Usare le interfacce per il riuso del codice

- Si possono usare *tipi interfaccia* per rendere il codice più riusabile
- Vediamo l'uso di una classe `DataSet` per trovare la media ed il massimo di un insieme di valori (*numeri*)
- Si potrà usare la stessa tecnica per trovare il valore medio ed il valore massimo di un insieme di valori `BankAccount`?

Continua

ch06/dataset/DataAnalyzer.java

```
01: import java.util.Scanner;
02:
03: /**
04:    This program computes the average and maximum of a set
05:    of input values.
06: */
07: public class DataAnalyzer
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       DataSet data = new DataSet();
13:
14:       boolean done = false;
15:       while (!done)
16:       {
17:          System.out.print("Enter value, Q to quit: ");
18:          String input = in.next();
19:          if (input.equalsIgnoreCase("Q"))
20:             done = true;
```

Continua

ch06/dataset/DataAnalyzer.java (cont.)

```
21:             else
22:             {
23:                double x = Double.parseDouble(input);
24:                data.add(x);
25:             }
26:          }
27:
28:          System.out.println("Average = " + data.getAverage());
29:          System.out.println("Maximum = " + data.getMaximum());
30:       }
31: }
```

ch06/dataset/DataSet.java

```
01: /**
02:    Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:    /**
07:       Constructs an empty data set.
08:    */
09:    public DataSet()
10:    {
11:       sum = 0;
12:       count = 0;
13:       maximum = 0;
14:    }
15:
16:    /**
17:       Adds a data value to the data set
18:       @param x a data value
19:    */
20:    public void add(double x)
21:    {
```

Continua

ch06/dataset/DataSet.java (continua)

```
22:          sum = sum + x;
23:          if (count == 0 || maximum < x) maximum = x;
24:          count++;
25:       }
26:
27:       /**
28:          Gets the average of the added data.
29:          @return the average or 0 if no data has been added
30:       */
31:       public double getAverage()
32:       {
33:          if (count == 0) return 0;
34:          else return sum / count;
35:       }
36:
37:       /**
38:          Gets the largest of the added data.
39:          @return the maximum or 0 if no data has been added
40:       */
```

Continua

ch06/dataset/DataSet.java (continua)

```
41:    public double getMaximum()
42:    {
43:       return maximum;
44:    }
45:
46:    private double sum;
47:    private double maximum;
48:    private int count;
49: }
```

Output:

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

Usare le interfacce per il riuso del codice (continua)

```
public class DataSet // Modified for BankAccount objects
{
   . . .
   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;
}
```

Usare le interfacce per il riuso del codice

O si supponga di voler trovare la moneta con il valore più alto tra un insieme di monete. Sarebbe necessario modificare nuovamente la classe `DataSet`:

```
public class DataSet // Modified for Coin objects
{
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() <
         x.getValue()) maximum = x;
      count++;
   }
```

Continua

Usare le interfacce per il riuso del codice

```
   public Coin getMaximum()
   {
      return maximum;
   }

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

Usare le interfacce per il riuso del codice

- Il meccanismo per l'analisi dei dati è lo stesso in tutti I casi; I dettagli del confronto cambiano
- Le classi potrebbero accordarsi su un metodo `getMeasure` che fornisca la misura da usare nell'analisi
- Si può realizzare una singola classe riusabile `DataSet` il cui metodo `add` ha il seguente aspetto:

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

Continua

Usare le interfacce per il riuso del codice (continua)

- Qual è il tipo della variabile `x`?
  `x` dovrebbe riferirsi a qualunque classe abbia un metodo `getMeasure`
- In Java, un *tipo interfaccia* viene usato per specificare le operazioni richieste

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

- La dichiarazione di interfaccia elenca tutti i metodi (e le loro intestazioni) che il tipo interfaccia richiede

Interfacce e classi

Un’interfaccia è simile ad una classe, ma ci sono delle importanti differenze:

- *Tutti i metodi di un’interfaccia sono astratti (abstract); essi non hanno una implementazione*
- *Tutti i metodi di un interfaccia sono automaticamente pubblici*
- *Un’interfaccia non ha dati di istanza*

Un generico `DataSet` per oggetti Measurable

```
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;
   }
```

Continua

Un generico `DataSet` per oggetti Measurable

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

Implementare un'interfaccia

- Si usa la parola chiave `implements` per indicare che una classe implementa un'interfaccia

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   // Additional methods and fields
}
```

- Una classe può implementare più di una interfaccia
  - *La classe **deve** definire tutti i metodi che sono richiesti da tutte le interfacce che essa implementa*

Continua

Sintassi 9.1 Definizione di un interfaccia

```
public interface InterfaceName
{
   // method signatures
}
```

Esempio:

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

Scopo:

Definire un'interfaccia e l'intestazione dei suoi metodi. I metodi sono automaticamente pubblici.

Sintassi 9.2 Implementare un'interfaccia

```
public class ClassName
implements InterfaceName, InterfaceName, ...
{
      // methods
      // instance variables
}
```

Esempio:

```
public class BankAccount implements Measurable
{
   // Other BankAccount methods
   public double getMeasure()
   {
      // Method implementation
   }
}
```

Continua

Sintassi 9.2 Implementare un interfaccia (continua)

Scopo:

Definire una nuova classe che implementa i metodi di un interfaccia.

ch09/measure1/DataSetTester.java

```
01: /**
02:    This program tests the DataSet class.
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:
```

ch09/measure1/DataSetTester.java (cont.)

```
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: }
```

ch09/measure1/DataSetTester.java (cont.)

Output:

```
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 tra classi ed interfacce

- Si può effettuare la conversione da una classe ad un'interfaccia, purché la classe implementi l'interfaccia

- ```
  BankAccount account = new BankAccount(10000);
  Measurable x = account; // OK
  ```

- ```
  Coin dime = new Coin(0.1, "dime");
  Measurable x = dime; // Also OK
  ```

- Non si possono effettuare conversioni tra tipi scorrelati

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

- Dipende dal fatto che `Rectangle` non implementa `Measurable`

Conversioni (cast)

- Aggiungere oggetti Coin a `DataSet`

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
. . .
Measurable max = coinData.getMaximum(); // Get the
   largest coin
```

- Cosa si può fare con questo? Non è di tipo `Coin`

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

- E' necessaria una conversione per passare da un tipo interfaccia ad un tipo classe
- E' noto che è un oggetto Coin, ma il compilatore non lo sa. Si effettui allora una conversione:

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

Continua

Conversioni (cast)

- Se si sbaglia e `max` non è un oggetto Coin, il compilatore lancia un'eccezione
- Differenze con la conversione di numeri:

  nella conversione di tipi numerici si accettano perdite di informazioni
  nella conversione di oggetti si accetta il rischio di originare un'eccezione

Polimorfismo

- Una variabile interfaccia può contenere il riferimento ad un oggetto di una classe che implementa tale interfaccia

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

- Si noti che l'oggetto a cui `x` si riferisce non ha il tipo `Measurable`; il tipo dell'oggetto è una classe che implementa l'interfaccia `Measurable`
- Si può chiamare qualunque tra i metodi della classe interfaccia:

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

- Quale metodo è stato chiamato?

Polimorfismo

- Dipende dall'oggetto effettivo
- Se x si riferisce a un oggetto BankAccount, chiama `BankAccount.getMeasure`
- Se x si riferisce ad un oggetto Coin, chiama `Coin.getMeasure`
- Polimorfismo (= molte forme): comportamento che varia in base al tipo effettivo dell'oggetto
- Chiamato anche *late binding* (selezione posticipata): la selezione del metodo viene effettuata in esecuzione
- Differente dall'overloading (sovraccaricamento); l'overloading viene risolto dal compilatore (*early binding - selezione anticipata*)

Usare le interfacce di smistamento (callback)

- Limitazioni dell'interfaccia `Measurable`:
  - *Si può aggiungere l'interfaccia `Measurable` solo a classi sotto il proprio controllo*
  - *Si può "misurare" un oggetto in un unico modo*
    Ad esempio, non si può analizzare un insieme di conti di risparmio sia per saldo bancario sia per tasso di interesse
- Meccanismo del callback: permette ad una classe di richiamare un metodo specifico quando necessita di maggiore informazione
- Nella precedente implementazione di `DataSet`, la responsabilità della misurazione ricade sugli oggetti stessi aggiunti all'insieme di dati

Continua

Usare le interfacce di smistamento

- Alternativa: consegnare l'oggetto da misurare ad un metodo:

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

- `Object` è il "minimo comune denominatore" di tutte le classi

Usare le interfacce di smistamento

`add` è un metodo che chiede al misuratore (non all'oggetto aggiunto) di effettuare la misurazione:

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

Usare le interfacce di smistamento

- Si possono definire i misuratori per effettuare un qualunque tipo di misura

public class `RectangleMeasurer implements Measurer`

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

- Bisogna effettuare le conversione da `Object` a `Rectangle`

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

Continua

Usare le interfacce di smistamento

- Si passa il misuratore al costruttore del data set:

```
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)); . . .
```

ch09/measure2/DataSet.java

```
01: /**
02:    Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:    /**
07:       Constructs an empty data set with a given measurer.
08:       @param aMeasurer the measurer that is used to measure data
values
09:    */
10:    public DataSet(Measurer aMeasurer)
11:    {
12:       sum = 0;
13:       count = 0;
14:       maximum = null;
15:       measurer = aMeasurer;
16:    }
17:
18:    /**
19:       Adds a data value to the data set.
20:       @param x a data value
21:    */
```

Continua

ch09/measure2/DataSet.java (continua)

```
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:       Gets the average of the added data.
33:       @return the average or 0 if no data has been added
34:    */
35:    public double getAverage()
36:    {
37:       if (count == 0) return 0;
38:       else return sum / count;
39:    }
40:
```

Continua

ch09/measure2/DataSet.java (continua)

```
41:    /**
42:       Gets the largest of the added data.
43:       @return the maximum or 0 if no data has been added
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: }
```

ch09/measure2/DataSetTester2.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    This program demonstrates the use of a 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, 15));
17:
18:       System.out.println("Average area: " + data.getAverage());
19:       System.out.println("Expected: 625");
20:
```

Continua

ch09/measure2/DataSetTester2.java (continua)

```
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: }
```

ch09/measure2/Measurer.java

```
01: /**
02:    Describes any class whose objects can measure other objects.
03: */
04: public interface Measurer
05: {
06:    /**
07:       Computes the measure of an object.
08:       @param anObject the object to be measured
09:       @return the measure
10:    */
11:    double measure(Object anObject);
12: }
```

ch09/measure2/RectangleMeasurer.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    Objects of this class measure rectangles by 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: }
```

ch09/measure2/RectangleMeasurer.java (continua)

Output:

```
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

- Classi di uso limitato possono essere definite all'interno di un metodo

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

Continua

Classi interne (continua)

- Se la classe interna è definita dentro una classe che la racchiude, ma fuori dai suoi metodi, essa è disponibile a tutti i metodi della classe che la racchiude
- Il compilatore traduce la classe interna in un normale file di tipo class:
  `DataSetTester$1$RectangleMeasurer.class`

Sintassi 9.3 Classi interne

Dichiarata entro un metodo

```
class OuterClassName
{
   method signature
   {
      . . .
      class InnerClassName
      {
         // methods
         // fields
      }
      . . .
   }
   . . .
}
```

Dichiarata entro una classe

```
class OuterClassName
{
   // methods
   // fields
   accessSpecifier class
      InnerClassName
   {
      // methods
      // fields
   }
   . . .
}
```

Continua

Sintassi 9.3 Classi interne

Esempio:

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

Scopo:

Definire una classe interna il cui ambito si limita ad un singolo metodo o ai metodi di una singola classe.

ch09/measure3/DataSetTester3.java

```
01: import java.awt.Rectangle;
02:
03: /**
04:    This program demonstrates the use of an inner class.
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;
18:          }
19:       }
20:
```

Continua

ch09/measure3/DataSetTester3.java (continua)

```
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, 15));
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, sorgenti di eventi ed ascoltatori di eventi

- Gli *eventi* dell'interfaccia utente includono pressioni dei tasti, movimenti del mouse, click su pulsanti, ecc. ecc.
- La maggior parte dei programmi non vogliono essere "importunati" da eventi inutili. Un programma può indicare che si prende cura soltanto di certi specifici eventi
- Ascoltatore di eventi:
  - *Riceve notifica quando un evento si verifica*
  - *Appartiene ad una classe che è fornita dal programmatore*
  - *I suoi metodi descrivono le azioni che devono essere intraprese quando un evento ha luogo*
  - *Un programma indica quali eventi necessita di gestire installando oggetti ascoltatori di eventi*
- Sorgente di eventi:
  - *Le sorgenti di evento riferiscono sugli eventi*
  - *Quando un evento si verifica, la sorgente dell'evento lo notifica a tutti gli ascoltatori dell'evento*

Eventi, sorgenti di eventi ed ascoltatori di eventi

- Esempio: Usare componenti `JButton` per pulsanti; assegnare un `ActionListener` a ciascun pulsante
- `ActionListener` è un'interfaccia:

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

- Bisogna fornire una classe il cui metodo `actionPerformed` contiene istruzioni che vanno eseguite quando si effettua il click sul pulsante
- `event` è un parametro che contiene i dettagli sull'evento, come ad esempio l'istante nel quale ha avuto luogo

Continua

Eventi, sorgenti di eventi ed ascoltatori di eventi

- Costruire un oggetto dell'ascoltatore ed aggiungerlo al pulsante:

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

ch09/button1/ClickListener.java

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03:
04: /**
05:    An action listener that prints a message.
06: */
07: public class ClickListener implements ActionListener
08: {
09:    public void actionPerformed(ActionEvent event)
10:    {
11:       System.out.println("I was clicked.");
12:    }
13: }
```

ch09/button1/ButtonViewer.java

```
01: import java.awt.event.ActionListener;
02: import javax.swing.JButton;
03: import javax.swing.JFrame;
04:
05: /**
06:    This program demonstrates how to install an action listener.
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:    }
```

Continua

ch09/button1/ButtonViewer.java (continua)

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

Output:

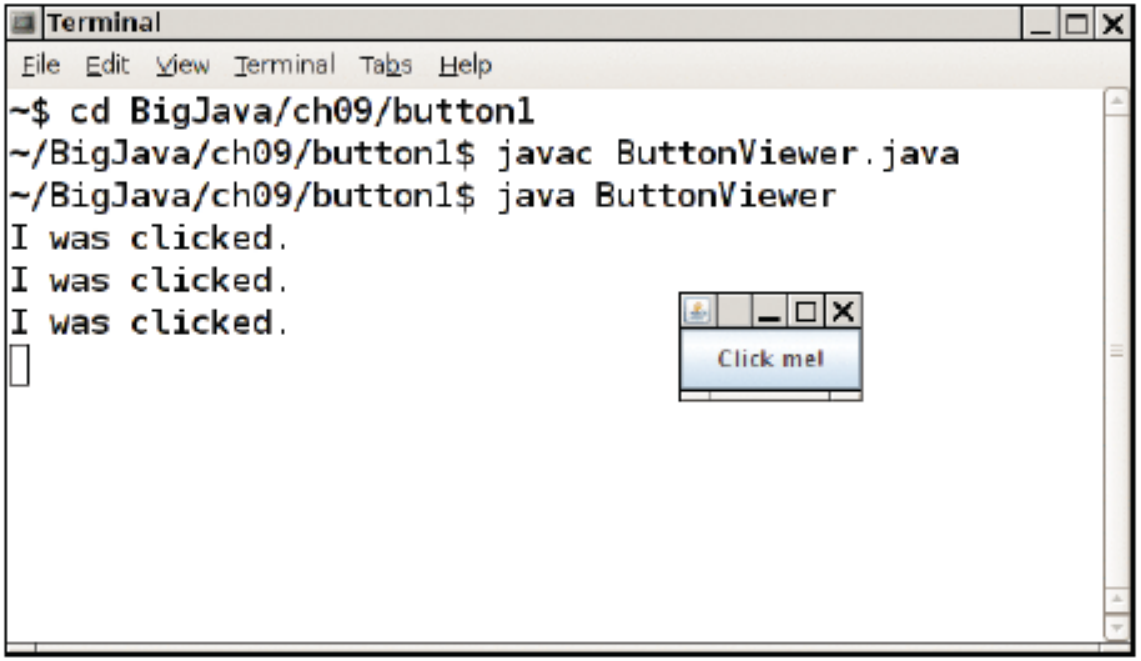

Figure 3 Implementing an Action Listener

Usare classi interne per gli ascoltatori

- Implementare semplici classi ascoltatori come classi interne:

```
JButton button = new JButton(". . .");
// This inner class is declared in the same method as the
button variable
class MyListener implements ActionListener
{
   . . .
};
ActionListener listener = new MyListener();
button.addActionListener(listener);
```

- Così la semplice classe ascoltatore è sistemata esattamente dove serve, senza “infastidire” il resto del progetto
- I metodi di una classe interna possono accedere alle variabili locali dei blocchi che la circondano e alle variabili di istanza delle classi che la circondano

Usare classi interne per gli ascoltatori

- Le variabili locali alle quali si accede da un metodo di una classe interna devono essere dichiarate come final
- **Esempio:** aggiungere l'interesse ad un conto corrente ogni volta che viene fatto click su un pulsante:

```
JButton button = new JButton("Add Interest");
final BankAccount account = new
BankAccount(INITIAL_BALANCE);
// This inner class is declared in the same method as the
   account
// and button variables.
class AddInterestListener implements ActionListener
{
```

Continua

Usare classi interne per gli ascoltatori

```
   public void actionPerformed(ActionEvent event)
   {
      // The listener method accesses the account
         variable
      // from the surrounding block
      double interest = account.getBalance() *
         INTEREST_RATE / 100;
      account.deposit(interest);
   }
};
ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

ch09/button2/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:    This program demonstrates how an action listener can access
08:    a variable from a surrounding block.
09: */
10: public class InvestmentViewer1
11: {
12:    public static void main(String[] args)
13:    {
14:       JFrame frame = new JFrame();
15:
16:       // The button to trigger the calculation
17:       JButton button = new JButton("Add Interest");
18:       frame.add(button);
19:
```

Continua

ch09/button2/InvestmentViewer1.java (continua)

```
20:       // The application adds interest to this bank account
21:       final BankAccount account = new BankAccount(INITIAL_BALANCE);
22:
23:       class AddInterestListener implements ActionListener
24:       {
25:          public void actionPerformed(ActionEvent event)
26:          {
27:             // The listener method accesses the account variable
28:             // from the surrounding block
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:
```

Continua

ch09/button2/InvestmentViewer1.java (continua)

```
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: }
```

Output:

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

Self Check 9.15

Perché un metodo di una classe interna può voler accedere ad una variabile dell'ambito che lo circonda?

Risposta: L'accesso diretto è più semplice che l'alternativa, passare la variabile come parametro ad un costruttore o ad un metodo.

Costruire applicazioni con pulsanti

- Esempio: ogni volta che si fa click sul pulsante, viene aggiunto l'interesse ed il nuovo saldo viene visualizzato

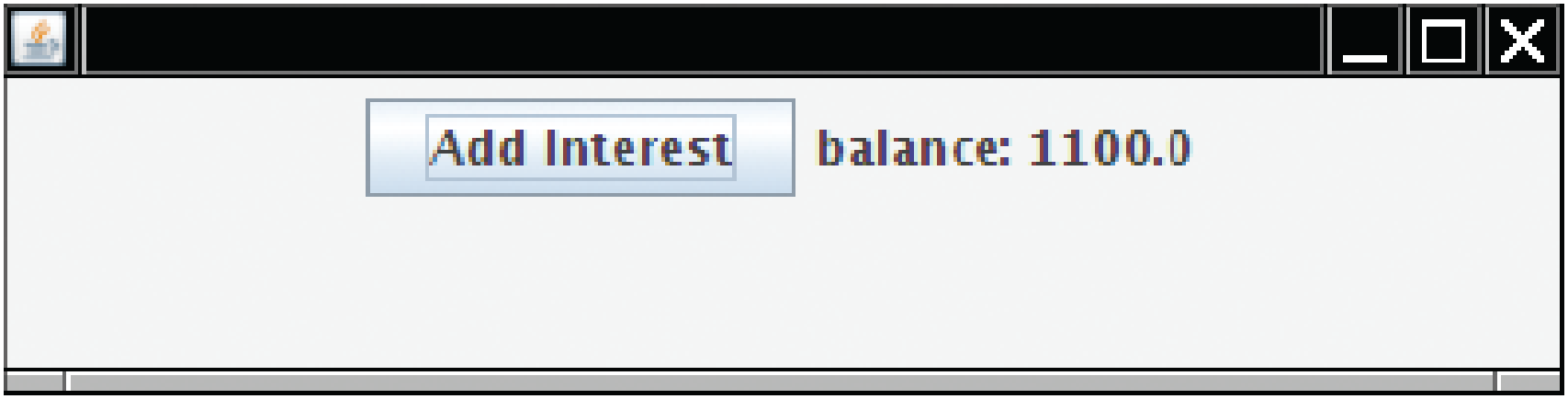

Figure 4 An Application with a Button

Continua

Costruire applicazioni con pulsanti

- Costruire un oggetto della classe `JButton`:

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

- Si usa un componente dell'interfaccia utente per visualizzare un messaggio:

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

- Usare un contenitore `JPanel` per raggruppare più componenti dell'interfaccia utente:

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

Costruire applicazioni con pulsanti

- La classe `Listener` aggiunge l'interesse e visualizza il nuovo saldo:

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

- Si aggiunga `AddInterestListener` come classe interna in modo da avere accesso alle variabili `final` che la circondano (`account` e `label`)

ch09/button3/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:    This program displays the growth of an investment.
11: */
12: public class InvestmentViewer2
13: {
14:    public static void main(String[] args)
15:    {
16:       JFrame frame = new JFrame();
17:
18:       // The button to trigger the calculation
19:       JButton button = new JButton("Add Interest");
```

Continua

ch09/button3/InvestmentViewer2.java (cont.)

```
20:
21:    // The application adds interest to this bank account
22:    final BankAccount account = new BankAccount(INITIAL_BALANCE);
23:
24:       // The label for displaying the results
25:       final JLabel label = new JLabel(
26:             "balance: " + account.getBalance());
27:
28:       // The panel that holds the user interface components
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;
```

Continua

ch09/button3/InvestmentViewer2.java (continua)

```
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: }
```

Gestire eventi del Timer

- `javax.swing.Timer` genera eventi equidistanti nel tempo
- Utile quando si vuole avere l'aggiornamento regolare nel tempo di un oggetto
- Notifica l'evento all'ascoltatore di eventi azione (action listener)

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

Continua

Gestire eventi del Timer

- Definire una classe che implementa l'interfaccia `ActionListener`

```
class MyListener implements ActionListener
{
   void actionPerformed(ActionEvent event)
   {
      // This action will be executed at each timer
         event
      Place listener action here
   }
}
```

- Aggiungere l'oggetto ascoltatore all'oggetto Timer

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

ch09/timer/RectangleComponent.java (continua)

```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import java.awt.Rectangle;
04: import javax.swing.JComponent;
05:
06: /**
07:    This component displays a rectangle that can be moved.
08: */
09: public class RectangleComponent extends JComponent
10: {
11:    public RectangleComponent()
12:    {
13:       // The rectangle that the paint method draws
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:
```

Continua

ch09/timer/RectangleComponent.java

```
23:          g2.draw(box);
24:       }
25:
26:       /**
27:          Moves the rectangle by a given amount.
28:          @param x the amount to move in the x-direction
29:          @param y the amount to move in the y-direction
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: }
```

ch09/timer/RectangleComponent.java

- Visualizza un rettangolo che può essere mosso
- Il metodo `repaint` fa ridisegnare un componente. Si chiami questo metodo ogni volta che si modificano le forme che il metodo `paintComponent` disegna

Continua

ch09/timer/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:
```

Continua

ch09/timer/RectangleMover.java (continua)

```
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; // Milliseconds between timer ticks
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

- Usare un ascoltatore del mouse per catturare eventi del mouse
- Implementare l'interfaccia MouseListener:

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button has been pressed on a
      component
   void mouseReleased(MouseEvent event);
   // Called when a mouse button has been released on a
      component
   void mouseClicked(MouseEvent event);
   // Called when the mouse has been clicked on a
      component
   void mouseEntered(MouseEvent event);
   // Called when the mouse enters a component
```

Eventi del mouse

```
    void mouseExited(MouseEvent event);
    // Called when the mouse exits a component }
```

- `mousePressed`, `mouseReleased`: chiamati quando un pulsante del mouse è premuto o rilasciato
- `mouseClicked`: se il pulsante del mouse è premuto e rilasciato in rapida successione e il mouse non è stato mosso
- `mouseEntered`, `mouseExited`: il mouse è entrato o è uscito dall'area del componente

Eventi del mouse

- Aggiungere un ascoltatore del mouse ad un componente chiamando il metodo `addMouseListener`:

```
public class MyMouseListener implements MouseListener
{
   // Implements five methods
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

- Un semplice programma: migliorare `RectangleComponent` – quando l'utente fa click su un rettangolo, muovere il rettangolo

ch09/mouse/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:    This component displays a rectangle that can be moved.
08: */
09: public class RectangleComponent extends JComponent
10: {
11:    public RectangleComponent()
12:    {
13:       // The rectangle that the paint method draws
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:
```

Continua

ch09/mouse/RectangleComponent.java (continua)

```
23:          g2.draw(box);
24:       }
25:
26:       /**
27:          Moves the rectangle to the given location.
28:          @param x the x-position of the new location
29:          @param y the y-position of the new location
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

- Chiamare `repaint` quando si modificano le forme che il metodo `paintComponent` disegna

```
box.setLocation(x, y); repaint();
```

- Ascoltatore del mouse: se il mouse è premuto, `listener` muove il rettangolo alla posizione che la classe `MousePressListener` implementa

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

Continua

Eventi del mouse

```
// Do-nothing methods
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}
```

- Tutti e cinque i metodi dell'interfaccia devono essere implementati; se alcuni di questi non sono utilizzati, possono restare vuoti

Output del programma `RectangleComponentViewer`

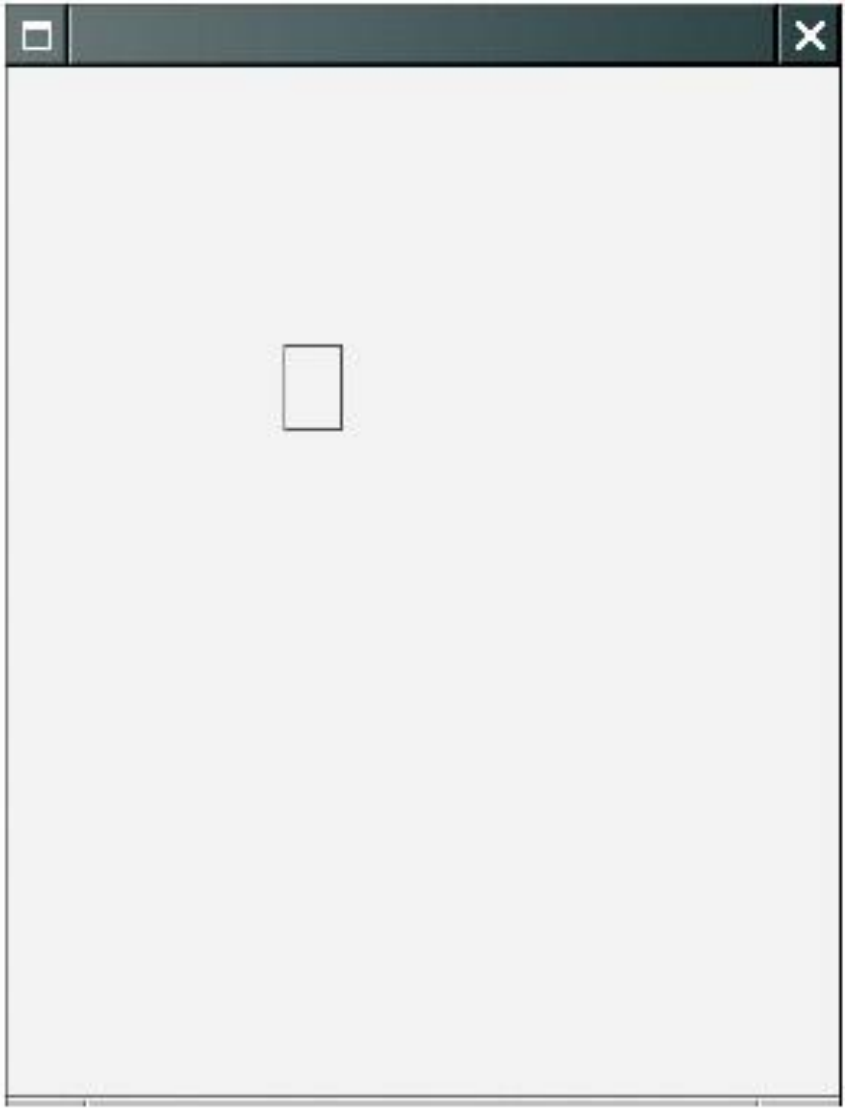

Figure 5 Clicking the Mouse Moves the Rectangle

ch09/mouse/RectangleComponentViewer.java

```
01: import java.awt.event.MouseListener;
02: import java.awt.event.MouseEvent;
03: import javax.swing.JFrame;
04:
05: /**
06:    This program displays a RectangleComponent.
07: */
08: public class RectangleComponentViewer
09: {
10:    public static void main(String[] args)
11:    {
12:       final RectangleComponent component = new RectangleComponent();
13:
14:       // Add mouse press listener
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:          }
```

Continua

ch09/mouse/RectangleComponentViewer.java (continua)

```
24:
25:          // Do-nothing methods
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: }
```