Programmation Objet

Florian Boudin

Révision 3 du 12 novembre 2014

Classe 4 : *Interfaces et héritage*

Plan

- Interfaces
- Héritage
- Polymorphisme
- `super` et héritage de `Object`
- Méthodes et classes finales/abstraites

Introduction

- Exemple : dans une société futuriste ou les voitures sont contrôlées par des ordinateurs
  - Un constructeur automobile écrit le programme Java qui fait fonctionner la voiture : arrêt, marche, accélérer, etc.
  - Une société de GPS veut utiliser l'information sur le trafic en temps réel pour conduire la voiture → le constructeur automobile doit publier une interface
  - Diffuser une interface permet de garder le code source propriétaire
- Des groupes de développeurs distants peuvent accepter un *contrat* qui précise comment leurs logiciels interagissent

Interfaces en Java

- Une interface définit un type de référence
- Une interface ne spécifie qu'une API : toutes ses méthodes sont abstraites et n'ont pas de corps
- Il n'est pas possible d'instancier une interface
- Le mot-clé *interface* est utilisé pour définir une interface
- Une interface peut être étendue avec le mot-clé *extends*
- Le mot-clé *implements* est utilisé lorsqu'une classe implémente une interface

Définir une interface

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

  // constant declarations
  // base of natural logarithms
  double E = 2.718282;

  // method signatures
  void doSomething(int i, double x);
  int doSomethingElse(String s);

}
```

- *public* indique que l'interface peut être utilisée par toutes les classes du *package*
- L'interface hérite des méthodes et des constantes des super-interfaces

Exemple d'interface : *Relatable*

```
public interface Relatable {

  // this (object calling isLargerThan)
  // and other must be instances of
  // the same class
  // returns 1, 0, -1 if this is greater
  // than, equal to, or less than other
  public int isLargerThan(Relatable other);

}
```

- Une classe peut implémenter Relatable si l'on peut comparer la taille des objets instanciés : strings → nombre de caractères; livres → nombre de pages; etc.
- Si vous savez qu'une classe implémente *Relatable*, vous savez que vous pouvez comparer la taille des objets instanciés

Rectangle implémentant *Relatable*

```
public class RectanglePlus implements Relatable {
  // ...
  public int isLargerThan(Relatable other) {
    RectanglePlus otherRect = (RectanglePlus) other;
    if (this.getArea() < otherRect.getArea())
      return -1;
    else if (this.getArea() > otherRect.getArea())
      return 1;
    else
      return 0;
  }
}
```

- La méthode `isLargerThan` prend un objet de type `Relatable`, `(RectanglePlus) other` convertit `other` en type `RectanglePlus`

Etendre une interface

- Considérons une interface que vous avez créée :

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
}
```

- Supposons que, plus tard, vous voulez ajouter une troisième méthode de sorte que l'interface devient

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
  boolean didItWork(int i, double x, String s);
}
```

- Si vous apportez cette modification, toutes les classes qui implémentent l'ancienne interface `DoIt` ne compilerons plus

Etendre une interface (cont.)

- Essayer d'anticiper les utilisations de l'interface et de les spécifier le plus complètement possible dès le début
- Considérant que cela est souvent impossible, vous serez amener à créer ou étendre de nouvelles interfaces
- Le mot-clé *extends* permet d'étendre une interface

```
public interface DoItPlus extends DoIt {
  boolean didItWork(int i, double x, String s);
}
```

- Les utilisateurs de ce code peuvent choisir de continuer à utiliser l'ancienne interface ou de mettre à jour pour utiliser la nouvelle

Questions

1. Quelles sont les méthodes qu'une classe implémentant l'interface `java.lang.CharSequence` devrait implémenter ?
2. Considérons l'interface suivante :

```
public interface SomethingIsWrong {
  void aMethod(int aValue) {
    System.out.println("Hi Mom");
  }
}
```

   1. Quelle est l'erreur dans l'interface ?
   2. Corrigez l'interface

Réponses

1. Quelles sont les méthodes qu'une classe implémentant l'interface `java.lang.CharSequence` devrait implémenter ?
   - Avec la javadoc : `charAt`, `length`, `subSequence`, `toString`
2. Considérons l'interface suivante :
   1. Quelle est l'erreur dans l'interface ? Il y a une implémentation de méthode
   2. Corrigez l'interface

```
public interface SomethingIsWrong {
  void aMethod(int aValue);
}
```

Plan

- Interfaces
- Héritage
- Polymorphisme
- `super` et héritage de `Object`
- Méthodes et classes finales/abstraites

Introduction

- L'idée d'héritage est simple mais puissante :
  - lorsque vous souhaitez créer une nouvelle classe et qu'une classe comprenant une partie du code que vous voulez existe, vous pouvez dériver votre nouvelle classe de la classe existante
- Une classe dérivée d'une autre classe est appelée *sous-classe*
- Réciproquement, une classe à partir de laquelle une sous-classe est dérivée est appelée *super-classe* (classe parent)

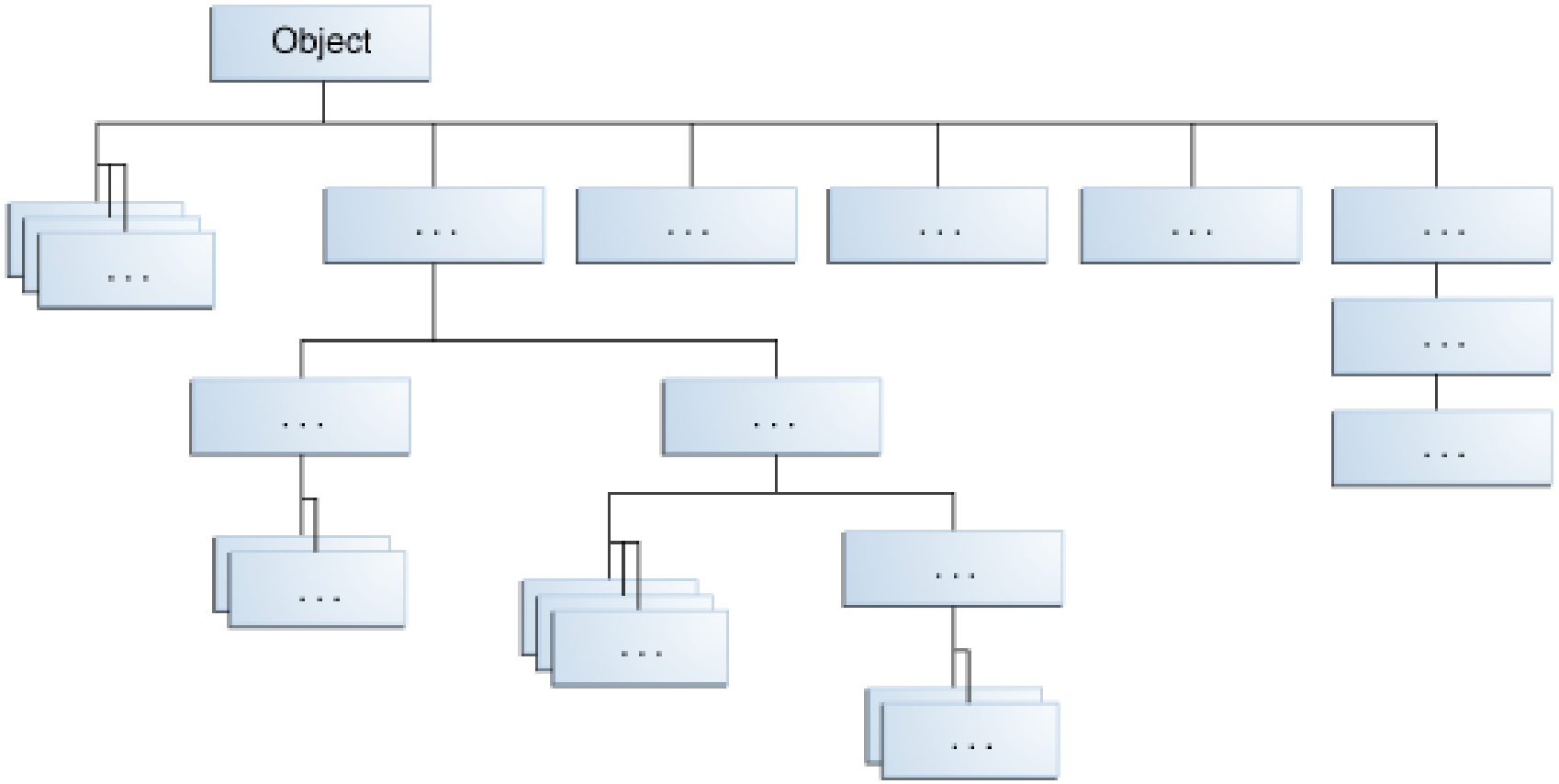
Introduction (cont.)

- Excepté la classe `Object`, qui n'a pas de super-classe, chaque classe a une et une seule super-classe directe (héritage simple)
- En l'absence de super-classe explicite, chaque classe est implicitement une sous-classe de `Object`
- Les classes peuvent être dérivées de classes qui sont issues de classes qui sont issues de classes, etc.
- Une telle classe est un descendant de toutes les classes de la hiérarchie jusqu'à la classe `Object`

Introduction (cont.)

- Une sous-classe hérite de tous les membres (champs, méthodes et classes imbriquées) de sa super-classe
- Les constructeurs ne sont pas des membres, et ne sont pas hérités mais un constructeur de la super-classe peut être invoqué de la sous-classe

Hiérarchie des classes en Java



La classe `Bicycle`

```
public class Bicycle {

  public int cadence;
  public int gear;
  public int speed;

  public Bicycle(int cadence, int speed, int gear) {
    this.gear = gear;
    this.cadence = cadence;
    this.speed = speed;
  }

  public void setCadence(int newValue) {
    cadence = newValue;
  }

  public void setGear(int newValue) {
    gear = newValue;
  }

  //...
}
```

La sous-classe `MountainBike`,

```
public class MountainBike extends Bicycle {

  // the MountainBike subclass adds one field
  public int seatHeight;

  // the MountainBike subclass has one constructor
  public MountainBike(int startHeight, int startCadence, int startSpeed,
                      int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
  }

  // the MountainBike subclass adds one method
  public void setHeight(int newValue) {
    seatHeight = newValue;
  }

}
```

Ce que vous pouvez faire dans une sous-classe

- Les champs (méthodes) hérités peuvent être utilisés directement
- Vous pouvez déclarer un champ (méthode) dans la sous-classe ayant le même nom qu'un champ (méthode) déclaré dans la super-classe, il devient alors masqué (redéfinie)
- Vous pouvez déclarer de nouveaux champs (méthodes)

Ce que vous pouvez faire dans une sous-classe (cont.)

- Le constructeur de la sous-classe peut invoquer le constructeur de la super-classe implicitement ou en utilisant le mot-clé *super*
- Une sous-classe n'hérite pas des membres privés de la classe parent, des méthodes d'accès (`public` ou `protected`) peuvent cependant être utilisées pour accéder aux champs privés

Conversion (*casting*) des objets

- Créer une instance de `MountainBike`

```
public MountainBike myBike = new MountainBike();
```

- `MountainBike` est une sous-classe de `Bicycle` et de `Object`, donc un objet `MountainBike` est aussi un `Bicycle` et `Object`

```
Object obj = new MountainBike();
// obj est un Object ET un MountainBike
// casting implicite

MountainBike myBike = obj;
// Erreur de compilation

MountainBike myBike = (MountainBike)obj;
```

Redéfinition de méthodes

- Une méthode d'instance dans une sous-classe avec la même signature (nom, nombre et type des paramètres) et valeur de retour qu'une méthode d'instance de la super-classe redéfinit (*overrides*) la méthode de la super-classe
- L'annotation de documentation `@Override` peut être utilisée pour indiquer au compilateur que la méthode est surchargée

Masquage de méthodes

- Une méthode de classe dans une sous-classe avec la même signature et valeur de retour qu'une méthode de classe de la super-classe masque la méthode de la super-classe
- Considérons la classe suivante :

```
public class Animal {
  public static void testClassMethod() {
    System.out.println("The class method in Animal");
  }

  public void testInstanceMethod() {
    System.out.println("The instance method in Animal");
  }
}
```

Masquage de méthodes (cont.)

```
public class Cat extends Animal {

  public static void testClassMethod() {
    System.out.println("The class method in Cat");
  }

  public void testInstanceMethod() {
    System.out.println("The instance method in Cat");
  }

  public static void main(String[] args) {
    Cat myCat = new Cat();
    Animal myAnimal = myCat;
    Animal.testClassMethod();
    myAnimal.testInstanceMethod();
    // The class method in Animal
    // The instance method in Cat
  }
}
```

Plan

- Interfaces
- Héritage
- Polymorphisme
- `super` et héritage de `Object`
- Méthodes et classes finales/abstraites

Polymorphisme

- En biologie le polymorphisme est la propriété qu'ont les individus d'une espèce de se présenter sous plusieurs formes différentes
- Ce principe peut également être appliquée à la programmation objet :
  - les sous-classes d'une classe peuvent définir leurs propres comportements uniques et pourtant partager une partie des fonctionnalités de la classe parente

Exemple

- Ajout de la méthode `printDescription` dans la classe `Bicycle`

```
public void printDescription(){
  System.out.println("\nBike is in gear " + this.gear
  + " with a cadence of " + this.cadence
  + " and travelling at a speed of " + this.speed);
}
```

- Deux sous-classes de `Bicycle` sont créées :
  - `MountainBike` ajoutant un champ `suspension` de type chaine de caractères indiquant le type de suspension
  - `RoadBike` ajoutant un champ `tireWidth` de type entier indiquant l'épaisseur des pneus en millimètres

Exemple (cont.)

```
public class MountainBike extends Bicycle{
  private String suspension;

  public MountainBike(int startCadence, int startSpeed,
                      int startGear, String suspType) {
    super(startCadence, startSpeed, startGear);
    this.setSuspension(suspType);
  }

  public String getSuspension() {
    return this.suspension;
  }

  public void printDescription() {
    super.printDescription();
    System.out.println("The MountainBike has a"
    + getSuspension() + " suspension");
  }
}
```

Exemple (cont.)

```
public class RoadBike extends Bicycle{
  private int tireWidth; // In millimeters (mm)

  public RoadBike(int startCadence, int startSpeed,
                  int startGear, int newTireWidth) {
    super(startCadence, startSpeed, startGear);
    this.setTireWidth(newTireWidth);
  }

  public int getTireWidth(){
    return this.tireWidth;
  }

  public void printDescription(){
    super.printDescription();
    System.out.println("The RoadBike has "
    + getTireWidth() + " MM tires");
  }
}
```

Exemple (cont.)

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;
    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);
    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}

// Bike is in gear 1 with a cadence of 20 and travelling at ...

// Bike is in gear 5 with a cadence of 20 and travelling at ...
// The MountainBike has a Dual suspension

// Bike is in gear 8 with a cadence of 40 and travelling at ...
// The RoadBike has 23 MM tires
```

Plan

- Interfaces
- Héritage
- Polymorphisme
- `super` et héritage de `Object`
- Méthodes et classes finales/abstraites

Accéder aux membres de la super-classe

```
public class Superclass {
  public void printMethod() {
    System.out.println("Printed in Superclass");
  }
}

public class Subclass extends Superclass {
  public void printMethod() {
    super.printMethod();
    System.out.println("Printed in Subclass");
  }

  public static void main(String[] args) {
    Subclass s = new Subclass();
    s.printMethod();
    // Printed in Superclass
    // Printed in Subclass
  }
}
```

Invoquer le constructeur de la super-classe

```
public MountainBike(int startHeight, int startCadence,
                    int startSpeed, int startGear) {
  super(startCadence, startSpeed, startGear);
  seatHeight = startHeight;
}
```

- La syntaxe pour appeller le constucteur de la super-classe est :
  - `super();` ou alors `super(parameter list);`
- Le compilateur Java insert automatiquement un appel vers le constructeur sans argument de la super-classe (attention aux erreurs de compilation)

Héritage de la classe `Object`

- Tous les classes sont des descendants de la classe `Object` et héritent des méthodes :

```
protected Object clone() throws CloneNotSupportedException {}
// Creates and returns a copy of this object

public boolean equals(Object obj) {}
// Indicates whether some other object is "equal to"

protected void finalize() throws Throwable {}
// Called by the garbage collector on an object when garbage collection
// determines that there are no more references to the object

public final Class getClass() {}
// Returns the runtime class of an object

public int hashCode() {}
// Returns a hash code value for the object

public String toString() {}
// Returns a string representation of the object
```

Redéfinir la méthode `equals`

```
public class Book {
  // ...
  public boolean equals(Object obj) {
    if (obj instanceof Book)
      return ISBN.equals((Book)obj.getISBN());
    else
      return false;
  }
}
```

- Deux instances de `Book` seront considérés comme égaux si ils ont le même ISBN :

```
Book firstBook = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
  System.out.println("objects are equal");
}
```

Redéfinir la méthode `toString`

```
public class Book {
  // ...
  public String toString() {
    return title + " (" + ISBN + ")";
  }
}
```

- Deux instances de `Book` seront considérés comme égaux si ils ont le même ISBN :

```
Book firstBook = new Book("Bilbo le Hobbit", "2253049417");
System.out.println(firstBook);
// Bilbo le Hobbit (2253049417)
```

Plan

- Interfaces
- Héritage
- Polymorphisme
- `super` et héritage de `Object`
- Méthodes et classes finales/abstraites

Méthodes et classes finales

- Les méthodes d'une classe peuvent être déclarées avec `final` afin qu'elles ne puissent être redéfinies

```
class ChessAlgorithm {

  enum ChessPlayer { WHITE, BLACK }

  final ChessPlayer getFirstPlayer() {
    return ChessPlayer.WHITE;
  }
}
```

- Une classe entière peut être définie comme `final`

```
final class ChessBoard {
  final int[][] board = new int[8][8];
}
```

Méthodes et classes abstraites

- Un classe abstraite est déclarée avec le mot-clé *abstract*, elle ne peut pas être instanciée mais elle peut être la super-classe d'une autre classe
- Une méthode abstraite est une méthode déclarée sans implémentation, e.g.

```
abstract void moveTo(double deltaX, double deltaY);
```

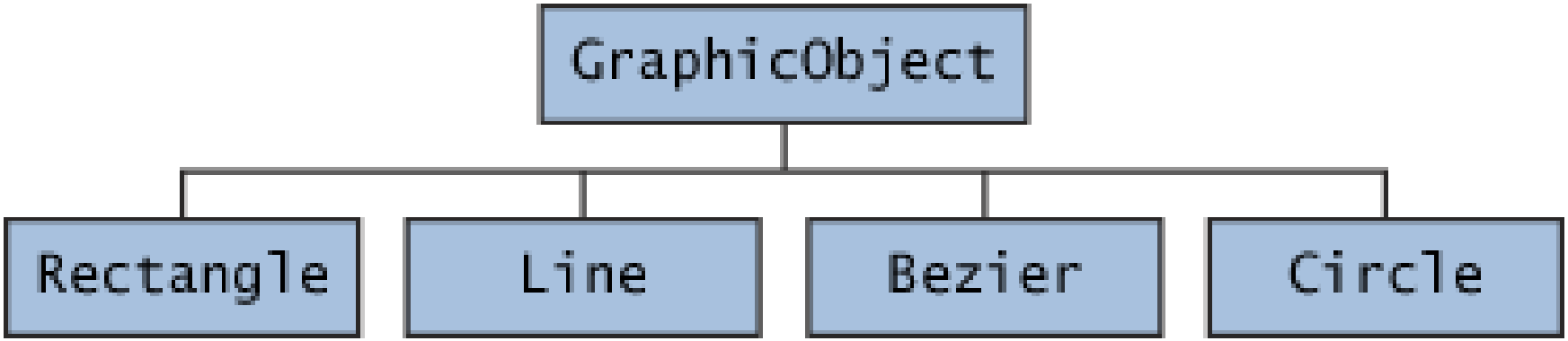
- Si une classe contient des méthodes abstraites alors elle doit être déclarée abstraite, e.g.

```
public abstract class GraphicObject {
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

Méthodes et classes abstraites (cont.)

- Contrairement aux interfaces, les classes abstraites peuvent contenir des champs qui ne sont pas `static` ou `final`
- Les classes abstraites contiennent une implémentation partielle
- *Exemple* : dans une application de dessin, vous pouvez dessiner des cercles, rectangles, lignes et autres objets graphiques
  - Les objets ont certains états (e.g. position, couleur) et comportements (e.g. déplacer, rotation) en commun

Méthodes et classes abstraites (cont.)



- On peut déclarer une classe abstraite `GraphicObject` afin de fournir les variables et les méthodes qui sont partagées par toutes les sous-classes

GraphicObject

- Certaines méthodes seront déclarées abstraites car leurs implémentations sont spécifiques aux sous-classes

```
abstract class GraphicObject {
  int x, y;
  ...

  void moveTo(int newX, int newY) {
    ...
  }

  abstract void draw();
  abstract void resize();
}
```

`Circle` et `Rectangle`

```
class Circle extends GraphicObject {
  void draw() {
    ...
  }

  void resize() {
    ...
  }
}

class Rectangle extends GraphicObject {
  void draw() {
    ...
  }

  void resize() {
    ...
  }
}
```

Résumé des notions abordées

- Excepté la classe `Object`, chaque classe a exactement une super-classe dont elle hérite les champs et méthodes
- Une sous-classe peut redéfinir les méthodes héritées et masquer les champs et méthodes hérités
- La classe `Object` est au sommet de la hiérarchie des classes, tous les descendants héritent de méthodes utiles comme `toString()` ou `equals()`
- Le mot-clé *final* peut être utilisé pour prévenir une méthode d'être surchargée ou une classe d'être dérivée
- Une classe abstraite ne peut pas être instanciée mais elle peut être la super-classe d'une autre classe, elle contient souvent une implémentation partielle

Questions

```
public class ClassA {
  public void methodOne(int i) { }
  public void methodTwo(int i) { }
  public static void methodThree(int i) { }
  public static void methodFour(int i) { }
}

public class ClassB extends ClassA {
  public static void methodOne(int i) { }
  public void methodTwo(int i) { }
  public void methodThree(int i) { }
  public static void methodFour(int i) { }
}
```

1. Quelle méthode sur-définit une méthode de la super-classe ?
2. Quelle méthode masque une méthode de la super-classe ?
3. Que font les autres méthodes ?

Réponses

```
public class ClassA {
  public void methodOne(int i) { }
  public void methodTwo(int i) { }
  public static void methodThree(int i) { }
  public static void methodFour(int i) { }
}

public class ClassB extends ClassA {
  public static void methodOne(int i) { }
  public void methodTwo(int i) { }
  public void methodThree(int i) { }
  public static void methodFour(int i) { }
}
```

1. Quelle méthode sur-définit une méthode de la super-classe ?
   - `methodTwo`
2. Quelle méthode masque une méthode de la super-classe ?
   - methodFour
3. Que font les autres méthodes ? des erreurs de compilation