

Programmation Orientée Objets avec Java

Prof. Yassin Rekik et Prof. Stéphane Malandain – 2023 - 2024

(Basé sur le cours de Joël Cavat)



Chapitre 2

Héritage et Polymorphisme

Concepts traités

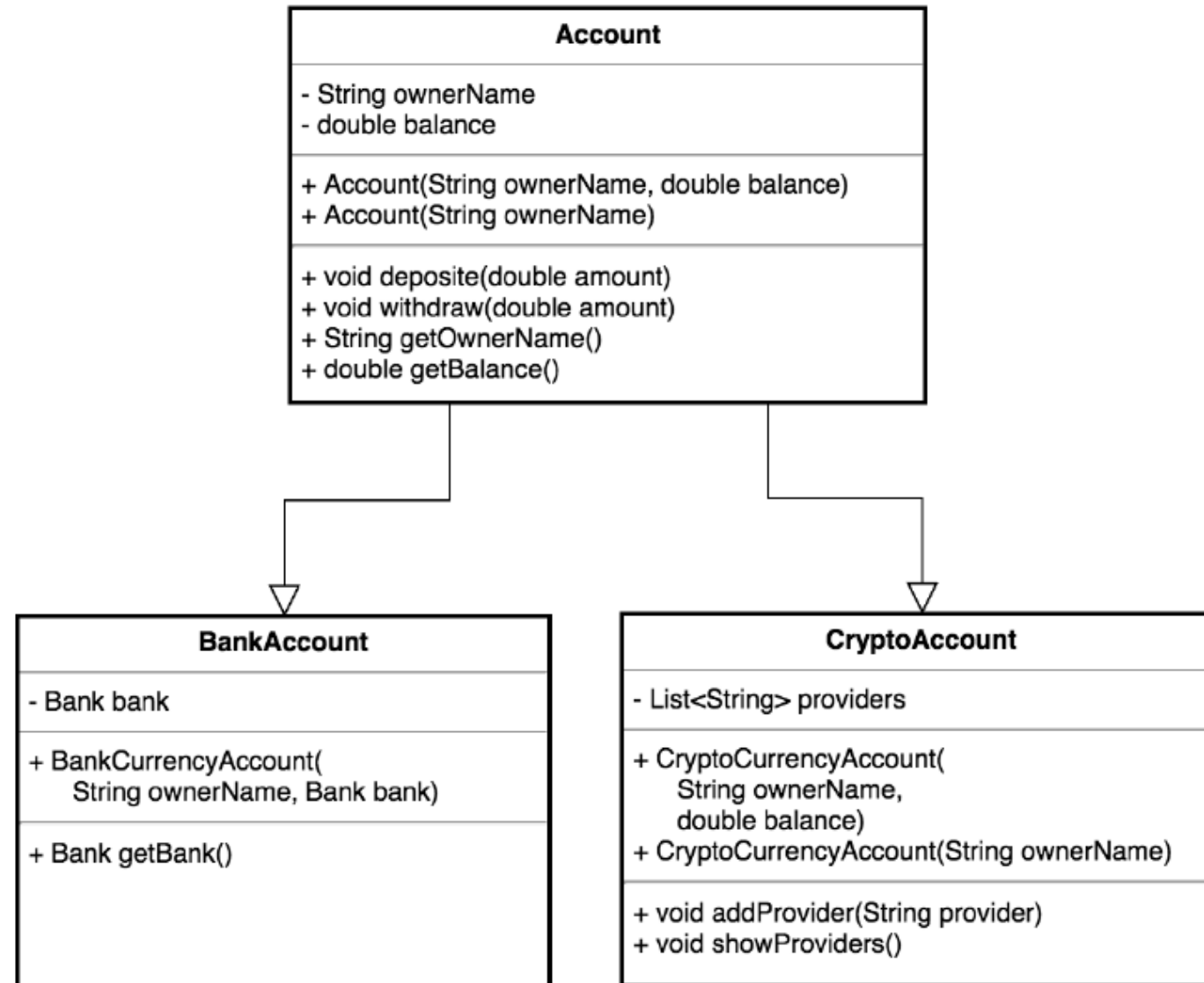
2

- L'héritage des classes
- Ajouts d'attributs et de méthodes / Surcharge de méthodes
- Redéfinition des implémentations des méthodes
- Analyse du typage entre classes et sous classes
- Les Interfaces Java
- Les classes Abstraites
- Analyse du typage entre classes – interfaces – classes abstraites
- Type déclaré / Type réel
- Polymorphisme
- Exemple d'utilisation

Héritage

Exemple de situation

4



Définition de l'héritage

5

- L'héritage est un **mécanisme permettant de créer une nouvelle classe à partir d'une classe existante en lui conférant ses propriétés et ses méthodes.**
- Ainsi, pour définir une nouvelle classe, il suffit de la faire hériter d'une classe existante et de lui ajouter de nouvelles propriétés/méthodes.
- En OO (Langage UML par exemple) on parle plutôt de Spécialisation-Généralisation
 - L'héritage est une manière d'implémenter la spécialisation
- L'héritage est un mécanisme permettant de maximiser la réutilisation de code

Syntaxe pour l'héritage

6

- Nouvelle classe : sous-classe de Account

```
public class BankAccount extends Account {  
  
    private Bank bank;  
  
    public BankAccount(String owner, Bank bank) {  
        super(owner, 0.0);  
        this.bank = bank;  
    }  
  
    public Bank getBank() { return this.bank; }  
  
}
```

Syntaxe pour l'héritage

7

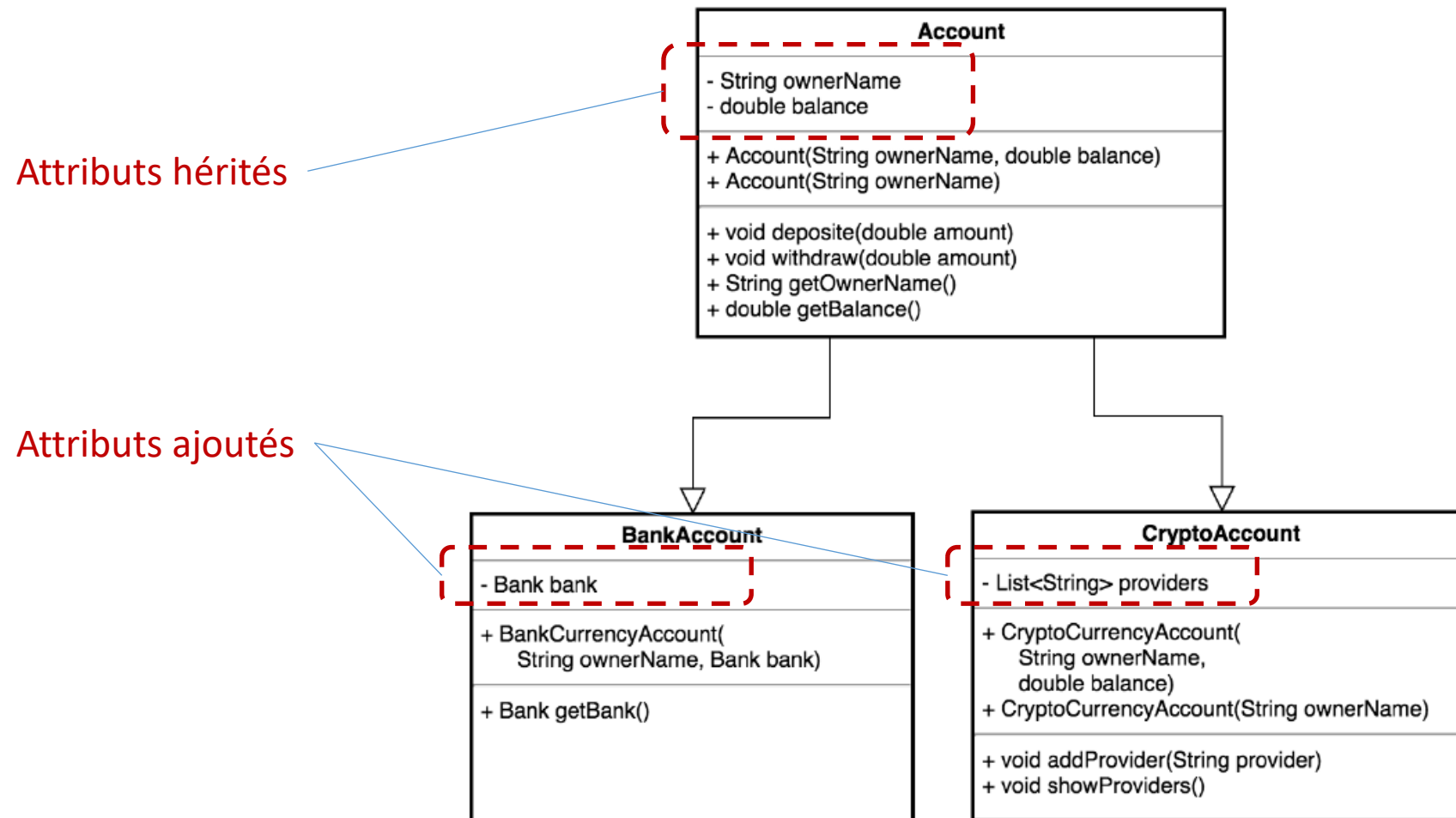
- Autre exemple

```
public class CryptoAccount extends Account {  
  
    private List<String> providers = new ArrayList<>()  
  
    public CryptoAccount(String owner, double balance) {  
        super(owner, balance);  
    }  
  
    public CryptoAccount(String owner) { this(owner, 0.0); }  
  
    public addProvider(String provider) {  
        this.providers.add(provider);  
    }  
  
    public void showProviders() {  
        for(String provider: providers) {  
            System.out.println(provider);  
        }  
    }  
}
```

Ajout d'attributs

8

- La spécialisation d'une classe en une autre classe nécessite souvent l'ajout de nouveaux attributs



Ajouts de méthodes

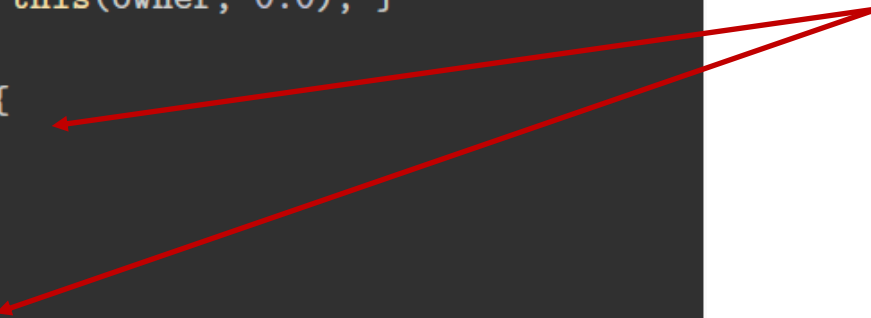
- Comme pour les attributs, une classe dérivée, qui hérite d'une super classe, peut
 - Ajouter des nouvelles méthodes
 - Surcharger des méthodes héritées
 - Redéfinir des méthodes héritées
- En plus, il est possible lors de l'héritage d'ouvrir la visibilité des méthodes héritées. L'inverse n'est pas possible et génère une erreur de compilation.

Ajout de méthodes

10

```
public class CryptoAccount extends Account {  
  
    private List<String> providers = new ArrayList<>()  
  
    public CryptoAccount(String owner, double balance) {  
        super(owner, balance);  
    }  
  
    public CryptoAccount(String owner) { this(owner, 0.0); }  
  
    public addProvider(String provider) {  
        this.providers.add(provider);  
    }  
  
    public void showProviders() {  
        for(String provider: providers) {  
            System.out.println(provider);  
        }  
    }  
}
```

Méthodes inexistantes
dans la super classe Account



Surcharge de méthode

11

- Utiliser un nom de méthode héritée, mais avec des paramètres différents
- Ça revient à ajouter une nouvelle méthode !!

```
5 public class Account {  
6     // ...  
7  
8     public void withdraw(double amount){  
9  
10         // Code 1  
11  
12     }  
13  
14 }
```

```
5 public class CryptoAccount extends Account{  
6     // ...  
7  
8     public void withdraw(double amount, int Commission){  
9  
10         // Code 2  
11  
12     }  
13  
14 }
```

Redéfinition de méthode

12

- La redéfinition est différente de la surcharge
 - La surcharge permet d'ajouter une nouvelle méthode, même si le nom existe
 - La redéfinition remplace l'implémentation interne d'une méthode existante héritée

```
5 public class Account {  
6     // ...  
7  
8     public void withdraw(double amount){  
9  
10        // Code 1  
11    }  
12  
13  
14 }
```

```
5 public class CryptoAccount extends Account {  
6     // ...  
7  
8     public void withdraw(double amount){  
9  
10        // Code 2 != Code 1  
11    }  
12  
13  
14 }
```

Redéfinition de méthodes

13

- Lorsqu'une méthode est redéfinie, c'est la nouvelle implémentation qui est utilisée lorsque l'objet admet réellement comme type la sous classe.
- Lors de l'exécution, c'est toujours le type réel et non pas déclaré qui est considéré.

```
2  class A {  
3      public String foo() { return "I'm A" }  
4  }  
5  
6  class B extends A {  
7      @Override  
8      public String foo() { return "I'm B" }  
9  }  
10 |
```

Appel Méthodes de la Super classe

14

- Dans une sous classe, les appels aux méthodes redéfinies permettent d'atteindre les nouvelles implémentations
- Si toutefois, vous souhaitez accéder à l'ancienne implémentation, se trouvant dans la super classe : vous pouvez utiliser le mot clef : `super`

```
3  class A {  
4      public String foo() { return "I'm A" }  
5  }  
6  
7  class B extends A {  
8      @Override  
9      public String foo() { return super.foo()+" and I'm B" }  
10 }  
11  
12 B b = new B();  
13 b.foo(); // return "I'm A and I'm B"  
14
```

Lien entre constructeurs

15

- Les constructeurs servent souvent à initialiser les attributs d'un objet
- Lors d'un héritage, il faut faire attention à l'initialisation des attributs hérités
 - Surtout ceux qui sont privés !!! Car ils ne sont pas accessibles dans la sous classe


Example : Point

16

```
3  class Point {  
4      private double x, y;  
5      public Point() {  
6          x = Math.random()*100;  
7          y = Math.random()*100;  
8      }  
9      public Point(double x, double y)  
10     {  
11         this.x = x; this.y = y;  
12     }  
13     public void add(int dx, double dy)  
14     {  
15         x += dx;  
16         y += dy;  
17     }  
18     public double getX() { return x; }  
19     public double getY() { return y; }  
20 }
```

Example : ColoredPoint

17

```
3  class ColoredPoint extends Point {  
4      private byte color;  
5      public void setColor(byte c) { color = c; }  
6       public byte getColor() { return color; }  
7  }
```

Exemple : Constructeurs

18

- Dans l'exemple précédent, lorsqu'un objet de type `ColoredPoint` est créé, on peut se demander si un constructeur de la classe `Point` est appelé.

La réponse est oui

- Juste avant que le constructeur de la classe `ColoredPoint` soit appelé, un constructeur de la classe de base est appelé.

Exemple : Constructeurs

19

- Par défaut, le constructeur par défaut est appelé
- Mais on peut en choisir un autre avec l'instruction `super(...)`.
- Dans ce cas **ça doit être la première instruction du constructeur**
 - Ici, les choix 1 et 2 sont équivalents (c'est inutile d'écrire `super()`).
 - Par contre dans le choix 3, le constructeur de `Point` est appelé avec deux paramètres (les coordonnées ne sont donc plus aléatoires mais initialisées à 0).

```
3  class ColoredPoint extends Point {
4      //Choix 1
5      public ColoredPoint() {
6          color = (byte)(Math.random()*256);
7      }
8      //Choix 2
9      public ColoredPoint() {
10         super();
11         color = (byte)(Math.random()*256);
12     }
13     //Choix 3
14     public ColoredPoint() {
15         super(0, 0);
16         color = (byte)(Math.random()*256);
17     }
18 }
19
```

Exemple : Attention

20

- Remarque: Les constructeurs ne sont pas hérités au même titre que les méthodes publiques. Lors de la création d'un objet, un constructeur de la classe dérivée doit obligatoirement exister.

```
2  
3 class ColoredPoint extends Point {  
4     private byte color;  
5     public ColoredPoint() {  
6         color = (byte)(Math.random()*256);  
7     }  
8     public void setColor(byte c) { color = c; }  
9     public byte getColor() { return color; }  
10 }
```

- On ne peut pas faire `new ColoredPoint(10, 10)` même si un constructeur de `Point` existe pour ces arguments.
- bien sûr si aucun constructeur n'est défini, le compilateur en définit un par défaut qui ne fait rien.

Exemple : Solution plus cohérente

21

- Pour être cohérent avec la classe `Point`, on peut faire :

```
3  class ColoredPoint extends Point {  
4      private byte color;  
5      public ColoredPoint() {  
6          color = (byte)(Math.random()*256);  
7      }  
8      public ColoredPoint(double x, double y) {  
9          super(x, y);  
10         color = (byte)(Math.random()*256);  
11     }  
12     public ColoredPoint(double x, double y, byte c) {  
13         super(x, y);  
14         color = c;  
15     }  
16     public void setColor(byte c) { color = c; }  
17     public byte getColor() { return color; }  
18 }
```

Autre exemple

22

```
5   public class A {  
6       private int j;  
7       public A(int j){this.j=j;}  
8  
9       // ...  
10  }  
11  
12  public class B extends A{  
13      private int i;  
14      public B(int i){this.i=i;}  
15  
16      // ...  
17  
18  }
```

- Erreur de compilation, aucun constructeur par défaut n'existe pour A

Typage entre classes

23

- A super classe – Type A
- B sous classe – Type B est un Sous type de A

```
3 public class A { // déclarée dans le fichier A.java
4     protected String nom = "Je suis dans A" ;
5
6     public void uneMethode() {
7         System.out.println(nom) ; // imprime "Je suis dans A"
8     }
9 }
10
11 public class B extends A { // déclarée dans le fichier B.java
12     protected String nom = "Je suis dans B" ;
13
14     public void uneAutreMethode() {
15         System.out.println(nom) ; // imprime "Je suis dans B"
16         System.out.println(super.nom) ; // imprime "Je suis dans A"
17     }
18 }
```

Typage entre classes

24

- On peut affecter une variable d'un sous type dans une variable d'un super type
- L'inverse n'est pas réalisé implicitement (erreur de compilation)

```
3 public class Main { // déclarée dans le fichier Main.java
4
5     Run | Debug
6     public static void main(String... args) {
7         A a = new A() ; // la classe A est celle que nous venons de définir
8         B b = new B() ; // idem
9         A ba = b ;      // cette déclaration est légale, car b est un élément de A,
10
11         System.out.println("a.nom = " + a.nom) ;
12         System.out.println("b.nom = " + b.nom) ;
13         System.out.println("ba.nom = " + ba.nom) ;
14     }
```

Tester le type : instanceof

25

- L'opérateur `instanceof` permet de tester le type d'une variable complexe
- Attention : une variable de type B , est aussi de type A si B hérite de A !!!!!

```
3 public class TableModel2 extends TableModel1 {  
4  
5     // code  
6  
7 }  
8  
9 TableModel2 tableModel = new TableModel2();  
10  
11 boolean t1 = tableModel instanceof TableModel1;  
12 boolean t2 = tableModel instanceof TableModel2;
```

t1 et t2 sont true

Comment différencier le type ?

26

```
3 public class TableModel2 extends TableModel1 {
4
5     // code
6
7 }
8
9 TableModel2 tableModel = new TableModel2();
10
11 boolean t1 = tableModel instanceof TableModel1;
12 boolean t2 = tableModel instanceof TableModel2;
13
14 boolean t2 = tableModel.getClass().equals(obj: TableModel1.class); //False
15 boolean t2 = tableModel.getClass().equals(obj: TableModel2.class); //True
```

Interdire l'héritage

27

- Il est possible d'interdire la redéfinition d'une classe ou d'une méthode

```
2 public final class A { // déclarée dans le fichier A.java, ne peut être étendue
3     public void ouSuisJe() {
4         System.out.println(x: "Je suis dans A !") ;
5     }
6 }
```

```
3 public class B { // déclarée dans B.java
4     public final void ouSuisJe() { // ne peut être surchargée, bien que
5         // B puisse être étendue
6         System.out.println(x: "J'y suis j'y reste !") ;
7     }
8 }
9
```

Polymorphisme

- En général, il est possible d'envisager :
 - Un héritage simple : une classe B qui hérite d'une classe A
 - Un héritage multiple : une classe C qui hérite à la fois d'une classe A et d'une classe B
- L'héritage multiple est possible en C++ , mais il n'est **pas autorisé en Java !!!**
- Pour remédier à cette limitation, et permettre de créer un sous typage multiple, Java propose un mécanisme similaire à l'héritage : **l'implémentation d'interface**

- Une interface permet de définir un ensemble de services qu'un client peut obtenir d'un objet
 - Une interface introduit une abstraction pure qui permet un découplage maximal entre un service et son implémentation
 - On retrouve ainsi les interfaces au cœur de l'implémentation de beaucoup de bibliothèques et de frameworks.
 - Le mécanisme des interfaces permet d'introduire également une forme simplifiée d'héritage multiple

Exemple d'Interfaces

31

- Comme pour une classe, une interface a une portée, un nom et un bloc de déclaration.
- Une interface est déclarée dans son propre fichier qui porte le même nom que l'interface.
- Une interface décrit un ensemble de méthodes en fournissant uniquement leur signature.

```
3 public interface Compte {  
4  
5     void deposer(int montant) throws OperationInterrompueException,  
6         CompteBloqueException;  
7  
8     int retirer(int montant) throws OperationInterrompueException,  
9         CompteBloqueException;  
10  
11     int getBalance() throws OperationInterrompueException;  
12  
13 }
```

- Une classe signale les interfaces qu'elle implémente grâce au mot-clé **implements**.
- Une classe concrète doit fournir une implémentation pour toutes les méthodes d'une interface, soit dans sa déclaration, soit parce qu'elle en hérite.
- Une interface fournit un contrat que toutes les classes qui l'implémentent doivent le respecter.

Implémentation

33

```
3 public class CompteBancaire implements Compte {
4
5     private final String numero;
6     private int balance;
7
8     public CompteBancaire(String numero) {
9         this.numero = numero;
10    }
11
12    @Override
13    public void deposer(int montant) {
14        this.balance += montant;
15    }
16
17    @Override
18    public int retirer(int montant) throws OperationInterrompueException {
19        if (balance < montant) {
20            throw new OperationInterrompueException();
21        }
22        return this.balance -= montant;
23    }
24
25    @Override
26    public int getBalance() {
27        return this.balance;
28    }
29
30    public String getNumero() {
31        return numero;
32    }
33
34 }
```

Lien entre Interface/Classe

34

- Comme pour l'héritage, l'implémentation d'interface permet de créer une relation de type sous-type / sur-type
- Si la classe *Humain* implémente les interfaces *Carnivore* et *Herbivore*. Donc une instance de la classe *Humain* peut être utilisée dans une application partout où les types *Carnivore* et *Herbivore* sont attendus.

Lien entre Interface/Classe

35

```
3 public interface Herbivore {
4
5     void manger(Vegetal vegetal);
6
7 }
8
9 public interface Carnivore {
10
11     void manger(Animal animal);
12
13 }
14
15 public class Humain extends Animal implements Carnivore, Herbivore {
16
17     @Override
18     public void manger(Animal animal) {
19         // ...
20     }
21
22     @Override
23     public void manger(Vegetal vegetal) {
24         // ...
25     }
26
27 }
28
```

Héritage entre interfaces

36

- Une interface peut hériter d'autres interfaces. Contrairement aux classes qui ne peuvent avoir qu'une classe parente, une interface peut avoir autant d'interfaces parentes que nécessaire. Pour déclarer un héritage, on utilise le mot-clé **extends**.
- Une classe concrète qui implémente une interface doit donc disposer d'une implémentation pour les méthodes de cette interface mais également pour toutes les méthodes des interfaces dont cette dernière hérite.

```
4 public interface Omnivore extends Carnivore, Herbivore {  
5  
6 }
```

- Nous avons vu que l'héritage est un moyen de mutualiser du code dans une classe parente.
- Parfois cette classe représente une abstraction pour laquelle il n'y a pas vraiment de sens de créer une instance :
 - Abstraction pas réelle
 - Manque d'informations
- Dans ce cas, on peut considérer que la généralisation est *abstraite*.

- Une classe abstraite peut déclarer des méthodes abstraites.
- Une méthode abstraite possède une signature mais pas de corps.
- Cela signifie qu'une classe qui hérite de cette méthode doit la redéfinir pour en fournir une implémentation (sauf si cette classe est elle-même abstraite).

Exemple

39

```
3 public abstract class Vehicule {  
4  
5     private final String marque;  
6     protected float vitesse;  
7  
8     public Vehicule(String marque) {  
9         this.marque = marque;  
10    }  
11  
12     public abstract int getNbRoues();  
13  
14  
15     // ...  
16  
17 }
```

Héritage de classe abstraite

40

- Toutes les classes concrètes héritant de *Vehicule* doivent maintenant fournir une implémentation de la méthode *getNbRoues* pour pouvoir compiler.

```
3  public class Voiture extends Vehicule {  
4  
5      public Voiture(String marque) {  
6          super(marque);  
7      }  
8  
9      @Override  
10     public int getNbRoues() {  
11         return 4;  
12     }  
13  
14     // ...  
15  
16 }
```

- On dit que la classe *Voiture* est concrète

exemple

41

```
3  import java.awt.Point;
4  abstract public class Shape /* extends Object */ {
5      // --- The center attribute ---
6      private Point center;
7
8      // --- 2 constructors ---
9      public Shape() {
10         this( new Point( x: 0, y: 0 ) );
11     }
12
13     public Shape( Point center ) {
14         // super();      // Appel au constructeur parent sous-entendu
15         this.setCenter( center );
16     }
17
18     // --- The center property ---
19     public Point getCenter() {
20         return center;
21     }
22
23     public void setCenter( Point center ) {
24         if ( center == null ) {
25             throw new NullPointerException( s: "center parameter cannot be null" );
26         }
27         this.center = center;
28     }
29
30     // --- An abstract method for compute the area of the shape ---
31     public abstract double area();
32
33 }
```

Example, suite

42

```
3 public class Circle extends Shape {
4
5     // Begin of the class: attributes, constructors, properties, ...
6
7
8     @Override
9     public double area() {
10         return Math.PI * this.radius * this.radius;
11     }
12
13 }
```

```
3 public class Square extends Shape {
4
5     // Begin of the class: attributes, constructors, properties, ...
6
7
8     @Override
9     public double area() {
10         return this.length * this.length;
11     }
12
13 }
14
```

- Comme pour l'héritage normal entre classes concrètes, une classe concrète qui hérite et concrétise une classe abstraite sera considérée comme un sous-type du type abstrait
- Dans l'exemple précédent :
 - La classe abstraite `Shape` : définit un type
 - Les classes `Cercle` et `Carré` définissent deux sous-types du type `Shape`
- Pour résumer, une classe définit un sous type lorsque
 - Elle hérite d'une classe concrète
 - Elle hérite et concrétise une classe abstraite
 - Elle implémente une interface

Type réel / Type déclaré

44

- En Java, une variable objet a à la fois un type déclaré ou un type à la compilation et un type à l'exécution ou un type réel.
- Le type déclaré ou le type au moment de la compilation d'une variable est le type utilisé dans la déclaration.
- Le type d'exécution ou le type réel est la classe qui crée réellement l'objet.

Type réel / Type déclaré

45

- Dans le cas ci-contre, notre programme affiche "I'm B".
 - Car l'objet a a été défini par la classe B même s'il est stocké dans une variable de type A.
 - Donc lors de l'exécution de la méthode foo, la classe la plus dérivée (ici B) qui possède la méthode et donc utilisée.
- C'est ce qu'on appelle le polymorphisme.

```
3  class A {  
4      String foo() { return "I'm A" }  
5  }  
6  class B extends A {  
7      String foo() { return "I'm B" }  
8  }  
9  A generateSomethin() { return new B(); }  
10  
11  
12  A a = generateSomethin();  
13  System.out.println(a.foo());  
14
```

- Le polymorphisme est le concept consistant à fournir une interface unique à des entités pouvant avoir différents types (wikipedia).
 - On pourra remarquer que la surcharge de méthode est aussi considérée comme du polymorphisme.
 - Lorsqu'il provient d'une redéfinition de méthode, on l'appelle polymorphisme par sous-typage.
- Polymorphisme
 - En clair, ça signifie que lorsqu'on appelle une méthode sur un objet, l'implémentation choisie dépend du type réel de l'objet (la classe qui a créé l'objet) et ne dépend pas du type de la variable qui stocke l'objet.
 - Cela permet d'appeler la même méthode sur des objets sans se soucier de leur type !!

Exemple d'utilisation

47

- Gérer des objets de sous-types différents dans une même collection
 - Tableau de Shape (donc il y aura des cercles, des carrés, des rectangles,)
- Itérer sur des structures de données complexes contenant des objets de même type, mais de sous-types différents
 - Arbres de Nodes XML , mais ensuite les nodes sont des CDATA, Attributes, ...
- Manipuler des objets de manière similaire, mais avec des implémentations multiples
 - Jeu d'échec, avec des pièces, mais les déplacements sont implémentées différemment puisque le déplacement de chaque type de pièce est différent !