

# Programmation Orientée Objet

## **4ème Partie**

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
- Généricité
- (Collections)

# 6<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 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;
}
```

## 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-utilisabilité 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

---

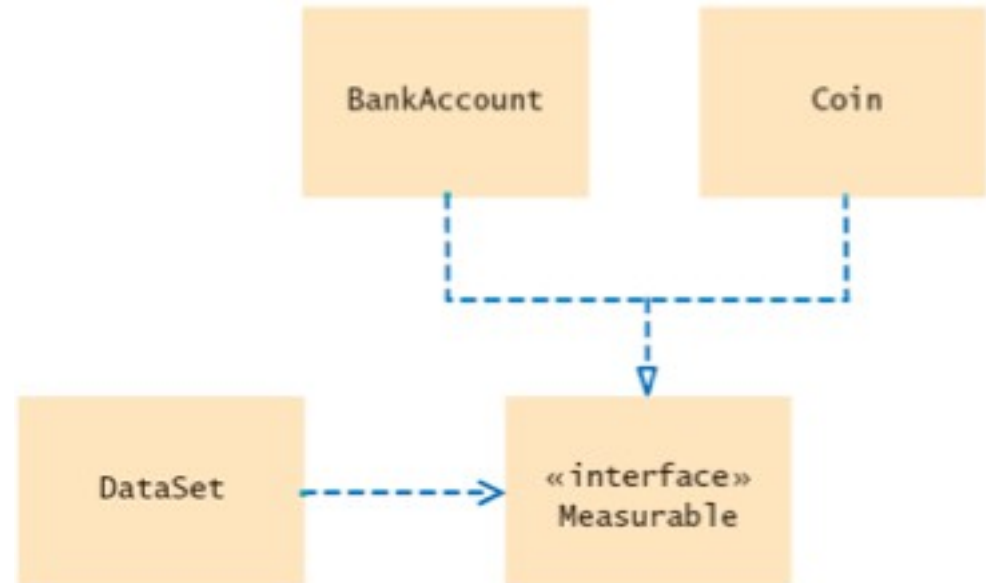
- Utiliser le mot-clé `implements` pour indiquer qu'une classe implémente une interface

```
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 une 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 `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 ?

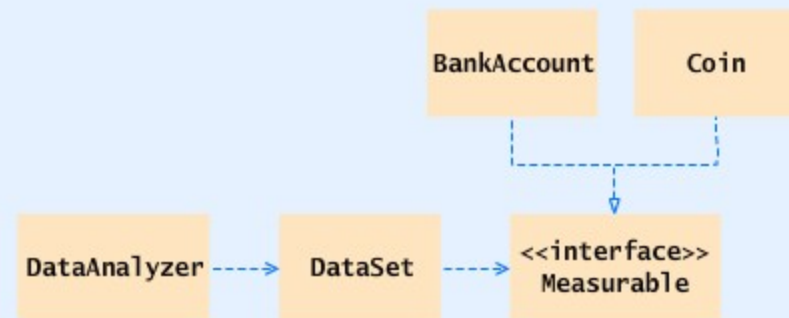
## Polymorphisme /2

---

- 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*)

# Animation 9.1 –

```
public interface Measurable
{
    double getMeasure();
}
public class BankAccount
{
    . . .
    public double getMeasure()
    {
        return balance;
    }
    . . .
}
public class Coin
{
    . . .
    public double getMeasure()
    {
        return value;
    }
    . . .
}
```



This animation demonstrates the phenomenon of polymorphism.

## 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();
```

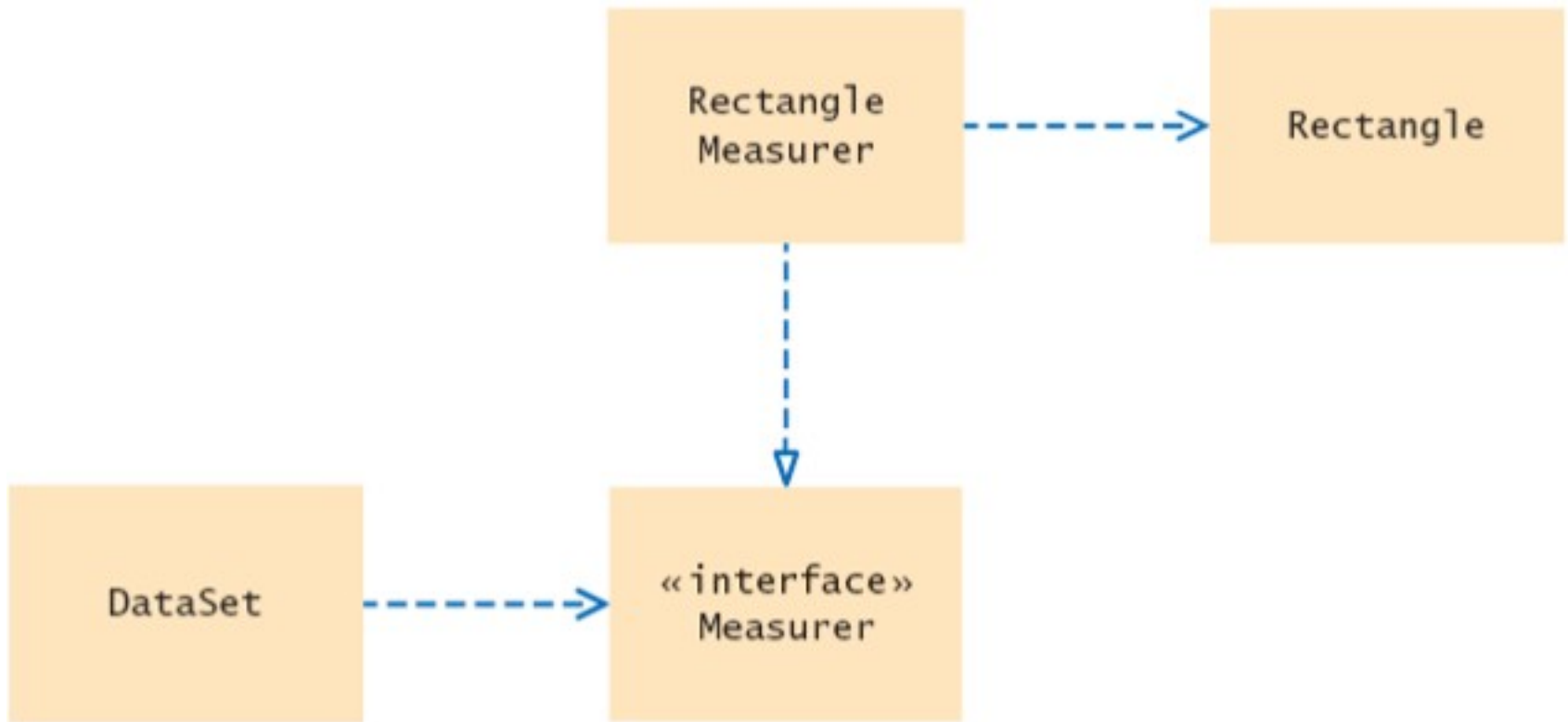
```
DataSet data = new DataSet(m);
```

```
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/Measurerer.java

```
01: /**
02:     Describes any class whose objects can measure other objects.
03: */
04: public interface Measurerer
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 /2

---

### 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 le 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.

# 7<sup>ème</sup> Partie : Héritage

## Objectifs de cette partie

---

- Découvrir la notion d'héritage
- Comprendre comment hériter ou redéfinir des méthodes d'une classe mère
- Savoir quand appeler les constructeurs des classes mères
- Apprendre l'effet du mot clé `protected` et ses effets sur le contrôle d'accès des paquetages
- Découvrir le comportement commun à tout objet défini dans la classe `Object` et comment redéfinir les méthodes telles que `toString` et `equals`



## Introduction à l'héritage

---

- Héritage : étendre des classes en ajoutant des méthodes et des variables d'instance
- Exemple : Compte d'épargne = compte bancaire avec des intérêts

```
class SavingsAccount extends BankAccount
{
    new methods
    new instance fields
}
```

- SavingsAccount hérite automatiquement de toutes les méthodes et variables d'instance de la classe BankAccount

```
SavingsAccount collegeFund = new SavingsAccount(10);
// Savings account with 10% interest
collegeFund.deposit(500);
// OK to use BankAccount method with SavingsAccount
object
```

## Introduction à l'héritage /2

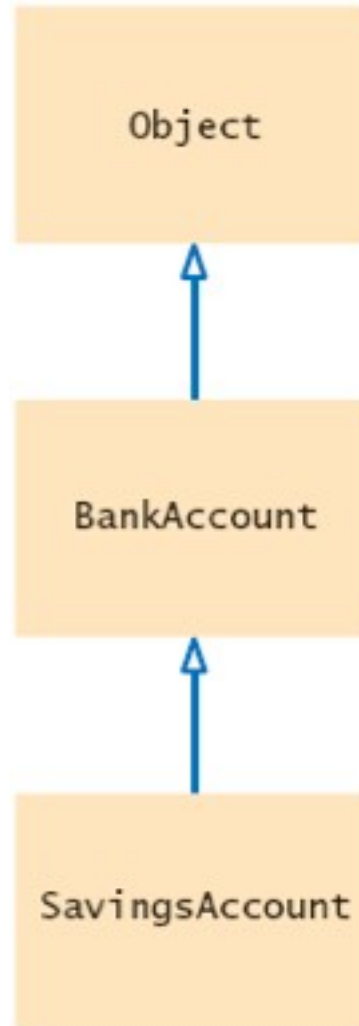
---

- Classe étendue = Classe mère = *super classe* (BankAccount),  
Classe étendant = Sous classe (Savings)
- Hériter d'une classe  $\neq$  d'implémenter une interface : une sous classe hérite de l'implémentation des méthodes et de l'état (variables d'instance)
- Un des avantages de l'héritage : la réutilisation de code

## Héritage : Diagramme

---

Toute classe hérite de la classe `Object` soit directement soit indirectement



## Introduction à l'héritage /3

---

- Dans la sous classe, sont spécifiés :
  - Les variables d'instance que l'on ajoute
  - Les méthodes que l'on ajoute
  - Les méthodes que l'on redéfinit (dont on change le comportement)

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }

    public void addInterest()
    {
        double interest = getBalance() * interestRate/100;
        deposit(interest);
    }

    private double interestRate;
}
```

- Encapsulation : La méthode `addInterest` appelle `getBalance` plutôt que de mettre à jour directement la variable `balance` de la classe mère (la variable est déclarée `private`)
- Remarquer que `addInterest` appelle `getBalance` sans spécifier le receveur (l'appel s'applique à l'objet lui-même)

## Sous classe

L'objet `SavingsAccount` hérite de la variable d'instance `balance` de la classe `BankAccount`, et gagne une variable additionnelle : `interestRate`:

### SavingsAccount

`balance = 10000`

`interestRate = 10`

BankAccount portion

## Syntaxe Héritage

---

```
class SubclassName extends SuperclassName
{
    methods
    instance fields
}
```

### Exemple :

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }
}
```

## Syntaxe Héritage

```
public void addInterest()
{
    double interest = getBalance() * interestRate / 100;

    deposit(interest);
}

private double interestRate;
}
```

### Objectifs :

Définir une nouvelle classe en héritant du comportement (les méthodes) et de l'état (les variables d'instance) d'une classe existante (la classe mère).



## Questions

---

Combien de variables d'instance possède un objet de la classe `SavingsAccount` ?

**Réponse :** 2 variables d'instance : `balance` et `interestRate`.

## Questions

---

Donnez quatre noms de méthode que vous pouvez appeler sur un objet de type `SavingsAccount`.

**Réponse :** `deposit`, `withdraw`, `getBalance`, et `addInterest`.

## Questions

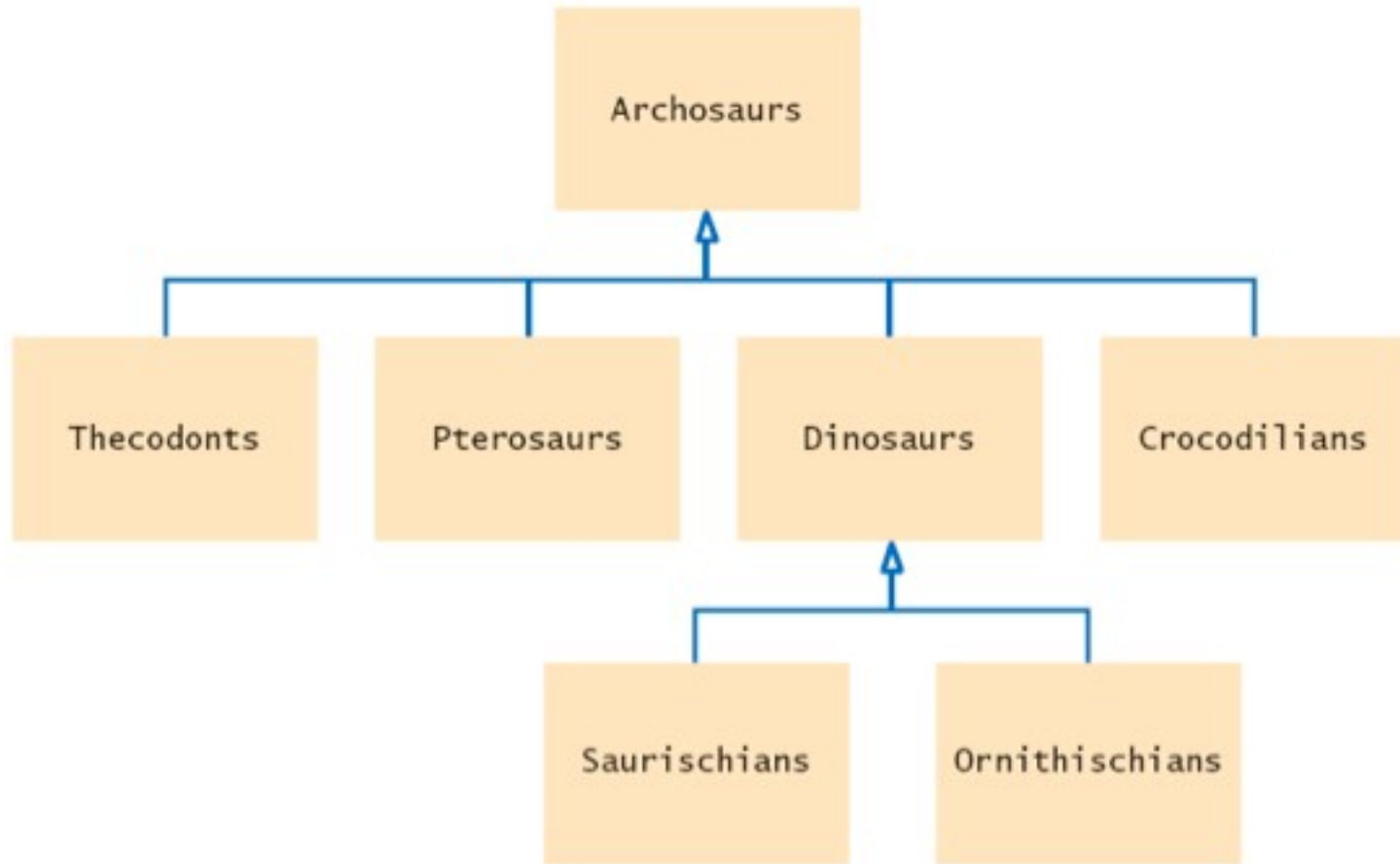
---

Si la classe `Manager` étend la classe `Employee`, quelle est la classe mère et quelle est la classe fille ?

**Réponse :** `Manager` est la classe fille (sous classe); `Employee` est la classe mère (super classe).

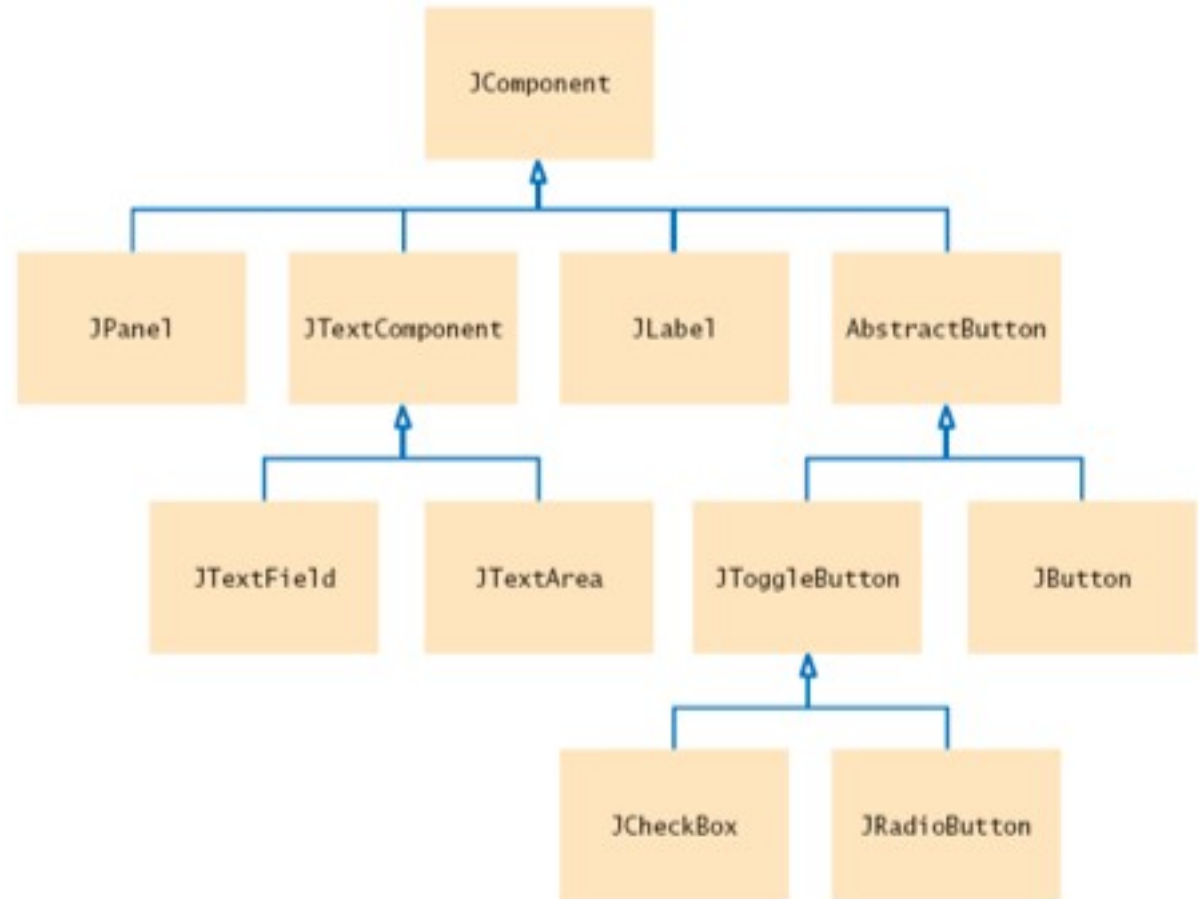
## Hiérarchie de classes

- Ensemble de classes qui forme arbre d'héritage
- Exemple :



## Hierarchie de classes /2

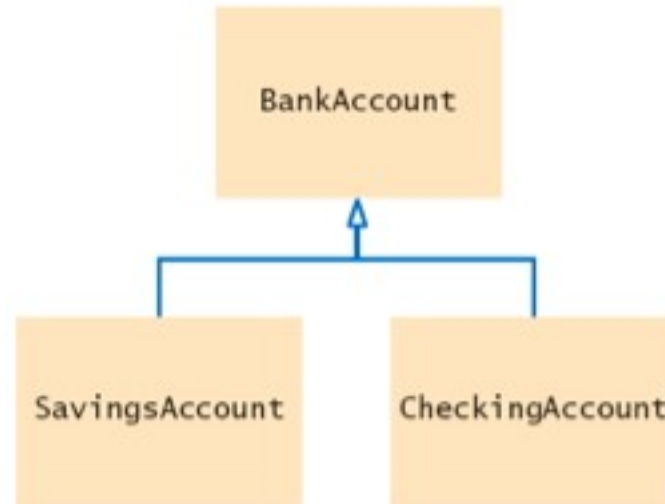
- La classe mère JComponent possède les méthodes getWidth, getHeight
- La classe AbstractButton fournit les méthodes pour consulter/modifier le texte d'un bouton et son icône



## Hiérarchie de classes /3

- Considérons une banque qui offre à ses clients deux types de compte :
  1. *Compte courant (Checking account): pas d'intérêt; un nombre (peu élevé) de transactions gratuites, des frais additionnels pour chaque transaction supplémentaire*
  2. *Compte d'épargne (Savings account) : des intérêts chaque mois*

- Hiérarchie de classe :



- Tous les comptes supportent la méthode `getBalance`
- Tous les comptes supportent les méthodes `deposit` et `withdraw`, mais leur implémentation diffère
- Compte courant requiert une méthode `deductFees`; Compte d'épargne requiert une méthode `addInterest`

## Questions

---

Quel est le rôle de la classe `JTextComponent` dans la hiérarchie présentée précédemment ?

**Réponse :** Exprimer et factoriser le comportement commun à tous les composants graphiques

## Questions

---

Quelle variable d'instance doit-on ajouter dans la classe `CheckingAccount`?

**Réponse :** On doit ajouter un compteur qui compte le nombre de dépôts et de retraits effectués.



## Héritage de méthodes

---

- Rédéfinition de méthodes (overriding) :
  - *Fournir une implémentation différente d'une méthode existante dans la classe mère*
  - *Doit avoir la même signature (même nom et même nombre et type de paramètres)*
  - *Si une méthode est appliquée sur un objet de la sous classe, la redéfinition de cette méthode est exécutée (cf. TD)*
- Méthodes héritées :
  - *Ne pas fournir de nouvelle implémentation pour une méthode de la classe mère*
  - *Les méthodes de la classes mère peuvent être appliquée sur des instances de la classe fille*
- Méthodes ajoutées :
  - *Fournir une méthode qui n'existe pas dans la classe mère*
  - *Cette nouvelle méthode ne peut être appliquée que sur les objets de la classe fille*

## Héritage des variables d'instance

---

- On ne peut redéfinir les variables d'instance de la classe mère
- Variables héritées: Toutes les variables de la classe mère sont automatiquement héritées
- Variables ajoutées : Définir de nouvelles variables qui n'existaient pas dans la classe mère
- Que se passe-t-il si l'on définit une nouvelle variable avec le même nom qu'une variable de la classe mère ?
  - *Chaque objet possèdera deux variables d'instances avec le même nom*
  - *Ces variables pourront contenir des valeurs différentes*
  - *Possible mais clairement déconseillé*

## Implémentation de la classe `CheckingAccount`

- Rédéfinir les méthodes `deposit` et `withdraw` pour incrémenter le compteur de transactions :

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public void deductFees() { . . . }
    // new method private int transactionCount;
    // new instance field
}
```

- Chaque objet `CheckingAccount` possède deux variables d'instance :
  - *balance* (hérité de *BankAccount*)
  - *transactionCount* (nouvellement ajouté à *CheckingAccount*)

## Implémentation de la classe `CheckingAccount` /2

---

- On peut appliquer 4 méthodes aux objets de la classe `CheckingAccount` :
  - *`getBalance()`* (hérité de *`BankAccount`*)
  - *`deposit(double amount)`* (rédéfini *`BankAccount`*)
  - *`withdraw(double amount)`* (rédéfini *`BankAccount`*)
  - *`deductFees()`* (ajouté à *`CheckingAccount`*)

## Variables d'instance héritées sont privées (Private)

---

- Considérons la méthode `deposit` de `CheckingAccount`

```
public void deposit(double amount)
{
    transactionCount++;
    // now add amount to balance
    . . .
}
```
- On ne peut pas ajouter simplement `amount` à `balance`
- `balance` est une variable *privée* de la classe mère
- Une sous classe n'a pas accès aux variables privées de sa classe mère
- Une sous classe doit donc utiliser l'interface publique de la classe mère

## Appel d'une méthode de la classe mère

---

- On ne peut pas appeler simplement `deposit(amount)` dans `deposit` de `CheckingAccount`
- Cela reviendrait à appeler `this.deposit(amount)`
- Et donc exécuter une boucle d'appel récursive infinie
- À la place, pour invoquer la méthode la classe mère `super.deposit(amount)`
- Maintenant cela appelle bien la méthode `deposit` telle que définie dans la classe `BankAccount`

## Appel d'une méthode de la classe mère /2

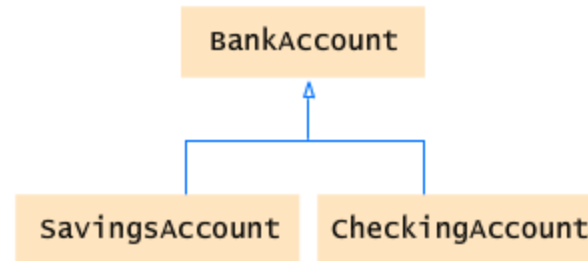
---

- Méthode complète :

```
public void deposit(double amount)
{
    transactionCount++;
    // Now add amount to balance
    super.deposit(amount);
}
```

## Animation 10.1 –

```
public class BankAccount
{
    public BankAccount() { . . . }
    public BankAccount(double initialBalance) { . . . }
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public double getBalance() { . . . }
    private double balance;
}
```



```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate) { . . . }
    public void addInterest() { . . . }
    private double interestRate;
}
```

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount() { . . . }
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public void deductFees() { . . . }
    private int transactionCount;
}
```

This animation demonstrates the process of inheritance.





## Syntaxe Appel d'une méthode de la classe mère

---

```
super.methodName(parameters)
```

### Exemple :

```
public void deposit(double amount)
{
    transactionCount++;
    super.deposit(amount);
}
```

### Objectif :

Appeler une méthode de la classe mère éventuellement masquée par une méthode de la fille.

## Implémentation des autres méthodes

---

```
public class CheckingAccount extends BankAccount
{
    . . .
    public void withdraw(double amount)
    {
        transactionCount++;
        // Now subtract amount from balance
        super.withdraw(amount);
    }

    public void deductFees()
    {
        if (transactionCount > FREE_TRANSACTIONS)
        {
            double fees = TRANSACTION_FEE
                * (transactionCount - FREE_TRANSACTIONS);
            super.withdraw(fees);
        }
    }
}
```

## Implémentation des autres méthodes /2

---

```
        transactionCount = 0;
    }
    . . .
    private static final
    int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;
}
```

## Erreur classique : Masquer une variable d'instance

- Une sous classe n'a pas accès aux variables privées de sa classe mère
- Erreur du débutant : “résoudre” le problème en ajoutant une variable d'instance dans la classe fille portant le même nom :

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount)
    {
        transactionCount++;
        balance = balance + amount;
    }
    . . .
    private double balance; // NON !!!
}
```

## Erreur classique : Masquer une variable d'instance /2

- Maintenant, la méthode deposit compile correctement, mais elle ne met plus à jour correctement le solde du compte!

### CheckingAccount

balance = 10000

transactionCount = 1

balance = 5000

BankAccount portion

## Construction d'une classe fille

---

- `super` suivi de parenthèses désigne l'appel au constructeur de la classe mère (super constructeur)

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        // Construct superclass
        super(initialBalance);
        // Initialize transaction count
        transactionCount = 0;
    }
    . . .
}
```

- Cette instruction doit être la *première* instruction du constructeur de la classe fille

## Construction d'une classe fille /2

---

- Si le constructeur d'une classe fille ne fait pas appel explicitement à un des constructeurs de la classe mère, le constructeur par défaut est appelé
  - *Constructeur par défaut = constructeur sans paramètre*
  - *Constructeur par défaut d'une classe est ajouté si il n'existe pas d'autres constructeurs déjà défini*
  - *Attention aux appels implicites du construteurs par défaut qui ne serait pas présent*

## Syntaxe Appel du super constructeur

```
ClassName(parameters)  
{  
    super(parameters);  
    . . .  
}
```

### Exemple :

```
public CheckingAccount(double initialBalance)  
{  
    super(initialBalance);  
    transactionCount = 0;  
}
```

### Objectif :

Appeler le constructeur de la classe mère. Cette instruction doit être la première instruction du constructeur de la classe fille.



## Questions

---

Quand vous faites appel à une méthode de la classe mère en utilisant le mot-clé `super`, cet appel doit-il être la première instruction de la méthode de la sous classe ?

**Réponse :** Non – c'est une obligation uniquement pour les constructeurs. Par exemple, la méthode `deposit` de la classe `SavingsAccount` incrémente d'abord le compteur de transactions avant de faire appel à la méthode de la classe mère.

## Conversion entre les types de la sous classe et de la classe mère

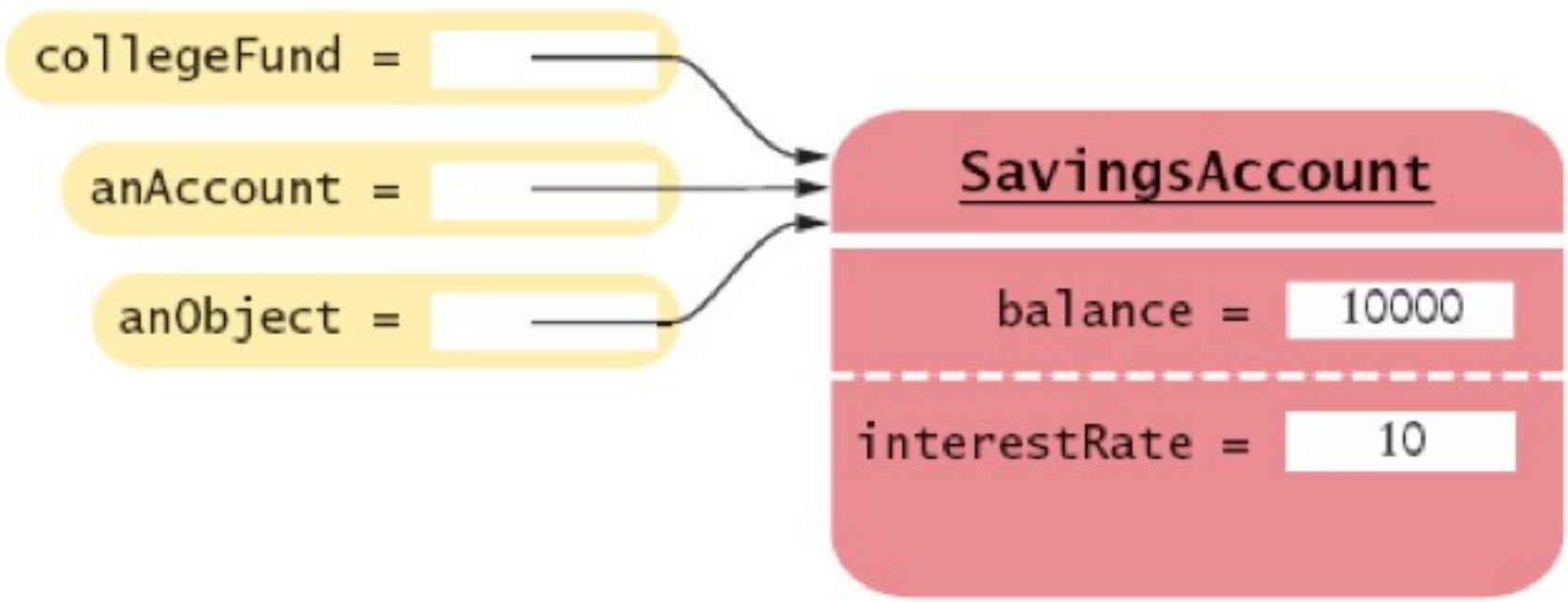
---

- Ok de convertir une référence de la sous classe en une référence de la classe mère

```
SavingsAccount collegeFund = new SavingsAccount(10);  
BankAccount anAccount = collegeFund;  
Object anObject = collegeFund;
```

- Les trois références d'objet stockées dans `collegeFund`, `anAccount`, et `anObject` font référence au même objet de type `SavingsAccount`

# Conversion entre les types de la sous classe et de la classe mère /2



## Conversion entre les types de la sous classe et de la classe mère /3

---

- Les références de la classe mère “ne connaissent pas toute l’histoire” :  
`anAccount.deposit(1000); // OK`  
`anAccount.addInterest();`  
`// Non ce n’est pas une méthode définie dans la classe`  
`dont anAccount est de type`
- Quand vous effectuez une conversion entre le type d’une sous classe et d’une classe mère :
  - *La valeur de la référence reste la même ! (en quelque sorte la position de l’objet dans la mémoire) t*
  - *Mais, moins d’information est connu sur cet objet*

## Conversion entre les types de la sous classe et de la classe mère /4

---

- Pourquoi souhaiterions connaître *moins* de chose d'un objet ?
  - *Pour réutiliser du code qui ne connaît que la classe mère :*

```
public void transfer(double amount, BankAccount other)  
  
{  
    withdraw(amount);  
    other.deposit(amount);  
}
```

*Peut être utilisé pour transférer de l'argent entre n'importe quel type de compte*

## Conversion entre les types de la sous classe et de la classe mère /5

---

- Parfois, on doit convertir une référence dont le type est la classe mère en une référence dont le type est la classe fille

```
BankAccount anAccount = (BankAccount) anObject;
```

- Ce transtypage (cast) est dangereux. Une erreur peut être levée!
- Solution : utiliser l'opérateur `instanceof`

- `instanceof`: test si l'objet appartient bien à un type particulier

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    . . .
}
```

## Syntaxe Opérateur instanceof

*object* instanceof *TypeName*

### Example :

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    . . .
}
```

### Objectif :

Retourner `true` si l'objet est une instance du type *TypeName* (ou de l'un de ses sous-types), retourne `false` dans tous les autres cas.

## Questions

---

Pourquoi le second paramètre de la méthode `transfer` doit être du type `BankAccount` et non pas par exemple, `SavingsAccount`?

**Réponse :** On souhaite utiliser cette méthode pour tous les types de comptes bancaires.



## Questions

---

Pourquoi ne pas changer dans ce cas le type du second paramètre de la méthode `transfer` en le type `Object`?

**Réponse :** On ne peut appeler la méthode `deposit` sur une variable de type `Object`.

## Polymorphisme

---

- En Java, le type d'une variable ne détermine pas complètement le type de l'objet à lequel cette variable fait référence

```
BankAccount aBankAccount = new SavingsAccount(1000);  
    // aBankAccount holds a reference to a SavingsAccount
```

- Les appels de méthodes sont déterminés par le type réel de l'objet (type dynamique) et non pas le type de la référence (type statique)

```
BankAccount anAccount = new CheckingAccount();  
anAccount.deposit(1000);  
    // Calls "deposit" from CheckingAccount
```

- Le compilateur doit vérifier quelles sont les méthodes autorisées à être appelées

```
Object anObject = new BankAccount();  
anObject.deposit(1000); // Faux!
```

## Polymorphisme

---

- Polymorphisme : aptitude à référencer un plusieurs types d'objets dont le comportement peut varier
- Polymorphisme :

```
public void transfer(double amount, BankAccount other)
{
    withdraw(amount);
    // Racourci pour this.withdraw(amount);
    other.deposit(amount);
}
```
- En fonction des types de `amount` et `other`, différentes versions des méthodes `withdraw` et `deposit` sont appelées

## ch10/accounts/AccountTester.java

```
01: /**
02:     This program tests the BankAccount class and
03:     its subclasses.
04: */
05: public class AccountTester
06: {
07:     public static void main(String[] args)
08:     {
09:         SavingsAccount momsSavings
10:             = new SavingsAccount(0.5);
11:
12:         CheckingAccount harrysChecking
13:             = new CheckingAccount(100);
14:
15:         momsSavings.deposit(10000);
16:
17:         momsSavings.transfer(2000, harrysChecking);
18:         harrysChecking.withdraw(1500);
19:         harrysChecking.withdraw(80);
20:
```

## ch10/accounts/AccountTester.java /2

```
21:     momsSavings.transfer(1000, harrysChecking);
22:     harrysChecking.withdraw(400);
23:
24:     // Simulate end of month
25:     momsSavings.addInterest();
26:     harrysChecking.deductFees();
27:
28:     System.out.println("Mom's savings balance: "
29:         + momsSavings.getBalance());
30:     System.out.println("Expected: 7035");
31:
32:     System.out.println("Harry's checking balance: "
33:         + harrysChecking.getBalance());
34:     System.out.println("Expected: 1116");
35: }
36: }
```

## ch10/accounts/CheckingAccount.java

```
01: /**
02:     A checking account that charges transaction fees.
03: */
04: public class CheckingAccount extends BankAccount
05: {
06:     /**
07:         Constructs a checking account with a given balance.
08:         @param initialBalance the initial balance
09:     */
10:     public CheckingAccount(double initialBalance)
11:     {
12:         // Construct superclass
13:         super(initialBalance);
14:
15:         // Initialize transaction count
16:         transactionCount = 0;
17:     }
18:
19:     public void deposit(double amount)
20:     {
21:         transactionCount++;
```

## ch10/accounts/CheckingAccount.java /2

```
22:         // Now add amount to balance
23:         super.deposit(amount);
24:     }
25:
26:     public void withdraw(double amount)
27:     {
28:         transactionCount++;
29:         // Now subtract amount from balance
30:         super.withdraw(amount);
31:     }
32:
33:     /**
34:      * Deducts the accumulated fees and resets the
35:      * transaction count.
36:      */
37:     public void deductFees()
38:     {
39:         if (transactionCount > FREE_TRANSACTIONS)
40:         {
41:             double fees = TRANSACTION_FEE *
42:                 (transactionCount - FREE_TRANSACTIONS);
43:             super.withdraw(fees);
44:         }
```

## ch10/accounts/CheckingAccount.java /3

---

```
45:         transactionCount = 0;
46:     }
47:
48:     private int transactionCount;
49:
50:     private static final int FREE_TRANSACTIONS = 3;
51:     private static final double TRANSACTION_FEE = 2.0;
52: }
```



## ch10/accounts/BankAccount.java

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance.
09:     */
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
14:
15:     /**
16:         Constructs a bank account with a given balance.
17:         @param initialBalance the initial balance
18:     */
19:     public BankAccount(double initialBalance)
20:     {
21:         balance = initialBalance;
22:     }
23:
```

## ch10/accounts/BankAccount.java /2

```
24:     /**
25:         Deposits money into the bank account.
26:         @param amount the amount to deposit
27:     */
28:     public void deposit(double amount)
29:     {
30:         balance = balance + amount;
31:     }
32:
33:     /**
34:         Withdraws money from the bank account.
35:         @param amount the amount to withdraw
36:     */
37:     public void withdraw(double amount)
38:     {
39:         balance = balance - amount;
40:     }
41:
42:     /**
43:         Gets the current balance of the bank account.
44:         @return the current balance
45:     */
```

## ch10/accounts/BankAccount.java /3

```
46:     public double getBalance()
47:     {
48:         return balance;
49:     }
50:
51:     /**
52:      * Transfers money from the bank account to another account
53:      * @param amount the amount to transfer
54:      * @param other the other account
55:      */
56:     public void transfer(double amount, BankAccount other)
57:     {
58:         withdraw(amount);
59:         other.deposit(amount);
60:     }
61:
62:     private double balance;
63: }
```

## ch10/accounts/SavingsAccount.java

```
01: /**
02:     An account that earns interest at a fixed rate.
03: */
04: public class SavingsAccount extends BankAccount
05: {
06:     /**
07:         Constructs a bank account with a given interest rate.
08:         @param rate the interest rate
09:     */
10:     public SavingsAccount(double rate)
11:     {
12:         interestRate = rate;
13:     }
14:
15:     /**
16:         Adds the earned interest to the account balance.
17:     */
```

## ch10/accounts/SavingsAccount.java /2

---

```
18:     public void addInterest()  
19:     {  
20:         double interest = getBalance() * interestRate / 100;  
21:         deposit(interest);  
22:     }  
23:  
24:     private double interestRate;  
25: }
```

### Output:

Mom's savings balance: 7035.0

Expected: 7035

Harry's checking balance: 1116.0

Expected: 1116

## Questions

---

Si `a` est une variable du type `BankAccount` qui contient une référence non-`null`, que sait-on à propos de l'objet auquel cette variable réfère ?

**Réponse :** L'objet est une instance de la classe `BankAccount` ou d'une de ses sous classes.

## Questions

---

Si `a` référence un compte courant, quel est l'effet de l'appel de méthode `a.transfer(1000, a)`?

**Réponse :** Le solde du compte (balance) n'est pas changé et le compteur de transaction est incrémenté deux fois.

## Contrôle d'accès / Encapsulation

---

- Java possède 4 niveaux de contrôle d'accès aux variables, méthodes et classes :
  - *public*
    - Peut être accédé par toutes les méthodes de toutes les classes
  - *private*
    - Peut être accédé par les méthodes de la même classe
  - *protected*
    - Peut être accédé par toutes les méthodes du même paquetage et par toutes les méthodes d'une sous-classe
  - *package*
    - Par défaut, quand aucun mot clé n'est précisé
    - Peut être accédé par toutes les méthodes du même paquetage
    - Bon point pour les classes par défaut, mais dommageable pour les variables d'instance



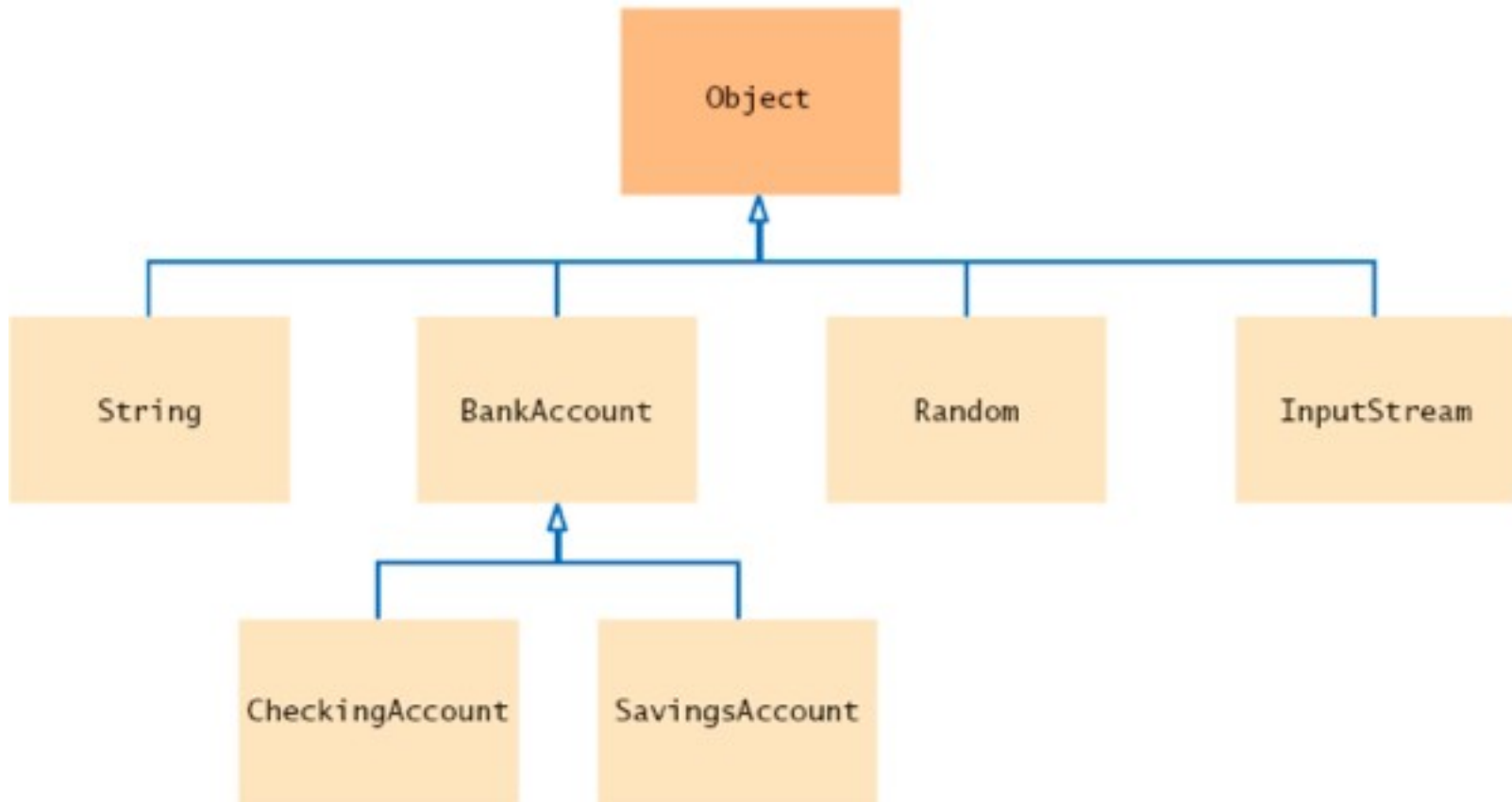
## Niveaux d'accès recommandés

---

- Variables d'instance et de classes (`static`) : Toujours privé. Exceptions :
  - *`public static final` pour les constantes (utiles et non dangereux)*
  - *Certains objets, tels que `System.out`, doivent être accessible à tous les programmes (`public`)*
  - *Parfois, les classes d'un paquetage doivent collaborer de manière très étroite (donner l'accès package aux variables concernées) ; classes internes sont généralement recommandées dans ce cas*
- Méthodes : `public` ou `private`
- Classes and interfaces: `public` ou `package`
  - Meilleure alternative à l'accès package : les classes internes
    - En générale, les classes internes ne sont pas `public` (des exceptions existent ex `Ellipse2D.Double`)
- Attention aux accès par défaut !!!!

## La super classe Object

- Toutes les classes définies dans une clause explicite `extends` dérive automatiquement de la classe `Object`



## La super classe Object /2

---

- Toutes les classes définies dans une clause explicite `extends` dérive automatiquement de la classe `Object`
- Les méthodes les plus communes de cette classes sont :
  - *String toString()*
  - *boolean equals(Object otherObject)*
  - *Object clone()*
- C'est une bonne idée de redéfinir ces méthodes dans vos propres classes

## Redéfinition de la méthode `toString`

---

- Retourne une représentation sous forme d'une chaîne de caractères de l'objet
- Pratique pour "debugger" :

```
Rectangle box = new Rectangle(5, 10, 20, 30);  
String s = box.toString();  
// Sets s to java.awt.Rectangle[x=5,y=10,width=20,  
    height=30]"
```
- `toString` est appelée quand vous concatérez une chaîne avec une référence à un objet :

```
"box=" + box;  
// Result: "box=java.awt.Rectangle[x=5,y=10,width=20,  
    height=30]"
```

## Redéfinition de la méthode `toString` /2

---

- `Object.toString` affiche le nom de la classe et le code de “hachage de l’objet”

```
BankAccount momsSavings = new BankAccount(5000);  
String s = momsSavings.toString();  
// Sets s to something like "BankAccount@d24606bf"
```

## Redéfinition de la méthode toString /3

---

- Pour fournir une version plus détaillée ou plus lisible, il suffit de redéfinir la méthode toString:

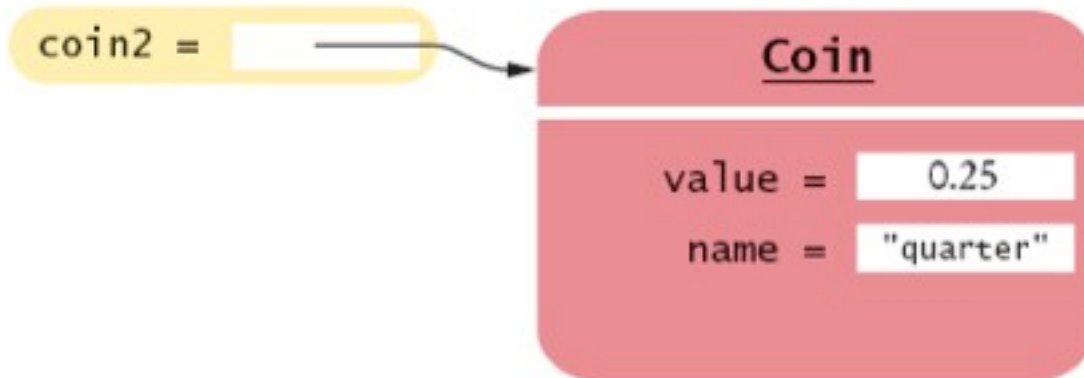
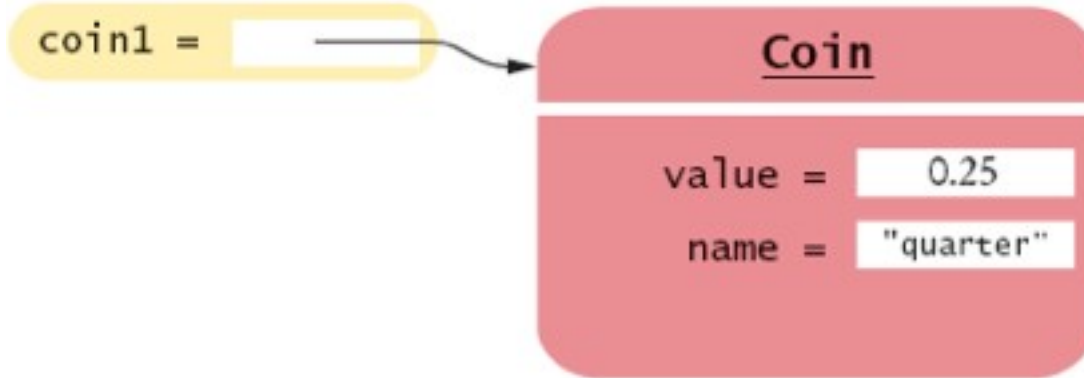
```
public String toString()  
{  
    return "BankAccount[balance=" + balance + "];"  
}
```

- Ce qui donne :

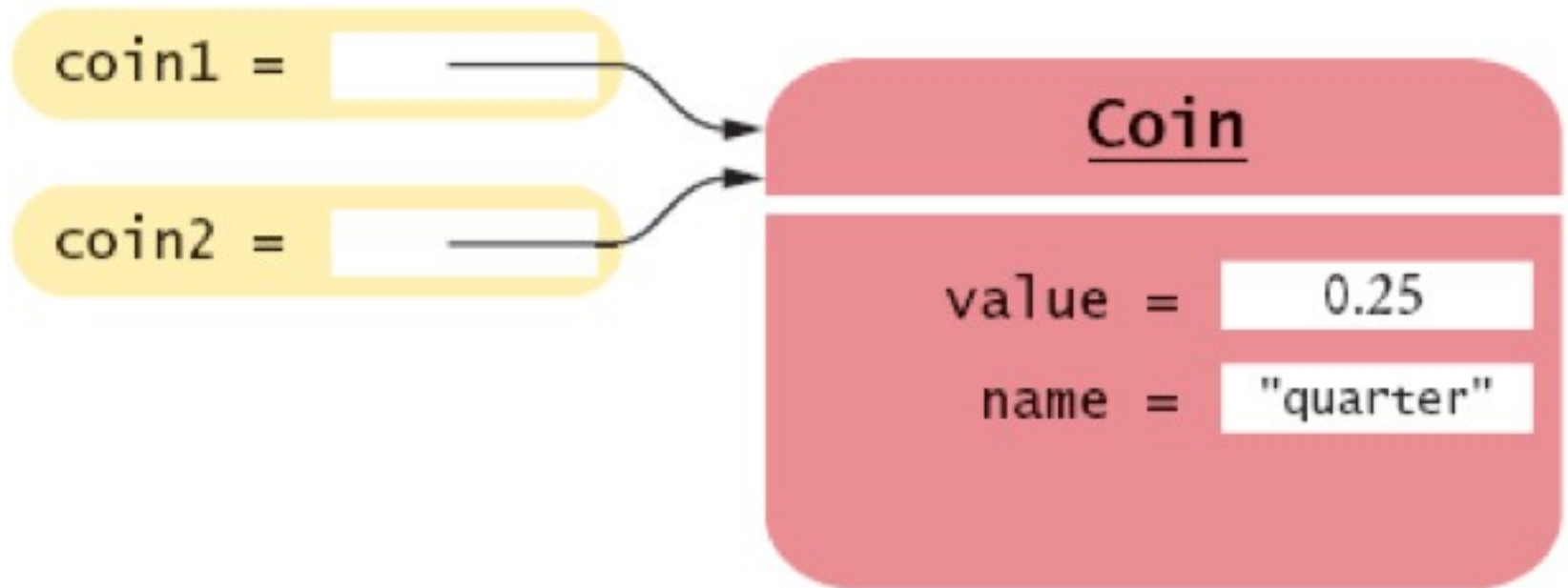
```
BankAccount momsSavings = new BankAccount(5000);  
String s = momsSavings.toString();  
// Sets s to "BankAccount[balance=5000]"
```

## Redéfinition de la méthode equals

- Equals teste l'égalité de contenu



## Redéfinition de la méthode equals /2





## Redéfinition de la méthode `equals` /3

---

- Définir la méthode `equals` pour tester si deux objets sont égaux (au sens état)
- Lorsque l'on redéfinit la méthode `equals`, on ne peut pas changer la signature de la méthode (type du paramètre); utiliser le transtypage (*cast*) :

```
public class Coin
{
    . . .
    public boolean equals(Object otherObject)
    {
        Coin other = (Coin) otherObject;
        return name.equals(other.name) && value ==
            other.value;
    }
    . . .
}
```

## Redéfinition de la méthode `equals` /4

---

- On doit également redéfinir la méthode `hashCode` pour que deux objets égaux (au sense de la méthode `equal`) retourne le même code de hachage

## Questions

---

L'appel `x.equals(x)` doit-il toujours retourner `true`?

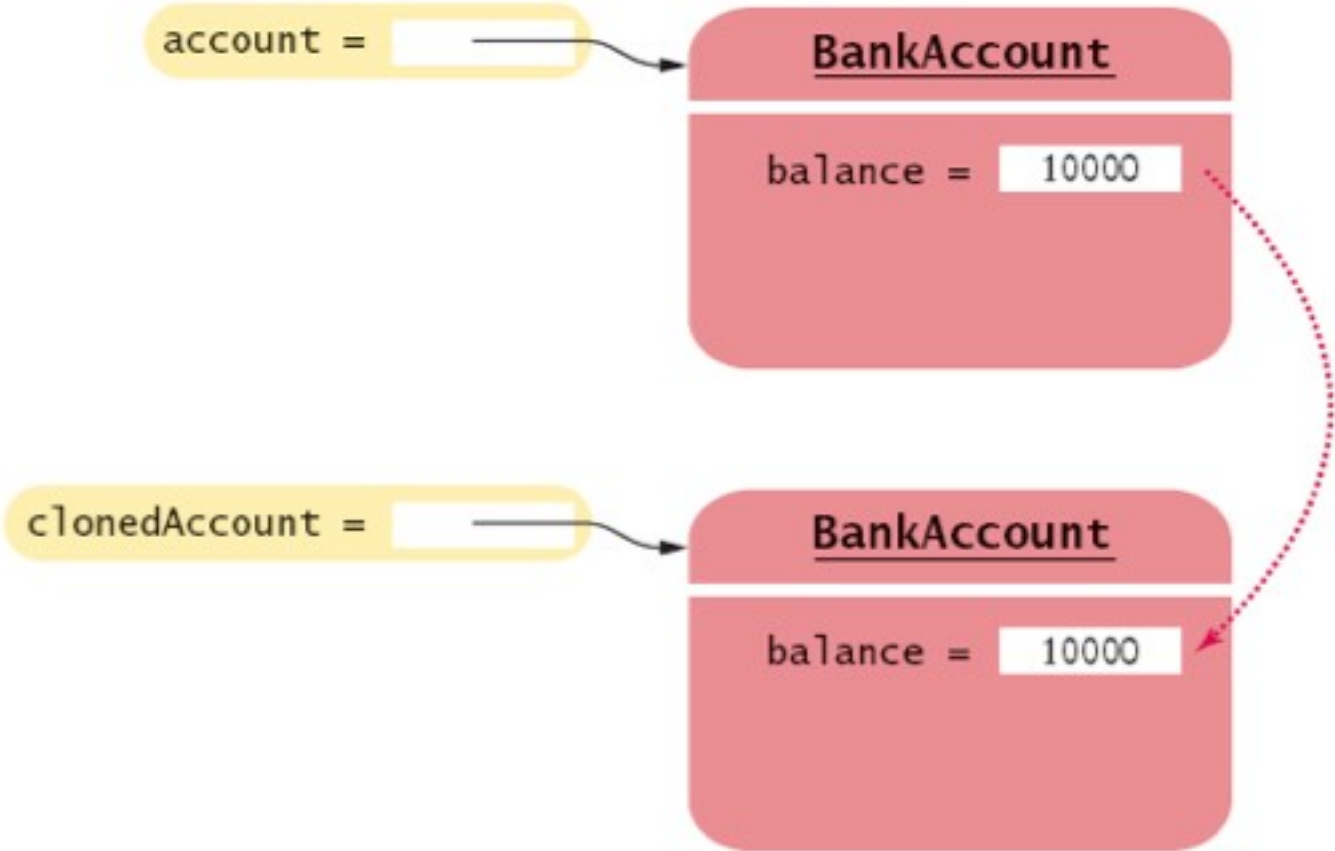
**Réponse :** Certainement, il devrait. Sauf si bien sûr `x` est `null`.

## Redéfinition de la méthode clone

---

- Copier une référence, donne deux références vers le même objet  
`BankAccount account2 = account;`
- Parfois, on souhaite copier l'objet

# Redéfinition de la méthode clone /2



## Redéfinition de la méthode `clone` /3

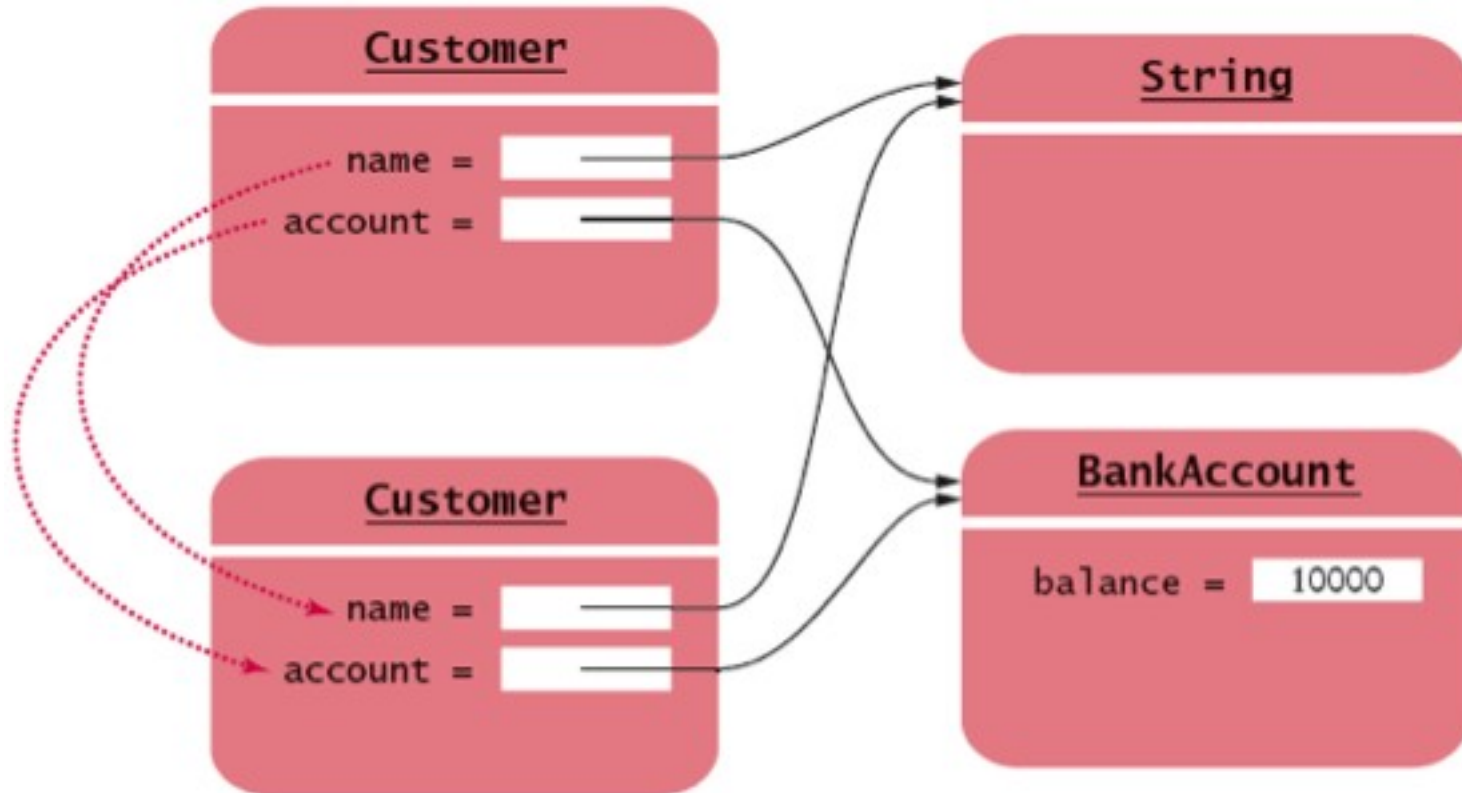
---

- Définir la méthode `clone` pour créer un nouvel objet
- Utilisation de la méthode `clone` :  

```
BankAccount clonedAccount =  
    (BankAccount)account.clone();
```
- Obligation de transtyper le résultat car le type de retour de la méthode `clone` est `Object`

## La méthode `Object.clone`

- Ne réalise pas une copie en profondeur



## La méthode `Object.clone` /2

---

- Ne clone pas systématiquement les objets sous-jacents
- Doit être utilisé avec attention
- Elle est déclarée comme `protected` pour prévenir les appels accidentels à `x.clone()` si la classe à laquelle `x` appartient ne redéfinit pas `clone` pour être `public`
- Vous devez redéfinir la méthode `clone` avec prudence