Capitolo 9
Interfacce e polimorfismo

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

- Conoscere le interfacce
- Saper effettuare conversioni tra riferimenti a classi e a interfacce
- Capire il concetto di polimorfismo
- Utilizzare le interfacce per ridurre l'accoppiamento tra classi
- Imparare a realizzare classi ausiliarie e classi interne
- Capire come le classi interne accedono alle variabili dell'ambito circostante
- Realizzare ricevitori di eventi in applicazioni grafiche

Utilizzo di interfacce per il riutilizzo del codice

- I tipi interfaccia vengono utilizzati per rendere il codice maggiormente riutilizzabile.
- Si è utilizzata la classe `DataSet` (del Capitolo 6) per calcolare il valore medio e il valore massimo di un insieme di valori numerici
- Se volessimo esaminare conti bancari per trovare il conto con il saldo più elevato, dovremmo modificare la classe

Segue...

File DataSet.java

```
01: /**
02:    Calcola informazioni relative a un insieme di dati.
03: */
04: public class DataSet
05: {
06:    /**
07:       Costruisce un insieme di dati vuoto.
08:    */
09:    public DataSet()
10:    {
11:       sum = 0;
12:       count = 0;
13:       maximum = 0;
14:    }
15:
16:    /**
17:       Aggiunge un valore all'insieme dei dati
18:       @param x un valore
19:    */
```

Segue

File DataSet.java

```
20:     public void add(double x)
21:     {
22:        sum = sum + x;
23:        if (count == 0 || maximum < x) maximum = x;
24:        count++;
25:     }
26:
27:     /**
28:        Restituisce la media dei valori inseriti.
29:        @return la media, o 0 se non ci sono dati
30:     */
31:     public double getAverage()
32:     {
33:        if (count == 0) return 0;
34:        else return sum / count;
35:     }
36:
```

Segue

File DataSet.java

```
37:    /**
38:       Restituisce il valore massimo tra i valori inseriti.
39:       @return il massimo, o 0 se non ci sono dati
40:    */
41:    public double getMaximum()
42:    {
43:       return maximum;
44:    }
45:
46:    private double sum;
47:    private double maximum;
48:    private int count;
49: }
```

Utilizzo di interfacce per il riutilizzo del codice

```
public class DataSet // per oggetti di tipo BankAccount
{
   . . .
   public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() < x.getBalance())
         maximum = x;
      count++;
   }

   public BankAccount getMaximum()
   {
      return maximum;
   }
   private double sum;
   private BankAccount maximum;
   private int count;
}
```

Utilizzo di interfacce per il riutilizzo del codice

- Supponete, ora, che volessimo trovare la moneta con il valore più elevato in un insieme di monete: dovremmo modificare nuovamente la classe DataSet.

Segue...

Utilizzo di interfacce per il riutilizzo del codice

```
public class DataSet // modificata per oggetti di tipo Coin
{
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() < x.getValue())
         maximum = x;
      count++;
   }

   public Coin getMaximum()
   {
      return maximum;
   }
   private double sum;
   private Coin maximum;
   private int count;
}
```

Utilizzo di interfacce per il riutilizzo del codice

- Il meccanismo fondamentale per l'analisi dei dati è, evidentemente, lo stesso in tutti i casi, ma i dettagli del confronto cambiano.
- Supponete che le diverse classi potessero accordarsi su un unico metodo **`getMeasure`** che fornisca la misura da usare nell'analisi dei dati
- Potremmo realizzare **un'unica classe `DataSet`**, riutilizzabile, il cui metodo `add` assomiglierebbe a questo:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
   maximum = x;
count++;
```

Utilizzo di interfacce per il riutilizzo del codice

- Qual è il tipo della variabile `x`? Idealmente, `x` dovrebbe riferirsi a qualsiasi classe che abbia un metodo `getMeasure`
- Per esprimere il concetto di una funzionalità necessaria per una classe, in Java si usa un'*interfaccia*.

```
public interface Measurable
{
   double getMeasure();
}
```

- In Java, un'interfaccia dichiara un insieme di metodi e le loro firme.

Interfacce e classi

Un’interfaccia è simile a una classe, ma ci sono parecchie differenze importanti:

- Tutti i metodi di un’interfaccia sono *astratti*, cioè hanno un nome, un elenco di parametri,un tipo di valore restituito, ma **non hanno un’implementazione**
- Tutti i metodi di un’interfaccia sono automaticamente pubblici.
  (NB: non bisogna usare il modificatore **public**).
- Un’interfaccia **non ha variabili di esemplare**.
- Non è possibile creare oggetti del tipo di un'interfaccia. Un’interfaccia **non ha costruttori**.

Un'interfaccia può avere delle costanti, utilizzabili dalle classi che la implementano.

- non bisogna scrivere **public static final int COSTANTE = 1**, // NO
- si scrive direttamente: **int COSTANTE = 1**. // OK

Usare il tipo `Measurable` per dichiarare le variabili `x` e `maximum`

```
public class DataSet
{
   . . .
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum()
   {
      return maximum;
   }

   private double sum;
   private Measurable maximum;
   private int count;
}
```

Implementazione di interfaccia

- Per indicare che una classe realizza un'interfaccia si usa la parola chiave `implements`

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   // altri metodi e campi
}
```

- Una classe può realizzare più di una interfaccia
  - NOTA BENE: la classe deve definire **tutti** i metodi richiesti da **tutte** le interfacce che realizza.

Segue..

Implementazione di interfaccia

- Un altro esempio:

```
public class Coin implements Measurable
{
   public double getMeasure()
   {
      return value;
   }
   . . .
}
```

Schema UML della classe `DataSet` e delle classi che realizzano l'interfaccia `Measurable`

- Le interfacce possono ridurre l'accoppiamento tra le classi.
- Nella notazioneUML:

1. le interfacce vengono contrassegnate dall'indicazione di stereotipo «`interface`»
2. Una freccia tratteggiata con punta triangolare segnala la relazione di tipo "è un" che esiste tra un'interfaccia e una classe che la realizza.
3. Occorre fare molta attenzione alla punta delle frecce: una linea tratteggiata con la freccia a V aperta indica, invece, una dipendenza (relazione "usa")

- La classe `DataSet` dipende solamente dall'interfaccia `Measurable` e non è accoppiata alle classi `BankAccount` e `Coin`.

Segue..

Schema UML della classe DataSet e delle classi che realizzano l'interfaccia Measurable

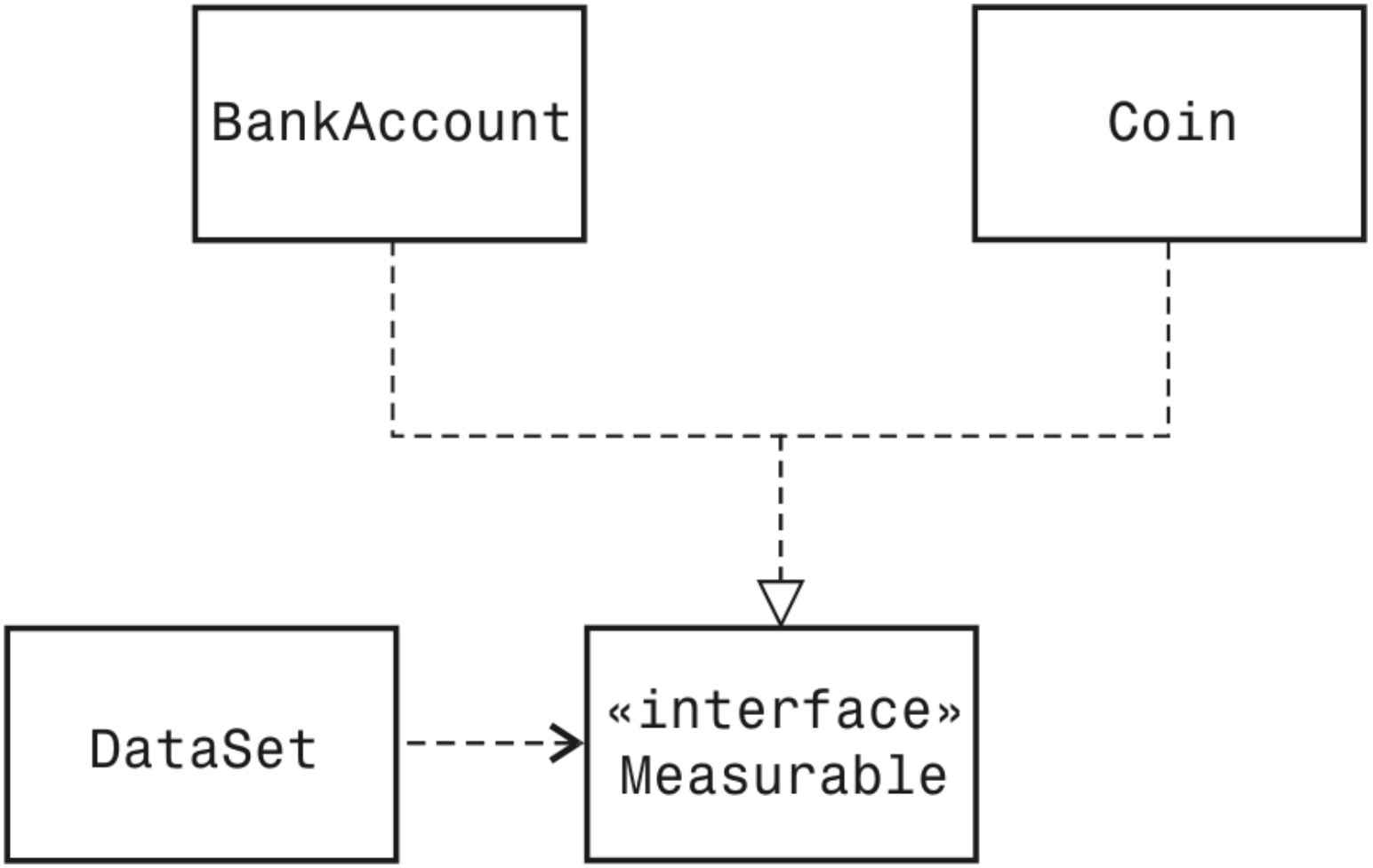

Figura 1

Sintassi di Java 9.1
Definizione di interfaccia

```
public interface NomeInterfaccia
{
   // firme dei metodi
}
```

Esempio:

```
public interface Measurable
{
   double getMeasure();
}
```

Obiettivo:

Definire un'interfaccia e le firme dei suoi metodi, che sono automaticamente pubblici.

Sintassi di Java 9.2
Implementazione di interfaccia

```
public class Nomeclasse
    implements NomeInterfaccia,NomeInterfaccia, ...
{
        // metodi
        // variabili di esemplare
}
```

Esempio:

```
public class BankAccount implements Measurable
{
   // altri metodi di BankAccount
   public double getMeasure()
   {
      // realizzazione del metodo
   }
}
```

Obiettivo:

Definire una classe che realizzi i metodi di un' interfaccia.

File DataSetTester.java

```
01: /**
02:    Questo programma collauda la classe DataSet.
03: */
04: public class DataSetTester
05: {
06:    public static void main(String[] args)
07:    {
08:       DataSet bankData = new DataSet();
09:
10:       bankData.add(new BankAccount(0));
11:       bankData.add(new BankAccount(10000));
12:       bankData.add(new BankAccount(2000));
13:
14:       System.out.println("Average balance: "
15:             + bankData.getAverage());
16:       System.out.println("Expected: 4000");
17:       Measurable max = bankData.getMaximum();
18:       System.out.println("Highest balance: "
19:          + max.getMeasure());
20:       System.out.println("Expected: 10000");
21:
```

Segue...

File DataSetTester.java

```
22:         DataSet coinData = new DataSet();
23:
24:         coinData.add(new Coin(0.25, "quarter"));
25:         coinData.add(new Coin(0.1, "dime"));
26:         coinData.add(new Coin(0.05, "nickel"));
27:
28:         System.out.println("Average coin value: "
29:               + coinData.getAverage());
30:         System.out.println("Expected: 0.133");
31:         max = coinData.getMaximum();
32:         System.out.println("Highest coin value: "
33:               + max.getMeasure());
34:         System.out.println("Expected: 0.25");
35:     }
36: }
```

Segue...

File DataSetTester.java

Visualizza:

```
Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
```

Conversione di tipo fra classi e interfacce

- Potete effettuare conversioni dal tipo di una classe al tipo di un'interfaccia che sia realizzata dalla classe.

```
BankAccount account = new BankAccount(10000);
Measurable x = account; // va bene
```

```
Coin dime = new Coin(0.1, "dime");
Measurable x = dime; // anche questo va bene
```

- Non è però possibile fare conversioni fra tipi non correlati

```
Measurable x = new Rectangle(5, 10, 20, 30); // ERRORE
```

Perché la classe `Rectangle` non realizza l'interfaccia `Measurable`.

Forzature (cast)

- Il metodo `getMaximum` della classe `DataSet` memorizza l'oggetto con la dimensione maggiore come riferimento di tipo `Measurable`

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, "nickel"));
Measurable max = coinData.getMaximum(); // la moneta di maggior valore
```

- Ora, cosa potete fare con il riferimento `max`?

```
String name = max.getName(); // ERRORE
```

Segue…

Forzature (cast)

- Per convertire un riferimento a interfaccia in un riferimento a classe serve un *cast*.
- Voi sapete che si riferisce a un oggetto di tipo `Coin`, ma il compilatore non lo sa. Potete usare la notazione di *forzatura* (*cast*)

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

- Se vi siete sbagliati e l'oggetto in realtà non è una moneta, il vostro programma lancerà un'eccezione e terminerà.

Segue…

Forzature (cast)

- Differenze fra tipi numerici e tipi di classe:
  - Quando convertite tipi numerici, c'è una potenziale *perdita di informazioni*, e usate il cast per dire al compilatore che ne siete al corrente.
  - quando convertite tipi di oggetto, *affrontate il rischio* di provocare il lancio di un'eccezione, e dite al compilatore che siete disposti a correre questo rischio.

Polimorfismo

- Quando più classi realizzano la medesima interfaccia, ciascuna classe realizza i metodi propri dell'interfaccia in modi diversi.
- E' assolutamente lecito, e in effetti molto comune, usare variabili il cui tipo sia un'interfaccia, come `Measurable x;`

```
x = new BankAccount(10000);
x = new Coin(0.1, "dime");
```

- **NOTA BENE**: Occorre, però, sempre ricordare che l'oggetto a cui si riferisce `x` non è di tipo `Measurable`. Il tipo dell'oggetto sarà sempre quello di una classe che realizza l'interfaccia `Measurable`.

Segue...

Polimorfismo

- Cosa potete fare con una variabile di tipo interfaccia se non conoscete la classe dell'oggetto a cui si riferisce?
  - Potete invocare i metodi dell'interfaccia

```
double m = x.getMeasure();
```

Polimorfismo

- Il principio secondo cui il tipo effettivo di un oggetto determina il metodo da chiamare è detto *polimorfismo*.
- Se `x` si riferisce a un oggetto di tipo `BankAccount`, allora viene invocato il metodo `BankAccount.getMeasure`
- Se `x` si riferisce a un oggetto di tipo `Coin`, viene invocato il metodo `Coin.getMeasure`.
- Il polimorfismo è un principio secondo cui il comportamento di un programma può variare in relazione al tipo effettivo di un oggetto.

Segue...

Polimorfismo

- *La selezione posticipata (late binding*) si ha quando la scelta del metodo avviene al momento dell'esecuzione del programma.
- Esiste una differenza importante fra polimorfismo e sovraccarico. Il compilatore sceglie un metodo sovraccarico quando traduce il programma, prima che il programma venga eseguito.Questa selezione del metodo é detta *selezione anticipata (early binding).*

Usare interfacce di smistamento

- Consideriamo queste importanti limitazioni dovute all'utilizzo dell'interfaccia `Measurable`:

1. Potete aggiungere l'interfaccia `Measurable` soltanto a classi che sono sotto il vostro controllo.
2. Potete "misurare" un oggetto in un unico modo. Se volete prendere in esame un insieme di conti bancari di risparmio sia in base al saldo che in base al tasso di interesse, siete bloccati.

- I meccanismi di smistamento e di richiamata consentono a una classe di richiamare uno specifico metodo quando necessita di maggiori informazioni.

Usare interfacce di smistamento

- Ripensiamo, quindi, alla classe `DataSet`: un insieme di dati deve poter misurare gli oggetti che vi vengono inseriti.
- Alternativa: un oggetto può effettuare le misurazioni

```
public interface Measurer
{
   double measure(Object anObject);
}
```

- Tutti i riferimenti ad oggetto possono essere convertiti nel tipo `Object`, il "minimo comun denominatore" delle classi in Java

Usare interfacce di smistamento

- La classe `DataSet` migliorata viene costruita fornendo un oggetto di tipo `Measurer`, cioè un oggetto di una classe che realizzi l'interfaccia `Measurer`, memorizzato nella variabile di esemplare `measurer` e utilizzato per eseguire le misurazioni, in questo modo:

```
public void add(Object x)
{
   sum = sum + measurer.measure(x);
   if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
      maximum = x;
   count++;
}
```

Usare interfacce di smistamento

- Ora siete in grado di definire misuratori per qualsiasi tipo di misurazione
- Osservazioni:
  - Meglio usare RectangleAreaMeasurer come nome della classe
  - Riusare il più possibile i metodi già definiti (e.g. getArea()) .

```
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() * aRectangle.getHeight();
      return area;
   }
}
```

Usare interfacce di smistamento

- Il parametro di tipo `Object` deve essere convertito in un `Rectangle` con un cast:

```
Rectangle aRectangle = (Rectangle) anObject;
```

- Costruite un oggetto di tipo `RectangleMeasurer` e passatelo al costruttore di `DataSet`:

```
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
. . .
```

Diagramma UML delle classi e delle interfacce

- La classe `Rectangle` non è più accoppiata a un'altra classe: per elaborare rettangoli, dovete usare una piccola classe "ausiliaria", `RectangleMeasurer` che esplicita come si misurano gli oggetti

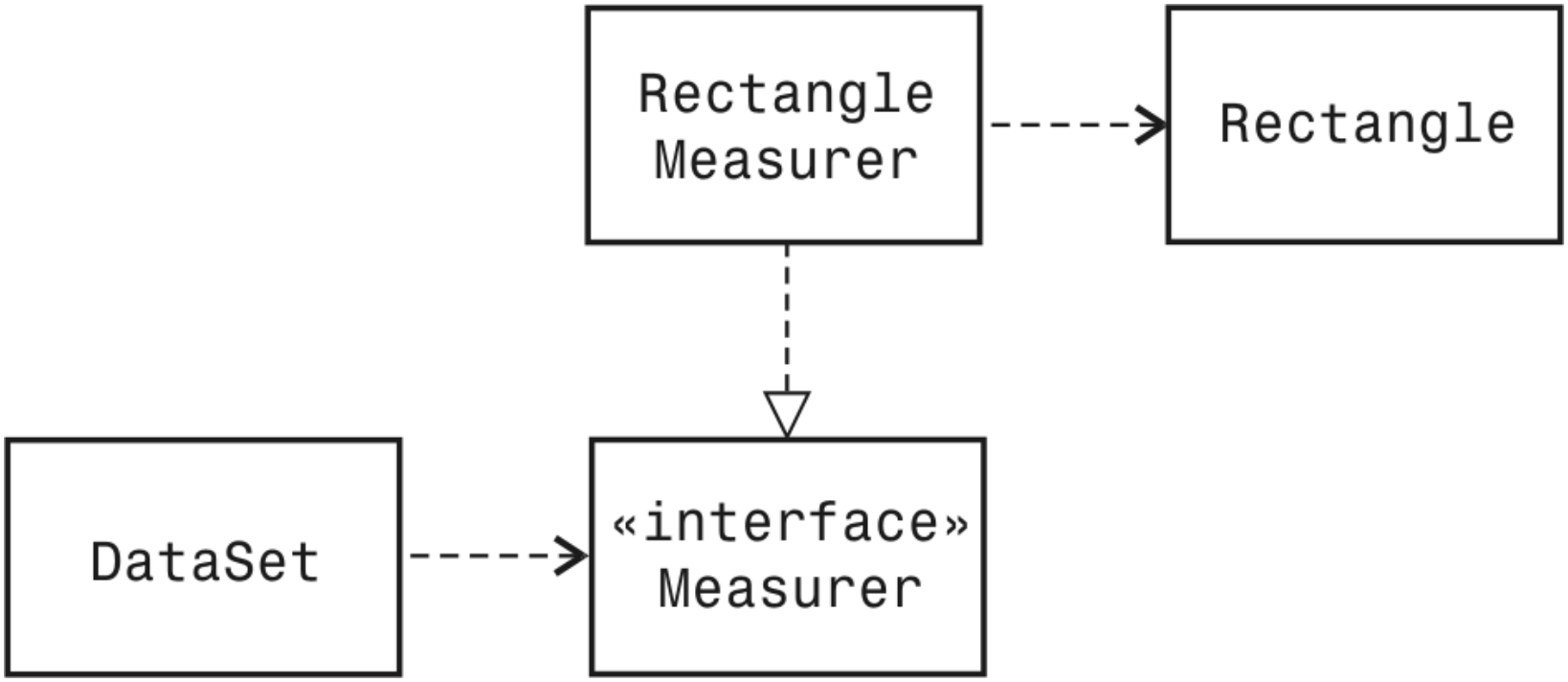

Figura 2:
Schema UML della classe *DataSet* e dell'interfaccia *Measurer*

File DataSet.java

```
01: /**
02:    Calcola la media di un insieme di valori.
03: */
04: public class DataSet
05: {
06:    /**
07:       Costruisce un insieme vuoto di dati con un misuratore assegnato.
08:       @param aMeasures il misuratore che viene usato
                           per misurare i valori dei dati
09:    */
10:    public DataSet(Measurer aMeasurer)
11:    {
12:       sum = 0;
13:       count = 0;
14:       maximum = null;
15:       measurer = aMeasurer;
16:    }
17:
```

Segue...

File DataSet.java

```
18:    /**
19:       Aggiunge il valore all'insieme dei dati.
20:       @param x il valore da aggiungere
21:    */
22:    public void add(Object x)
23:    {
24:       sum = sum + measurer.measure(x);
25:       if (count == 0
26:             || measurer.measure(maximum) < measurer.measure(x))
27:          maximum = x;
28:       count++;
29:    }
30:
31:    /**
32:       Restituisce la media dei dati inseriti.
33:       @return la media, o 0 se non sono stati inseriti valori
34:    */
```

Segue...

File DataSet.java

```
35:    public double getAverage()
36:    {
37:       if (count == 0) return 0;
38:       else return sum / count;
39:    }
40:
41:    /**
42:       Restituisce il dato maggiore tra i dati inseriti.
43:       @return il dato maggiore, null se non ci sono valori
44:    */
45:    public Object getMaximum()
46:    {
47:       return maximum;
48:    }
49:
50:    private double sum;
51:    private Object maximum;
52:    private int count;
53:    private Measurer measurer;
54: }
```

File DataSetTester2.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    Questo programma illustra l'utilizzo di un oggetto di tipo Measurer.
05: */
06: public class DataSetTester2
07: {
08:    public static void main(String[] args)
09:    {
10:       Measurer m = new RectangleMeasurer();
11:
12:       DataSet data = new DataSet(m);
13:
14:       data.add(new Rectangle(5, 10, 20, 30));
15:       data.add(new Rectangle(10, 20, 30, 40));
16:       data.add(new Rectangle(20, 30, 5, 10));
17:
```

Segue...

File DataSetTester2.java

```
18:         System.out.println("Average area: " + data.getAverage());
19:         System.out.println("Expected: 625");
20:
21:         Rectangle max = (Rectangle) data.getMaximum();
22:         System.out.println("Maximum area rectangle: " + max);
23:         System.out.println("Expected: java.awt.Rectangle"
              + "[x=10,y=20,width=30,height=40]");
24:     }
25: }
```

File Measurer.java

```
01: /**
02:    Descrive una qualsiasi classe i cui esemplari possano
      misurare altri oggetti.
03: */
04: public interface Measurer
05: {
06:    /**
07:       Calcola la misura di un oggetto.
08:       @param anObject lìoggetto da misurare
09:       @return la misura
10:    */
11:    double measure(Object anObject);
12: }
```

File RectangleMeasurer.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    Gli oggetti di questa classe misurano rettangoli in base
                                          alla loro area.
05: */
06: public class RectangleMeasurer implements Measurer
07: {
08:    public double measure(Object anObject)
09:    {
10:       Rectangle aRectangle = (Rectangle) anObject;
11:       double area = aRectangle.getWidth()
                * aRectangle.getHeight();
12:       return area;
13:    }
14: }
15:
```

Segue...

File RectangleMeasurer.java

Visualizza:

```
Average area: 625
Expected: 625
Maximum area rectangle:java.awt.Rectangle[x=10,y=20,
   width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]
```

Classi interne

- Quando avete una classe che serve a uno scopo molto limitato, potete dichiararla all'interno del metodo che ne ha bisogno

```
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m); . . .
   }
}
```

Segue…

Classi interne

- Si può anche definire un classe interna contenuta in un'altra, ma al di fuori dei metodi di quest'ultima: in questo modo la classe interna sarà visibile a tutti i metodi della classe che la contiene.
- Quando compilate i file sorgenti di questo programma, vedrete che le classi interne vengono memorizzate in file con nomi curiosi:

```
DataSetTester$1$RectangleMeasurer.class
```

- Definite classi come classi interne solo se sono molto semplici e non riutilizzabili.

Sintassi di Java 9.3 Classi interne

Dichiarata all'interno di un metodo

```
class NomeClasseEsterna
{
    firma del metodo
    {
        . . .
        class NomeClasseInterna
        {
            // metodi
            // Variabili
        }
        . . .
    }
    . . .
}
```

Dichiarata all'interno di una classe

```
class NomeClasseEsterna
{
            metodi
            variabili
    specificatoreDiaccesso
class
            NomeClasseInterna
        {
            // metodi
            // variabili
        }
    . . .
}
```

Segue…

Sintassi di Java 9.3 Classe interna

Esempio:

```
public class Tester
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      . . .
   }
}
```

Obiettivo:

Definire una classe interna il cui ambito di visibilità sia limitato a un solo metodo o ai metodi di una sola classe.

File DataSetTester3.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    Questo programma illustra l'utilizzo di una classe interna.
05: */
06: public class DataSetTester3
07: {
08:    public static void main(String[] args)
09:    {
10:       class RectangleMeasurer implements Measurer
11:       {
12:          public double measure(Object anObject)
13:          {
14:             Rectangle aRectangle = (Rectangle) anObject;
15:             double area
16:                   = aRectangle.getWidth()
                            * aRectangle.getHeight();
17:             return area;
```

Segue...

File FileTester3.java

```
18:             }
19:          }
20:
21:          Measurer m = new RectangleMeasurer();
22:
23:          DataSet data = new DataSet(m);
24:
25:          data.add(new Rectangle(5, 10, 20, 30));
26:          data.add(new Rectangle(10, 20, 30, 40));
27:          data.add(new Rectangle(20, 30, 5, 10));
28:
29:          System.out.println("Average area: " + data.getAverage());
30:          System.out.println("Expected: 625");
31:
32:          Rectangle max = (Rectangle) data.getMaximum();
33:          System.out.println("Maximum area rectangle: " + max);
34:          System.out.println("Expected: java.awt.Rectangle"
               + "[x=10,y=20,width=30,height=40]");
35:       }
36: }
```

Eventi: ricevitori e sorgenti

- Gli eventi dell'interfaccia utente comprendono pressioni di tasti, movimenti del mouse, pressioni di pulsanti, selezioni di voci e così via.

- La maggior parte dei programmi non vuole essere sommersa da eventi inutili.

- Un programma può indicare che gli interessano soltanto determinati eventi.

- *Ricevitore di eventi:*
  - in risposta ad un evento
  - è un esemplare di una classe definita dal programmatore dell'applicazione
  - i suoi metodi descrivono le istruzioni che vanno eseguite quando accade un particolare evento
  - ogni programma indica quali eventi gradisce ricevere mediante l'installazione di oggetti ricevitori di eventi

- *Sorgente di eventi:*
  - le sorgenti di eventi generano gli eventi stessi
  - quando accade un evento, invocano tutti i ricevitori di tale evento

Eventi: ricevitori e sorgenti

- Usate componenti di tipo `JButton` per realizzare pulsanti grafici e connettete un `ActionListener` a ogni pulsante.
- L'interfaccia `ActionListener`

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

- Progettare una classe il cui metodo `actionPerformed` contenga le istruzioni che volete eseguire ogni volta che viene premuto il pulsante
- Il parametro `event` contiene ulteriori dettagli relativi all'evento, come l'istante in cui esso è avvenuto
- Costruire un esemplare e associarlo al pulsante grafico

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

File ClickListener.java

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03:
04: /**
05:    Un ricevitore di azioni che visualizza un messaggio.
06: */
07: public class ClickListener implements ActionListener
08: {
09:    public void actionPerformed(ActionEvent event)
10:    {
11:       System.out.println("I was clicked.");
12:    }
13: }
```

File ButtonViewer.java

```
01: import java.awt.event.ActionListener;
02: import javax.swing.JButton;
03: import javax.swing.JFrame;
04:
05: /**
06:    Questo programma mostra come si installa un ricevitore di azioni.
07: */
08: public class ButtonViewer
09: {
10:    public static void main(String[] args)
11:    {
12:       JFrame frame = new JFrame();
13:       JButton button = new JButton("Click me!");
14:       frame.add(button);
15:
16:       ActionListener listener = new ClickListener();
17:       button.addActionListener(listener);
18:
19:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
20:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21:       frame.setVisible(true);
22:    }
```

Segue...

File ButtonViewer.java

```
23:
24:    private static final int FRAME_WIDTH = 100;
25:    private static final int FRAME_HEIGHT = 60;
26: }
```

Visualizza:

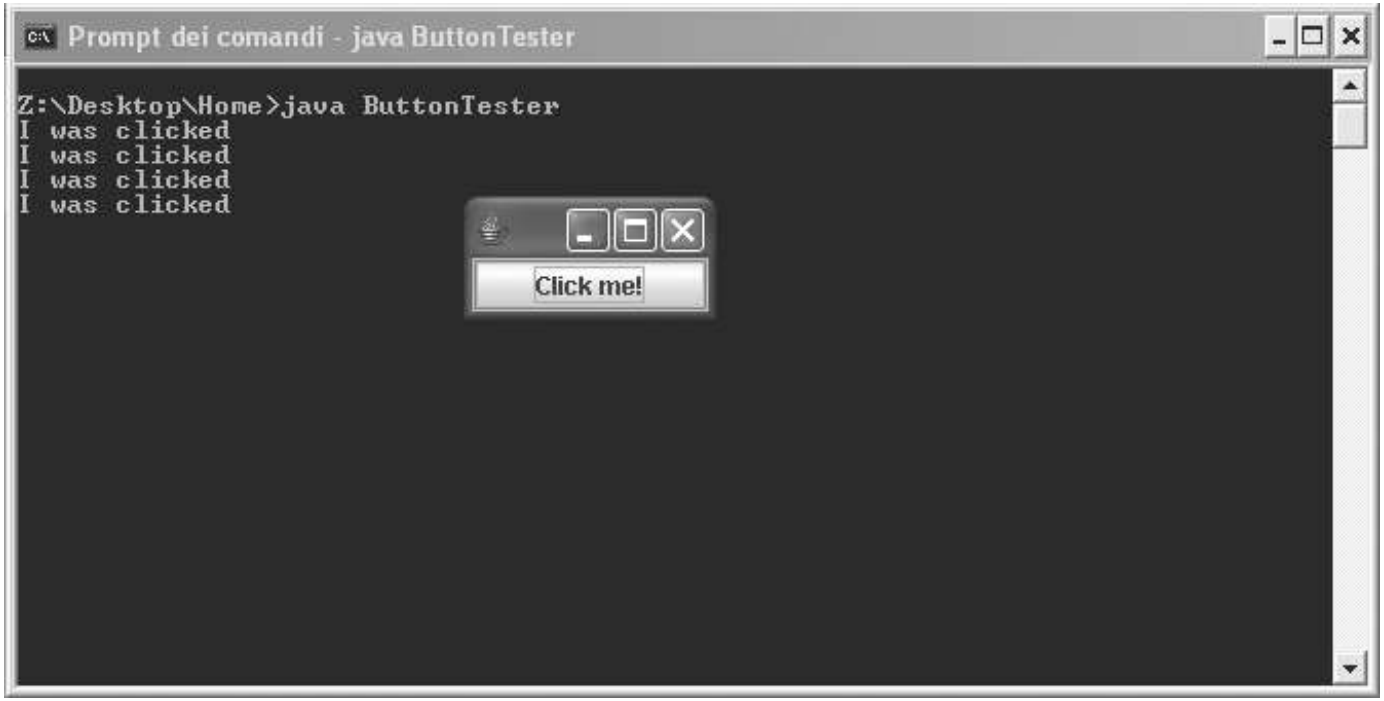

Figura 3: Realizzazione di un ricevitore di azioni

Classi interne come ricevitori di eventi

- Spesso una classe che funge da ricevitore di eventi viene realizzata come classe interna:

```
JButton button = new JButton(". . .");
//Questa classe interna viene definita all'interno dello
stesso //metodo in cui si trova la variabile che contiene il
pulsante
class MyListener implements ActionListener
{
   . . .
};
ActionListener listener = new MyListener();
button.addActionListener(listener);
```

- La classe che riceve gli eventi si viene a trovare nel posto esatto in cui serve al suo scopo senza creare confusione nella restante parte del progetto.

- I metodi delle classi interne possono accedere alle variabili definite nei blocchi circostanti

Classi interne come ricevitori di eventi

- Le variabili locali a cui si accede da un metodo di una classe interna devono essere dichiarate `final`

- Per esempio, immaginiamo di accreditare gli interessi a un conto bancario ogni volta che viene premuto un pulsante:

```
JButton button = new JButton("Add Interest");
final BankAccount account =
      new BankAccount(INITIAL_BALANCE);
//Questa classe interna viene definita all'interno
dello //stesso metodo in cui si trovano le variabili
account e
//button.
class AddInterestListener implements ActionListener
{
```

Segue…

Classi interne come ricevitori di eventi

```
    public void actionPerformed(ActionEvent event)
    {
       // il metodo ricevitore di eventi accede alla
       // variabile definita nel blocco circostante
       double interest = account.getBalance() *
          INTEREST_RATE / 100;
       account.deposit(interest);
    }
};
ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

File InvestmentViewer1.java

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03: import javax.swing.JButton;
04: import javax.swing.JFrame;
05:
06: /**
07:    Questo programma illustra il funzionamento di una classe interna
08:    che accede a una variabile definita nell'ambito circostante.
09: */
10: public class InvestmentViewer1
11: {
12:    public static void main(String[] args)
13:    {
14:       JFrame frame = new JFrame();
15:
16:       // Il pulsante che innesca il calcolo
17:       JButton button = new JButton("Add Interest");
18:       frame.add(button);
19:
```

Segue…

File InvestmentViewer1.java

```
20:    // l'applicazione accredita gli interessi a questo conto bancario
21:    final BankAccount account = new BankAccount(INITIAL_BALANCE);
22:
23:    class AddInterestListener implements ActionListener
24:    {
25:         public void actionPerformed(ActionEvent event)
26:         {
27:            // Il metodo del ricevitore accede alla variabile account
28:            // definita nell'ambito di visibilità circostante
29:             double interest = account.getBalance()
30:                   * INTEREST_RATE / 100;
31:             account.deposit(interest);
32:             System.out.println("balance: " + account.getBalance());
33:          }
34:       }
35:
36:       ActionListener listener = new AddInterestListener();
37:       button.addActionListener(listener);
38:
```

Segue...

File InvestmentViewer1.java

```
39:         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
40:         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
41:         frame.setVisible(true);
42:      }
43:
44:      private static final double INTEREST_RATE = 10;
45:      private static final double INITIAL_BALANCE = 1000;
46:
47:      private static final int FRAME_WIDTH = 120;
48:      private static final int FRAME_HEIGHT = 60;
49: }
```

Visualizza:

```
balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1
```

Costruire applicazioni dotate di pulsanti

Esempio di programma che visualizza investimenti: ogni volta che il pulsante grafico viene premuto, vengono accreditati gli interessi maturati sul conto bancario e viene visualizzato il saldo aggiornato.

Figura 4: Un'applicazione dotata di pulsante grafico

Costruire applicazioni dotate di pulsanti

- Costruiamo un esemplare della classe `JButton`:

```
JButton button = new JButton("Add Interest");
```

- Abbiamo anche bisogno di un componente dell'interfaccia utente che visualizzi un messaggio:

```
JLabel label = new JLabel("balance: " +
account.getBalance());
```

- Usate un contenitore di tipo `JPanel` per raggruppare insieme più componenti dell'interfaccia utente:

```
JPanel panel = new JPanel(); panel.add(button);
panel.add(label); frame.add(panel);
```

Costruire applicazioni dotate di pulsanti

- La classe `AddInterestListener` aggiunge gli interessi al conto e ne visualizza il saldo aggiornato:

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double interest = account.getBalance() *
         INTEREST_RATE / 100;
      account.deposit(interest);
      label.setText("balance=" +
            account.getBalance());
   }
}
```

- Definire la classe `AddInterestListener` come classe interna del metodo `main` in modo che il metodo `actionPerformed` possa accedere alle variabili `final` (`account` e `label`)

File InvestmentViewer2.java

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03: import javax.swing.JButton;
04: import javax.swing.JFrame;
05: import javax.swing.JLabel;
06: import javax.swing.JPanel;
07: import javax.swing.JTextField;
08:
09: /**
10:    Questo programma visualizza la crescita di un investimento.
11: */
12: public class InvestmentViewer2
13: {
14:    public static void main(String[] args)
15:    {
16:       JFrame frame = new JFrame();
17:
18:       // Il pulsante che innesca l'elaborazione
19:       JButton button = new JButton("Add Interest");
```

Segue…

File InvestmentViewer2.java

```
20:
21:     //l'applicazione aggiunge gli interessi a questo conto bancario
22:     final BankAccount account = new BankAccount(INITIAL_BALANCE);
23:
24:     // L'etichetta che visualizza i risultati
25:     final JLabel label = new JLabel(
26:           "balance: " + account.getBalance());
27:
28:     //Il pannello che contiene i componenti dell'interfaccia utente
29:     JPanel panel = new JPanel();
30:     panel.add(button);
31:     panel.add(label);
32:     frame.add(panel);
33:
34:     class AddInterestListener implements ActionListener
35:     {
36:        public void actionPerformed(ActionEvent event)
37:        {
38:           double interest = account.getBalance()
39:                 * INTEREST_RATE / 100;
```

Segue…

File InvestmentViewer2.java

```
40:             account.deposit(interest);
41:             label.setText(
42:                   "balance: " + account.getBalance());
43:          }
44:       }
45:
46:       ActionListener listener = new AddInterestListener();
47:       button.addActionListener(listener);
48:
49:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
50:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51:       frame.setVisible(true);
52:    }
53:
54:    private static final double INTEREST_RATE = 10;
55:    private static final double INITIAL_BALANCE = 1000;
56:
57:    private static final int FRAME_WIDTH = 400;
58:    private static final int FRAME_HEIGHT = 100;
59: }
```

Elaborare eventi di temporizzazione

- La classe `Timer` nel pacchetto `javax.swing` genera una sequenza di *eventi*, separati da intervalli di tempo tutti uguali fra loro
- Ciò è utile ogni volta che volete disporre di oggetti aggiornati a intervalli regolari.
- Un ricevitore di eventi riceve una notifica quando accade un particolare evento.

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

Segue...

Elaborare eventi di temporizzazione

- Dovete definire una classe che realizzi l'interfaccia `ActionListener`

```
class MioRicevitore implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // questa azione verrà eseguita a ogni
      // evento di temporizzazione
      Inserite qui le azioni del ricevitore di eventi
   }
}
```

Segue...

Elaborare eventi di temporizzazione

- Costruite un oggetto di tale classe e passatelo al costruttore di `Timer`.

```
MioRicevitore listener = new MioRicevitore();
Timer t = new Timer(interval, listener);
t.start();
```

File RectangleComponent.java

```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import java.awt.Rectangle;
04: import javax.swing.JComponent;
05:
06: /**
07:    Componente che visualizza un rettangolo che può essere spostato.
08: */
09: public class RectangleComponent extends JComponent
10: {
11:    public RectangleComponent()
12:    {
13:       // il rettangolo che viene disegnato dal metodo paint.
14:       box = new Rectangle(BOX_X, BOX_Y,
15:             BOX_WIDTH, BOX_HEIGHT);
16:    }
17:
18:    public void paintComponent(Graphics g)
19:    {
20:       super.paintComponent(g);
21:       Graphics2D g2 = (Graphics2D) g;
```

File RectangleComponent.java

```
22:
23:          g2.draw(box);
24:       }
25:
26:       /**
27:          Sposta il rettangolo della quantità specificata.
28:          @param x l'entità dello spostamento nella direzione x
29:          @param y l'entità dello spostamento nella direzione y
30:       */
31:       public void moveBy(int dx, int dy)
32:       {
33:          box.translate(dx, dy);
34:          repaint();
35:       }
36:
37:       private Rectangle box;
38:
39:       private static final int BOX_X = 100;
40:       private static final int BOX_Y = 100;
41:       private static final int BOX_WIDTH = 20;
42:       private static final int BOX_HEIGHT = 30;
43: }
```

File RectangleComponent.java

- Mostra un rettangolo che può essere spostato

- Il metodo `repaint` chiede a un componente di ridisegnare se stesso: invocatelo ogni volta che modificate le forme grafiche disegnate dal metodo `paintComponent`

File RectangleMover.java

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03: import javax.swing.JFrame;
04: import javax.swing.Timer;
05:
06: public class RectangleMover
07: {
08:    public static void main(String[] args)
09:    {
10:       JFrame frame = new JFrame();
11:
12:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
13:       frame.setTitle("An animated rectangle");
14:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:
16:       final RectangleComponent component = new RectangleComponent();
17:       frame.add(component);
18:
19:       frame.setVisible(true);
20:
```

Segue…

File RectangleMover.java

```
21:       class TimerListener implements ActionListener
22:       {
23:          public void actionPerformed(ActionEvent event)
24:          {
25:             component.moveBy(1, 1);
26:          }
27:       }
28:
29:       ActionListener listener = new TimerListener();
30:
31:       final int DELAY = 100; // Millisecondi tra due eventi
32:       Timer t = new Timer(DELAY, listener);
33:       t.start();
34:    }
35:
36:    private static final int FRAME_WIDTH = 300;
37:    private static final int FRAME_HEIGHT = 400;
38: }
```

Eventi del mouse

- Per catturare gli eventi del mouse usate un ricevitore di eventi del mouse (`MouseListener`)
- Un ricevitore di eventi del mouse deve realizzare l'interfaccia `MouseListener`, che definisce i cinque metodi seguenti:

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   //Chiamato quando un pulsante del mouse
   //è stato premuto su un componente
   void mouseReleased(MouseEvent event);
   //Chiamato quando un pulsante del mouse
   //è stato rilasciato su un componente
   void mouseClicked(MouseEvent event);
   //Chiamato quando un pulsante del mouse
   //è stato premuto e rilasciato in rapida
   //successione su un componente
   void mouseEntered(MouseEvent event);
   //Chiamato quando il mouse entra in un componente
   void mouseExited(MouseEvent event);
   //Chiamato quando il mouse esce da un componente  }
```

Eventi del mouse

- `mousePressed` e `mouseReleased` vengono invocati quando un pulsante del mouse viene premuto o rilasciato

- `mouseClicked` viene invocato se il pulsante viene premuto e rilasciato in rapida successione senza che il mouse si sia spostato

- `mouseEntered` e `mouseExited` possono essere utilizzati per visualizzare in modo speciale un componente dell'interfaccia utente quando il puntatore del mouse si trova al suo interno

Eventi del mouse

• Per aggiungere a un componente un ricevitore di eventi del mouse si invoca il metodo `addMouseListener`:

```
public class MyMouseListener implements MouseListener
{
   // realizza i cinque metodi
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

• Nel programma di esempio, ogni volta che l'utente preme e rilascia il pulsante del mouse sul componente rettangolare, vogliamo che il rettangolo si sposti e si posizioni nel punto in cui si trova il mouse

File RectangleComponent.java

```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import java.awt.Rectangle;
04: import javax.swing.JComponent;
05:
06: /**
07:    Componente che visualizza un rettangolo che può essere spostato.
08: */
09: public class RectangleComponent extends JComponent
10: {
11:    public RectangleComponent()
12:    {
13:       // Il rettangolo disegnato del metodo paint
14:       box = new Rectangle(BOX_X, BOX_Y,
15:             BOX_WIDTH, BOX_HEIGHT);
16:    }
17:
18:    public void paintComponent(Graphics g)
19:    {
20:       super.paintComponent(g);
21:       Graphics2D g2 = (Graphics2D) g;
22:
```

Segue...

File RectangleComponent.java

```
23:          g2.draw(box);
24:       }
25:
26:       /**
27:          Sposta il rettangolo nella posizione indicata.
28:          @param x la coordinata x della nuova posizione
29:          @param y la coordinata y della nuova posizione
30:       */
31:       public void moveTo(int x, int y)
32:       {
33:          box.setLocation(x, y);
34:          repaint();
35:       }
36:
37:       private Rectangle box;
38:
39:       private static final int BOX_X = 100;
40:       private static final int BOX_Y = 100;
41:       private static final int BOX_WIDTH = 20;
42:       private static final int BOX_HEIGHT = 30;
43: }
```

Eventi del mouse

- Invocare `repaint` per essere certi che il componente venga ridisegnato dopo aver modificato lo stato del rettangolo.
- Aggiungete al componente un ricevitore di eventi del mouse: ogni volta che viene premuto un pulsante del mouse, il ricevitore sposta il rettangolo nella posizione del mouse

```
class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   {
      int x = event.getX();
      int y = event.getY();
      component.moveTo(x, y);
   }
```

Eventi del mouse

```
    // metodi che non fanno niente
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}
```

- I cinque metodi dell'interfaccia vanno realizzati tutti; i metodi inutilizzati vengono semplicemente realizzati sotto forma di metodi che non fanno nulla

Un clic del mouse sposta il rettangolo

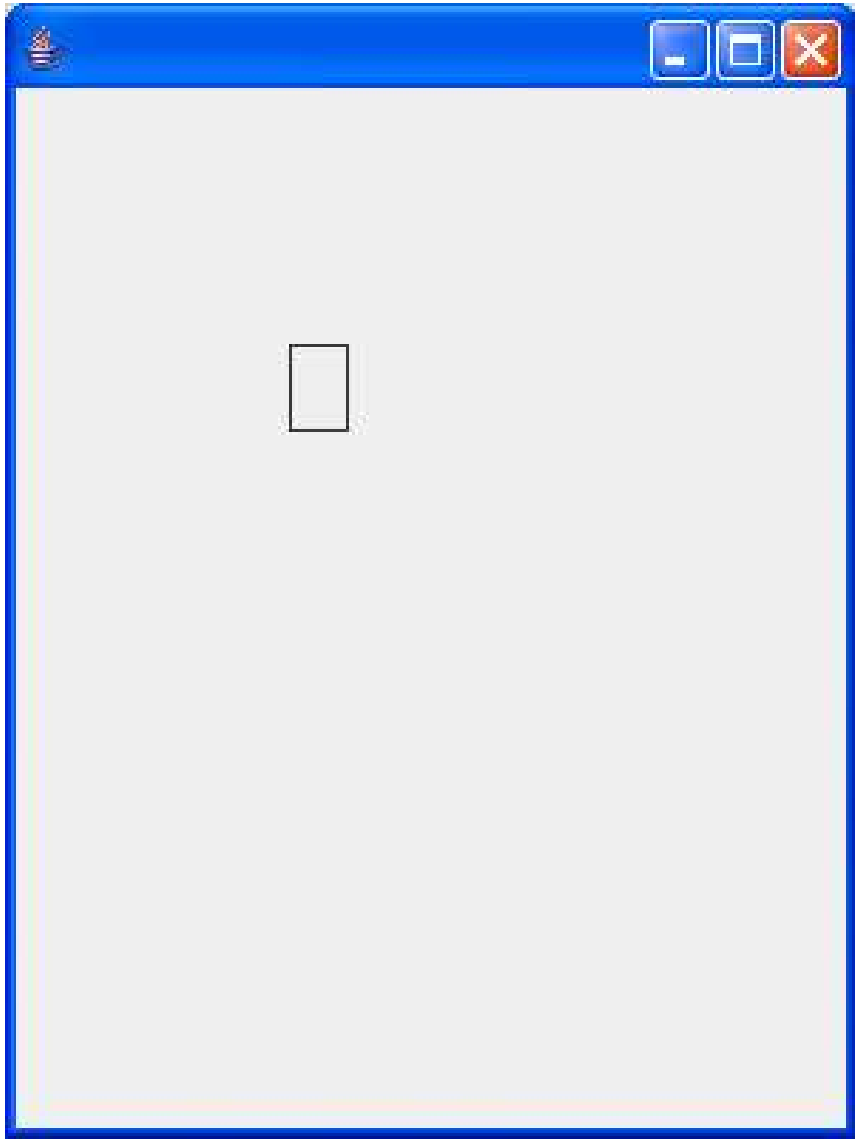

Figura 5

File RectangleComponentViewer.java

```
01: import java.awt.event.MouseListener;
02: import java.awt.event.MouseEvent;
03: import javax.swing.JFrame;
04:
05: /**
06:    Questo programma visualizza un RectangleComponent.
07: */
08: public class RectangleComponentViewer
09: {
10:    public static void main(String[] args)
11:    {
12:       final RectangleComponent component = new RectangleComponent();
13:
14:       // associa il ricevitore di eventi del mouse
15:
16:       class MousePressListener implements MouseListener
17:       {
18:          public void mousePressed(MouseEvent event)
19:          {
20:             int x = event.getX();
21:             int y = event.getY();
22:             component.moveTo(x, y);
23:          }
```

File RectangleComponentViewer.java

```
24:
25:          // metodi che non fanno niente
26:          public void mouseReleased(MouseEvent event) {}
27:          public void mouseClicked(MouseEvent event) {}
28:          public void mouseEntered(MouseEvent event) {}
29:          public void mouseExited(MouseEvent event) {}
30:       }
31:
32:       MouseListener listener = new MousePressListener();
33:       component.addMouseListener(listener);
34:
35:       JFrame frame = new JFrame();
36:       frame.add(component);
37:
38:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
39:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40:       frame.setVisible(true);
41:    }
42:
43:    private static final int FRAME_WIDTH = 300;
44:    private static final int FRAME_HEIGHT = 400;
45: }
```