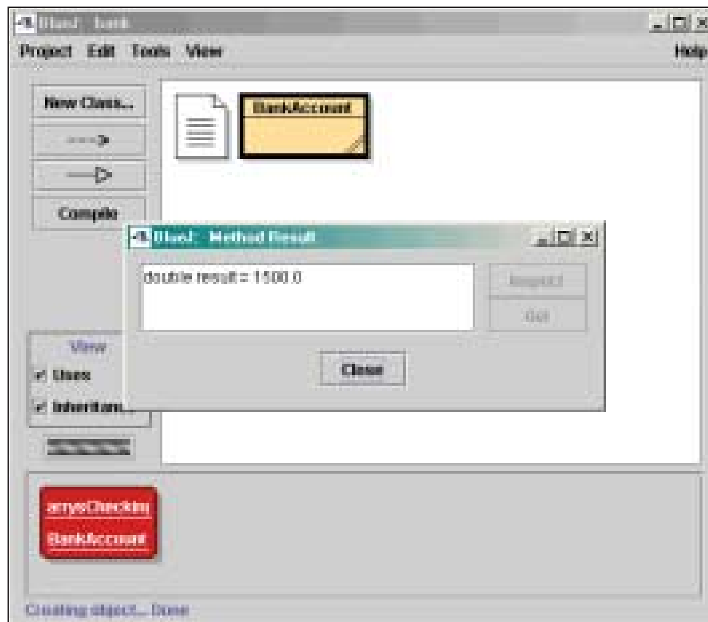


Figura 11
Il valore restituito dal metodo `getBalance` in BlueJ

```
{
   balance = 0;
}

/**
   Costruisce un conto bancario con un saldo assegnato
   @param initialBalance il saldo iniziale
*/
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}

/**
   Versa denaro nel conto bancario
   @param amount l'importo da versare
*/
public void deposit(double amount)
{
   double newBalance = balance + amount;
   balance = newBalance;
}

/**
   Preleva denaro dal conto bancario
   @param amount l'importo da prelevare
*/
public void withdraw(double amount)
{
```

```
        double newBalance = balance – amount;
        balance = newBalance;
    }

    /**
        Ispeziona il valore del saldo attuale del conto bancario
        @return il saldo attuale
    */
    public double getBalance()
    {
        return balance;
    }

    private double balance;
}
```

File BankAccountTest.java

```
/**
    Una classe per il collaudo della classe BankAccount
*/
public class BankAccountTest
{
    /**
        Collauda i metodi della classe BankAccount
        @param args non utilizzato
    */
    public static void main(String[] args)
    {
        BankAccount harrysChecking = new BankAccount();
        harrysChecking.deposit(2000);
        harrysChecking.withdraw(500);
        System.out.println(harrysChecking.getBalance());
    }
}
```

Errori comuni 2.2

Tentare di impostare nuovamente un oggetto mediante la chiamata di un costruttore

Il costruttore si invoca solo quando un oggetto viene creato per la prima volta. Non potete chiamare il costruttore per “ricreare” un oggetto, cioè per riportarlo al suo stato iniziale:

```
BankAccount harrysChecking = new BankAccount();
harrysChecking.withdraw(500);
harrysChecking.BankAccount();   // Errore: non si può ricostruire
                                // l'oggetto
```

Il costruttore assegna a un *nuovo* oggetto di tipo conto bancario un saldo uguale a zero, ma non si può invocare un costruttore su un oggetto *esistente*. La soluzione è semplice: costruite un nuovo oggetto che sovrascriva quello attuale.

```
harrysChecking = new BankAccount(); // Va bene
```

Consigli pratici 2.1

Progettare e realizzare una classe

Questa è la prima sezione di tipo "Consigli pratici" ("HOWTO") che trovate in questo libro. Gli utenti del sistema operativo Linux dispongono di guide "HOWTO" che forniscono risposte alle domande ricorrenti, quali "Da dove inizio?" e "Qual è il passo successivo?", che sorgono nella soluzione dei problemi più vari. In modo assai simile, le sezioni "Consigli pratici" di questo libro forniscono il procedimento passo per passo per portare a termine alcuni compiti specifici.

Vi sarà chiesto spesso di progettare e di realizzare una classe. Ad esempio, in un compito a casa vi si potrebbe chiedere di progettare una classe `Car`.

Passo 1. Capire cosa si deve fare con un oggetto della classe

Supponiamo, ad esempio, che vi sia chiesto di realizzare una classe `Car`. Non avrete bisogno di prendere in considerazione tutte le caratteristiche di un'automobile vera, sono troppe. Il compito assegnato vi dovrebbe specificare *quali aspetti* di un'automobile devono essere simulati dalla vostra classe. Fate un elenco, in linguaggio naturale, delle operazioni che dovrebbero essere svolte da un oggetto della vostra classe, come il seguente:

- Aggiungi carburante al serbatoio.
- Percorri una certa distanza.
- Verifica la quantità di carburante rimasto nel serbatoio.

Passo 2. Assegnare i nomi ai metodi

Assegnate i nomi ai metodi e applicateli a un oggetto in un esempio:

```
Car myBeemer = new Car(...);
myBeemer.addGas(20);
myBeemer.drive(100);
myBeemer.getGas();
```

Passo 3. Scrivere la documentazione per l'interfaccia pubblica

Ecco la documentazione, con i commenti che descrivono la classe e i suoi metodi:

```
/**
   Un'automobile può percorrere una certa distanza e consumare carburante.
*/
```

```
public class Car
{
   /**
      Aggiunge carburante al serbatoio.
      @param amount la quantità di carburante aggiunta
   */
   public void addGas(double amount)
   {
   }

   /**
      Percorre una certa distanza, consumando carburante.
      @param distance la distanza percorsa
   */
   public void drive(double distance)
   {
   }

   /**
      Ispeziona la quantità di carburante rimasta nel serbatoio.
      @return la quantità di carburante
   */
   public double getGas()
   {
   }
}
```

Passo 4. Identificare le variabili istanza

Chiedetevi quali informazioni debba memorizzare un oggetto al proprio interno per svolgere il suo compito. Ricordate che i metodi possono essere invocati in qualsiasi ordine! L'oggetto deve avere memoria interna a sufficienza per gestire tutti i metodi, usando soltanto le sue variabili istanza e i parametri dei metodi. Analizzate ciascun metodo, preferibilmente iniziando dal più semplice o da un metodo particolarmente significativo, e chiedetevi di cosa avete bisogno per portare a termine il compito assegnato al metodo. Create variabili istanza per conservare le informazioni necessarie ai metodi.

Nell'esempio dell'automobile abbiamo bisogno di sapere (o di calcolare) la quantità di carburante presente nel serbatoio, perché ce lo chiede il metodo `getGas`. Ha quindi senso che un oggetto `Car` la conservi:

```
public class Car
{
   ...
   private double gas;
}
```

Conseguentemente, il metodo `addGas` aggiungerà semplicemente una quantità a tale valore. Il metodo `drive` deve ridurre la quantità di carburante nel serbatoio, ma di quanto? Dipende dall'efficienza dell'automobile. Se percorrete 100 miglia e l'automobile può percorrere 20 miglia con un gallone, allora verranno consumati 5 galloni.

Non riceviamo informazioni sull'efficienza dell'automobile dal metodo `drive`, l'automobile la deve conservare:

```
public class Car
{
   ...
   private double gas;
   private double efficiency;
}
```

Passo 5. Identificare i costruttori

Chiedetevi di cosa avete bisogno per costruire un oggetto. Spesso è possibile impostare semplicemente tutti i campi a zero o a un valore costante. Altre volte si ha bisogno di informazioni essenziali, per cui è necessario impostare tali dati all'interno di un costruttore. In alcuni casi serviranno due costruttori: uno che imposta tutti i campi a un valore predefinito e un altro che li imposta a valori forniti dall'utente. Progettate i costruttori che sono necessari.

Nel caso dell'esempio dell'automobile, possiamo iniziare con un serbatoio vuoto, ma abbiamo bisogno di conoscere l'efficienza dell'automobile. Non c'è nessun valore predefinito che sia valido in questo caso, per cui dovrebbe essere un parametro di costruzione. Di solito i nomi dei parametri di costruzione hanno un prefisso "a" o "an", in modo da non entrare in conflitto con i nomi delle variabili istanza.

```
/**
   Costruisce un'automobile con un'efficienza combustibile assegnata.
   @param anEfficiency l'efficienza combustibile dell'automobile
*/
public Car(double anEfficiency)
{
}
```

Passo 6. Realizzare i metodi

Realizzate i metodi e i costruttori della vostra classe, uno alla volta, iniziando dai più semplici. Se vi accorgete di avere problemi con l'implementazione, può darsi che sia necessario tornare a uno dei passi precedenti. Forse il vostro insieme di metodi, individuato nei passi 1, 2 e 3, non era adeguato? Forse non avete i campi giusti? Per un principiante è molto frequente incorrere in un paio di problemi che richiedono di tornare sui propri passi.

Compilate la vostra classe e correggete tutti gli errori di compilazione.

Passo 7. Collaudare la classe

Scrivete un breve programma di collaudo ed eseguitelo. Il programma di collaudo può eseguire semplicemente le invocazioni di metodi che avete visto nel passo 2.

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

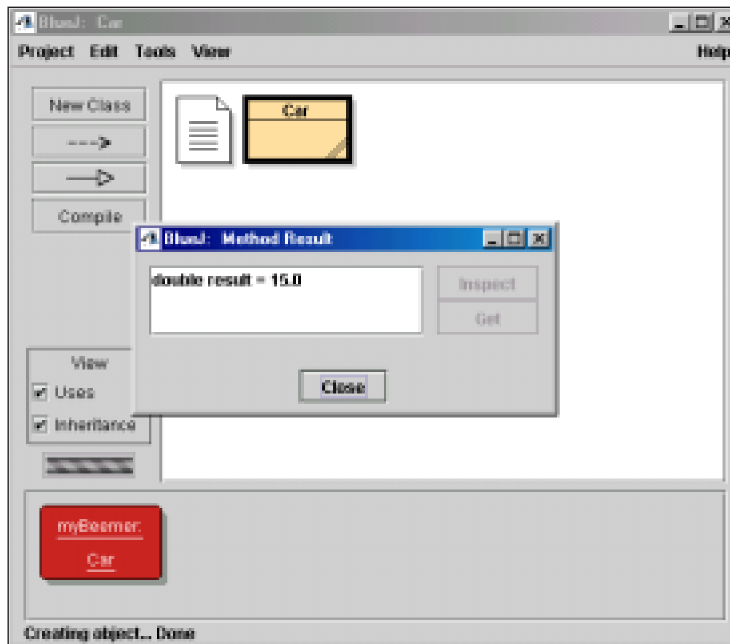


Figura 12
Collaudare una classe con BlueJ

```
            Car myBeemer = new Car(20); // 20 miglia con un gallone
            myBeemer.addGas(20);
            myBeemer.drive(100);
            double gasLeft = myBeemer.getGas();
            System.out.println(gasLeft);
        }
    }
```

In alternativa, se usate un programma che vi consente di collaudare oggetti interattivamente, come BlueJ, costruite un oggetto e applicatevi le invocazioni dei metodi (osservare la Figura 12).

2.10 Tipi di variabili

Avete visto tre diversi tipi di variabili in questo capitolo:

1. Variabili istanza, come la variabile `balance` della classe `BankAccount`
2. Variabili locali, come la variabile `newBalance` del metodo `deposit`
3. Variabili parametro, come la variabile `amount` del metodo `deposit`

Le variabili istanza appartengono a un oggetto. Le variabili parametro e le variabili locali appartengono a un metodo, per cui vengono eliminate quando il metodo termina.

Queste variabili sono simili, tutte conservano un valore di un tipo specifico, ma hanno un paio di differenze importanti. La prima differenza è il *tempo di vita*.

Una variabile istanza appartiene a un oggetto, e ogni oggetto ha la sua copia di ciascuna variabile di istanza.

Ad esempio, se avete due oggetti di tipo `BankAccount` (diciamo `harrysChecking` e `momsSavings`), ciascuno di essi ha il proprio campo `balance`. Quando viene costruito un oggetto, si creano i suoi campi, che rimangono in vita finché qualche metodo usa ancora l'oggetto.

Le variabili locali e la variabili parametro appartengono a un metodo. Quando il metodo inizia la sua esecuzione, le variabili prendono vita; quando il metodo termina, esse muoiono. Ad esempio, se invocate

```
harrysChecking.deposit(500);
```

viene creata una variabile parametro che si chiama `amount`, inizializzata con il valore del parametro, 500. Quando il metodo termina, la variabile scompare. Quando eseguite un'altra invocazione del metodo

```
momsSavings.deposit(1000);
```

viene creata una diversa variabile parametro, chiamata anch'essa `amount`. Anch'essa scompare alla fine del metodo. Lo stesso discorso vale per la variabile locale `newBalance`: quando il metodo `deposit` raggiunge la riga

```
double newBalance = balance + amount;
```

la variabile prende vita e viene inizializzata con la somma del saldo dell'oggetto e dell'importo versato. La durata della vita di tale variabile si estende fino alla fine del metodo, tuttavia il metodo `deposit` ha anche un effetto duraturo. La sua linea successiva

```
balance = newBalance;
```

Le variabili istanza vengono inizializzate a un valore predefinito, ma le variabili locali devono essere inizializzate esplicitamente.

imposta la variabile istanza `balance`, che sopravvive alla fine del metodo `deposit`, finché l'oggetto di tipo `BankAccount` è in uso.

La seconda differenza sostanziale fra le variabili istanza e locali sta nella loro *inizializzazione*.

Le variabili locali devono essere tutte inizializzate. Se non inizializzate una variabile locale, il compilatore segnala l'errore quando tentate di usarla.

Le variabili parametro vengono inizializzate con i valori che sono forniti nell'invocazione del metodo.

La variabili istanza, se non viene esplicitamente assegnato loro un valore in un costruttore, vengono inizializzate con un valore predefinito. I campi numerici vengono inizializzati a 0, mentre i riferimenti a oggetti vengono impostati a un valore speciale chiamato `null`. Se un riferimento a oggetto è `null`, allora esso non si riferisce ad alcun oggetto. Discuteremo il valore `null` in maggior dettaglio nel Paragrafo 5.2. L'inizializzazione inconsapevole a 0 o a `null` è una causa di errori frequente, perciò è un buono stile di programmazione l'inizializzazione di tutte le variabili istanza in tutti i costruttori.

Errori comuni 2.3

Dimenticarsi di inizializzare i riferimenti agli oggetti in un costruttore

Se dimenticarsi di inizializzare una variabile locale è un errore comune, è altrettanto facile trascurare l'inizializzazione delle variabili istanza. Ciascun costruttore deve essere sicuro che tutte le variabili istanza siano impostate su valori corretti.

Se non inizializzate una variabile istanza, il compilatore Java provvederà a farlo per voi. I numeri saranno impostati a 0, ma i riferimenti agli oggetti, quali le variabili stringa, diventeranno un riferimento `null`.

Ovviamente, spesso lo zero è un comodo valore predefinito per i numeri, mentre `null` difficilmente costituisce una scelta opportuna per gli oggetti. Esaminate questo costruttore "pigro" per la classe `Greeter`:

```
public class Greeter
{
   public Greeter() {} // non fa nulla
   ...
   private String name;
}
```

Il campo `name` è impostato al riferimento `null`. Quando invocate `sayHello`, il metodo restituisce `"Hello, null!"`.

Se dimenticate di inizializzare una variabile *locale* in un *metodo*, il compilatore lo segnalerà come un errore, che dovrete correggere prima di eseguire il programma. Se fate lo stesso errore con una variabile *istanza*, il compilatore fornisce l'inizializzazione predefinita, e l'errore diventa evidente solo al momento dell'esecuzione del programma.

Per evitare questo problema, prendete l'abitudine di inizializzare ciascuna variabile istanza in ciascun costruttore.

2.11 Parametri espliciti e impliciti nei metodi

Osservate la particolare invocazione del metodo `deposit`:

```
momsSavings.deposit(500);
```

Ora guardiamo di nuovo il codice del metodo `deposit`:

```
public void deposit(double amount)
{
```

```
    double newBalance = balance + amount;
    balance = newBalance;
}
```

Quando inizia l'esecuzione del metodo, la variabile parametro `amount` viene impostata a 500. Ma cosa significa esattamente `balance`? In fin dei conti, il nostro programma può avere diversi oggetti di tipo `BankAccount`, e *ciascuno di loro* ha il suo saldo.

Il parametro implicito di un metodo è l'oggetto con il quale il metodo viene invocato. Il riferimento `this` indica il parametro implicito.

Dato che versiamo i soldi in `momsSavings`, `balance` deve rappresentare `momsSavings.balance`. Più in generale, quando all'interno di un metodo ci si riferisce a una variabile istanza, si intende la variabile istanza dell'oggetto con il quale è stato invocato il metodo.

L'invocazione del metodo `deposit` dipende quindi da due valori: l'oggetto a cui si riferisce `momsSavings` e il valore `500`. Il parametro `amount` all'interno delle parentesi viene detto parametro *esplicito*, perché viene menzionato esplicitamente nella definizione del metodo. Il riferimento all'oggetto di tipo conto bancario, invece, non è esplicito nella definizione del metodo, e viene quindi detto parametro *implicito* del metodo stesso.

Se è necessario, è possibile accedere al parametro implicito (l'oggetto con cui è stato invocato il metodo) usando la parola chiave `this`. Ad esempio, nella precedente invocazione del metodo, `this` aveva il valore `momsSavings`, mentre `amount` valeva 500.

Ogni metodo ha un parametro implicito, il quale non ha un nome: si chiama sempre `this`. (C'è un'eccezione alla regola che ogni metodo ha un parametro implicito: i metodi `static` non ce l'hanno, come vedremo nel Capitolo 7). I metodi possono, invece, avere un numero qualsiasi di parametri espliciti, ai quali è possibile assegnare un nome qualsiasi, oppure possono anche non avere parametri espliciti.

Procediamo, osservando di nuovo con attenzione l'implementazione del metodo `deposit`. L'enunciato

```
double newBalance = balance + amount;
```

in realtà significa

```
double newBalance = this.balance + amount;
```

L'uso del nome di una variabile istanza in un metodo si riferisce alla variabile istanza del parametro implicito.

Quando in un metodo ci si riferisce a un campo, il compilatore lo considera automaticamente una variabile istanza del parametro `this`. Alcuni programmatori preferiscono inserire manualmente il parametro `this` prima di ogni variabile istanza, perché ritengono che ciò renda il codice più comprensibile. Ecco un esempio:

```
public void deposit(double amount)
{
    double newBalance = this.balance + amount;
    this.balance = newBalance;
}
```

Potete provare anche questo stile e vedere se vi piace.

Errori comuni 2.4

Tentare di chiamare un metodo senza un parametro implicito

Supponiamo che il metodo `main` contenga questa istruzione:

```
withdraw(30); // Errore
```

Il compilatore non sa a quale conto accedere per prelevare il denaro. Pertanto, dobbiamo fornire un riferimento a un oggetto di tipo `BankAccount`:

```
BankAccount harrysChecking = new BankAccount();
harrysChecking.withdraw(30);
```

Nondimeno, esiste un caso in cui è ammesso invocare un metodo, apparentemente senza un parametro implicito. Esaminiamo la seguente modifica della classe `BankAccount`: aggiungiamo un metodo per applicare il canone mensile del conto:

```
class BankAccount
{  ...
   public void monthlyFee()
   {
      withdraw(10); // preleva da questo conto
   }
}
```

Questi enunciati indicano il prelievo dallo *stesso* oggetto di tipo conto corrente che sta eseguendo l'operazione `monthlyFee`. In altre parole, il parametro implicito del metodo `withdraw` è il parametro implicito (che non vediamo) del metodo `monthlyFee`.

Se pensate che un parametro implicito di questo tipo generi confusione, potete sempre usare il parametro `this`, per rendere il metodo più leggibile:

```
class BankAccount
{  ...
   public void monthlyFee()
   {
      this.withdraw(10); // preleva da questo conto
   }
}
```

Argomenti avanzati 2.3

Chiamare un costruttore da un altro

Esaminiamo la classe `BankAccount`, che ha due costruttori: un costruttore senza parametri, per inizializzare il saldo a zero, e un altro costruttore, per fornire un saldo ini-

ziale. Invece di impostare esplicitamente il saldo a zero, un costruttore può invocare un altro costruttore della stessa classe. Per questa operazione, esiste una notazione abbreviata:

```
class BankAccount
{
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   public BankAccount()
   {
      this(0);
   }
   ...
}
```

L'enunciato `this(0);` significa "Chiama un altro costruttore di questa classe e fornisci il valore `0`". Una chiamata di costruttore di questo tipo può comparire solamente *quale prima riga di un altro costruttore*.

Questa sintassi rappresenta una comodità trascurabile, quindi non la useremo nel libro, perché in realtà l'utilizzo della parola chiave `this` genera un po' di confusione. Normalmente, `this` indica un riferimento al parametro implicito, ma, se `this` è seguito da parentesi, esprime la chiamata di un altro costruttore della stessa classe.

Note di cronaca 2.1

I mainframe: quando i dinosauri dominavano la terra

Quando, nei primi anni 50, l'International Business Machines Corporation, un'affermata società che costruiva apparecchiature a schede perforate per l'archiviazione di dati, cominciò a interessarsi alla progettazione di computer, i suoi responsabili della pianificazione stimarono che forse esisteva un mercato per non più di 50 apparecchiature di quel tipo, destinati a organizzazioni governative e militari, e a qualcuna delle più grandi aziende nazionali. Invece, vendettero circa 1500 macchine del loro modello System 650 e iniziarono a produrre e vendere computer più potenti.

I cosiddetti computer *mainframe* degli anni 50, 60 e 70, erano enormi. Riempivano stanze intere, che bisognava climatizzare per proteggerne i delicati meccanismi (osservate la Figura 13). Attualmente, grazie alle tecnologie di miniaturizzazione, anche i mainframe sono diventati più piccoli, però sono ancora molto costosi. (Quando questo libro veniva scritto, un computer IBM 3090 di fascia media costava all'incirca 4 milioni di dollari).

Questi sistemi enormi e costosi ebbero un immediato successo appena apparvero sul mercato, perché prendevano il posto di molti uffici affollati da impiegati ancora

più costosi, che prima svolgevano le attività a mano. Sono pochi fra questi computer a eseguire operazioni di qualche interesse. Per lo più conservano informazioni banali, quali registrazioni di fatture o prenotazioni aeree, e semplicemente ne contengono in gran quantità.

L'IBM non fu la prima società a costruire computer mainframe: questo primato appartiene all'Univac Corporation. Tuttavia, ben presto l'IBM guadagnò il ruolo principale, in parte grazie alla qualità tecnologica e all'attenzione alle necessità dei clienti, e in parte perché sfruttò il suo predominio, strutturando i suoi prodotti e i suoi servizi in modo da rendere difficile ai clienti mescolarli con quelli di altri fornitori. Negli anni 60, i concorrenti dell'IBM, i cosiddetti "Sette Nani" (GE, RCA, Univac, Honeywell, Burroughs, Control Data e NCR), se la videro brutta. Alcuni uscirono completamente dal mercato dei computer, mentre altri tentarono senza successo di combinare le loro forze per fondere le iniziative. Le previsioni generali indicavano una sconfitta finale per tutti loro. Fu in questa atmosfera che il governo degli Stati Uniti intentò una causa antitrust contro l'IBM, nel 1969. Il dibattimento iniziò nel 1975, trascinandosi fino al 1982, quando l'amministrazione Reagan l'abbandonò, dichiarandola "priva di valore".

Naturalmente, da allora il panorama informatico è cambiato completamente. Proprio come i dinosauri cedettero il passo a creature più piccole e agili, sono apparse tre nuove ondate di computer: i minicomputer, le workstation e i microcomputer, tutti progettati da nuove società e non dai Sette Nani. Al giorno d'oggi, il ruolo dei mainframe si è ridimensionato, e l'IBM, che è tuttora una grande società ricca di risorse, non domina più il mercato dei computer.

Oggi i mainframe sono ancora in uso per due ragioni. La prima è che eccellono tuttora nell'elaborazione di grandi volumi di dati. Maggiormente importante, i programmi per la gestione di dati commerciali si sono perfezionati nel corso di 20 e più anni, risolvendo un problema alla volta. Trasferire questi programmi su computer meno costosi, con linguaggi e sistemi operativi diversi, è difficoltoso e genera errori. La Sun

Figura 13
Un computer mainframe

Microsystems, un costruttore di primo piano di workstation, era ansiosa di dimostrare che il suo sistema mainframe si poteva ridimensionare e sostituire con le proprie apparecchiature. La Sun alla fine ci riuscì, ma impiegò oltre cinque anni, molto più di quanto avesse previsto.

Riepilogo del capitolo

1. Gli oggetti sono entità di un programma che possono essere manipolate invocando metodi.
2. L'interfaccia pubblica di una classe specifica ciò che è possibile fare con i suoi oggetti. L'implementazione nascosta descrive come portare a termine questi compiti.
3. Le classi sono fabbriche di oggetti: un nuovo oggetto di una classe viene costruito con l'operatore `new`.
4. Le posizioni degli oggetti vengono memorizzate in variabili oggetto.
5. Tutte le variabili oggetto devono essere inizializzate prima di utilizzarle.
6. Un riferimento a oggetto si riferisce alla posizione dell'oggetto.
7. Più variabili oggetto possono contenere un riferimento allo stesso oggetto.
8. Le classi Java sono raggruppate in pacchetti. Se si usa una classe di un pacchetto diverso da `java.lang` occorre importare la classe.
9. La definizione di un metodo specifica il nome del metodo, i suoi parametri e gli enunciati che devono essere eseguiti per compiere l'azione prevista dal metodo.
10. Per fare in modo che un metodo restituisca un valore a chi l'ha invocato si usa l'enunciato `return`.
11. Per collaudare una classe si usa un ambiente per il collaudo interattivo, oppure si scrive una seconda classe che esegue istruzioni di collaudo.
12. Un oggetto usa le variabili istanza per memorizzare il suo stato, cioè i dati che gli servono per eseguire i suoi metodi.
13. Ogni oggetto di una classe ha il proprio insieme di variabili istanza.
14. L'incapsulamento è la strategia che prevede di nascondere i dati degli oggetti e di fornire eventualmente metodi per accedervi.
15. Le variabili istanza dovrebbero essere sempre dichiarate `private`.
16. I costruttori contengono istruzioni per inizializzare gli oggetti. Il nome di un costruttore è sempre uguale al nome della classe.
17. L'operatore `new` invoca il costruttore.
18. I metodi sovraccarichi sono metodi aventi lo stesso nome ma parametri di tipo diverso.
19. L'astrazione è il processo che identifica l'insieme delle caratteristiche essenziali di una classe.
20. Usate i commenti di documentazione per descrivere le classi e i metodi pubblici dei vostri programmi.

21. Fornite i commenti di documentazione per ogni classe, ogni metodo, ogni parametro e ogni valore di ritorno.
22. Le variabili istanza appartengono a un oggetto. Le variabili parametro e le variabili locali appartengono a un metodo, e scompaiono quando il metodo termina.
23. Le variabili istanza vengono inizializzate a un valore predefinito, mentre le variabili locali devono essere inizializzate esplicitamente.
24. Il parametro implicito di un metodo è l'oggetto con il quale il metodo viene invocato. Il riferimento `this` rappresenta il parametro implicito.
25. Una variabile istanza usata all'interno di un metodo rappresenta la variabile istanza del parametro implicito.

Esercizi di ripasso

Esercizio R2.1. Spiegate la differenza fra un oggetto e un riferimento a oggetto.

Esercizio R2.2. Spiegate la differenza fra un oggetto e una variabile oggetto.

Esercizio R2.3. Spiegate la differenza fra un oggetto e una classe.

Esercizio R2.4. Spiegate la differenza fra un costruttore e un metodo.

Esercizio R2.5. Fornite il codice Java per un *oggetto* della classe `BankAccount` e per una *variabile oggetto* della stessa classe.

Esercizio R2.6. Spiegate la differenza fra una variabile istanza e una variabile locale.

Esercizio R2.7. Spiegate la differenza fra una variabile locale e una variabile parametro.

Esercizio R2.8. Spiegate la differenza fra i seguenti enunciati:

```
new BankAccount(5000);
```

e

```
BankAccount b = new BankAccount(5000);
```

Esercizio R2.9. Spiegate la differenza fra i seguenti enunciati:

```
BankAccount b;
```

e

```
BankAccount b = new BankAccount(5000);
```

Esercizio R2.10. Quali sono i parametri per la costruzione di un oggetto `BankAccount`?

Esercizio R2.11. Fornite il codice Java per costruire questi oggetti:

- Un rettangolo con le coordinate del centro uguali a (100, 100) e con la lunghezza di tutti i lati uguale a 25.
- Un generatore di saluti che dica `"Hello, Mars!"`
- Un conto bancario con un saldo di $ 5000.

Create soltanto gli oggetti, non le variabili oggetto.

Esercizio R2.12. Ripetete l'esercizio precedente, ma definite le variabili oggetto che vengono inizializzate con gli oggetti richiesti.

Esercizio R2.13. Trovate gli errori nei seguenti enunciati:

```
Rectangle r = (5, 10, 15, 20);

double x = BankAccount(10000).getBalance();

BankAccount b;
b.deposit(10000);
b = new BankAccount(10000);
b.add("un sacco di soldi");
```

Esercizio R2.14. Descrivete tutti i costruttori della classe `BankAccount`. Elencate tutti i metodi che si possono usare per modificare un oggetto `BankAccount`. Elencate tutti i metodi che non modificano l'oggetto `BankAccount`.

Esercizio R2.15. Qual è il valore di `b`, dopo le operazioni seguenti?

```
BankAccount b = new BankAccount(10);
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
```

Esercizio R2.16. Esaminate gli enunciati seguenti, in cui `b1` e `b2` contengono oggetti di tipo `BankAccount`.

```
b1.deposit(b2.getBalance());
b2.deposit(b1.getBalance());
```

Dopo l'elaborazione, i saldi di `b1` e `b2` sono identici? Spiegate il perché.

Esercizio R2.17. Che cos'è il riferimento `this`?

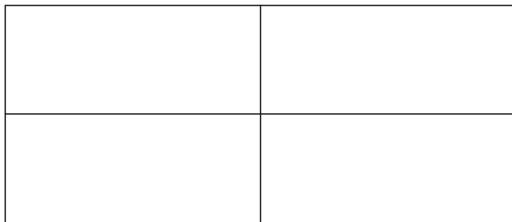
Esercizio R2.18. Cosa fa il metodo seguente? Date un esempio di come si possa invocarlo.

```
public class BankAccount
{
   public void mystery(BankAccount that, double amount)
   {
      this.balance = this.balance - amount;
      that.balance = that.balance + amount;
```

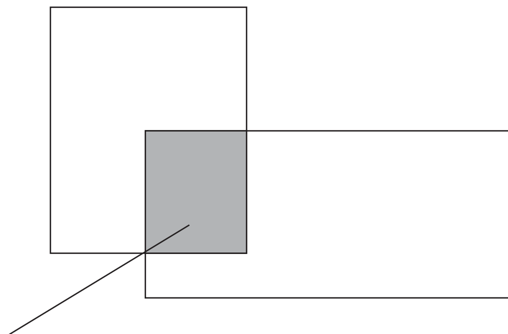
```
    }
    ... // altri metodi del conto bancario
}
```

Esercizi di programmazione

Esercizio P2.1. Scrivete un programma che costruisca un oggetto `Rectangle`, lo stampi e che quindi lo sposti e lo stampi per altre tre volte, in modo che, se i rettangoli fossero disegnati, formerebbero un unico grande rettangolo:



Esercizio P2.2. Il metodo `intersection` calcola l'intersezione di due rettangoli, ovvero il rettangolo formato dalla sovrapposizione parziale di altri due rettangoli:



Il metodo viene invocato in questo modo:

```
Rectangle r3 = r1.intersection(r2);
```

Scrivete un programma che costruisca due oggetti rettangolo, li stampi e che quindi stampi la loro intersezione. Che cosa succede quando i rettangoli non si sovrappongono?

Esercizio P2.3. Aggiungete alla classe `Greeter` il metodo `sayGoodbye`.

Esercizio P2.4. Aggiungete alla classe `Greeter` il metodo `refuseHelp`, che dovrebbe restituire una stringa del tipo `"I am sorry, Dave. I am afraid I can't do that."`

Esercizio P2.5. Scrivete un programma che costruisca un conto bancario, vi versi \$ 1000, prelevi \$ 500, prelevi altri \$ 400 e infine stampi il saldo rimasto.

Esercizio P2.6. Aggiungete alla classe `BankAccount` il metodo

```
void addInterest(double rate)
```

che aggiunga gli interessi calcolati con il tasso indicato. Ad esempio, dopo gli enunciati

```
BankAccount momsSavings = new BankAccount(1000);
momsSavings.addInterest(10); // interesse del 10%
```

il saldo di `momsSavings` è \$ 1100.

Esercizio P2.7. Scrivete una classe `SavingsAccount` (conto di risparmio) che sia simile alla classe `BankAccount`, tranne per una variabile istanza aggiuntiva `interest`. Fornite un costruttore che imposti sia il saldo iniziale sia il tasso di interesse. Fornite un metodo `addInterest` (senza parametri espliciti) che aggiunga gli interessi al conto. Scrivete un programma che costruisca un conto di risparmio con un saldo iniziale di \$ 1000 e tasso di interesse 10%. Quindi, applicate il metodo `addInterest` cinque volte e stampate il saldo risultante.

Esercizio P2.8. Implementate una classe `Employee` (dipendente). Ciascun dipendente ha un nome (di tipo stringa) e uno stipendio (di tipo `double`). Scrivete un costruttore senza parametri, un costruttore con due parametri (nome e stipendio), e i metodi per reperire nome e stipendio. Scrivete un breve programma per collaudare la classe.

Esercizio P2.9. Sviluppate la classe dell'esercizio precedente, aggiungendo un metodo `raiseSalary(double byPercent)`, che incrementi lo stipendio del dipendente secondo una certa percentuale. Ecco un esempio di utilizzo:

```
Employee harry = new Employee("Hacker, Harry", 55000);
harry.raiseSalary(10); // Harry ottiene un aumento del 10 per cento
```

Esercizio P2.10. Implementate una classe `Car` (automobile) con le proprietà seguenti. Un'automobile ha una determinata resa del carburante (misurata in miglia/galloni o in litri/chilometri: scegliete il sistema che preferite) e una certa quantità di carburante nel serbatoio. La resa è specificata dal costruttore e il livello iniziale del carburante è a zero. Fornite questi metodi: un metodo `drive` per simulare il percorso di un'automobile per una certa distanza, riducendo il livello di carburante nel serbatoio; un metodo `getGas`, per ispezionare il livello corrente del carburante; un metodo `addGas`, per fare rifornimento. Ecco un esempio di utilizzo:

```
Car myBeemer = new Car(29); // 29 miglia per gallone
myBeemer.addGas(20); // carica 20 galloni di carburante
myBeemer.drive(100); // viaggia per 100 miglia
System.out.println(myBeemer.getGas());
// stampa la quantità di carburante rimasto
```

Esercizio P2.11. Implementate una classe `Student`. Ai fini di questo esercizio, ciascuno studente ha un nome e un punteggio totale delle risposte. Fornite un costruttore appropriato e i metodi seguenti: `getName()`, per reperire il nome; `addQuiz(int score)`, per sommare il punteggio di ciascuna risposta; `getTotalScore()`, per reperire il punteggio totale; `getAverageScore()`, per ottenere la media dei punteggi. Per calcolare la media, dovete registrare anche il *numero delle domande* ricevute dallo studente.

Esercizio P2.12. Implementate una classe `Product` (prodotto). Ciascun prodotto ha un nome e un prezzo, come per esempio `new Product("Tostapane", 29.95)`. Fornite i seguenti metodi: `getName()`, per ispezionare il nome del prodotto; `getPrice()`, per ispezionarne il prezzo; `setPrice()`, per impostarne il prezzo. Scrivete un programma che crea due prodotti e ne stampa il nome e il prezzo, per poi ridurre i loro prezzi di $5.00 e stamparli nuovamente.

Esercizio P2.13. Implementate una classe `Circle` (cerchio), con i metodi `getArea()`, per ispezionare l'area, e `getPerimeter()`, per ottenere la circonferenza. Nel costruttore, indicate il raggio del cerchio.

Esercizio P2.14. Implementate una classe `Square` (quadrato), con i metodi `getArea()`, per ispezionare l'area, e `getPerimeter()`, per ottenere il perimetro. Nel costruttore, indicate la lunghezza del lato del quadrato.

Esercizio P2.15. Implementate una classe `SodaCan` (lattina di bibita), con i metodi `getSurfaceArea()`, per ottenere l'area della superficie, e `getVolume()`, per il volume. Nel costruttore, specificate l'altezza e il raggio della lattina.

Esercizio P2.16. Implementate una classe `RoachPopulation` che simuli la crescita di una popolazione di scarafaggi. Il costruttore riceve la dimensione della popolazione iniziale di scarafaggi. Il metodo `waitForDoubling` simula un periodo di tempo in cui la popolazione raddoppia. Il metodo `spray` simula una spruzzata di insetticida, che riduce la popolazione del 10%. Il metodo `getRoaches` restituisce il numero attuale di scarafaggi. Realizzate la classe e un programma di collaudo che simuli una cucina che inizia con 10 scarafaggi. Attendete un periodo, spruzzate l'insetticida e stampate il numero di scarafaggi. Ripetete tre volte.

Esercizio P2.17. Implementate una classe `RabbitPopulation` che simuli la crescita di una popolazione di conigli, secondo le regole seguenti. Si inizia con una coppia di conigli. I conigli sono in grado di accoppiarsi all'età di un mese. Un mese dopo, ogni femmina genera un'altra coppia di conigli. Ipotizzate che i conigli non muoiano mai e che le femmine generino sempre una nuova coppia (un maschio e una femmina) ogni mese a partire dal secondo mese. Realizzate un metodo `waitAMonth` che fa trascorrere un mese, e un metodo `getPairs` che stampa il numero attuale di coppie di conigli. Scrivete un programma di collaudo che mostri la crescita della popolazione di conigli per dieci mesi. *Suggerimento*: usate una variabile istanza per le coppie di conigli neonate e un'altra variabile istanza per le coppie di conigli con un'età di almeno un mese.