L’héritage

L'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- Une classe parent ou superclasse
- une classe fille ou sous classe qui hérite de sa classe mère

Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

L'héritage

L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super classes et de sous classes. Une classe qui hérite d'une autre est une **sous-classe** et celle dont elle hérite est une **super classe**. Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.

Object est la classe parent de toutes les classes Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

L'héritage

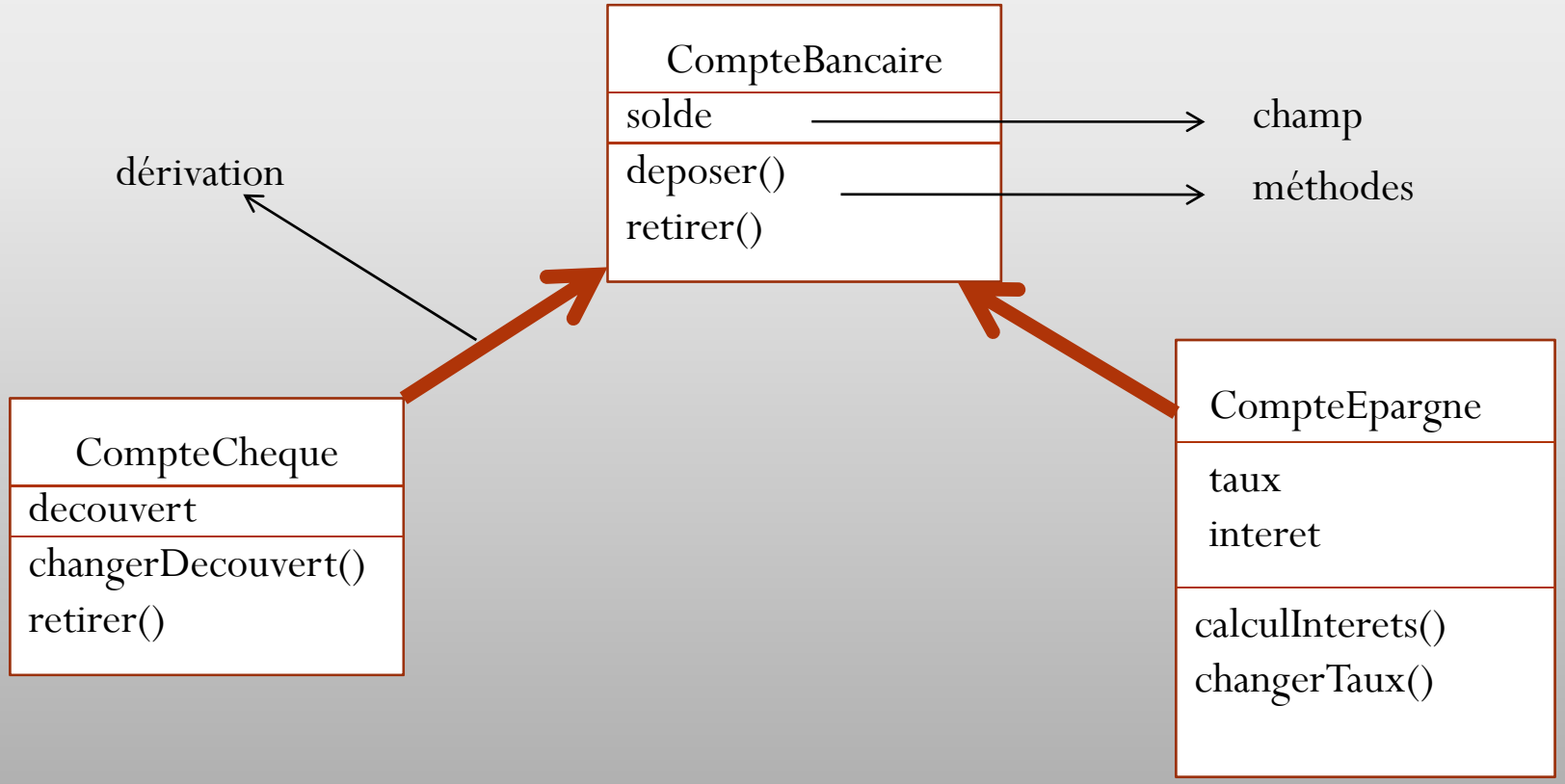
Considérons la classe **CompteBancaire** ci-dessous:

```
public class CompteBancaire {
     double solde ;

     public CompteBancaire (double solde ){
             this.solde = solde;
     }
     void deposer (double montant){
            solde += montant;
     }
     void retirer (double montant){
            if (solde >= montant)
            solde -= montant ;
     }
}
```

L'héritage

On se propose de spécialiser la gestion des comptes. On crée alors une classe CompteCheque et une autre classe CompteEpargne qui dérivent de la classe CompteBancaire.



L'héritage

```
public class CompteCheque extends CompteBancaire {
   private double decouvert ;

   public CompteCheque(double solde, double decouvert){
            super (solde);
            this.decouvert = decouvert ;
   }
   void retirer (double montant){ // methode redéfinie
            if (solde + decouvert >= montant )
                  solde -= montant ;
   }
//.....
}
```

on utilise le mot clé **extends** pour signaler au compilateur que la classe `CompteCheque` dérive de la classe `CompteBancaire`.

On rajoute un nouveau champ `decouvert` et on redéfinit la méthode `retirer()`

L'héritage

```
public class CompteEpargne extends CompteBancaire {
     private double  taux ;       //on rajoute un champ taux

     public CompteEpargne (double solde, double taux) {
            super (solde);
            this.taux = taux;
     }
     // pas de methode retirer
     //……
}
```

L'héritage

La méthode retirer() n'est pas définie dans la classe CompteEpargne mais on peut bien faire:

```
CompteEpargne ce = new CompteEpargne ( 20000, 0.05) ;
ce.retirer (10000) ;
```

Un objet d'une classe dérivée accède aux membres publics de sa classe de base, exactement comme s'ils étaient dans la classe dérivée elle-même.

Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base.

Construction des objets dérivés

La construction d'un objet dérivé est intégralement prise en compte par le constructeur de la classe dérivée.

Par exemple, le constructeur de `CompteCheque`:

- ✓ initialise le champ `decouvert` (déjà membre de `CompteCheque`);
- ✓ appelle le constructeur de `CompteBancaire` pour initialiser le champ `solde` (hérité)

Dans l'initialisation des champs d'un objet dérivé , il est fondamental et très important de respecter une contrainte majeure: si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur et ce dernier est désigné par le mot clé **`super`**.

Redéfinition des méthodes

```
public class CompteBancaire {
      double solde ;
      // ……
      void retirer (double montant){
               if (solde >= montant ;
               solde -= montant ;
      }
      void imprimeHistorique( ){
               System.out.print (" solde  = "+solde );
}
```

Redéfinition des méthodes

```
public class CompteCheque extends CompteBancaire {
      double decouvert;

      // méthode redéfinie
      void retirer (double montant){
              if (solde + decouvert >=montant ;
                    solde -= montant ;
      }
      void imprimeHistoriqueCheque(){
              System.out.print ("solde =" +solde + " " +
                                    "decouvert =" +decouvert);
      }
}
```

Redéfinition des méthodes

Avec :

```
CompteBancaire cb; CompteCheque cc;
```

`cb.retirer (20000)` appelle la méthode retirer de `CompteBancaire`.

`cc.retirer ( 20000)` appelle la méthode retirer de `CompteCheque`.

On se base tout simplement sur le type de l'objet pour déterminer la classe de la méthode appelée.

Pour bien voir l'intérêt de la redéfinition des méthodes, examinons la méthode **`imprimeHistorique`** de la classe **`CompteBancaire`** qui permet d'afficher le solde pour un compte et la méthode **`imprimeHistoriqueCheque`** de la classe **`CompteCheque`** qui affiche non seulement le solde (qui est un membre hérité) mais aussi le **`decouvert`**.

Dans cette dernière, il y a une information qui est déjà prise en compte dans la méthode `imprimeHistorique`. La situation précédente peut être améliorée de cette façon:

Redéfinition des méthodes

```
public class CompteCheque extends CompteBancaire {
double decouvert;

// méthode redéfinie
void retirer (double montant){
        if (solde + decouvert >=montant ;
              solde -=montant ;
}
void imprimeHistorique(){
       super.imprimeHistorique() ; //super obligatoire
        System.out.print ("solde =" +solde + " " +
                                "decouvert =" +decouvert);
}
}
```

Règles sur la redéfinition

Si une méthode d'une clase dérivée a la même signature qu'une méthode d'une classe ascendante :

- ✓ les valeurs de retour des deux méthodes doivent être exactement de même type,
- ✓ le droit d'accès de la méthode de la classe dérivée ne doit pas être plus élevé que celui de la classe ascendante,
- ✓ la clause throws de la méthode de la classe dérivée ne doit pas mentionner des exceptions non mentionnées dans la clause throws de la méthode de la classe ascendante(la clause throws sera étudiée ultérieurement).

Si ces trois conditions sont remplies, on a affaire à une redéfinition. Sinon, il s'agit d'une erreur.

- ➢ Lorsqu'une méthode d'une classe dérivée a le même nom qu'une méthode d'une classe ascendante, avec une signature différente, on a affaire à une **surdéfinition**
- ➢ Une méthode de classe (static) ne peut pas être redéfinie dans une classe dérivée.

Duplication de champs

Bien que cela soit d'un usage peu courant, une classe dérivée peut définir un champ portant le même nom qu'un champ d'une classe de base ou d'une classe ascendante . Ce phénomène est appelé **duplication de champs**.

```
class A{
      public int resultat;
      // instructions
}


class B extends A{

      /* le champ resultat est dupliqué */
      public int resultat ;

      float calcul( int n){
              return resultat + super.resultat+n;
      }
}
//On utilise super pour accéder au champ resultat de la classe A
```

Surclassement

Une classe B qui hérite d'une classe A peut être vue comme un sous-type (sous -ensemble) du type défini par la classe A.

Exemple: un `CompteCheque` est un `CompteBancaire`. L'ensemble des comptes chèques est inclus dans l'ensemble des comptes bancaires.

Tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.

Cette relation est directement supportée par le langage JAVA.

A une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B (surclassement ou upcasting) .

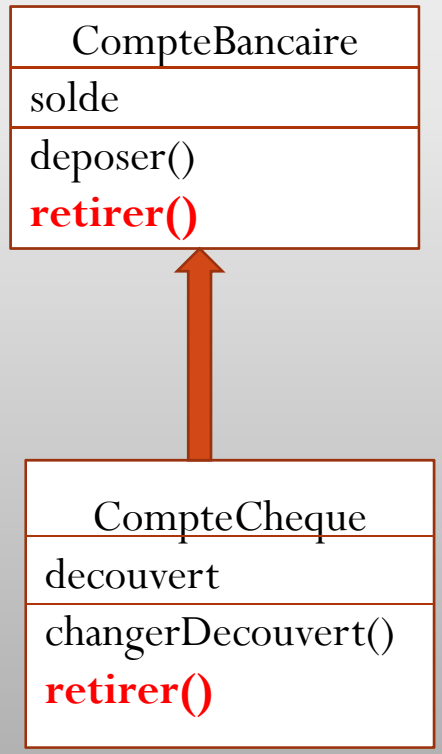
```
CompteBancaire cb;
cb = new CompteCheque(100000, 50000);
```

Plus généralement, à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte du type de la référence.

Surclassement

Lorsqu'un objet est surclassé il est vu comme un objet du type de la référence utilisée pour le désigner: ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence.

```
CompteCheque cc = new CompteCheque(100,50);
CompteBancaire cb;
cb = cc;    //surclassement
cb.retirer(50);
cc.retirer(25);
cb.deposer(500);
cc.deposer(250);
cb.changerDecouvert(); //erreur de compilation
cc.changerDecouvert();
```



Liaison dynamique: Résolution des message

Lorsqu'une méthode d'un objet est accédée au travers d'une référence surclassée, c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est réellement invoquée et donc exécutée.

Les messages sont résolus à l'exécution.

- ✓ La méthode à exécuter est déterminée à l'exécution (run–time) et non pas à la compilation .
- ✓ La méthode définie pour le type réel de l'objet recevant le message est appelée et non pas celle définie pour son type déclaré.

Ce mécanisme est désigné sous le terme de liaison dynamique (dynamic binding, late binding ou run-time binding).

Le polymorphisme

Le surclassement et la liaison dynamique (ligature dynamique) servent à mettre en œuvre le polymorphisme.

Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, à l'image de l'eau qui peut être à l'état liquide, solide ou gazeux.

En programmation Objet, on appelle polymorphisme:

- ✓ le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe
- ✓ le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.

Remarques:

- ✓ Une méthode déclarée **final** ne peut pas être redéfinie dans une classe dérivée.
- ✓ Une classe déclarée **final** ne peut plus être dérivée .

Classes abstraites

Une classe abstraite est une classe qui ne peut instancier aucun objet. Une telle classe ne peut servir qu'à une dérivation.

Dans une classe abstraite on peut retrouver:

- ✓ des champs et des méthodes dont héritera toute classe dérivée
- ✓ des méthodes abstraites (avec signature et type de retour)

Les classes abstraites facilitent largement la Conception Orientée Objet. En effet, on peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

- ✓ soit sous forme d'une implémentation complète de méthodes(non abstraites) et de champs(privés ou non) lorsqu'ils sont communs à toutes ses descendantes ,
- ✓ soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existent dans toute classe dérivée instanciable.

Classes abstraites

On définit une classe abstraite en Java en utilisant le mot clé **abstract** devant le nom de la classe:

```
abstract class Forme {
     // champs usuels et constantes
     // méthodes définies ou méthodes non définies
     // MAIS AU MOINS UNE METHODE NON DEFINIE
}
```

Avec cette classe on peut écrire :

```
Forme f ;// declaration d'une reference de type forme
```

Par contre on ne peut écrire :

```
f = new Forme ( );// INTERDIT
```

Classes abstraites

Si on se retrouve dans la situation suivante:

```
class Rectangle extends Forme{
     /* ici on redefinit TOUTES les méthodes abstraites
        héritiées de  Forme
     */
}
```

Alors on peut instancier un objet de type Rectangle et placer sa référence dans une variable de type Forme (polymorphisme):

```
Forme f = new Rectangle ();// OK car polymorphisme
```

Classes abstraites

```
abstract class Forme{
      public abstract double perimetre( ) ;
}// fin de Forme

class Circle extends Forme {
      private double r;
      //…constructeur à definir
      public double perimetre () { return 2 * Math.PI * r ; }
}//fin de Circle

class Rectangle extends Forme {
      private double longu, larg;
      //constructeur à définir
      public double perimetre() { return 2 * (longu + larg); }
}//fin de Rectangle
```

Classes abstraites

```
/* dans le main d'une classe de test */
Forme [] formes = {new Circle(2), new Rectangle(2,3),
                                                  new Circle(5)};
double sommeDesPerimetres = 0;
for ( int i=0; i < formes.length; i++)
     sommeDesPerimetres += formes[i].perimetre ( );
```

Classes abstraites

Remarques:

- ✓ Une classe abstraite est une classe ayant au moins une méthode abstraite.
- ✓ Une méthode abstraite ne possède pas de définition.
- ✓ Une classe abstraite ne peut pas être instanciée (`new`).
- ✓ Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite.
- ✓ Une méthode abstraite ne doit pas être déclarée `final`, puisque sa vocation est d'être redéfinie. De même une classe abstraite ne doit pas être `final`, `private` ou `static`.
- ✓ Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base(elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite.
- ✓ Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites

Interfaces

- ✓ Une interface correspond à une classe où TOUTES les méthodes sont abstraites.
- ✓ Une classe peut implémenter (implements) une ou plusieurs interfaces tout en héritant (extends) d'une classe.
- ✓ Une interface peut hériter (extends) de plusieurs interfaces.
- ✓ Une interface n'est composée que de constantes et de méthodes abstraites

Interfaces

```
interface Operation{

      /*constantes*/
      public double nombre = 100 ;
      final float x = 1000;

      /* que des methodes abstraites*/
      public  double addition();
      public  float division(float a, float b);
      //private et protected  interdit
      abstract double multiplication (); //abstract non obligatoire

}//fin de Operation
```

Dans la définition d'une interface seuls les droits d'accès public et le droit de paquetage (vide) sont autorisés.

Interfaces

Les interfaces sont faites pour être implémenter.

Une contrainte dans l'implémentation d'une interface: il faut obligatoirement redéfinir toutes les méthodes de l'interface aucune définition de méthode ne peut être différée comme dans le cas des classes abstraites.

Lorsque qu'une interface est implémentée, seules les méthodes sont redéfinies, les constantes sont directement utilisables .

Interfaces

```
abstract class Forme{ public abstract double perimetre() ; }

class Circle extends Forme  implements Dessinable{
      private double r;
      public double perimetre(){ return 2 * Math.PI * r ; }
      void dessiner ( ){ //instructions de dessin d'un cercle}
}//fin de Circle

class Rectangle extends Forme implements Dessinable {
      private double longu, larg;
      public double perimetre() { return 2 * (long + larg); }
      void dessiner ( ){ //instructions de dessin d'un rectangle}
}//fin de Rectangle
```

Interfaces

```
/* dans le main d une classe de test */
Dessinable [] dessins = {new Circle (2), new Rectangle(2,3),
                                                  new Circle(5)};
for ( int i = 0; i < dessins.length; i++)
     dessins[i].dessiner ( );
```

Interfaces

Diverses situations avec les interfaces:

On dispose de deux interfaces :

```
interface I1 {…}
interface I2 {…}
```

On peut avoir:

```
interface I3 extends I1 {…} //dérivation dune interface

class A implements I2 {…}//implémenter une seule interface

class B implements I1, I2, I3 {…} //implémenter plusieurs
                                  //interfaces

class C extends A implements I3 {…} // dérivation d une classe
                              //et implementation d une interface
```

Application

Réaliser une classe Etudiant disposant des fonctionnalités suivantes:

On suppose qu'il y a trois sortes d'étudiants, les étudiants nationaux, les étudiants étrangers et les étudiants sportifs. Un étudiant étranger du premier type est caractérisé par son nom, son prénom et son âge; pour un étudiant du second type on ajoute le pays d'origine et pour les étudiants du troisième type on rajoute le sport pratiqué.

On considère qu'on a une classe Etudiant et deux autres classes EtudiantEtranger et EtudiantSportif qui dérivent toutes deux de la première classe.

On prévoit donc :

- un constructeur pour la classe Etudiant
- un constructeur pour la classe EtudiantEtranger(appel du premier constructeur)
- un constructeur pour la classe EtudiantSportif (idem)
- une méthode affiche() pour toutes les classes(redéfinir la méthode dans les deux classes)

On crée maintenant une classe GroupeTD qui permet de lier un étudiant quelconque à un groupe de TD. On prévoit une méthde ajouterEtudiant(Etudiant e) qui permet d'ajouter un étudiant à un groupe de TD. Pour ajouter un étudiant il faut s'assurer que le groupe ne dépasse pas les 30 étudiants.

Créer une méthode afficherListe pour afficher tous les étudiants d'un groupe de TD.