Polimorfismo

```
Measurable x;
x = new ... (BankAccount OR Coin)
double i = x.getMeasure();
```

- Quale metodo **getMeasure** viene invocato?
  - Le classi **BankAccount** e **Coin** forniscono due diverse implementazioni di **getMeasure**
- JVM utilizza il metodo **getMeasure()** della classe a cui si riferisce l'oggetto.

Polimorfismo

- L'invocazione

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

  può chiamare metodi diversi a seconda del tipo reale dell'oggetto **x**
- Il metodo **getMeasure()** viene detto polimorfico (multiforme)

- Realizzato in Java attraverso:
  - Uso di interfacce
  - Ereditarietà -- Overriding (prossime lezioni)
- Altro caso di polimorfismo in senso lato
  - Overloading -- metodi sono distinti dai parametri espliciti

Polimorfismo vs Overloading

- Entrambi invocano metodi distinti con lo stesso nome, ma...
  - Con l'overloading scelta del metodo appropriato avviene in fase di compilazione, esaminando il tipo dei parametri
    - early binding, effettuato dal compilatore
  - Con il polimorfismo avviene in fase di esecuzione
    - late binding, effettuato dalla JVM

Domande

- Supponiamo di voler utilizzare la classe **`DataSet`** per cercare la Stringa più lunga da un insieme di oggetti **`String`** in input. Funziona?
  - **Risposta:** No. La classe **`String`** non implementa l'interfaccia **`Measurable`**.

Riutilizzo di codice: problema 1

- Se vogliamo utilizzare il metodo **`getMeasure()`** per misurare oggetti di tipo **`Rectangle`**, come facciamo?
  - Non possiamo riscrivere la classe Rectangle in modo che implementi l'interfaccia Measurable

    (E' una classe standard: non abbiamo i permessi)

Riutilizzo di codice: problema 2

- Sappiamo misurare un oggetto in base ad un unico parametro
  - saldo, valore moneta, etc..
- Come facciamo a misurare un oggetto in base a parametri differenti?
  - un rettangolo con perimetro ed area
  - c/c bancario con saldo e tasso di interesse

Interfacce di smistamento

- Permettono ad una classe di richiamare un metodo prestabilito per ottenere maggiori informazioni
  - Con

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

    misurazione demandata all'oggetto stesso
  - Con

```
public interface Measurer{
    double measure(Object anObject);
    // restituisce la misura dell'oggetto anObject
}
```

    misurazione implementata in una classe dedicata

(Interfaccia di smistamento)

Misurazione dell'area dei rettangoli

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

Note su RectangleMeasurer

- La firma del metodo **`measure`** della nostra classe deve essere lo stesso del metodo omonimo nell'interfaccia **`Measurer`**
  - **`measure`** deve accettare un parametro di tipo **`Object`**
  - Occorre un cast per convertire il parametro di tipo **`Object`** in un **`Rectangle`**

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

Soluzione ai problemi

- La nuova classe **`DataSet`** viene costruita con un oggetto di una classe che realizza l'interfaccia **`Measurer`**

- Tale oggetto viene memorizzato nella variabile di istanza **`measurer`** ed è usato per eseguire le misurazioni

La classe DataSet con l'oggetto Measurer

```
/**
Serve a computare la media di un
insieme di valori
*/
public class DataSet {
/**
Costruisce un insieme vuoto
*/
public DataSet(Measurer M){
            sum = 0;
            count = 0;
            minimum = null;
            maximum = null;
            measurer = M;
}
```

```
// Restituisce la media dei valori
public double getAverage()
{
   if (count == 0) return 0;
   else return sum / count;
}
/**Restituisce un oggetto con il
valore più grande
*/
public Object getMaximum()
{
            return maximum;
}
```

La classe DataSet con l'oggetto Measurer

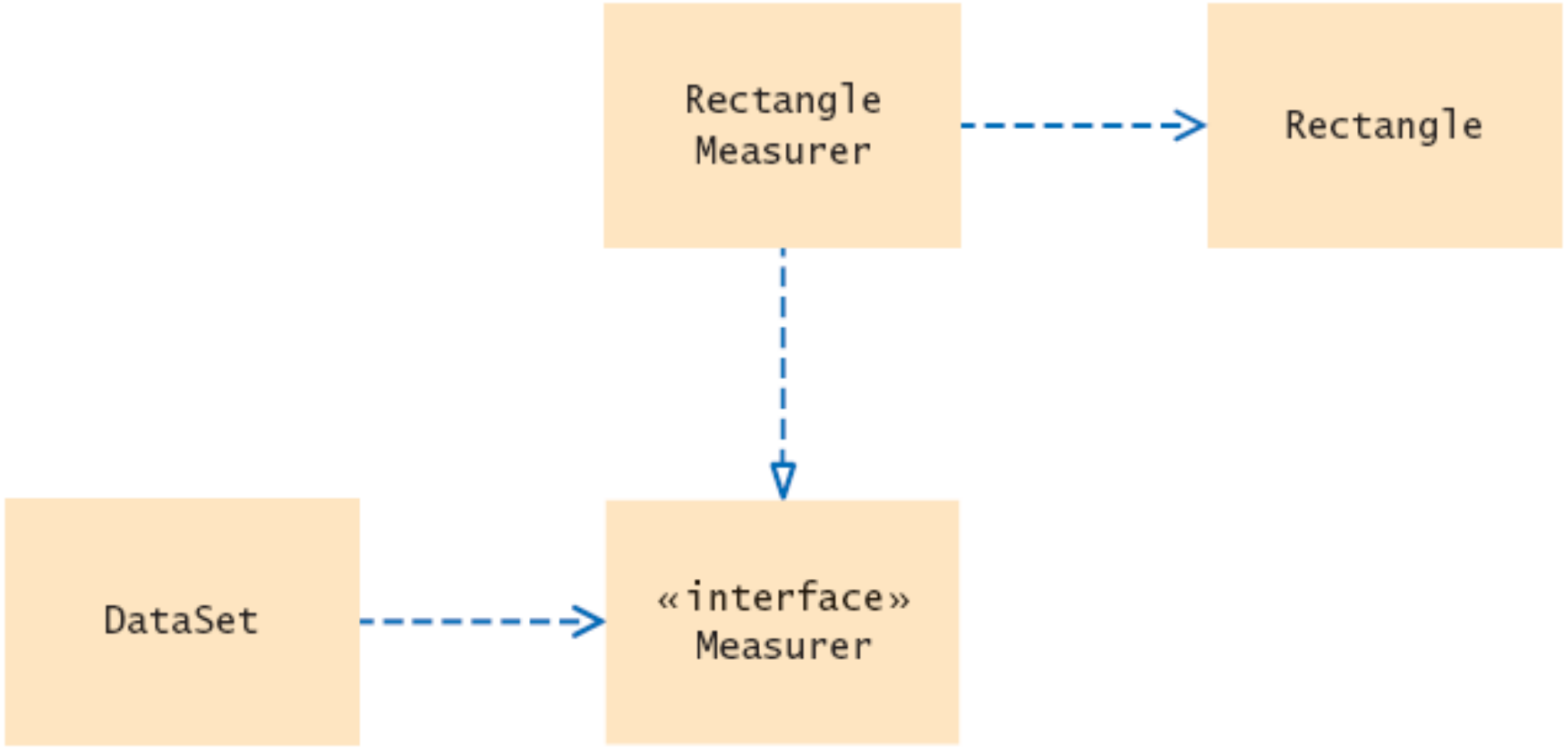
```
// Restituisce un oggetto con il valore più piccolo
public Object getMinimum() { return minimum; }

// Aggiunge un oggetto
public void add(Object x) {
            sum = sum + measurer.measure(x);
            if (count == 0 || measurer.measure(minimum) > measurer.measure(x))
                        minimum = x;
            if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
                        maximum = x;
            count++;
}

private double sum;
private Object minimum;
private Object maximum;
private int count;
private Measurer measurer;
}
```

Misurare i rettangoli

- Costruiamo un oggetto di tipo **`RectangleMeasurer`** e passiamolo al costruttore di **`DataSet`**
  - `Measurer m = new RectangleMeasurer();`
  - `DataSet data = new DataSet(m);`
- Aggiungiamo rettangoli all'insieme dei dati
  - `data.add(new Rectangle(5,10,20,30));`
  - `data.add(new Rectangle(10,20,30,40));`
- E se aggiungiamo dati di tipo diverso?
  - Viene sollevata un'eccezione nel metodo **`measure`** al punto in cui si tenta un cast a **`Rectangle`**



La classe **`Rectangle`** è separata dall'interfaccia **`Measurer`**

Misurare i rettangoli

- **`RectangleMeasurer`** è una classe ausiliaria
  - Utilizzata solo per creare oggetti di una classe che implementa l'interfaccia **`Measurer`**
  - Possiamo dichiararla all'interno del metodo che ne ha bisogno (classe interna)

Esempio

```
import java.awt.Rectangle;
public class DataSetTest {
public static void main(String[] args){
//classe interna
 class RectangleMeasurer implements Measure{
       public double measure(Object o){
         Rectangle aRectangle = (Rectangle) o;
         double area = aRectangle.getWidth() *
                                  aRectangle.getHeight();
         return area;
       }
    }
   Measurer m = new RectangleMeasurer();
   DataSet data = new DataSet(m);
   . . .
}
```

Classi interne

- Classi definite all'interno di altre classi
  - fuori dai metodi: visibile in tutti i metodi
  - all'interno di un metodo: visibile solo in questo metodo
- I metodi della classe interna
  - hanno accesso alle variabili e ai metodi a cui possono accedere i metodi della classe in cui sono definite (accesso all'ambiente in cui è definita)
    - se definite in un metodo statico accedono solo alle variabili statiche non alle variabili di istanza
  - possono accedere a variabili locali solo se sono state dichiarate **`final`**
    - Una variabile di tipo riferimento ad un oggetto è **`final`** quando si riferisce sempre allo stesso oggetto
    - Lo stato dell'oggetto può cambiare, ma la variabile non può riferirsi ad un altro oggetto

Sintassi classi interne

Dichiarata all'interno di un metodo

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

Dichiarata all'interno di una classe

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

Oggetti dimostrativi

- In molti casi si ha la necessità di testare una classe prima che l'intera applicazione sia stata completata
- Un oggetto di simulazione (*mock*) fornisce gli stessi servizi di un altro oggetto, ma in maniera semplificata
- **Esempio:** un'applicazione per gestire il registro dei voti, **`GradingProgram`**, gestisce i punteggi dei quiz (scores) usando la classe **`GradeBook`** avente i metodi:

```
public void addScore(int studentId, double score)
public double getAverageScore(int studentId)
public void save(String filename)
```

- Si vuole testare **`GradingProgram`** senza avere a disposizione tutte le funzionalità della classe **`GradeBook`**

Oggetti dimostrativi

- Bisogna dichiarare un tipo interfaccia con gli stessi metodi forniti dalla classe **`GradeBook`**
  - *In questi casi è opportuno usare la lettera* **`I`** *come prefisso per il nome dell'interfaccia:*

```
public interface IGradeBook
{
    void addScore(int studentId, double score);
    double getAverageScore(int studentId);
    void save(String filename);
    . . .
}
```

- La classe **`GradingProgram`** dovrebbe usare *solo* questa interfaccia, mai la classe **`GradeBook`** che implementa questa interfaccia

Oggetti dimostrativi

- In questo modo si può fornire una implementazione semplificata ristretta al caso di uno studente e senza funzionalità di salvataggio:

```
public class MockGradeBook implements IGradeBook
{
   private ArrayList<Double> scores;
   public void addScore(int studentId, double score)
   {
      // Ignore studentId
      scores.add(score);
   }
   double getAverageScore(int studentId)
   {
      double total = 0;
      for (double x : scores) { total = total + x; }
      return total / scores.size();
   }
   void save(String filename)
   {
      // Do nothing
   }
   . . .
}
```

Oggetti dimostrativi

- E' possibile costruire un'istanza di **`MockGradeBook`** ed usarla immediatamente per testare la classe **`GradingProgram`**

- Quando si è pronti a testare la classe reale, occorre semplicemente usare un'istanza di **`GradeBook`**

Domande

- Perchè è necessario che la classe reale e la classe dimostrativa implementano lo stesso tipo di interfaccia?
  - **Risposta:** Si vuole implementare la classe `GradingProgram` con quella interfaccia perchè così che non c'è bisogno di cambiare nulla quando si passa dalla classe `mock` alla classe reale.

Domande

- Perchè la tecnica degli oggetti dimostrativi è particolarmente efficace quando le classi **`GradeBook`** e **`GradingProgram`** sono sviluppate da due programmatori?
  - **Risposta:** Perchè lo sviluppatore di **`GradingProgram`** non deve aspettare che la classe **`GradeBook`** venga completata.

Eventi di temporizzazione

- La classe **`Timer`** in **`javax.swing`** genera una sequenza di eventi ad intervalli di tempo prefissati
  - Utile per la programmazione di una animazione
- Un evento di temporizzazione deve essere notificato ad un ricevitore di eventi
- Per creare un ricevitore bisogna definire una classe che implementa l'interfaccia **`ActionListener`** in **`java.awt.event`**

Esempio

```
class MioRicevitore implements ActionListener
 {
  public void actionPerformed(ActionEvent event)
  {
  // azione da eseguire ad ogni evento di
  // temporizzazione
  }
 }
```

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

Eventi di temporizzazione

- Un temporizzatore invoca il metodo `actionPerformed` dell'oggetto `listener` ad intervalli regolari
- Il parametro `interval` indica il lasso di tempo tra due eventi in millisecondi
- Vediamo un programma che conta all'indietro fino a zero con un secondo di ritardo tra un valore e l'altro

Programma CountDown

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JOptionPane;
import javax.swing.Timer;

public class TimerTest{ // Questo programma collauda la classe Timer
    public static void main(String[] args){
     class CountDown implements ActionListener {
          public CountDown(int initialCount){ count = initialCount;}
          public void actionPerformed(ActionEvent event){
                   if (count >= 0) System.out.println(count);
                   count--;
          }
          private int count;
     }
     CountDown listener = new CountDown(10);
     Timer t = new Timer(1000, listener); t.start();
     JOptionPane.showMessageDialog(null, "Quit?"); System.exit(0);
  }
}
```