TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

IMT Grand Est

UNIVERSITÉ DE LORRAINE

IA – 2019 / 2020

Programmation Orientée Objet

Cours 3

Gérald Oster <gerald.oster@telemcomnancy.eu>

Supports inspirés et traduits en partie de C. Horstmann

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

9ème 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 (`SavingsAccount`)
- Hériter d'une classe ≠ 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 redéfinie (dont on change le comportement)

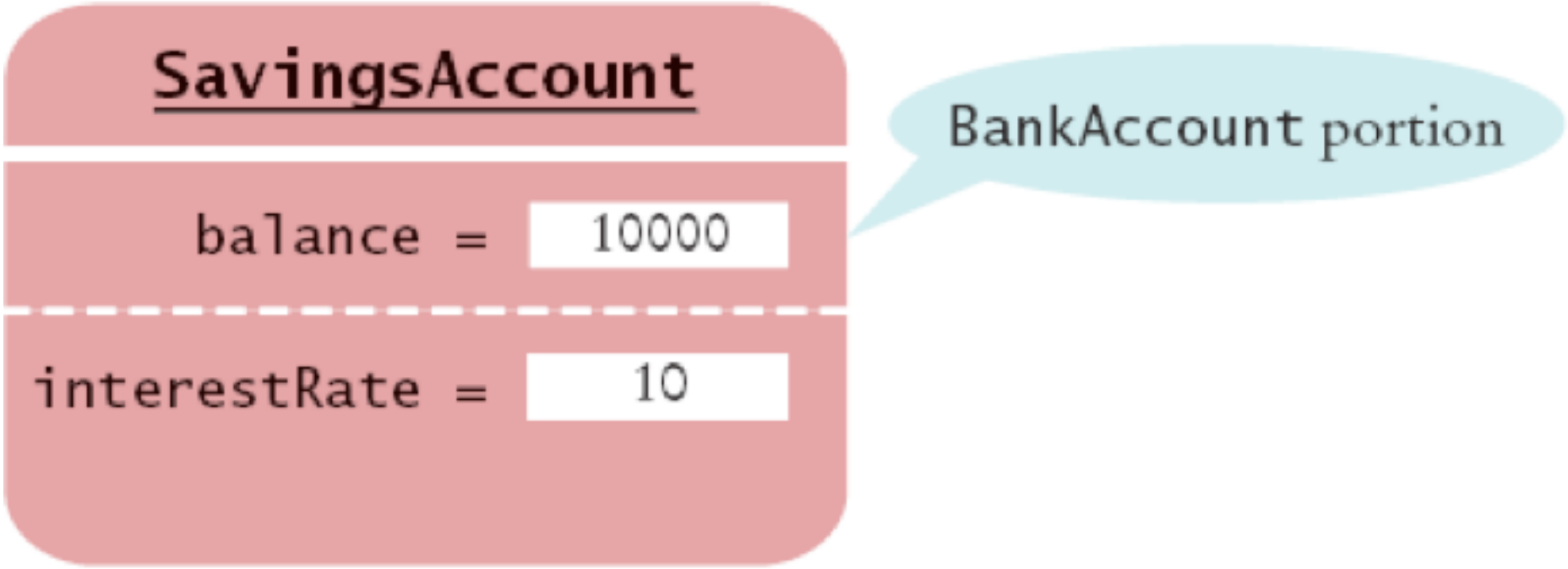
```
public class SavingsAccount extends BankAccount
 {
    public SavingsAccount(double rate)
    {
       interestRate = rate;
    }

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

    private double interestRate;
 }
```

Introduction à l'héritage /4

- 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`:



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` ?

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`.

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 ?

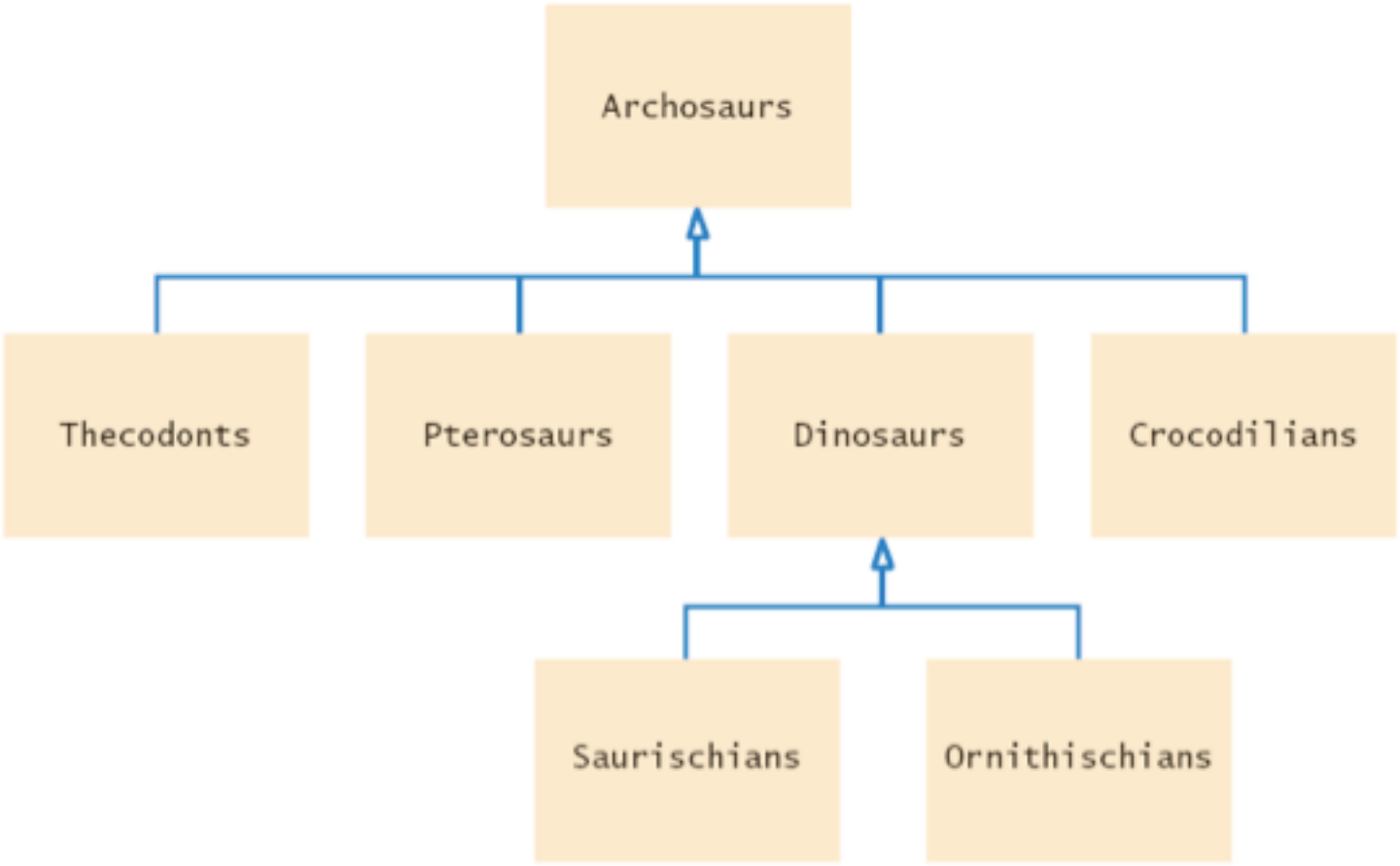
Questions

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).

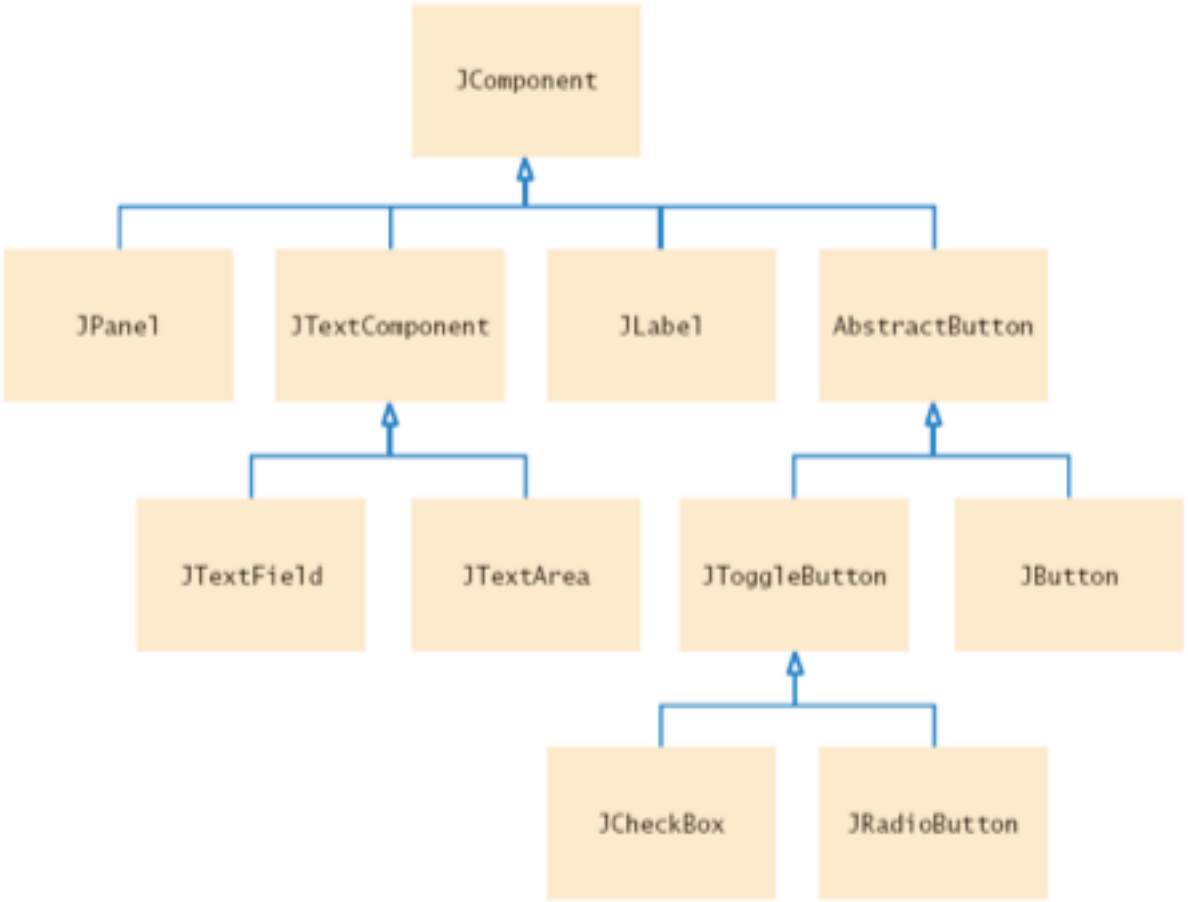
Hiérarchie de classes

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



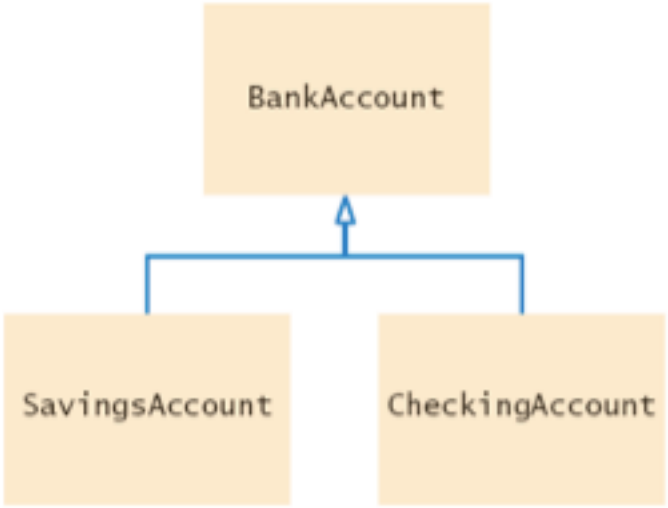
Hiérarchie de classes /2

- La classe mère `JComponent` possède les méthodes `getWidth`, `getHeight`
- La classe `AbstractButton` fournie 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 l'implémentation différe
- Compte courant requiert une méthode `deductFees`; Compte d'éparne requiert une méthode `addInterest`

Questions

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

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 qui permettent de rentrer du texte

Questions

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

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

- Redéfinition 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 rédéfinir 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`

- Redéfinir 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);
}
```

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;
}
```

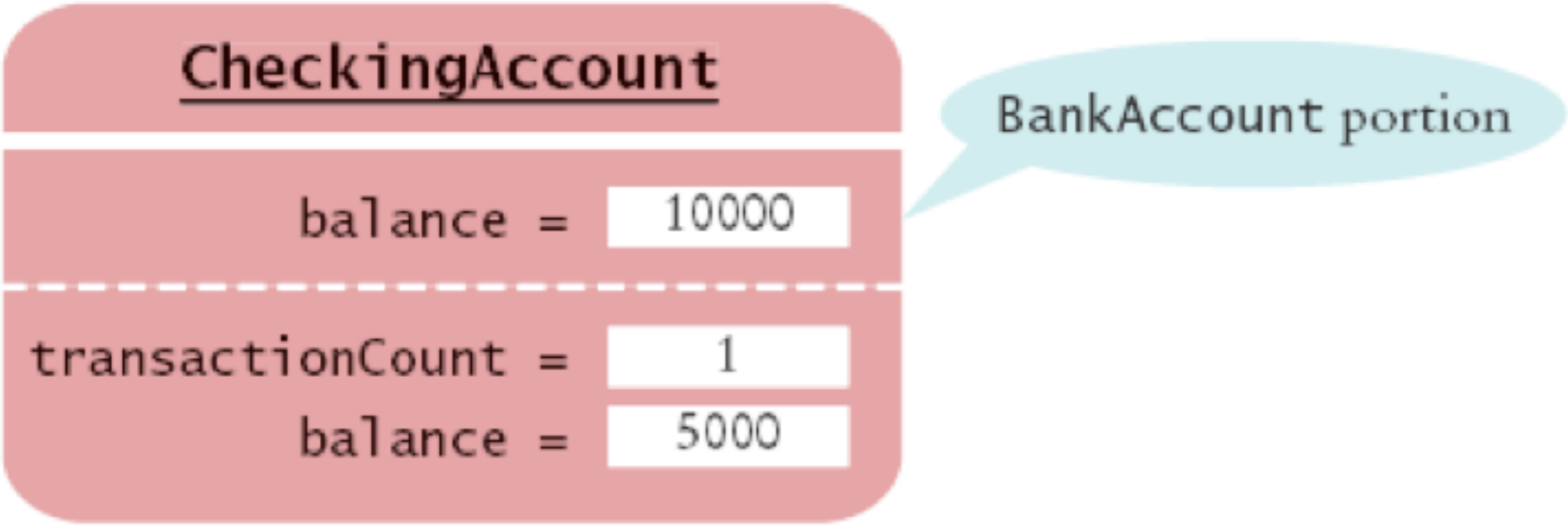
Erreur classique : Masquer une variable d'instance

- 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!



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 ?

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.

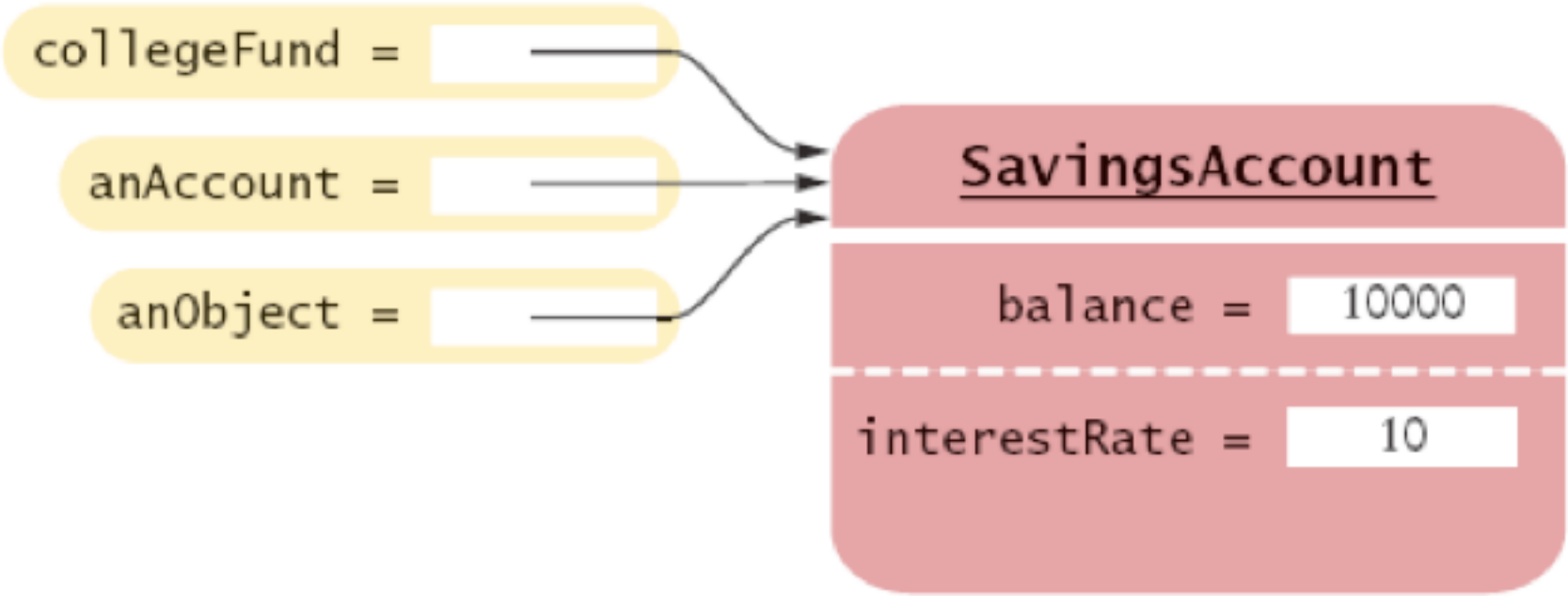
Conversion entre les types de la sous classe et de la classe mère

- 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 connait 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`?

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`?

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 complétement 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 référe ?

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 référe ?

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)`?

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

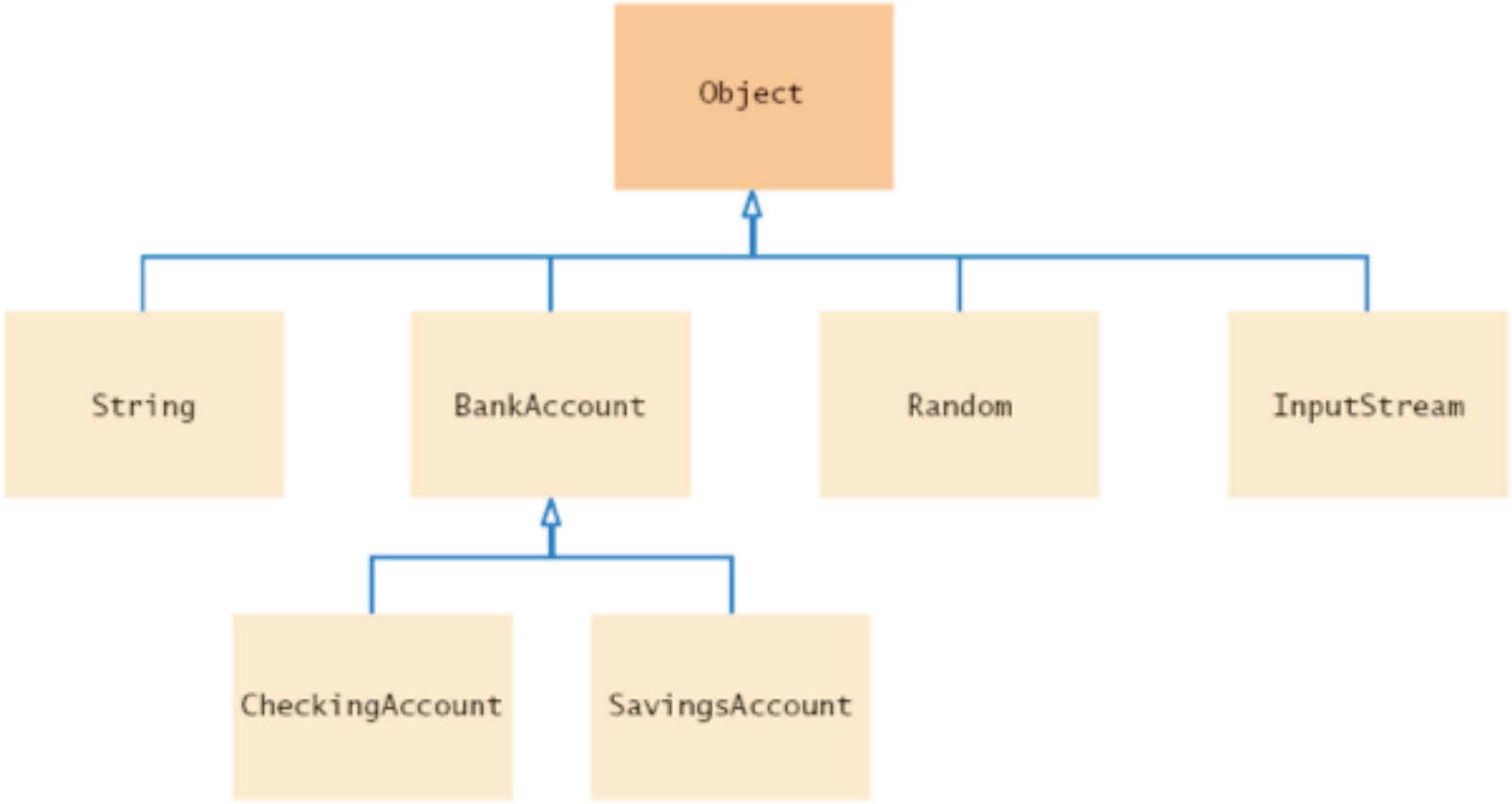
- Java possède 4 niveaux de contrôle d'accès aux variables, méthodes et classes :
  - `public`
    - *Peut être accédé par toutes les méthodes de toutes les classes*
  - `private`
    - *Peut être accédé par les méthodes de la même classe*
  - `protected`
    - *Peut être accédé par toutes les méthodes 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 dérivent automatiquement de la classe `Object`



La super classe Object /2

- Toutes les classes dérivent 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 concaténez 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`:

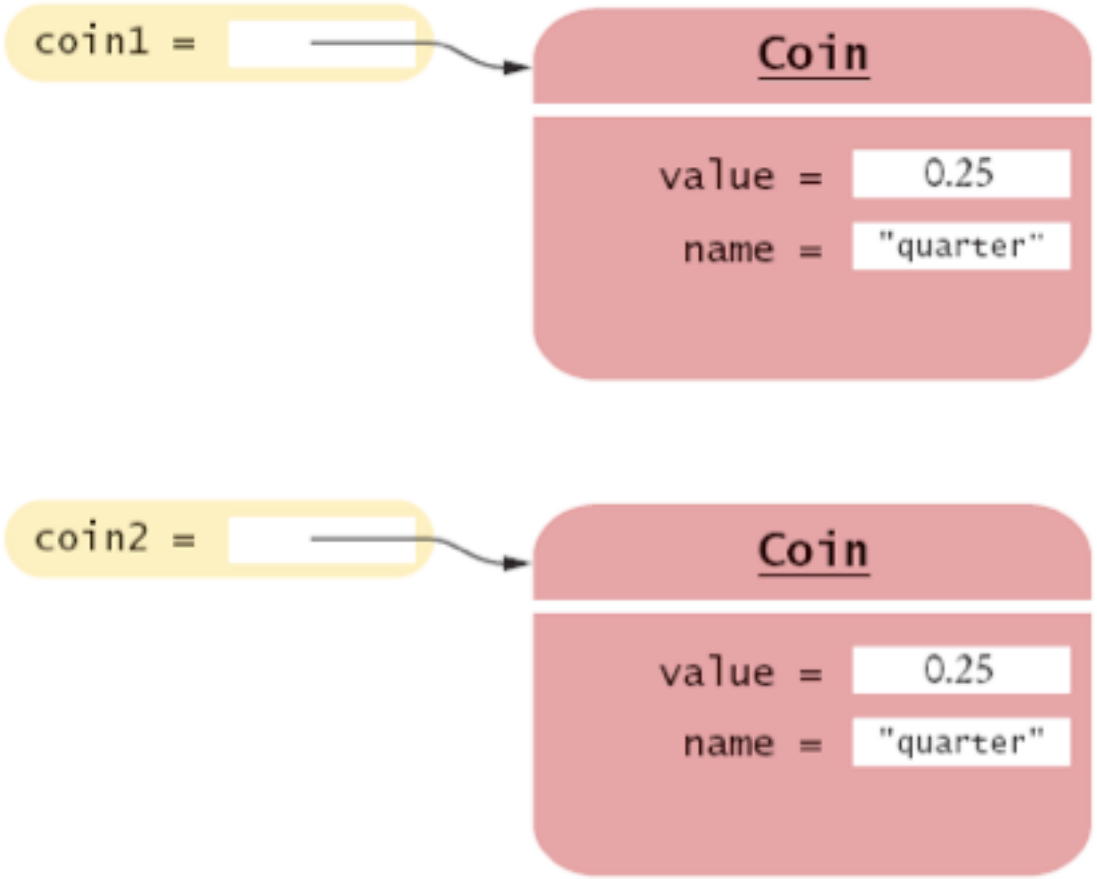
```
public String toString()
{
   return "BankAccount[balance=" + balance + "]";
}
```

- Ce qui donne :

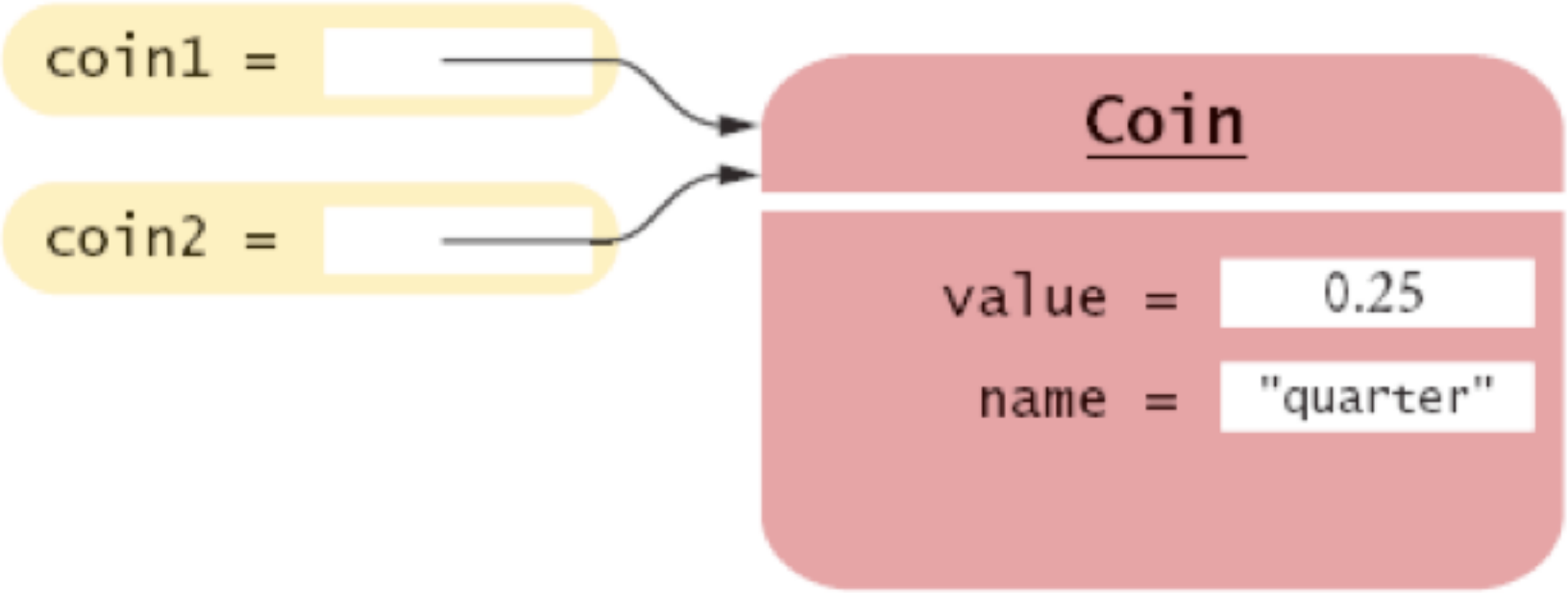
```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```

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 sens de la méthode `equal`) retourne le même code de hachage

Questions

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

Questions

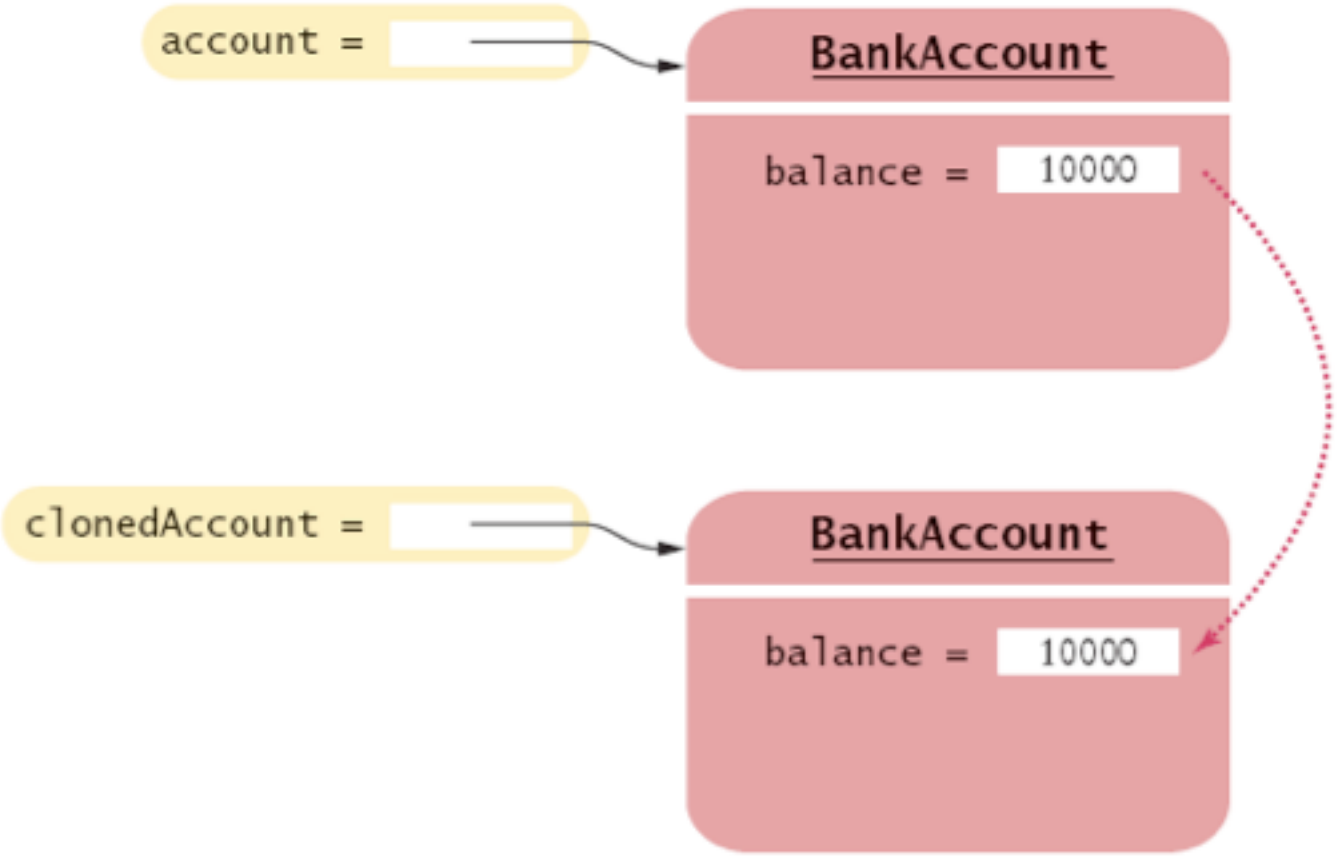
L’appel `x.equals(x)` doit-il toujours retourner `true`?

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

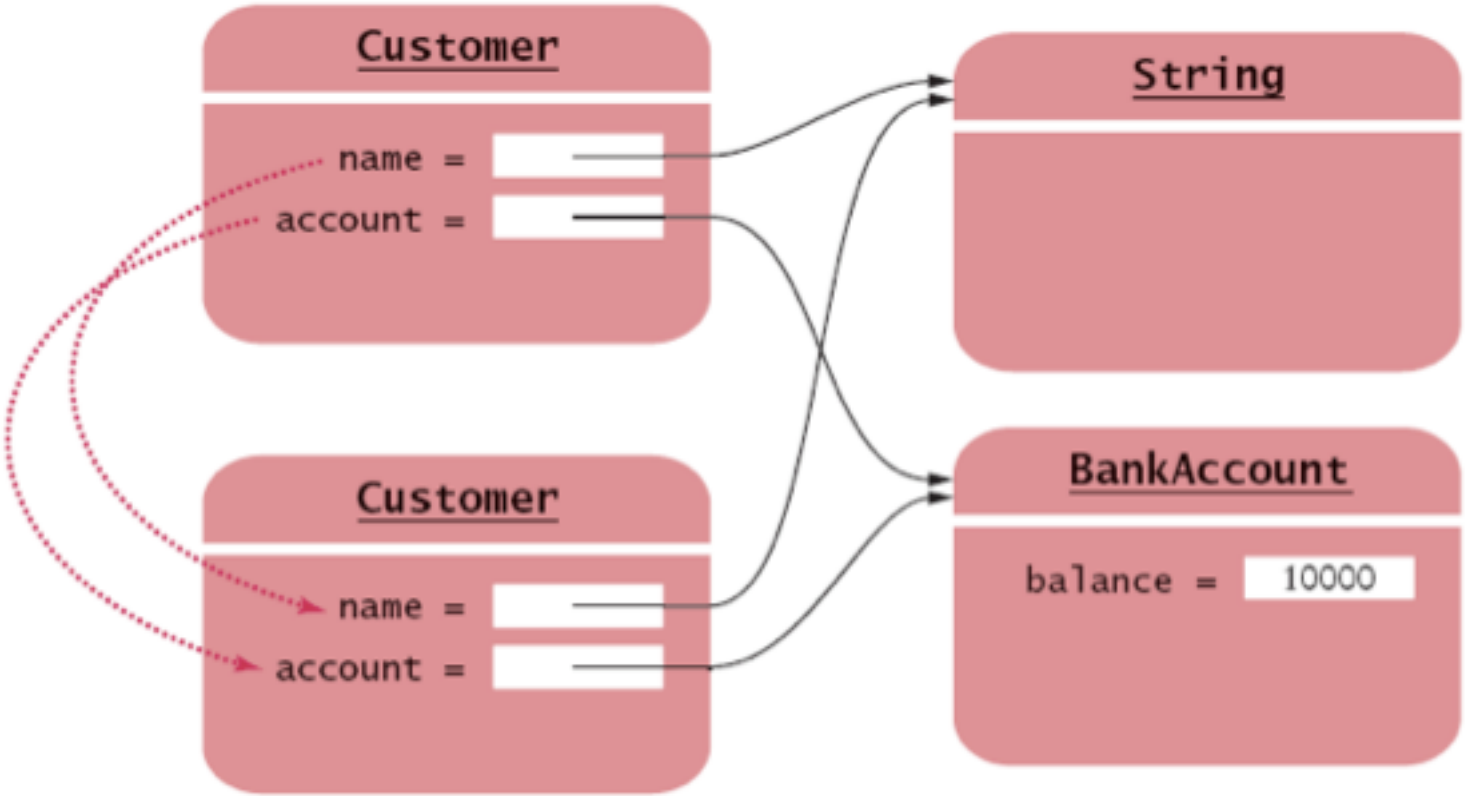
- Définir la méthode `clone` pour créer un nouvel objet
- Utilisation de la méthode `clone`:

```
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 redéfinie pas `clone` pour être `public`
- Vous devez redéfinir la méthode `clone` avec prudence