Chapitre 10

Le polymorphisme

1. Intérêt et définitions
2. Polymorphisme par redéfinition
3. Polymorphisme paramétrique de méthode
4. Polymorphisme par surcharge

1 Intérêt et définitions

- Le polymorphisme est un puissant mécanisme Java permettant aux objets d'avoir des comportements dynamiques
  - Il est fortement basé sur le concept de l'héritage, mais pas que
  - Il permet à un objet d'adapter son comportement en fonction des circonstances et du contexte
  - En termes plus simples :
    - Poly => plusieurs
    - Morphisme => comportement

    ➔ Un objet polymorphe ⇔ un objet (instance de classe) ayant des comportements dynamiques

- Le polymorphisme basé sur **l'héritage** (successif ou non) joue sur la **relation entre les classes** (super-classes ou sous-classes) pour apporter un dynamisme comportemental sur les **méthodes** qui sont définies :
  - Une méthode peut être redéfinie dans une sous-classe et être appelée de façon transparente
  - Une méthode peut manipuler un objet sans en connaître le type effectif au moment de l'implémentation

- Avec le polymorphisme, un objet a comme **type** non seulement sa classe mais aussi n'importe quelle classe située dans son ascendance. Une classe B est compatible en terme de typage à la classe A, si A figure dans son ascendance dans une relation d'héritage (parent, grands-parents ou arrière grands-parents, etc). L'inverse n'est pas vrai.

1 Intérêt et définitions

2 Polymorphisme par redéfinition

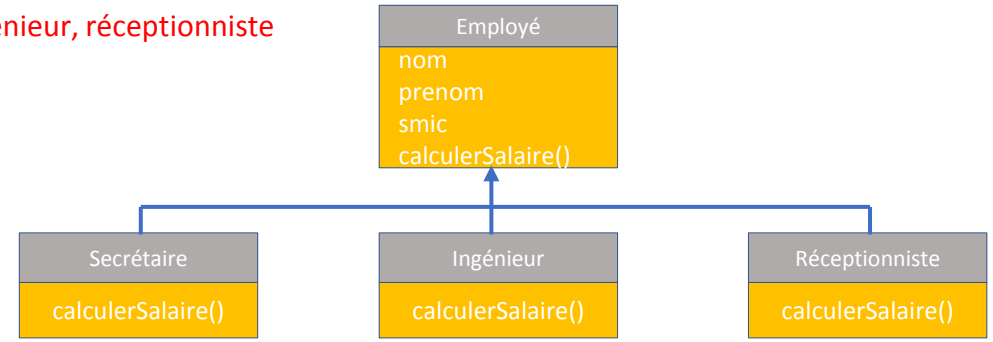
3 Polymorphisme paramétrique de méthode

4 Polymorphisme par surcharge

2 Polymorphisme par redéfinition (1/3)

- Java permet, en s'appuyant sur la règle de compatibilité, de rendre transparent l'appel d'une méthode **redéfinie** dans une sous-classe, au travers d'un objet de type déclarée comme étant celle de sa classe mère. C'est la notion d'**overriding**.

- Exemple :
  - Soit le diagramme ci-dessous modélisant à l'aide de l'héritage les employés d'une entreprise selon différents profiles : secrétaire, ingénieur, réceptionniste



  - La force du polymorphisme dans cet exemple est que :

    - la méthode calculerSalaire() de la classe Employé est **redéfinie** dans les sous-classes Secrétaire, Ingénieur et Réceptionniste. Ainsi, sachant que ces sous-classes sont aussi de type Employé, l'appelle à la méthode calculerSalaire() d'un objet Employé aura des comportements différents selon que l'employé effectif soit Secrétaire, Ingénieur ou Réceptionniste

2 Polymorphisme par redéfinition (2/3)

- Implémentation :

```
package emp.entreprise;

public class Employe {

  public String nom;
  public String prenom;
  public final Double smic  = 1300.00d;

  public Double calculerSalaire() {
     return 1.1 * smic;
  }
}
```

```
package emp.entreprise;

public class Receptionniste extends Employe {

   private float indicePoste = 1.2f;

   public Double calculerSalaire() {
     return indicePoste * super.smic;
   }
}
```

```
package emp.entreprise;

public class Secretaire extends Employe {

   private float indicePoste = 1.6f;

   public Double calculerSalaire() {
     return indicePoste * super.smic;
   }
}
```

```
package emp.entreprise;

public class Ingenieur extends Employe {

   private float indicePoste = 2.0f;

   public Double calculerSalaire() {
      return indicePoste * super.smic;
   }
}
```

2 Polymorphisme par redéfinition (3/3)

- Exploitabilité :

```
package emp.entreprise;

public class Test {

   public static void main(String[] args) {

       Double salaire = 0.00d;

       //bloc1
       Employe emp = new Employe();
       salaire = emp.calculerSalaire();
       System.out.println("Le salaire d'un " + emp.getClass().getSimpleName() + " est : " + salaire);

       //bloc2
       emp = new Receptionniste();
       salaire = emp.calculerSalaire();
       System.out.println("Le salaire d'un " + emp.getClass().getSimpleName() + " est : " + salaire);

       //bloc3
       emp = new Secretaire();
       salaire = emp.calculerSalaire();
       System.out.println("Le salaire d'un " + emp.getClass().getSimpleName() + " est : " + salaire);

       //bloc4
       emp = new Ingenieur();
       salaire = emp.calculerSalaire();
       System.out.println("Le salaire d'un " + emp.getClass().getSimpleName() + " est : " + salaire);

    }

}
```

Interprétation :

> Dans le bloc1, on définit un objet *emp* de type *Employé* et on appelle sa méthode *calculerSalaire()*. Dans ce cas, c'est bien la méthode définie dans la classe *Employé* qui est appelée

> Dans le bloc2, d'après la règle de compatibilité, un *Réceptionniste* est aussi un *Employé*. Il est donc possible d'affecter une instance de *Receptionniste* à l'objet *emp*. Dans ce cas, l'appelle de la méthode *calculerSalaire()* correspondra non plus à celle définie dans Employé mais bien celle de *Réceptionniste*. (idem pour les bloc3 et bloc4)

Constat :

> Avec la même référence *emp*, on a appelé la même méthode *calculerSalaire()* et en fonction de son **type effectif** (Secrétaire, Ingénieur, …), on obtient des comportements différents. Java effectue la conversion implicite de *emp* en un objet du type effectif.

Résultat affiché par le programme :

```
Le salaire d'un Employe est : 1430.0000000000002
Le salaire d'un Receptionniste est : 1560.0000619888306
Le salaire d'un Secretaire est : 2080.0000309944153
Le salaire d'un Ingenieur est : 2600.0
```

NB : la méthode *obj.getClass().getSimpleName()* renvoi le nom littéral d'une classe à partir de son instance obj

1 Intérêt et définitions

2 Polymorphisme par redéfinition

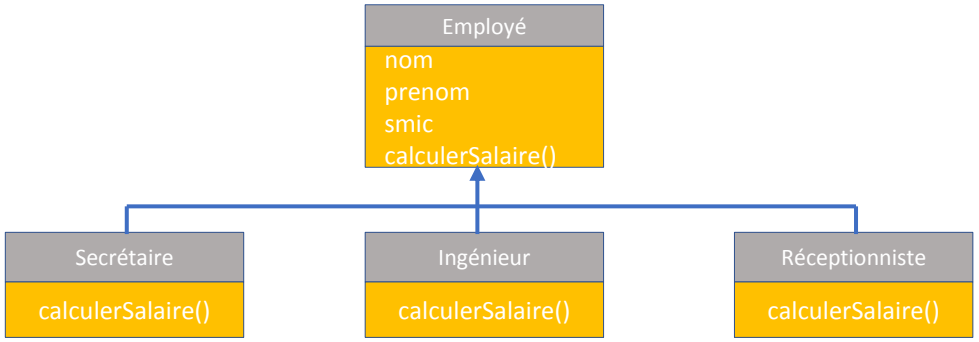
3 Polymorphisme paramétrique de méthode

4 Polymorphisme par surcharge

3 Polymorphisme paramétrique de méthode – cas 1 (1/2)

- Cas d'une méthode ayant un paramètre de type une classe C, mais appelée avec un objet dont le type explicite est une sous-classe de C :
  > Java peut détecter que l'objet effectivement passé en paramètre d'une méthode a été typé avec la classe fille de la classe attendue

- Exemple :
  - Soit toujours le diagramme ci-dessous modélisant à l'aide de l'héritage les employés d'une entreprise selon différents profiles (secrétaire, ingénieur, réceptionniste) et les implémentations exposées à la slide 7.



  - Nous allons écrire une classe **Test** qui dispose d'une méthode *afficherSalaire()* et qui prend en paramètre un objet de type Employé. Mais nous verrons que nous pouvons aussi lui passer en paramètre des sous-classes de Employé pour qu'elle puisse avoir des comportements différents

3 Polymorphisme paramétrique de méthode – cas 1(2/2)

```java
package emp.entreprise;

public class Employe {

   public String nom;
   public String prenom;
   public final Double smic  = 1300.00d;

   public Double calculerSalaire() {
      return 1.1 * smic;
   }
}
```

```java
package emp.entreprise;

public class Ingenieur extends Employe {

   private float indicePoste = 2.0f;

   public Double calculerSalaire() {
      return indicePoste * super.smic;
   }
}
```

```java
package emp.entreprise;

public class Secretaire extends Employe {

   private float indicePoste = 1.6f;

   public Double calculerSalaire() {
     return indicePoste * super.smic;
   }
}
```

```java
package emp.entreprise;

public class Test {

   public void afficherSalaire(Employe emp) {
      System.out.println(emp.getClass().getSimpleName() + " : " +
                                 emp.calculerSalaire());
   }

   public static void main(String[] args) {

     Test test = new Test();

     // bloc1
     Employe emp = new Employe();
     test.afficherSalaire(emp);

     // bloc2
     Ingenieur ing = new Ingenieur();
     test.afficherSalaire(ing);

     // bloc3
     Secretaire sec = new Secretaire();
     test.afficherSalaire(sec);

   }
}
```

Interprétation :

> Dans la classe Test, la méthode *afficherSalaire()* prend en paramètre une classe *Employé*. Mais comme Employé est parent des classes *Ingénieur*, *Secrétaire* et *Réceptionniste* (non représentée dans l'exemple), il est possible de passer une instance de chacune de ses sous-classes de Employé lors de l'appel de la méthode *afficherSalaire()*, Cf . bloc1, bloc2 et bloc3

Constat :

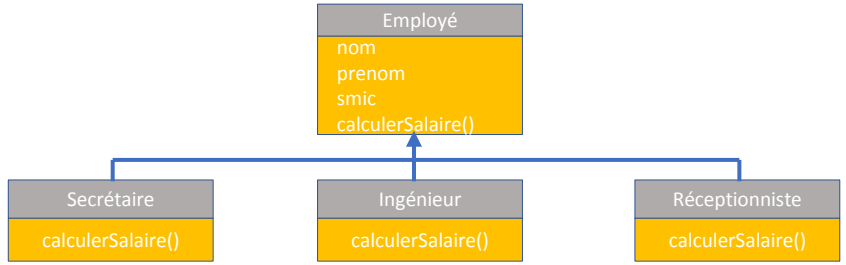
> Java est capable de détecter que l'objet effectivement passé en paramètre de la méthode *afficherSalaire(), du moment où ils ont été « typés » avec leurs véritables classes.* Il converti alors implicitement le type Employé attendu en le type effectif de l'objet passé en paramètre

Résultat affiché par le programme :

```
Employe : 1430.0000000000002
Ingenieur : 2600.0
Secretaire : 2080.0000309944153
```

3 Polymorphisme paramétrique de méthode - cas 2 (1/2)

- Cas d'une méthode ayant un paramètre de type une classe C, mais appelée avec un objet dont le type implicite est une sous-classe de C :
  > Sous réserve de compatibilité, Java permet de convertir un objet typé par une classe C en un objet dont le type est une sous-classe de C
- Pour cela, Java propose d'abord de tester la compatibilité de type avec l'instruction **instanceof** avant de réaliser la conversion
- Exemple :
  - Soit toujours le diagramme ci-dessous modélisant à l'aide de l'héritage les employés d'une entreprise selon différents profiles (secrétaire, ingénieur, réceptionniste) et les implémentations exposées à la slide 7.



  - Nous allons écrire une classe **Test** qui dispose d'une méthode *afficherSalaire()* et qui prend en paramètre un objet de type Employé. Nous allons lui passer différentes instances de Secrétaire, Ingénieur, Réceptionniste cachés derrière le type Employé et implémenterons la méthode afin qu'elle teste, détecte et convertisse explicitement l'objet passé. Ce qui influence la méthode effectivement appelée

3 Polymorphisme paramétrique de méthode – cas 2 (2/2)

```java
package emp.entreprise;

public class Employe {

   public String nom;
   public String prenom;
   public final Double smic  = 1300.00d;

   public Double calculerSalaire() {
      return 1.1 * smic;
   }
}
```

```java
package emp.entreprise;

public class Ingenieur extends Employe {

    private float indicePoste = 2.0f;

    public Double calculerSalaire() {
       return indicePoste * super.smic;
    }
}
```

```java
package emp.entreprise;

public class Secretaire extends Employe {

    private float indicePoste = 1.6f;

    public Double calculerSalaire() {
      return indicePoste * super.smic;
    }
}
```

```java
package emp.entreprise;

public class Test {

   public void afficherSalaire(Employe emp) {

      if(emp instanceof Ingenieur) {

           Ingenieur ing = (Ingenieur) emp;
           System.out.println(ing.getClass().getSimpleName() +
                              " : " + ing.calculerSalaire());
      }else if(emp instanceof Secretaire) {

           Secretaire sec = (Secretaire) emp;
           System.out.println(sec.getClass().getSimpleName() +
                              " : " + sec.calculerSalaire());
      }else {
           System.out.println(emp.getClass().getSimpleName() +
                              " : " + emp.calculerSalaire());
      }
   }

   public static void main(String[] args) {
      Test2 test = new Test2();

      // bloc1
      Employe emp1 = new Employe();
      test.afficherSalaire(emp1);

      // bloc2
      //on cache l'objet Ingenieur dans une référence emp2 de type Employe
      Employe emp2 = new Ingenieur();
      test.afficherSalaire(emp2);

      // bloc3
      //on cache l'objet Secretaire dans une référence emp3 du parent Employe
      Employe emp3 = new Secretaire();
      test.afficherSalaire(emp3);
   }
}
```

Interprétation :

> Dans la classe Test, la méthode *afficherSalaire()* prend en paramètre une objet *Employé*. Mais sachant que Employé est super-classe d'autres classes, par principe de compatibilité, il peut valablement représenter ses sous-classes et cacher l'effectivité du vrai type de l'objet passé en paramètre. Pour détecter cela, on vérifie explicitement le type de l'objet passé et on le converti explicitement. Ainsi, la méthode *main()*, peut appeler *afficherSalaire()* avec les objets *emp1*, *emp2* et *emp3* tous de type *Employé*, mais qui sont différents dans le fond. Cf. bloc1, bloc2, bloc3.

Constat :

> Java permet d'appeler une méthode et lui passer pour le même paramètre plusieurs objets fondamentalement différents, à condition d'avoir un ancêtre commun

Résultat affiché par le programme :

```
Employe : 1430.0000000000002
Ingenieur : 2600.0
Secretaire : 2080.0000309944153
```

1 Intérêt et définitions

2 Polymorphisme par redéfinition

3 Polymorphisme paramétrique de méthode

4 Polymorphisme par surcharge

3 Polymorphisme par surcharge

- Ce n'est ni plus ni moins que la notion déjà abordée au chapitre 9 qui donne la possibilité à une classe de définir en son sein plusieurs méthodes de même nom, mais avec des signatures différentes. C'est ce qu'on appelle l'**overloading**.
  - > Il est donc possible avec la même instance d'invoquer, en trompe l'œil, des méthodes différentes de même nom
  - > C'est le seul modèle de polymorphisme qui n'est pas forcément basé sur l'héritage

- Exemple :

```
public class Operation {

    //Division entière
    public int diviser(int num, int denom) {
        return num / denom;
    }

    //Division décimale
    public double diviser(double num, double denom) {
        return num / denom;
    }

    public static void main(String[] args) {
      Operation op = new Operation();

      System.out.println("Division entière : " + op.diviser(247, 3));
      System.out.println("Division décimal : " + op.diviser(606.90, 3.0));
    }

}
```

Interprétation :

> La classe *Operation* a deux méthodes diviser(). L'une qui prend deux entiers et donc effectue naturellement une division entière, et l'autre qui prendre deux nombres réels et effectue une division décimale. Ces deux méthodes ont le même nom, mais sont fondamentalement différentes

Constat :

> L'appelle à l'une ou l'autre de la méthode diviser() à travers le même objet aura un comportement différent selon les paramètres passé en fonction des paramètres passer en entrée

Résultat affiché par le programme :

Division entière : 82
Division décimale : 202.29999999999998


SUCCESS