Classi e Programmazione

Obiettivi del capitolo

- Acquisire familiarità con il procedimento di realizzazione di classi
- Essere in grado di realizzare semplici metodi
- Capire a cosa servono e come si usano i costruttori
- Capire come si accede a variabili di istanza e variabili locali
- Apprezzare l'importanza dei commenti per la documentazione
- Realizzare classi per disegnare forme grafiche

Scatole nere

- Scatola nera: qualcosa che svolge quasi magicamente i propri compiti
- Dispositivo di cui non si conoscono i meccanismi interni di funzionamento
- Dà luogo a incapsulamento, cioè nasconde i dettagli che non sono importanti.
- Va identificato il giusto concetto che rappresenti una certa scatola nera.

Continua…

Scatole nere

- I concetti vengono identificati durante il processo di astrazione.
- Astrazione: processo di eliminazione delle caratteristiche inessenziali, finché non rimanga soltanto l'essenza del concetto.
- Nella programmazione *orientata agli oggetti* ("object-oriented") le scatole nere con cui vengono costruiti i programmi vengono chiamate oggetti.

Livelli di astrazione: un esempio dalla vita reale

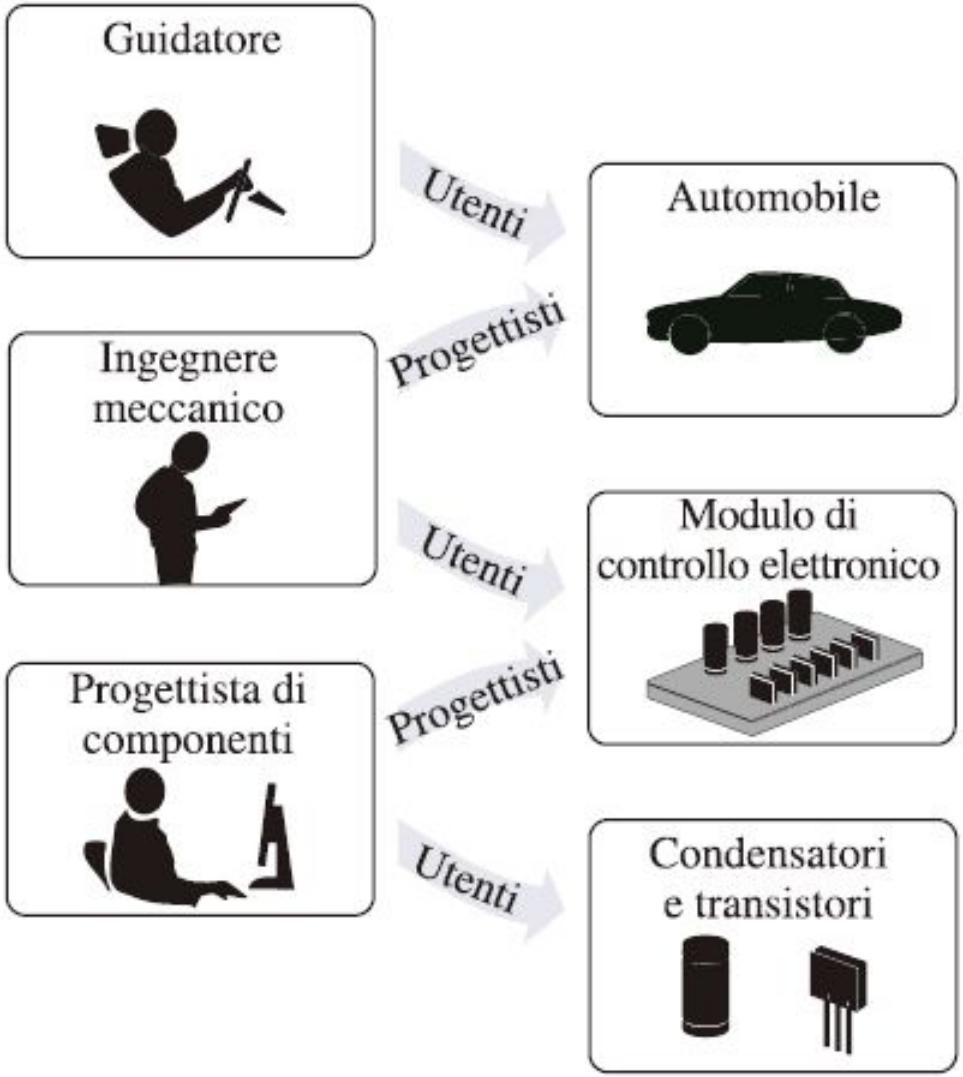


Figura 1
Livelli di astrazione nella progettazione di automobili

Livelli di astrazione: un esempio concreto

- I guidatori non hanno bisogno di capire come funzionano le scatole nere
- L'interazione di una scatola nera con il mondo esterno è ben definita:
  - I guidatori interagiscono con l'auto usando pedali, pulsanti,ecc.
  - I meccanici controllano che i moduli di controllo elettronico mandino il giusto segnale d'accensione alle candele
  - Per i produttori di moduli di controllo elettronico i transistori e i condensatori sono scatole nere magicamente costruite da un produttore di componenti elettronici
- L'incapsulamento porta all'efficienza:
  - Un meccanico si occupa solo di moduli per il controllo elettronico senza preoccuparsi di sensori e transistori
  - I guidatori si preoccupano solo di interagire con l'auto (mettere carburante nel serbatoio) e non dei moduli di controllo elettronico

Livelli di astrazione: progettazione del software

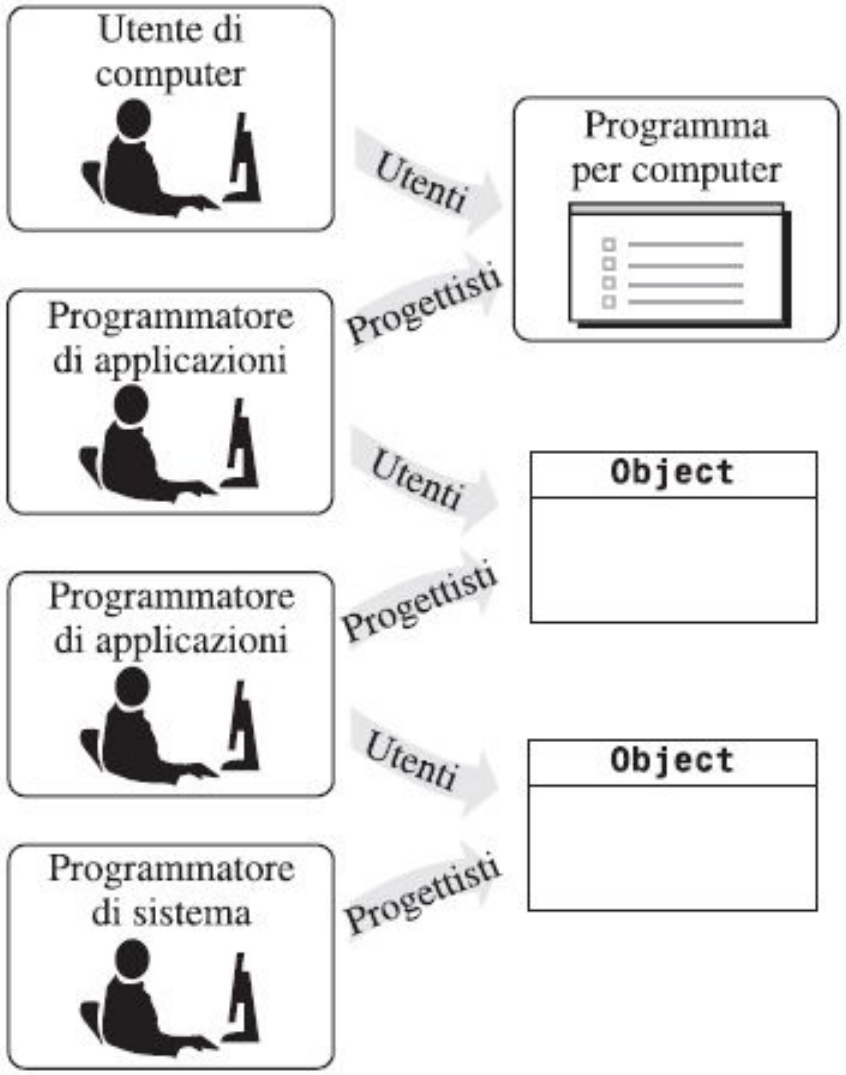


Figura 2
Livelli di astrazione nella progettazione del software

Livelli di astrazione: progettazione del software

- Ai primordi dell'informatica, i programmi per computer erano in grado di manipolare tipi di dati primitivi, come i numeri e i caratteri.
- Con programmi sempre più complessi, ci si trovò a dover manipolare quantità sempre più ingenti di questi dati di tipo primitivo, finché i programmatori non riuscirono più a gestirli.
- Soluzione: incapsularono le elaborazioni più frequenti, generando "scatole nere" software da poter utilizzare senza occuparsi di ciò che avviene all'interno,

Continua…

Livelli di astrazione: progettazione del software

- Fu usato il processo di astrazione per inventare tipi di dati a un livello superiore rispetto a numeri e caratteri.
- Nella programmazione orientata agli oggetti gli oggetti sono scatole nere.
- Incapsulamento: la struttura interna di un oggetto è nascosta al programmatore, che ne conosce però il comportamento.
- Nella progettazione del software si utilizza il processo di astrazione per definire il comportamento di oggetti non ancora esistenti e, dopo averne definito il comportamento, possono essere *realizzati* (o “implementati”)

Progettare l'interfaccia pubblica di una classe

- Progettare la classe `BankAccount`
- Procedimento di astrazione:
  - operazioni irrinunciabili per un conto bancario
    - versare denaro
    - prelevare denaro
    - conoscere il saldo attuale

Progettare l'interfaccia pubblica di una classe: i metodi

- Metodi della classe `BankAccount`:

```
deposit
withdraw
getBalance
```

- Vogliamo che i metodi possano funzionare nel modo seguente:

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

Progettare l'interfaccia pubblica di una classe: definizione di metodo

Ogni definizione di metodo contiene

- uno *specificatore di accesso* (solitamente `public`)
- il *tipo di dati restituito* (come `void` or `double`)
- il nome del metodo (come `deposit`)
- un elenco dei *parametri* del metodo, racchiusi fra parentesi (come `double amount`)
- il *corpo* del metodo: enunciati racchiusi fra parentesi graffe `{ }`

Esempi:

```
public void deposit(double amount) { . . . }
public void withdraw(double amount) { . . . }
public double getBalance() { . . . }
```

Sintassi 3.1: Definizione di metodo

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

Esempio:

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

Serve a:

Definire il comportamento di un metodo.

Progettare l'interfaccia pubblica di una classe: definizione di costruttore

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

```
public BankAccount()
{
   // corpo, che verrà riempito più tardi
}
```

Continua…

Progettare l'interfaccia pubblica di una classe: definizione di costruttore

- Il corpo del costruttore è una sequenza di enunciati che viene eseguita quando viene costruito un nuovo oggetto.
- Gli enunciati presenti nel corpo del costruttore imposteranno i valori dei dati interni dell'oggetto che è in fase di costruzione.
- Tutti i costruttori di una classe hanno lo stesso nome, che è il nome della classe.
- Il compilatore è in grado di distinguere i costruttori, perché richiedono parametri diversi.

Sintassi 3.2: Definizione di costruttore

```
specificatoreDiAccesso nomeClasse(tipoParametro nomeParametro,
                                  ...)
{
    corpo del costruttore
}
```

Esempio:

```
public BankAccount(double initialAmount)
{
   . . .
}
```

Serve a:

Definire il comportamento di un costruttore

Interfaccia pubblica

- I costruttori e i metodi pubblici di una classe costituiscono la sua interfaccia pubblica:

```
public class BankAccount
{
   // Costruttori
   public BankAccount()
   {
      // corpo, che verrà riempito più avanti
   }
   public BankAccount(double initialBalance)
   {
      // corpo, che verrà riempito più avanti
   }
   // Metodi
   public void deposit(double amount)
```

Continua

Interfaccia pubblica

```
   {
      // corpo, che verrà riempito più avanti
   }
   public void withdraw(double amount)
   {
      // corpo, che verrà riempito più avanti
   }
   public double getBalance()
   {
      // corpo, che verrà riempito più avanti
   }
   // campi privati, definiti più avanti
}
```

Sintassi 3.3: Definizione di classe

```
specificatoreDiAccesso class nomeClasse
{
    costruttori
    metodi
    campi
 }
```

Esempio:

```
public class BankAccount
{
    public BankAccount(double initialBalance) { . . . }
    public void deposit(double amount) { . . . }
   . . .
}
```

Serve a:

Definire una classe, la sua interfacia pubblica e i suoi dettagli realizzativi.

Commentare l'interfaccia pubblica

```
/**
   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)
}
```

Commento per la classe

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

- Scrivete commenti di documentazione
  - per ogni classe,
  - per ogni metodo,
  - per ogni parametro,
  - per ogni valore restituito.

Riassunto dei metodi generato da javadoc

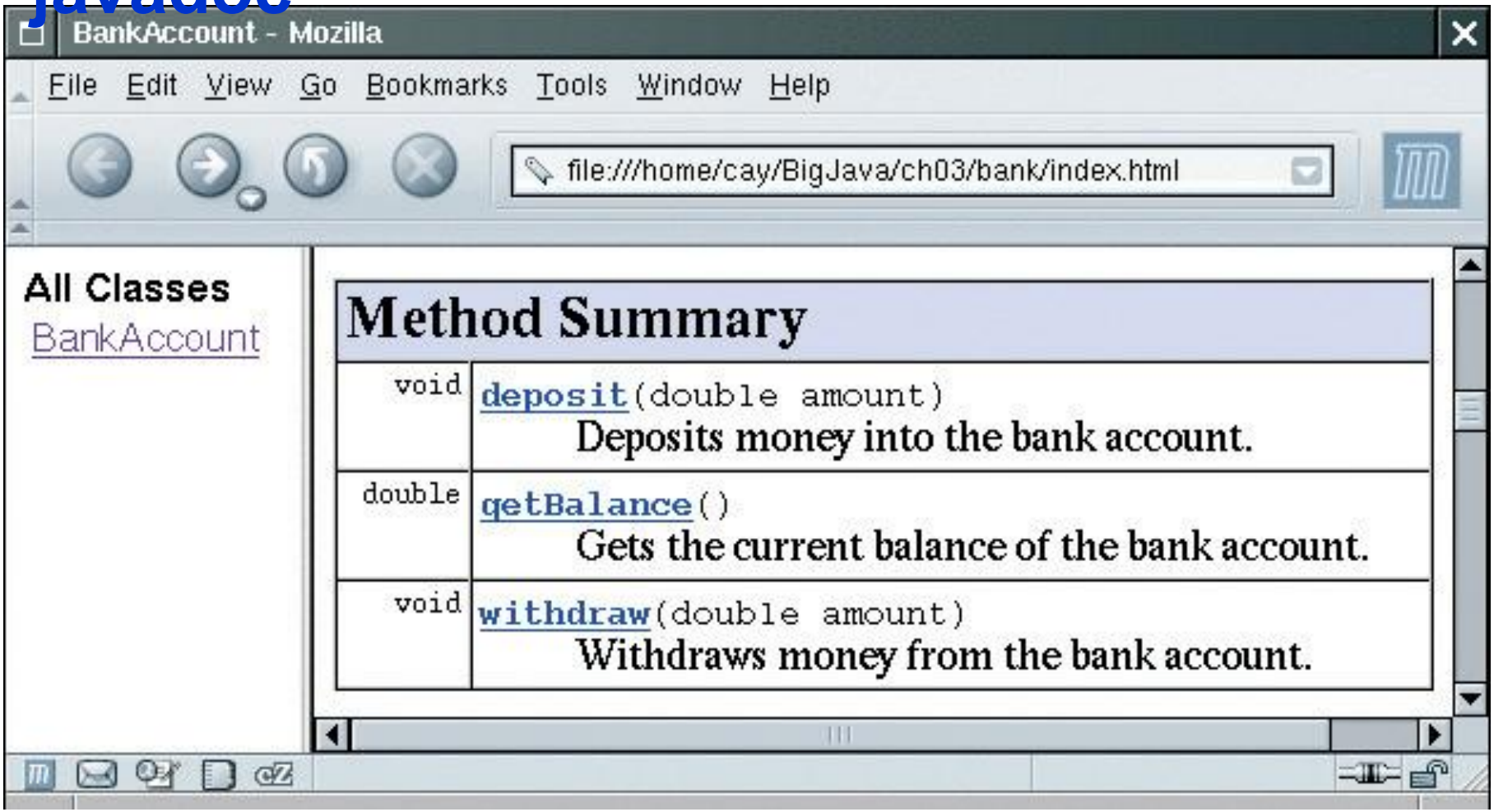


Figura 3 Un riassunto dei metodi generato da `javadoc`

Dettaglio di metodi generato da javadoc

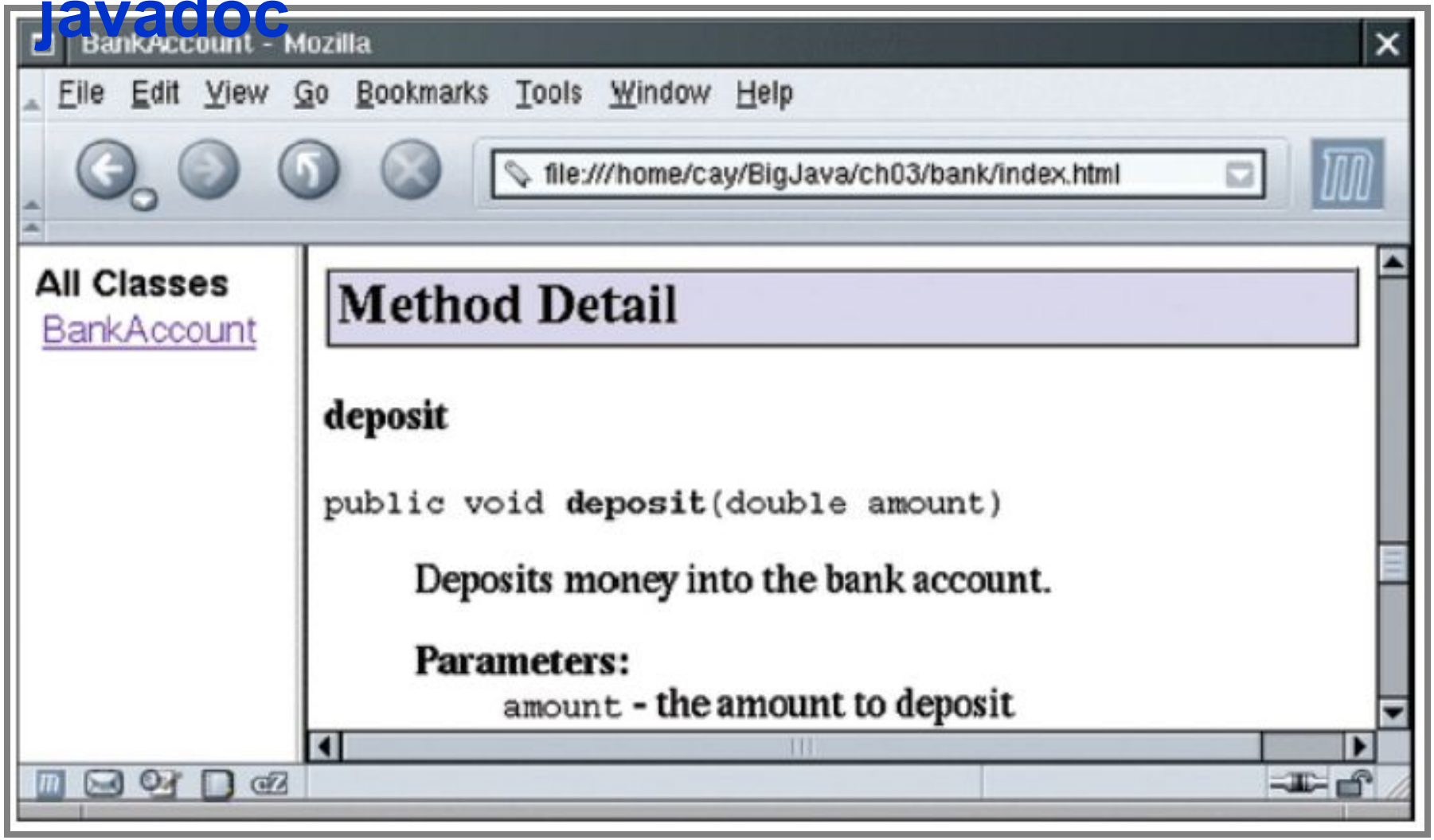


Figura 4 Dettaglio di metodi generato da `javadoc`

Variabili di Istanza (Campi di esemplare)

- Un oggetto memorizza i propri dati all'interno di *campi* (o *variabili*) *di esemplare* (o *di istanza*)
- *Campo* è un termine tecnico che identifica una posizione all'interno di un blocco di memoria
- Un *esemplare* (o *istanza*) di una classe è un oggetto creato da quella classe.
- La dichiarazione della classe specifica le sue variabili di istanza:

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

Variabili di Istanza

- La dichiarazione di una variabile di istanza è così composta:
  - Uno *specificatore d'accesso* (solitamente `private`)
  - Il *tipo* del campo di esemplare (come `double`)
  - Il nome del campo di esemplare (come `balance`)
- Ciascun oggetto di una classe ha il proprio insieme di variabili di istanza.
- Le variabili di istanza sono generalmente dichiarate con lo specificatore di accesso `private`

Variabili di Istanza

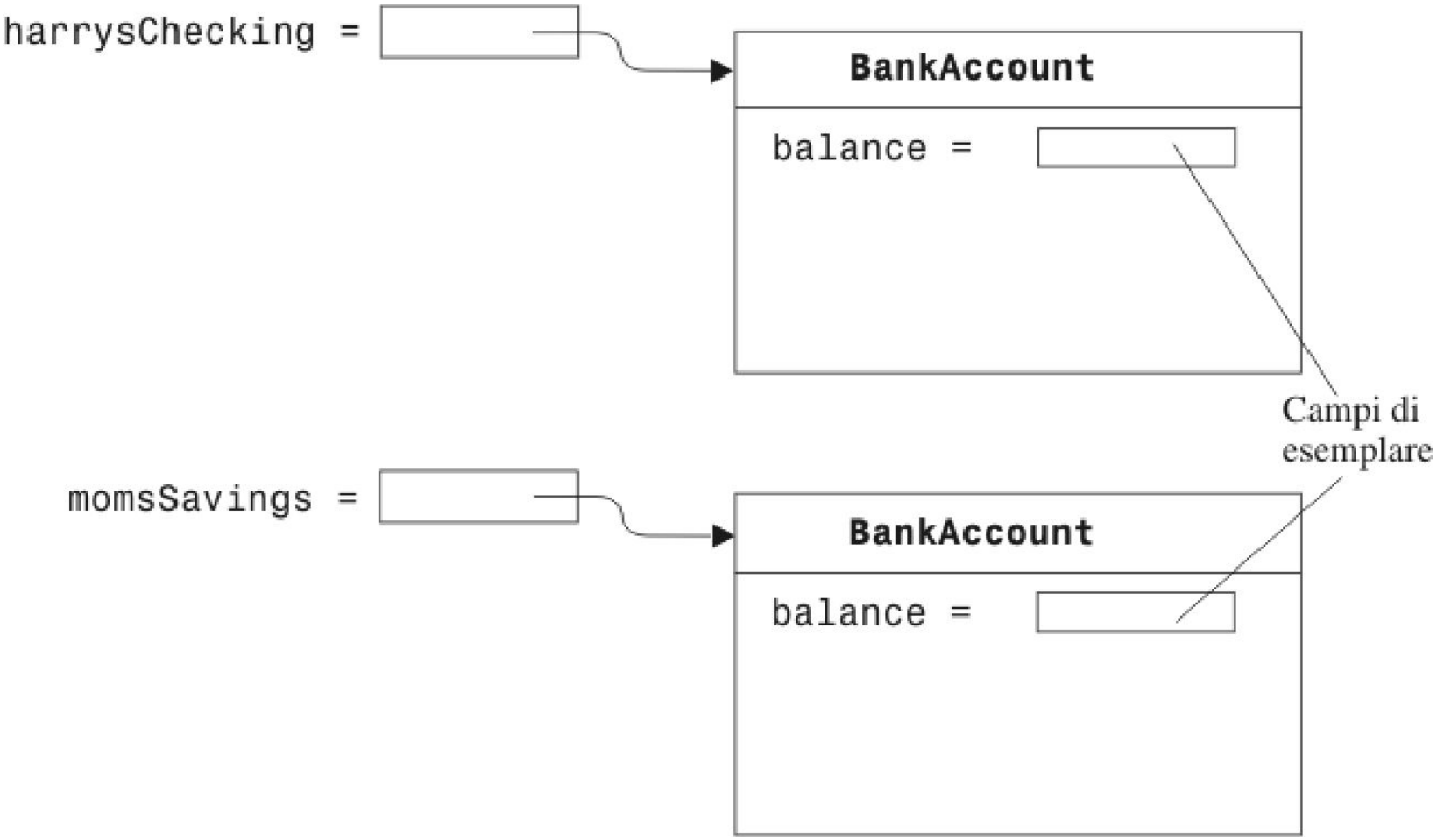


Figura 5
Campi di esemplare

Sintassi 3.4:
Dichiarazione di Variabile di Istanza

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

Esempio:

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

Serve a:

Definire un campo che sia presente in ciascun oggetto di una classe.

Accedere alle variabili di istanza

- Le variabili di istanza sono generalmente dichiarati con lo specificatore di accesso `private`: a essi si può accedere soltanto da metodi della medesima classe e da nessun altro metodo.
- Se i campi di esemplare (o variabili di istanza) vengono dichiarati privati, ogni accesso ai dati deve avvenire tramite metodi pubblici.
- L'incapsulamento prevede l'occultamento dei dati degli oggetti, fornendo metodi per accedervi.

Continua

Accedere ai campi di esemplare

- Ad esempio, alla variabile `balance` si può accedere dal metodo `deposit` della classe `BankAccount`, ma non dal metodo `main` di un'altra classe.

```
public class BankRobber
{
   public static void main(String[] args)
   {
     BankAccount momsSavings = new BankAccount(1000);
     . . .
     momsSavings.balance = -1000; // ERRORE
   }
}
```

Realizzare i costruttori

- Un costruttore assegna un valore iniziale alle variabili di istanza di un oggetto.

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

Esempio di invocazione di un costruttore

```
BankAccount harrysChecking = new BankAccount(1000);
```

- Creazione di un nuovo oggetto di tipo `BankAccount`.
- Invocazione del secondo costruttore (perché è stato fornito un parametro di costruzione).
- Assegnazione del valore 1000 alla variabile parametro `initialBalance`.
- Assegnazione del valore di initialBalance al campo di esemplare `balance` dell'oggetto appena creato.
- Restituzione, come valore dell'espressione `new`, di un riferimento a un oggetto, che è la posizione in memoria dell'oggetto appena creato.
- Memorizzazione del riferimento all'oggetto nella variabile `harrysChecking`.

Realizzare metodi

- Alcuni metodi non restituiscono un valore

```
public void withdraw(double amount)
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

- Alcuni metodi restituiscono un valore

```
public double getBalance()
{
   return balance;
}
```

Esempio di una invocazione di metodo

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

- Assegnazione del valore 500 alla variabile parametro `amount`.
- Lettura del campo balance dell'oggetto che si trova nella posizione memorizzata nella variabile `harrysChecking`.
- Addizione tra il valore di `amount` e il valore di `balance`, memorizzando il risultato nella variabile `newBalance`.
- Memorizzazione del valore di `newBalance` nel campo di esemplare `balance`, sovrascrivendo il vecchio valore.

Sintassi 3.5: L'enunciato `return`

```
  return espressione;
```
oppure
```
  return;
```

Esempio:
```
 return balance;
```

Serve a:
Specificare il valore restituito da un metodo e terminare l'esecuzione del metodo immediatamente. Il valore restituito diventa il valore dell'espressione di invocazione del metodo.

File BankAccount.java

```
01: /**
02:    Un conto bancario ha un saldo che può essere modificato
03:    da depositi e prelievi.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Costruisce un conto bancario con saldo uguale a zero.
09:    */
10:    public BankAccount()
11:    {
12:       balance = 0;
13:    }
14:
15:    /**
16:       Costruisce un conto bancario con un saldo assegnato.
17:       @param initialBalance il saldo iniziale
18:    */
```

Continua

File BankAccount.java

```
19:    public BankAccount(double initialBalance)
20:    {
21:       balance = initialBalance;
22:    }
23:
24:    /**
25:       Versa denaro nel conto bancario.
26:       @param amount l'importo da versare
27:    */
28:    public void deposit(double amount)
29:    {
30:       double newBalance = balance + amount;
31:       balance = newBalance;
32:    }
33:
34:    /**
35:       Preleva denaro dal conto bancario.
36:       @param amount l'importo da versare
```

Continua

File BankAccount.java

```
37:     */
38:     public void withdraw(double amount)
39:     {
40:        double newBalance = balance - amount;
41:        balance = newBalance;
42:     }
43:
44:     /**
45:        Ispeziona il valore del saldo attuale del conto bancario.
46:        @return il saldo attuale
47:     */
48:     public double getBalance()
49:     {
50:        return balance;
51:     }
52:
53:     private double balance;
54: }
```

Collaudare una classe

- *Collaudo di unità:* verifica che la classe a sè stante funzioni correttamente, al di fuori di un programma completo
- Per collaudare una classe usate un ambiente per il collaudo interattivo oppure scrivete un'altra classe che esegua istruzioni di collaudo
- *Classe di test*: una classe con un metodo main che contiene gli enunciati che servono al collaudo di un'altra classe.
- Esegue 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 risultati.
  4. Visualizzazione dei risultati previsti.

Continua

Collaudare una classe

- I dettagli per costruire un 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.

File BankAccountTester.java

```
01: /**
02:    Classe di collaudo per la classe BankAccount.
03: */
04: public class BankAccountTester
05: {
06:    /**
07:       Collauda i metodi della classe BankAccount.
08:       @param args non utilizzato
09:    */
10:    public static void main(String[] args)
11:    {
12:       BankAccount harrysChecking = new BankAccount();
13:       harrysChecking.deposit(2000);
14:       harrysChecking.withdraw(500);
15:       System.out.println(harrysChecking.getBalance());
16:    }
17: }
```

Collaudo con BlueJ

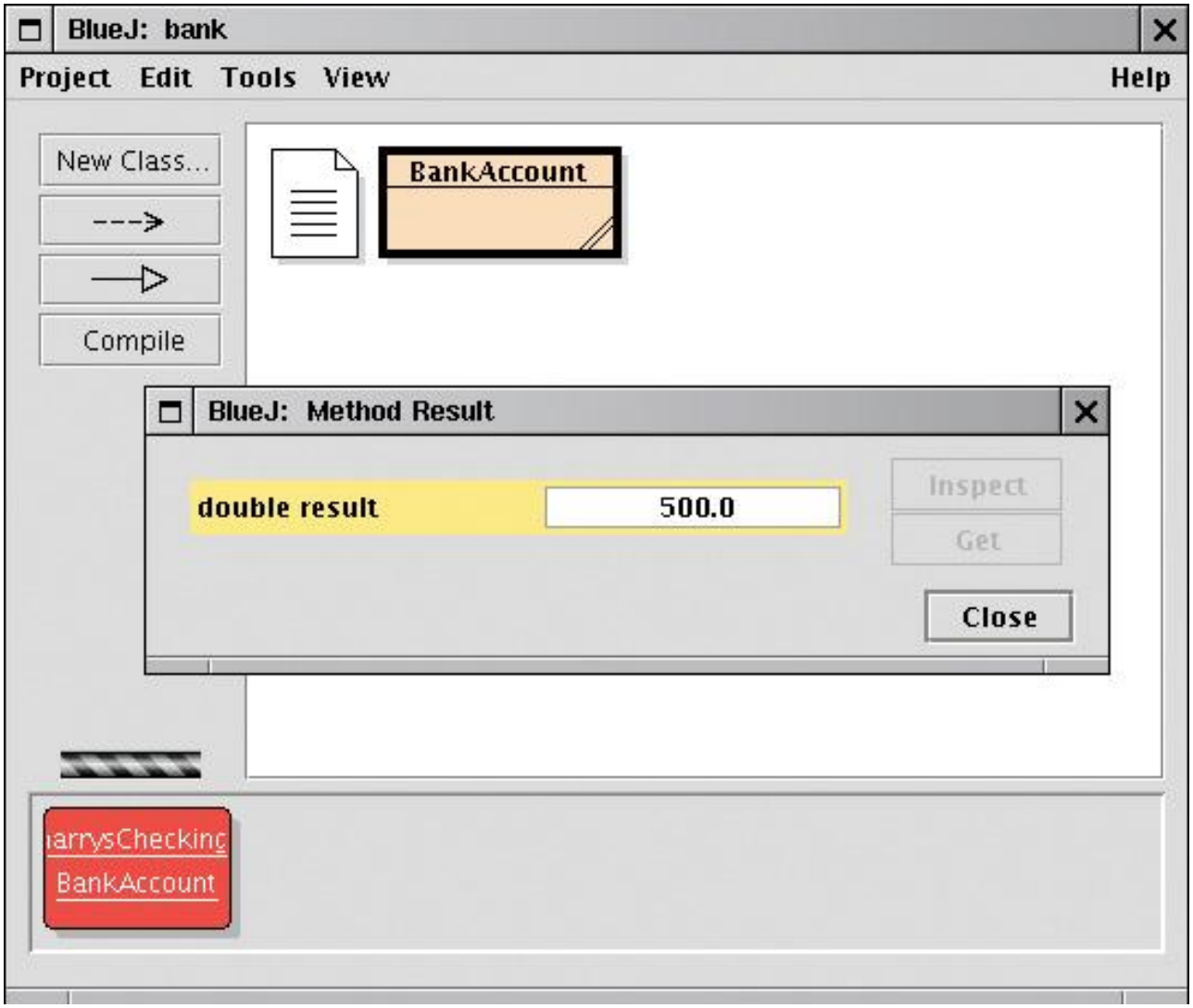


Figura 6: Il valore restituito dal metodo `getBalance` con BlueJ

Categorie di variabili

- Categorie di variabili:
  - *Campi di esemplare* (a volte chiamati anche *variabili di esemplare* o *di istanza*), come la variabile `balance` della classe `BankAccount`.
  - *Variabili locali*, come la variabile `newBalance` del metodo `deposit`
  - *Variabili parametro*, come la variabile `amount` del metodo `deposit`.
- Un campo di esemplare appartiene a un oggetto
- Quando viene costruito un oggetto, vengono creati anche i suoi campi di esemplare, che continuano a “vivere” finché c’è almeno un metodo che sta usando l’oggetto.

Categorie di variabili

- La macchina virtuale Java è dotata di un agente, chiamato *garbage collector*, “raccoglitore di spazzatura”, che periodicamente elimina gli oggetti che non sono più utilizzati.
- Le variabili locali e le variabili parametro appartengono a un metodo
- I campi di esemplare vengono inizializzati a un valore predefinito, mentre le variabili locali devono essere inizializzate esplicitamente.

Tempo di vita (*lifetime*) delle variabili

```
harrysChecking.deposit(500);
double newBalance = balance + amount;
balance = newBalance;
```

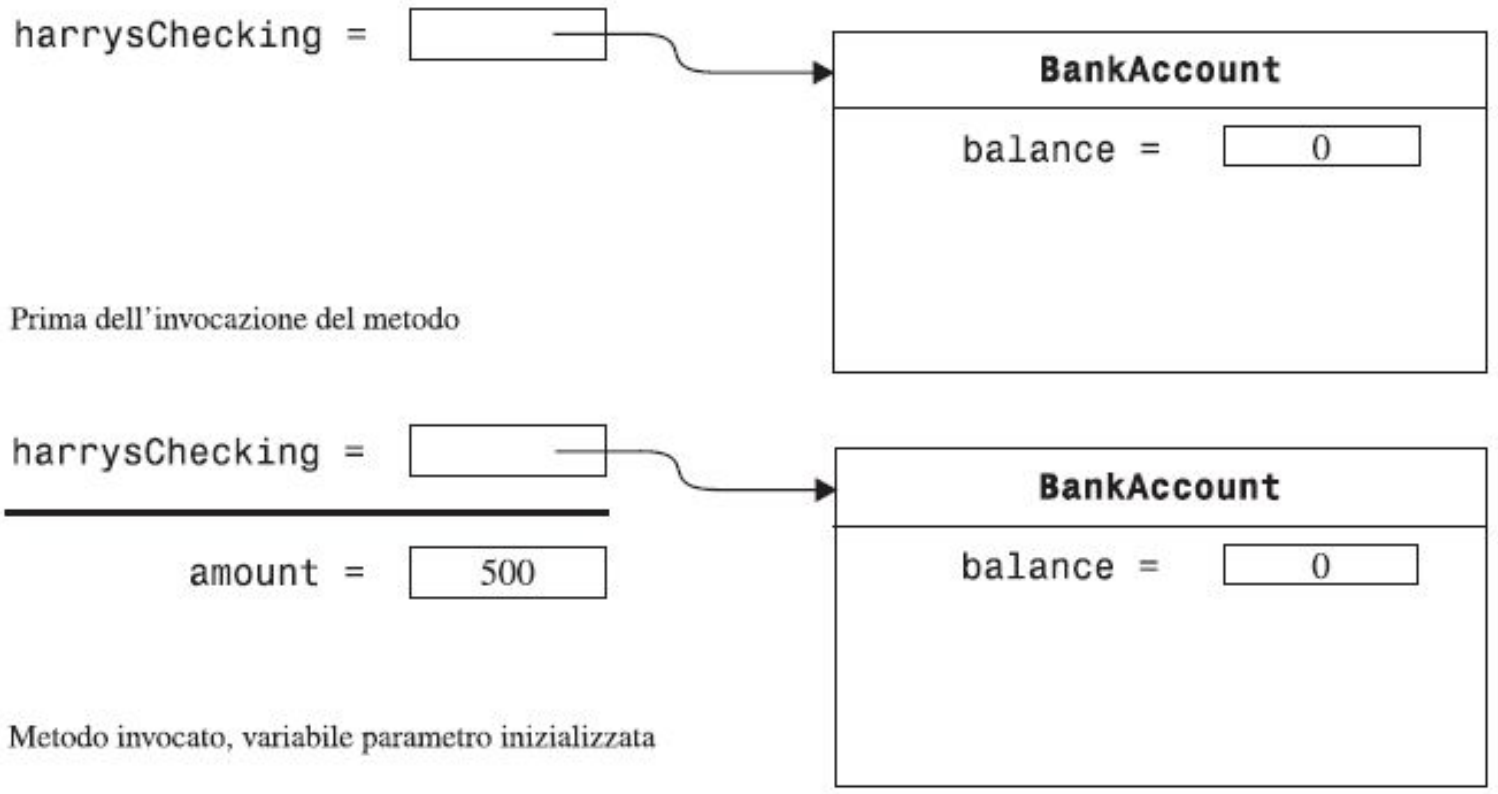


Figura 7a Tempo di vita delle variabili

Tempo di vita (*lifetime*) delle variabili

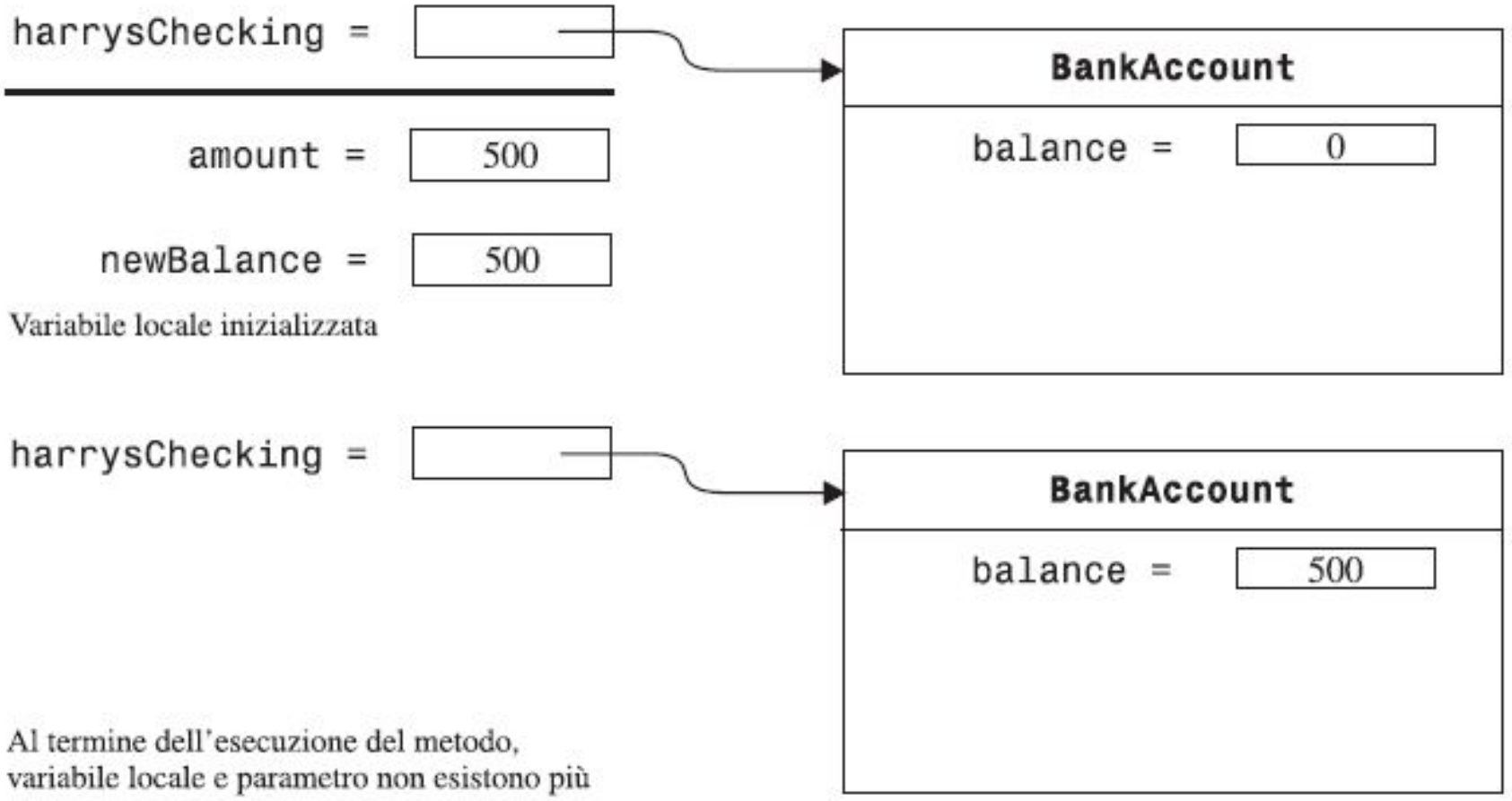


Figura 7b Tempo di vita delle variabili

Parametri impliciti ed espliciti nei metodi

- Il parametro implicito di un metodo è l'oggetto con cui il metodo viene invocato ed è rappresentato dal riferimento `this`.
- Il riferimento `this` permette di accedere al parametro implicito.
- All'interno di un metodo, il nome di un campo di esemplare rappresenta il campo di esemplare del parametro implicito.

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

Parametri impliciti ed espliciti nei metodi

- `balance` rappresenta il saldo dell'oggetto a sinistra del punto:

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

significa

```
double newBalance = momsSavings.balance - amount;
momsSavings.balance = newBalance;
```

Implicit Parameters and `this`

- Ogni metodo ha un suo parametro implicito.
- Il parametro implicito si chiama sempre `this`.
- Eccezione: i metodi dichiarati `static` non hanno alcun parametro implicito, come vedremo nel cap.8.

```
double newBalance = balance + amount;
// ha il seguente significato:
double newBalance = this.balance + amount;
```

- Quando all'interno di un metodo si fa riferimento a un campo di esemplare, il compilatore fa riferimento automaticamente al parametro `this`.

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

Parametri impliciti e `this`

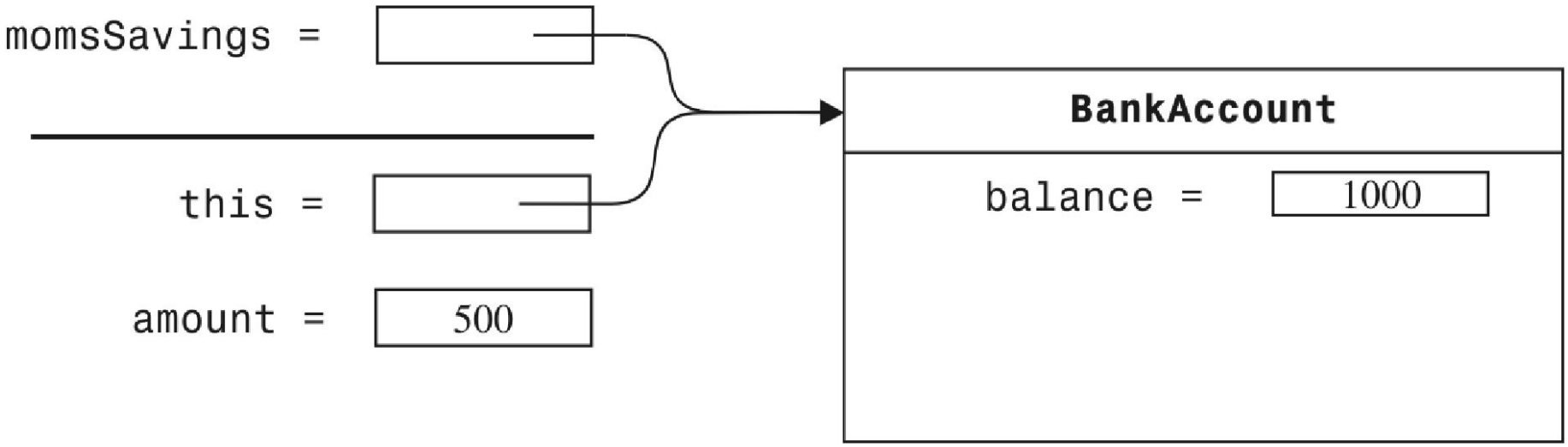
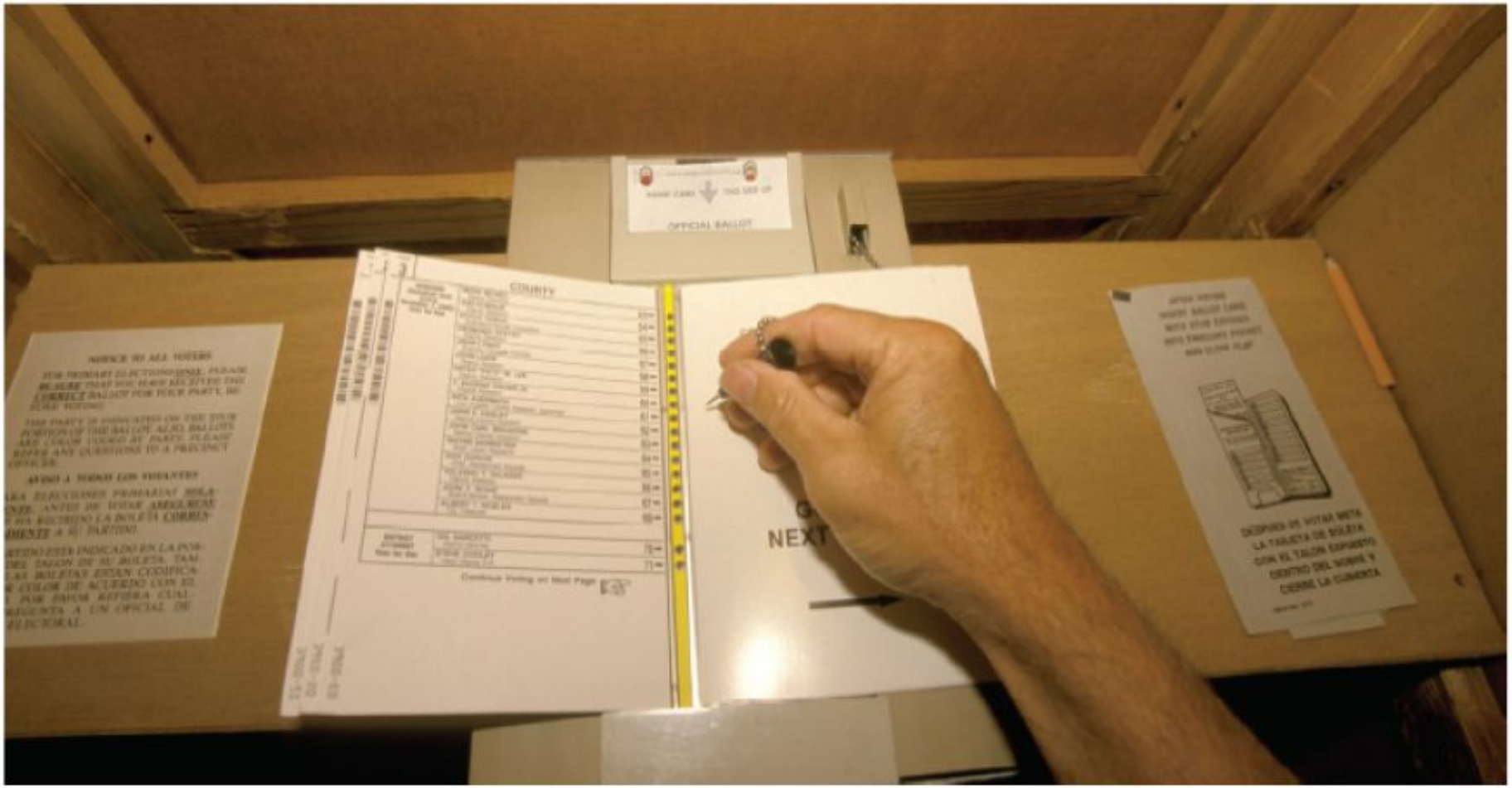
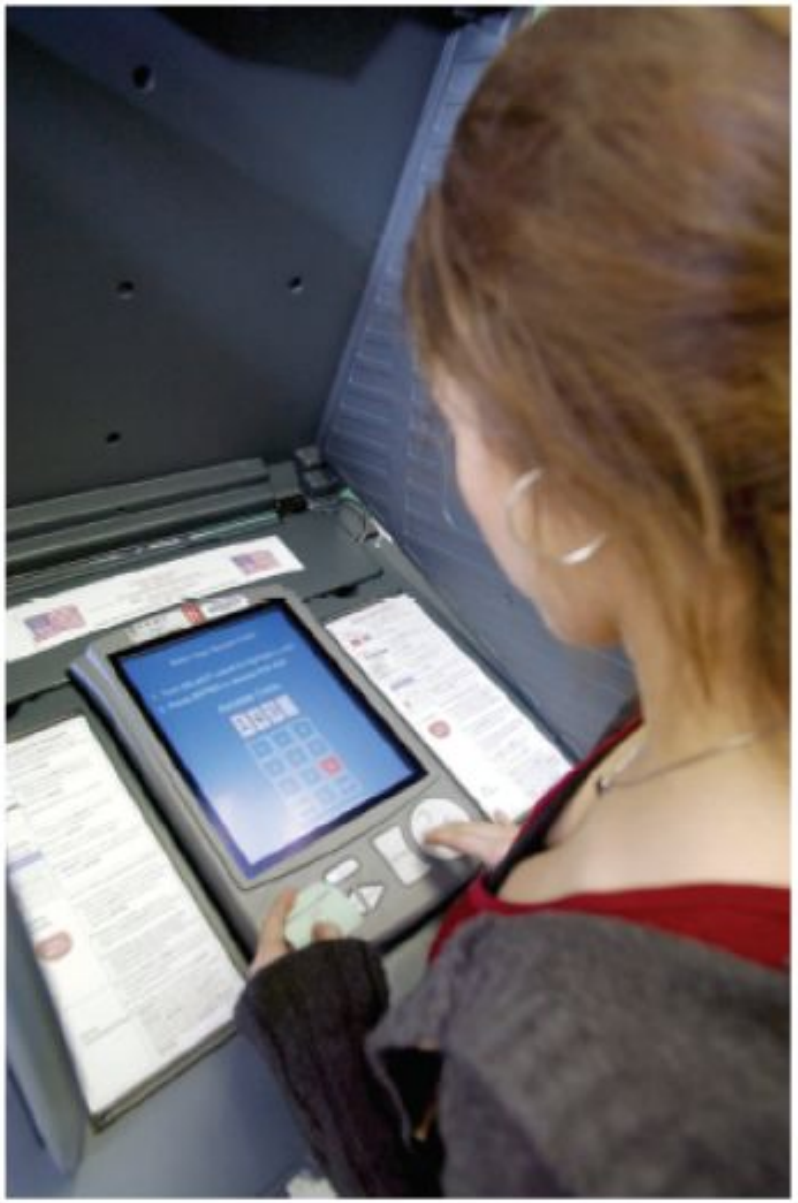


Figura 8: Il parametro implicito nell'invocazione di un metodo

Scheda elettorale a perforazione



Votazione con schermo a sfioramento (touch screen)



Disegnare figure complesse

E' un'ottima idea creare una classe separata per ogni figura complessa

```
public class Car
{
    public Car(int x, int y)
    {
       // Posizione
       . . .
    }
    public void draw(Graphics2D g2)
    {
       // istruzioni per il disegno
       . . .
    }
}
```

Disegnare automobili

- Disegnare due automobili: una nell'angolo superiore sinistro della finestra e un'altra nell'angolo inferiore destro.

- Calcolare la posizione in basso a destra nel metodo `paintComponent`:

```
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```

- `getWidth` e `getHeight` vengono applicati ad un oggetto che sta utilizzando il metodo `paintComponent`.

- Se la finestra viene ridimensionata, viene nuovamente invocato il metodo `paintComponent` e la posizione viene ricalcolata.

Disegnare automobili

Figura 11: Un componente che disegna due automobili

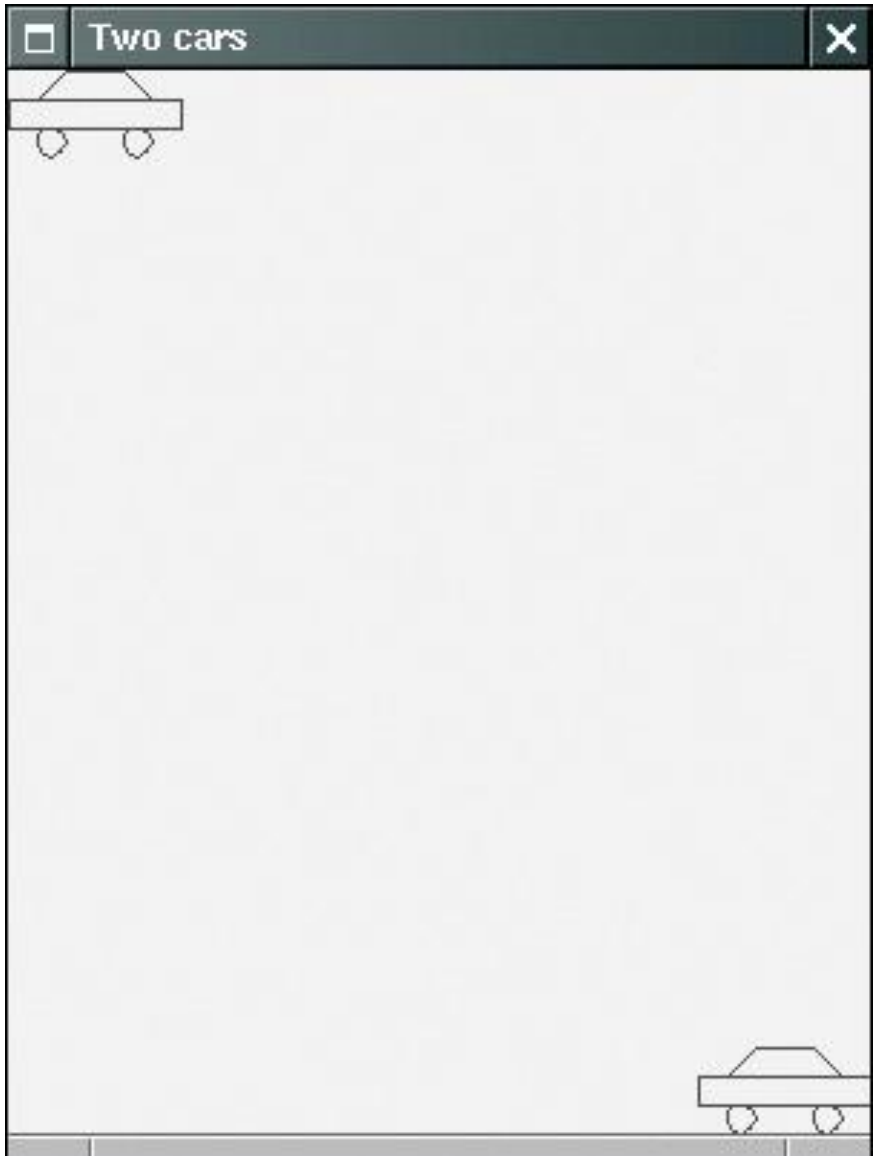


Figure complesse su carta millimetrata

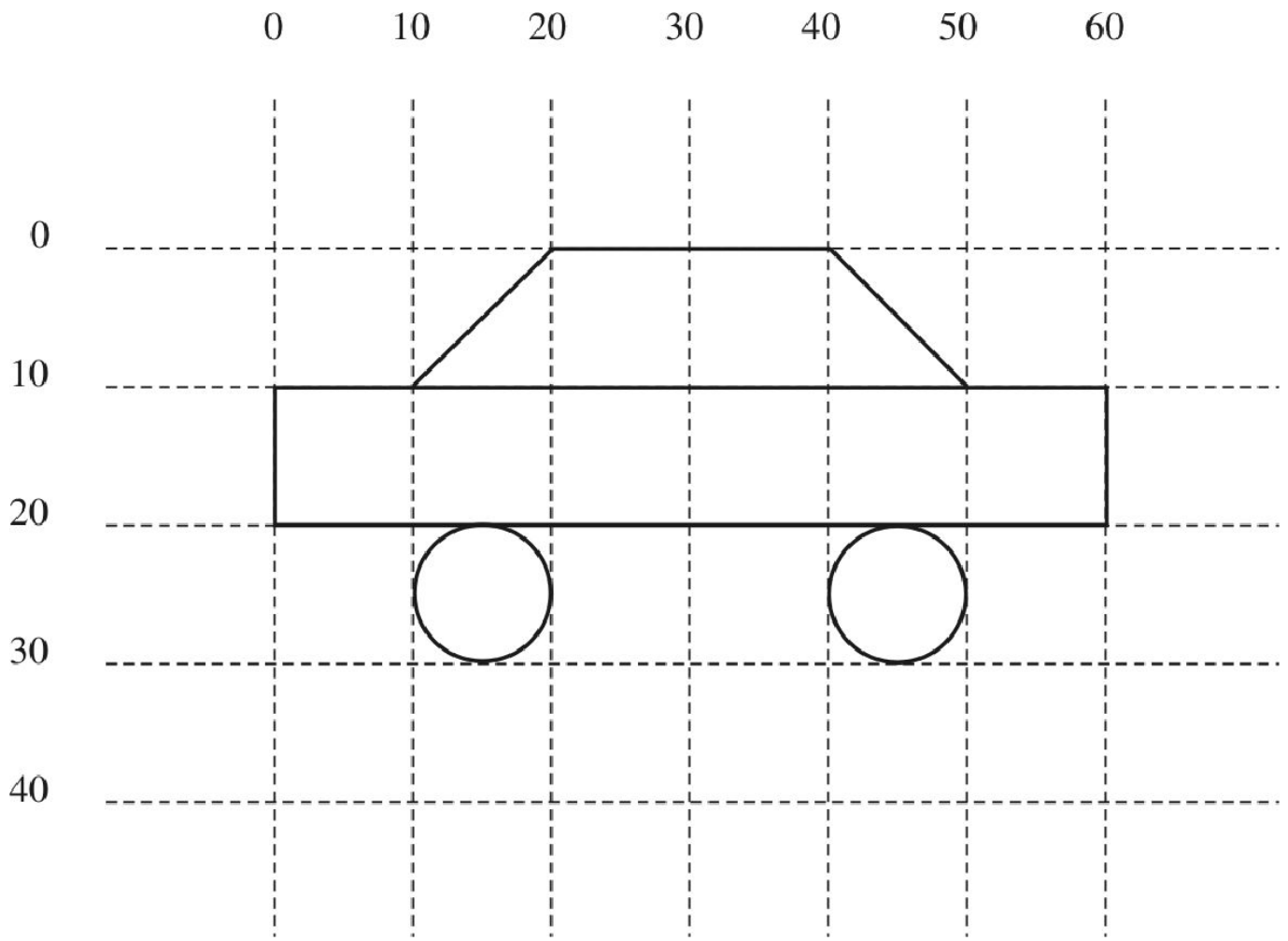


Figura 12: Utilizzo di carta millimetrata per individuare le coordinate delle figure

Classi di programmi per disegnare automobili

- La classe `Car` ha il compito di disegnare un'automobile. Vengono creati due oggetti di tale classe, uno per ogni automobile.
- La classe `CarComponent` visualizza il disegno completo.
- La classe `CarViewer` visualizza un frame che contiene `CarComponent`.

File Car.java

```
01: import java.awt.Graphics2D;
02: import java.awt.Rectangle;
03: import java.awt.geom.Ellipse2D;
04: import java.awt.geom.Line2D;
05: import java.awt.geom.Point2D;
06:
07: /**
08:    Un'automobile posizionabile ovunque sullo schermo.
09: */
10: public class Car
11: {
12:    /**
13:       Costruisce un'automobile.
14:       @param x la coordinata x dell'angolo in alto a sinistra
15:       @param y la coordinata y dell'angolo in alto a sinistra
16:    */
17:    public Car(int x, int y)
18:    {
19:       xLeft = x;
20:       yTop = y;
21:    }
22:
```

Continua

File Car.java

```
23:    /**
24:       Disegna l'automobile.
25:       @param g2 il contesto grafico
26:    */
27:    public void draw(Graphics2D g2)
28:    {
29:       Rectangle body
30:             = new Rectangle(xLeft, yTop + 10, 60, 10);
31:       Ellipse2D.Double frontTire
32:             = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
33:       Ellipse2D.Double rearTire
34:             = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
35:
36:       // La base del parabrezza
37:       Point2D.Double r1
38:             = new Point2D.Double(xLeft + 10, yTop + 10);
39:       // L'inizio del tettuccio
40:       Point2D.Double r2
41:             = new Point2D.Double(xLeft + 20, yTop);
42:       // La fine del tettuccio
43:       Point2D.Double r3
44:             = new Point2D.Double(xLeft + 40, yTop);
45:       // La base del lunotto posteriore
```

File Car.java

```
46:         Point2D.Double r4
47:               = new Point2D.Double(xLeft + 50, yTop + 10);
48:
49:         Line2D.Double frontWindshield
50:               = new Line2D.Double(r1, r2);
51:         Line2D.Double roofTop
52:               = new Line2D.Double(r2, r3);
53:         Line2D.Double rearWindshield
54:               = new Line2D.Double(r3, r4);
55:
56:         g2.draw(body);
57:         g2.draw(frontTire);
58:         g2.draw(rearTire);
59:         g2.draw(frontWindshield);
60:         g2.draw(roofTop);
61:         g2.draw(rearWindshield);
62:      }
63:
64:      private int xLeft;
65:      private int yTop;
66: }
```

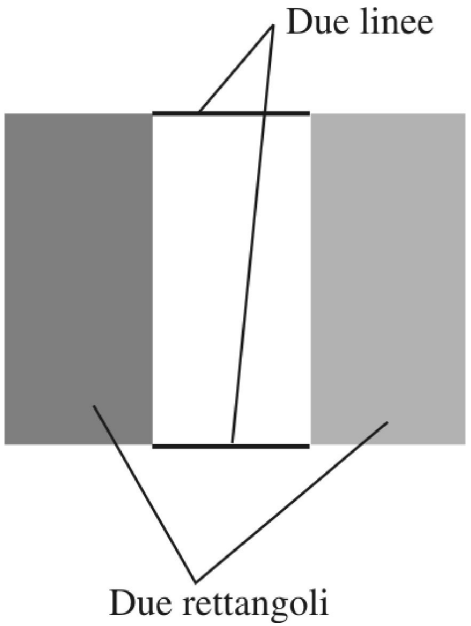
File CarComponent.java

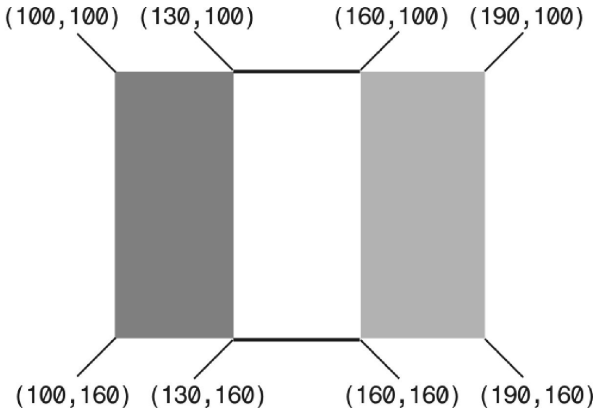
```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import javax.swing.JComponent;
04:
05: /**
06:    Questo componente disegna due automobili.
07: */
08: public class CarComponent extends JComponent
09: {
10:    public void paintComponent(Graphics g)
11:    {
12:       Graphics2D g2 = (Graphics2D) g;
13:
14:       Car car1 = new Car(0, 0);
15:
16:       int x = getWidth() - 60;
17:       int y = getHeight() - 30;
18:
19:       Car car2 = new Car(x, y);
20:
21:       car1.draw(g2);
22:       car2.draw(g2);
23:    }
24: }
```

File CarViewer.java

```
01: import javax.swing.JFrame;
02:
03: public class CarViewer
04: {
05:    public static void main(String[] args)
06:    {
07:       JFrame frame = new JFrame();
08:
09:       frame.setSize(300, 400);
10:       frame.setTitle("Two cars");
11:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:
13:       CarComponent component = new CarComponent();
14:       frame.add(component);
15:
16:       frame.setVisible(true);
17:    }
18: }
19:
```

Disegnare forme grafiche






```
Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
Rectangle rightRectangle = new Rectangle(160, 100, 30, 60);
Line2D.Double topLine = new Line2D.Double(130, 100, 160,100);
Line2D.Double bottomLine
      = new Line2D.Double(130, 160, 160, 160);
```