

IA – 2014 / 2015

# Programmation Orientée Objet

## **Cours 4**

Gérald Oster <oster@loria.fr>

# Plan du cours

- Introduction
- Programmation orientée objet :
  - Classes, objets, encapsulation, composition
  - 1. Utilisation
  - 2. Définition
- Héritage et polymorphisme :
  - Interface, classe abstraite, liaison dynamique
- Exceptions
- Généricité

# 8<sup>ème</sup> Partie : Interface et polymorphisme

## Objectifs de cette partie

---

- Découvrir la notion d'interface
- Être capable de réaliser de convertir des références d'interfaces en références de classes
- Découvrir et comprendre le concept de “polymorphisme”
- Apprécier comment les interfaces peuvent découpler les classes
- Apprendre à implémenter des classes “helper” en utilisant des classes internes (*inner classes*)
- Savoir comment les classes internes accèdent aux variables de portées englobantes

# Les interfaces pour améliorer la ré-utilisabilité du code

---

- Un cas d'utilisation des *types* interfaces : rendre du code réutilisable
- À la fin du 2ème cours, nous avons défini une classe `DataSet` pour calculer la moyenne et le maximum d'un ensemble de valeurs (*double*)
- Que devons nous faire si nous souhaitons calculer le solde moyen et le solde maximum d'un ensemble de `BankAccount` ?

# Les interfaces pour améliorer la ré-utilisabilité du code /2

```
public class DataSet // Modifiée pour des objets BankAccou
{
    . . .
    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;
}
```

## Les interfaces pour améliorer la ré-utilisabilité du code /3

Et si l'on suppose que l'on veuille faire le même genre de calcul pour la classe `Coin`. On devrait encore apporter les mêmes modifications à classe `DataSet` :

```
public class DataSet // Modifiée pour des objets Coin
{
    . . .
    public void add(Coin x)
    {
        sum = sum + x.getValue();
        if (count == 0 || maximum.getValue() <
            x.getValue()) maximum = x;
        count++;
    }
}
```

# Les interfaces pour améliorer la ré-utilisatibilité du code /4

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

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



# Les interfaces pour améliorer la ré-utilisabilité du code /5

- Dans tous les cas, les mécanismes d'analyse sont les mêmes; seule la façon précise de mesurer diffère
- Les classes peuvent “se mettre d'accord” sur une méthode `getMeasure` qui permettrait d'obtenir la mesure à analyser
- On peut implémenter une seule classe réutilisable `DataSet` dont le corps de la méthode `add` ressemblerait à:

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

# Les interfaces pour améliorer la ré-utilisabilité du code /6

- Mais quel est le type de la variable `x`?  
`x` devrait référencer n'importe quelle classe qui fournisse la méthode `getMeasure`
- En Java, un *type* interface est utilisé pour spécifier les opérations obligatoires

```
public interface Measurable
{
    double getMeasure();
}
```
- La déclaration d'une interface liste toutes les méthodes (et leur signature) que le type interface requiert

# Interfaces vs. Classes

---

Un type interface est similaire à une classe, mais il y a des différences fondamentales :

- *Toutes les méthodes d'une interface sont abstraites ; elles n'ont pas d'implémentation*
- *Toutes les méthodes d'une interface sont publiques*
- *Une interface ne possède pas de variables d'instance*

# Classe générique DataSet pour des objets “mesurable”

```
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;
    }
}
```

## Classe générique DataSet pour des objets “mesurable” /2

---

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

# Implémenter une interface

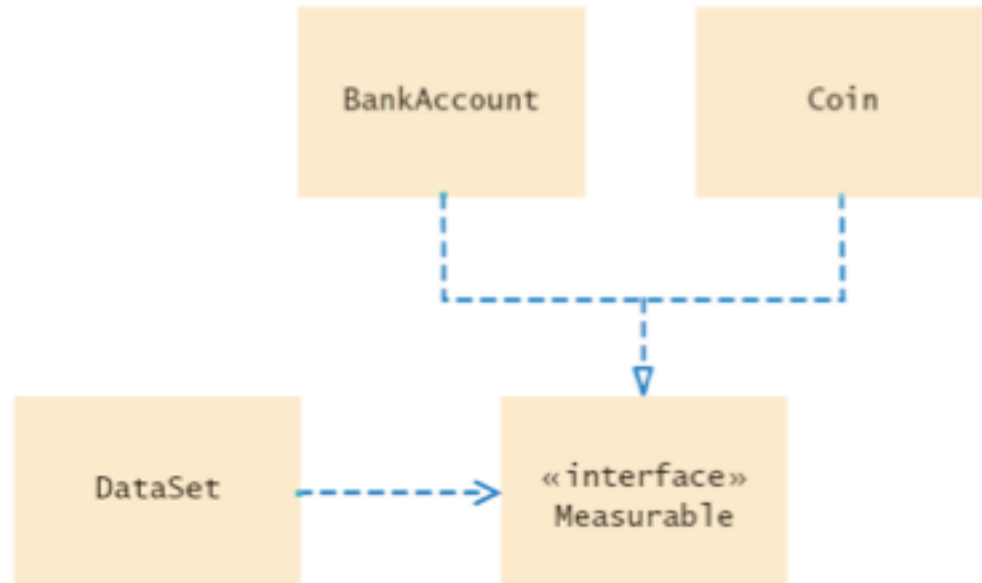
- Utiliser le mot-clé `implements` pour indiquer qu'une classe implémente une interface

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

- Une classe peut implémenter plus d'une interface
  - *Une classe doit obligatoirement définir toutes les méthodes qui sont requises par les interfaces qu'elle implémente*

# Diagramme de classes UML (`DataSet` et les classes en relation)

- Les interfaces réduisent le couplage entre classes
- Notation UML :
  - *Les interfaces sont étiquetées avec un "stereotype" indiquant «interface»*
  - *Une ligne se terminant par un triangle vide dénote une relation “est-un” entre une classe et une interface*
  - *Un ligne se terminant par une flèche dénote une relation “est client de” ou “utilise”*
- Remarque : `DataSet` est découplé de `BankAccount` et de `Coin`



# Syntaxe Définition d'une interface

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

## Exemple :

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

## Objectif :

Définir une interface et la signature de ses méthodes. Toutes les méthodes sont obligatoirement/automatiquement publiques.



# Syntaxe Implémentation d'une interface

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

## Exemple :

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

## Objectifs :

Définir une classe qui implémente (réalise) une interface.

## 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 /2

```
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: }
```

### Output:

Average balance: 4000.0

Expected: 4000

Highest balance: 10000.0

Expected: 10000

Average coin value: 0.13333333333333333333

Expected: 0.133

Highest coin value: 0.25

Expected: 0.25

## Questions

---

Supposons que l'on souhaite utiliser la classe `DataSet` pour connaître le pays (`Country`) qui possède la plus grande population. Quelle condition la classe `Country` doit-elle remplir ?

**Réponse :** Elle doit implémenter l'interface `Measurable` et sa méthode `getMeasure` doit retourner la population du pays.

## Questions

---

Pourquoi la méthode `add` de la classe `DataSet` ne peut tout simplement pas avoir un paramètre de type `Object` ?

**Réponse :** La classe `Object` n'a pas de méthode `getMeasure`, et la méthode `add` invoque la méthode `getMeasure`.

# Conversion entre types d' une classe et une interface

- On peut convertir une référence d' une classe en une référence d' une interface si la classe implémente l' interface
- ```
BankAccount account = new BankAccount(10000);  
Measurable x = account; // OK
```
- ```
Coin dime = new Coin(0.1, "dime");  
Measurable x = dime; // OK
```
- **Conversion interdite entre types qui n' ont aucune relation**  

```
Measurable x = new Rectangle(5, 10, 20, 30); // ERREUR
```
- **Car Rectangle n' implémente pas Measurable**

# Transtypage (Cast)

- Ajout d'objet Coin à un 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
```
- Et maintenant comment on utilise cette référence ? Ce n'est plus une référence vers Coin

```
String name = max.getName(); // ERREUR
```
- On doit "transtyper" la référence pour la convertir vers le type de l'objet (dynamique)
- On sait que c'est un objet de type Coin, mais le compilateur ne le sait pas. Transtyper (cast) :

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



## Transtypage (Cast) /2

---

- Si on s' est trompé et que  $\text{max } n'$  est pas un Coin, le compilateur renvoie une erreur (sous forme d' exception)
- Différence par rapport au cast avec les nombres :
  - Quand on cast un nombre on s' accorde sur la perte d' information
  - Quand on cast une référence, on risque de déclencher une erreur

## Questions

---

Peut-on utiliser le transtypage `(BankAccount)` `x` pour convertir la `x` de type `Measurable` en une référence de type `BankAccount` ?

**Réponse :** Seulement si `x` référence réellement un objet de type `BankAccount`.

## Questions

---

Si `BankAccount` et `Coin` implémentent l'interface `Measurable`, peut-on convertir une référence de type `Coin` en une référence `BankAccount`?

**Réponse :** Non – une référence de type `Coin` peut être convertie en une référence de type `Measurable`, mais si l'on essaye de la convertir vers une référence de type `BankAccount`, alors une exception est levée

# Polymorphisme

- Une variable maintient une référence vers un objet dont la classe implémente une interface

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

- Noter que l'objet référencé par x n'est pas de type `Measurable` ; Le type de l'objet est une classe qui implémente l'interface `Measurable`

- On peut appeler n'importe quelle méthode de l'interface :

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

- Quelle méthode est appelée ?

# Polymorphisme /2

---

- Dépend du type de l'objet référencé (type dynamique)
- Si `x` référence un compte bancaire, alors la méthode `getMeasure` de `BankAccount` appelée
- Si `x` référence une pièce, alors `c'` est la méthode de la classe `Coin`
- Polymorphisme (plusieurs formes): Comportement varie en fonction du type réel de l'objet
- Appelé liaison dynamique (*late binding*) résolu à l'exécution
- Différent de la surcharge qui est résolue à la compilation (*early binding*)

## Questions

---

Pourquoi ne peut on pas construire d' objet de type `Measurable` ?

**Réponse :** `Measurable` est une interface. Les interfaces ne contiennent pas de variable d' instance, ni d' implémentation de méthodes.

## Questions

---

Pourquoi peut-on néanmoins déclarer une variable dont le type est Measurable?

**Réponse :** Une telle variable ne référence jamais un objet de type Measurable. Elle référence un objet d'une certaine classe qui implémente l'interface Measurable.

# Interfaces pour implémenter un mécanisme de rappel

- Limitations de l'interface `Measurable` :
  - *On ne peut ajouter l'interface `Measurable` qu'aux classes dont on a le contrôle*
  - *On ne peut mesurer un objet que d'une seule manière*
- Mécanisme de rappel (*Callback mechanism*) : permet à une classe de rappeler une méthode spécifique lorsque l'on a besoin d'information supplémentaire
- Dans l'implémentation précédente `DataSet`, la responsabilité de mesurer revient aux objets eux-mêmes



## Interfaces pour implémenter un mécanisme de rappel /2

- Alternative : Passer l'objet à mesurer à une méthode :

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

- `Object` est “le plus petit dénominateur” de toutes les classes

## Interfaces pour implémenter un mécanisme de rappel /3

méthode `add` fait appel à “`measurer`” (et non l’objet ajouté) pour effectuer la mesure :

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

## Interfaces pour implémenter un mécanisme de rappel /4

- On peut définir des “measurer” pour tout type de mesure

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

- On doit transtyper (cast) de Object vers Rectangle  
`Rectangle aRectangle = (Rectangle) anObject;`

## Interfaces pour implémenter un mécanisme de rappel /5

- Passage d'un "measurer" à la construction du DataSet :

```
Measurer m = new RectangleMeasurer();
```

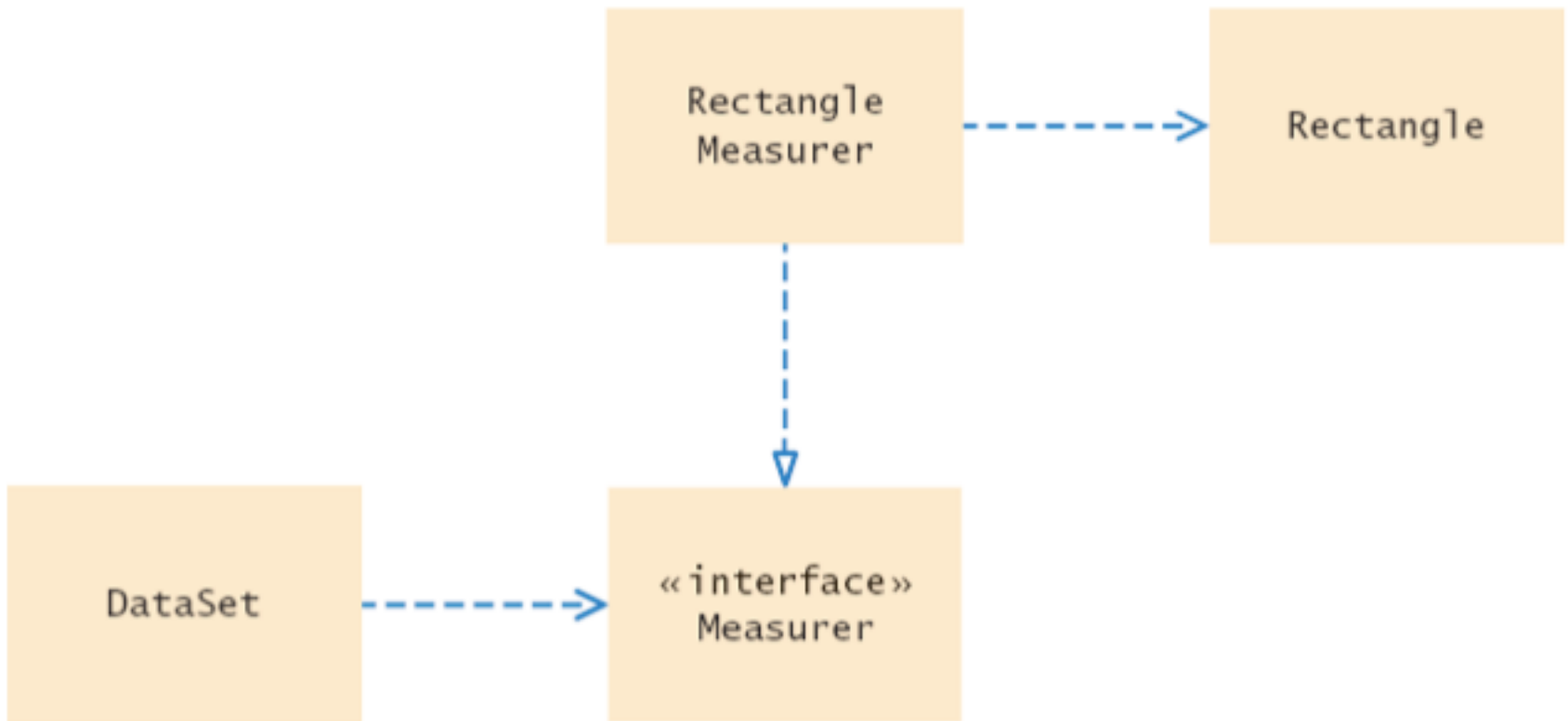
```
DataSet data = new DataSet(m);
```

```
data.add(new Rectangle(5, 10, 20, 30));
```

```
data.add(new Rectangle(10, 20, 30, 40)); . . .
```

## Diagramme de classes UML de l'interface `Measurer` ...

La classe `Rectangle` est bien découplée de l'interface `Measurer`



## 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:      */
```

## ch09/measure2/DataSet.java /2

```
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:
```

## ch09/measure2/DataSet.java /3

```
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:
```

## ch09/measure2/DataSetTester2.java /2

```
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: }
```

### 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]

## Questions

---

Pourquoi la méthode `measure` de l'interface `Measurer` possède un paramètre alors que la méthode `getMeasure` de l'interface `Measurable` n'en possède pas ?

**Réponse :** Un `Measurer` mesure un objet passé en paramètre, alors que la méthode `getMeasure` mesure son propre objet (le receveur de l'appel de méthode).

# Classes internes

- Une classe très simple peut être définie à l'intérieur d'une méthode

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

## Classes internes /2

---

- Si une classe interne est définie à l'intérieur d'une classe mais hors d'une méthode, elle est disponible pour toutes les méthodes de la classe englobante
- Le compilateur transforme la classe interne en des classes régulières :

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



# Syntaxe Classes internes

## Déclaration dans une méthode

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

## Déclaration dans une classe

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

# Syntaxe Classes internes

## Exemple :

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

## Objectif :

Déclarer une classe interne dont la portée est limitée à une seule méthode ou à une seule 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:
```

## ch09/measure3/DataSetTester3.java /2

```
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: }
```

## Questions

---

Pourquoi utiliseriez-vous une classe interne à la place d' une classe régulière ?

**Réponse :** Une classe interne se révèle utile pour implémenter des classes non significatives. De plus, les méthodes de cette classe peuvent accéder aux variables des blocs englobants.