Chapitre 5

L'orienté objet

1 Héritage en Java

Les relations d'héritage en java sont créées par l'extension d'une classe. Les deux raisons pour utiliser l'héritage sont:

- Permettre la réutilisation du code
- utiliser le polymorphisme

1.1 Réutilisation du Code

Code réutilisé par héritage signifie que les méthodes avec des fonctionnalités génériques ne sont pas ré-implémentées.

Exemple: la classe playerPiece hérite la méthode displayShape et l'utilise en tant que tel

```
class GameShape {
  public void displayShape() {
     System.out.println("displaying shape");    }
}
class PlayerPiece extends GameShape {
   public void movePiece() {
   System.out.println("moving game piece");    }
}
```

1.2 Polymorphisme ("plusieurs formes")

Il est possible de traiter une sous-classe comme la classe super (mère). Par exemple: n'importe quel sous classe de GameShape peut être traitée comme un Gameshape.

Exemple: en utilisant la classe GameShape de l'exemple précédent, quel est le résultat du code suivant?

```
public class TestShapes {
  public static void main (String[] args) {
 PlayerPiece player = new PlayerPiece();
      TilePiece tile = new TilePiece();
     doShapes(player);
      doShapes(tile);
  }
  public static void doShapes(GameShape shape) {
     shape.displayShape();    }
}
class TilePiece extends GameShape {
  public void getAdjacent() {
      System.out.println("getting adjacent tiles");     }
}
```

Résultat : ……

……

⇨ ……

……

- Les méthodes appelées avec une référence sont totalement dépendantes **du type de la référence**.

  ➔ Ce n'est pas possible d'utiliser une variable GameShape pour appeler la méthode getAdjacent() même si l'objet passé est de type TylePiece. Par exemple:

```
GameShape r=new TilePiece ();
    r. getAdjacent();//Compilation fails, Game Shapes does not
                     //have the method getAdjacent
```

2 Relation « est-un »

Est utilisée pour dire cette chose est de type tel chose. Cette relation en java est utilisée à travers l'héritage de classe et l'implémentation d'interfaces.

Exemple: soit les classes suivantes:
public class Vehicle { ... }
public class Car extends Vehicle { ... }
public class Subaru extends Car { ... }

Les relations suivantes sont correctes :

Car hérite de Vehicle.
Subaru hérite de Vehicle et Car.
Subaru est-un Vehicle et est-un Car.

3 Relation « a-un »

- Basée sur l'utilisation
- class A a-un B si le code dans la classe class A a une référence à une instance de la class B

Exemple:

```
public class Animal { }
public class Cheval extends Animal {
   private Halter myHalter;
}
```

Un cheval est-un Animal. Un cheval a-un Halter.

4 Polymorphisme

- Une variable de référence peut référer n'importe quel objet de même type que la référence déclarée ou bien **il peut référer n'importe quel** *sous-type* **du type declaré!**
- Une variable de référence peut être déclarée comme un type classe ou un type interface. Si la variable est déclarée comme un type interface, il peut référer n'importe quel objet de n'importe quel classe qui implémente l'interface.

```
Interface i{};
class A implements i{ i r;
}
```

- Une classe ne peut pas hériter plus qu'une classe
- N'importe quelle classe peut hériter des classes multiples à travers son arbre d'héritage

Exemple: Revenons à l'exemple précédent de Gameshape. Supposons que certaines sous-classes qui hérite de GameShape peuvent être animées en utilisant une méthode animate(). Quelle sera la solution ?

- Mettre la méthode animate() dans GameShape, et le mettre non accessible quand les classes qui ne sont pas animées.
  - ➔ Mauvaise conception et mauvais choix pour plusieurs raisons, ça permet d'introduire des erreurs
  - ➔ Sa signifie que les API GameShape "avertis" tous les shapes d'être animées alors que ce n'est pas correct puisque seulement quelques sous-classes de GameShape peuvent être capables d'exécuter avec réussite la méthode animate().

La solution est de créer une interface qui contiendra ma méthode animate(), seule les sous-classes de GameShape qui sont animées, implémentent l'interface. Dans ce cas, l'interface suivante est définie:

```
public interface Animatable {
  public void animate();}
```

Exemple: Soit la classe PlayerPiece héritant de GameShape et implementant l'interface Animatable

```
class PlayerPiece extends GameShape implements Animatable {
 public void movePiece() {
    System.out.println("moving game piece");    }
   public void animate() {
    System.out.println("animating...");
  }
}
```

Un objet PlayerPiece est-un GameShape et est-un Animatable. Il peut être traité polymorphiquement comme:

- un object (puisque chaque objet hérite d'Object)
- un GameShape (puisque PlayerPiece extends GameShape)
- un PlayerPiece (puisque c'est réellement c'est qu'il est)
- un Animatable (puisque PlayerPiece implemente Animatable)

Ces declarations sont toutes légales:

- PlayerPiece player = new PlayerPiece();
- Object o = player;
- GameShape shape = player;
- Animatable mover = player;

Il y'a uniquement un seul objet –une instance- de type PlayerPiece—mais ils existent quatre types différents de variable de référence qui réferent le même objet dans le segment.

Question 1: lequel parmi les variables de référence précédentes peut invoquer la méthode displayShape()?

Les deux classes GameShape et PlayerPiece class sont connues par le compilateur d'avoir une méthode displayShape(), donc chacune de ces références peut être utilisé pour invoquer displayShape().

Question 2: Quelles sont les méthodes qui peuvent être invoquées avec mover?

PlayerPiece player = new PlayerPiece();

Animatable mover = player;

=> Uniquement la méthode animate()

Si on redéfinie la méthode displayShape dans la classe PlayerPiece comme suit:

```
public void displayShape() {
     System.out.println("displaying player piece");
}
```

Et on crée un objet PlayerPiece avec une référence GameShape et on invoque la méthode displayShape

```
GameShape g = new PlayerPiece();
g.displayShape();
```

Quelle méthode displayShape sera exécutée celle de GameShape ou de PlayerPiece?

➔ displayShape de …… est invoquée

Même si l'objet est PlayerPiece, la méthode displayShape() est applée en utilisant une variable de référence GameShape,

```
Gameshape g=new playerPeace();
     g.displayShapes ();
```

➔ Si PlayerPiece redéfinie (override) la méthode displayShape(), le JVM invoquera la version de PlayerPiece!

Les invocations polymorphique des méthodes sont appliquées uniquement **aux méthodes d'instance.** Il est possible toujours de référencer un objet avec une variable de référence plus générale de type (une superclass ou une interface)**, mais en exécution,** Uniquement les choses qui sont dynamiquement sélectionnées en se basant sur l'objet actuel **(et non pas le type de la variable de *référence*) sont les méthodes d'instance.**

➔ Ce ne sont pas les méthodes *static*. Ni les *variables.* Uniquement les méthodes d'instances redéfinies (overridden) sont dynamiquement invoquées en se basant sur le type réel de l'objet.

5 Redéfinition(Overriding) / Surcharge(Overloading)

5.1 Méthodes redéfinies (Overridden)

Le bénéfice d'overriding (redéfinition) est la capabilité de définir le comportement qui est spécifique pour une sous-classe type.

Exemple:

```
public class Animal {
   public void eat() {
      System.out.println("Generic Animal Eating Generically");   }}
class Horse extends Animal {
   public void eat() {
   System.out.println("Horse eating hay");
 }}
```

Exemple d'utilisation de polymorphisme: soit les classes suivantes:

```
class Animal {
  public void eat () {
    System.out.println("Generic Animal Eating Generically‟);  }}
class Horse extends Animal {
public void eat() {
   System.out.println("Horse eating hay");   }
public void buck() { }
}
public class TestAnimals {
 public static void main (String [] args) {
   Animal a = new Animal();
   Animal b = new Horse();
   a.eat();
   b.eat();
}}
```

Est-ce que la classe suivante compile?

```
1. public class TestAnimals {
2. public static void main (String [] args) {
3.   Animal a = new Animal();
4.   Animal b = new Horse();
5.   a.eat();
6.   b.eat();
7.   b.buck(); //…….
8.}
9.}
```

……

……

Le compilateur regarde toujours le type de référence et non pas le type d'instance.

- Soit la classe Horse redéfinie la méthode héritée de Animal soit simplement l'hérite, n'importe qui avec une référence Animal à une instance Horse est libre d'appeler toutes les méthodes accessibles de Animal.
- Pour cette raison, une méthode redéfinie doit remplir le contrat de la classe super.
- Les règles pour redéfinir une méthode sont:
- la liste d'arguments doit être la même que celle de la méthode (overridden) originale de la classe super.
- Le type de retour doit être le même ou un sous- type comme celui déclaré dans la méthode originale de la classe super
- Le type de retour doit être le même ou un sous- type comme celui déclaré dans la méthode originale de la classe super
- Le niveau d'accès peut être moins restrictif que la méthode originale de la classe super.
- Les méthodes d'instance héritées par les sous-classes peuvent être redéfinies si elles ne sont pas marquées private ou final. Si elles sont dans un package différent, elles doivent être marquées public ou protected
- La méthode rdéfinie ne doit pas avoir des nouvelles exceptions ou des exeptions plus larges que celles de la méthode de la super-classe.
- La méthode redéfinie peut avoir moins d'exceptions ou des exceptions moins larges.

5.1.1 Invoquer la version de la classe super de la méthode redéfinie

Pour invoquer la méthode de la classe super de la méthode redéfinie, utiliser la syntaxe suivante dans la méthode redéfinie:

```
super.method();
```

Exemple: dans la classe Horse, on modifie la méthode eat() comme suit:

```
public void eat(){
super.eat();
System.out.println("Horse eating hay");
}
```

5.2 Méthodes surchargées (Overloaded)

Les méthodes surchargées permettent d'utiliser le même nom de méthode dans une classe mais avec des arguments différents, et optionnellement un type différent de retour. Les règles sont:

- Les méthodes surchargées doivent changer la liste d'arguments.
- Les méthodes surchargées peuvent changer leurs types de retour
- Les méthodes surchargées peuvent changer leurs modificateurs d'accès.
- Les méthodes surchargées peuvent déclarer des nouvelles exceptions ou des exceptions plus larges

Exemple: Est-ce que le code suivant compile?

```
public class Foo {
public void doStuff(int y, String s) { }
public void moreThings(int x) { }
}
class Bar extends Foo {
 public void doStuff(int y, long s) throws Exception {
 }
```

……..

- Une méthode peut être surchargée dans la même classe ou dans une sous-classe.
  - si une classe A définit une méthode dostuff(int i), une sous-classe B peut définir une méthode dostuff (String s) sans redéfinir la version de la superclass qui a un argument int.

5.2.1 Ambiguïté de surcharge

Le compilateur peut ne pas savoir laquelle des méthodes choisir. Par exemple:

```
public class Overloading {
public static void m(int i, long l) {
System.out.println("m(int, long)");
}
public static void m(long l, int i) {
System.out.println("m(long, int)");
}
public static void main(String[] args) {
int i = 1;
long l = 1L;
m(i,l); //……
m(l,i); //……
m(i,i); //……..
…….
}
}
```

- Une méthode peut être surchargée dans la même classe ou dans une sous-classe.
  - si une classe A défini une méthode dostuff(int i), une sous-classe B peut définir une méthode dostuff (String s) sans redéfinir la version de la superclasse qui a un argument int.

Soient les classes suivantes:

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
   public void doStuff(Animal a) {
      System.out.println("In the Animal version");   }
  public void doStuff(Horse h) {
      System.out.println("In the Horse version");   }
```

Soit le code suivant:

```
UseAnimals ua = new UseAnimals();
Animal a = new Horse();
ua.doStuff(a);
```

Laquelle des méthodes doStuff() sera invoquée?.....

Le type de *référence* (non le type d'objet) <u>détermine quelle méthode surchargée est invoquée!</u>

6 Constructeurs et Instanciation

Chaque classe, *incluant les classes abstract*, doivent avoir un constructeurr. Comme par exemple:

```
class Foo {
 Foo() { } // Constructeur de la classe Foo
}
```

Typiquement, les constructeurs sont utilisés pour initialiser l'état de variables comme montre l'exemple suivant:

Exemple:

```
class Foo {
   int size;
   String name;
   Foo(String name, int size) {
      this.name = name;
      this.size = size;
 }
}
```

La classe Foo n'a pas un constructeur sans arguments. Donc

```
Foo f = new Foo();
```

ne compile pas, le constructeur n'existe pas

Mais ce code compile:

```
Foo f = new Foo("Fred", 43);
```

=> Ne compile pas. invocation récursive du constructeur

Il est très commun et désirable pour une classe d'avoir un constructeur sans arguments

Exemple 1: 1) soient les classes A et B suivantes:

```
class A{}
class B extends A{
public static void main(String [] args) {
B ob=new B ();
}
```

⇒ Code compile même sans constructeur => un constructeur par défaut sans arguments est exécuté, ce constructeur a la forme suivante :

```
B() {super();}
```

2) On ajoute des constructeurs aux classes A et B comme suit quel sera le résultat?

```
class A{
A() {System.out.println("ici la classe A");}
}
class B extends A{
B() {
System.out.println("ici la classe B");}
}
```

Résultat :

Ici la classe A

Ici la classe B

⇒ les deux constructeurs s'exécutent: de la classe A et de la classe B => constructeur de super-class s'exécute implicitement

3) On modifie le constructeur de la classe A en lui ajoutant un argument:

```
class A{
A(int x) {System.out.println("constructeur avec arg"+x);}
}
class B extends A{
B() {
System.out.println("ici la classe B");}
}
public static void main(String [] args) {
B ob=new B ();
```

```
}
```

=> La classe B ne compile pas. La constructeur par défaut de la classe A est désactivé car elle a un constructeur avec argument.

Le constructeur de la classe B doit appeler le constructeur super en utilisant par exemple:

```
super (6);
```

⇨ L'instruction super doit être la première dans le code

```
class A{
A(int x) {System.out.println("constructeur avec arg"+x);}
}
class B extends A{
B() {
Super(6);
System.out.println("ici la classe B");}
}
public static void main(String [] args) {
B ob=new B ();
}
```

Résultat:
constructeur avec arg6
ici la classe B

4) Si on ajoute le modificateur abstract à la class A, est ce que B compile?

⇨ B compile, et le constructeur de la classe abstract A s'exécute et on aura le même résultat que la question précédente

Résultat:
constructeur avec arg6
ici la classe B

5) Ajouter la méthode suivante à la classe A

```
int compute(int r){
A(r);
int x=r*2;
return x;
}
```

=> Ne compile pas. Un Constructor ne peut être invoqué qu'à partir d'un constructeur.

6.1 Les Règles pour les constructeurs

- S'il n'y a pas un constructeur dans la classe, a default constructor will be automatically generated by the compiler.
- Le constructeur par défaut et toujours un constructeur sans arguments.
- Si un constructeur avec arguments est tapé, le constructeur par défaut n'existera pas, sauf si vous le tapez-vous même!
- Chaque constructeur a, comme sa première instruction, soit un appel à un constructeur surchargé (this()) ou un appel à la classe super (super()).
- Un appel à super() peut être soit un appel sans arguments soit avec arguments passés au constructeur super.
- Ce n'est pas possible d'appeler les méthodes d'instance, ou les variables d'instance, qu'après l'exécution du constructeur.

- Uniquement les variables et les méthodes static peuvent être accédés comme une partie d'appel de super() ou this().
- Les classes abstarct ont des constructeurs, et ces constructeurs sont toujours appelés quand une sous-classe est instanciée.
- Les interfaces n'ont pas constructeurs. Les interfaces ne sont pas parties de l'arbre d'héritage pour les objets.
- Un constructeur peut être invoqué uniquement à partir d'un autre constructeur.

N.B. Les ***constructeurs ne sont pas héritables.*** Ils ne peuvent pas être redéfinis mais ils peuvent être surchargés.

6.2 Le surcharge des constructeurs

Exemple: 1) Quel est le résultat du code suivant?

```
class Foo1{
int y;
Foo1(){
this(3);
System.out. println("constructeur Foo sans arguments");
}
Foo1 (int t){
this.y=t;
System.out.println(" constructeur avec un arg int"+t) ;
}
```

Règle clé:

La première ligne dans un constructeur doit être un appel à super() ou this() uniquement l'un d'eux peut être utilisé.

- **Super()** ou **super(args):** permet d'invoquer le constructeur de la classe super
- **this()** ou **this (args):** permet d'invoquer un constructeur surchargé dans la même classe.

Exemple 2: Quel est le résultat de la classe suivante?

```
class A1 {
  A1() {     this("Hello");   }
 A1(String s) {      this();   }
}
```

=> Ne compile pas. invocation récursive du constructeur