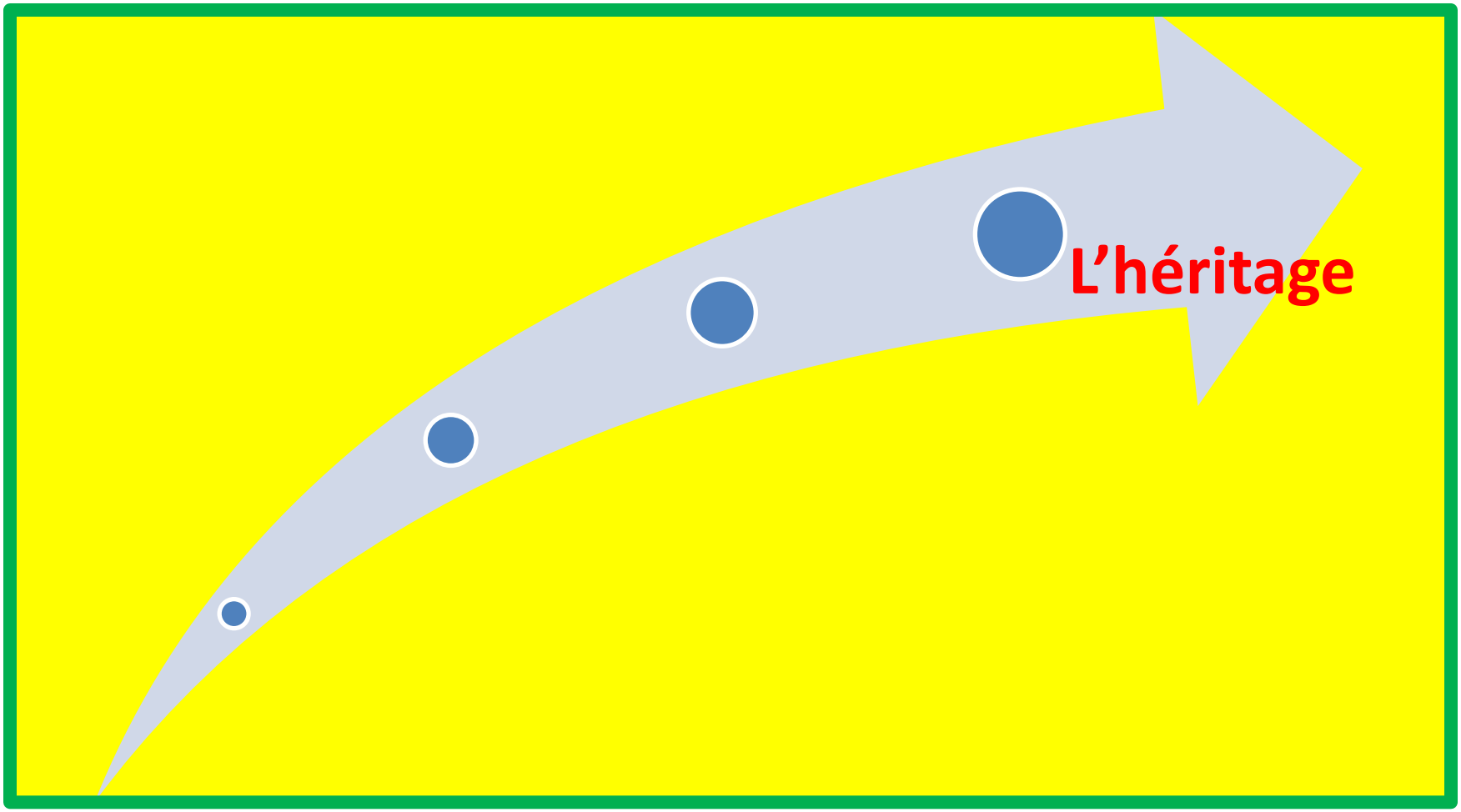
Cours de Programmation Orientée Objet JAVA

Demba SOW

Docteur en Codage Cryptologie Algèbre et Applications

L.A.C.G.A.A.

F.S.T. / U.C.A.D.


L’héritage

L'héritage en Java

Héritage → possibilité de réutilisation des composants logiciels.

Héritage = définir une **classe dérivée** (nouvelle classe) d'une **classe de base** (déjà existante) .

La classe dérivée hérite donc de toutes les fonctionnalités de sa classe de base: **champs** et **méthodes**. Elle peut avoir des *caractéristiques propres et redéfinir des caractéristiques héritées*.

Le concept d'héritage

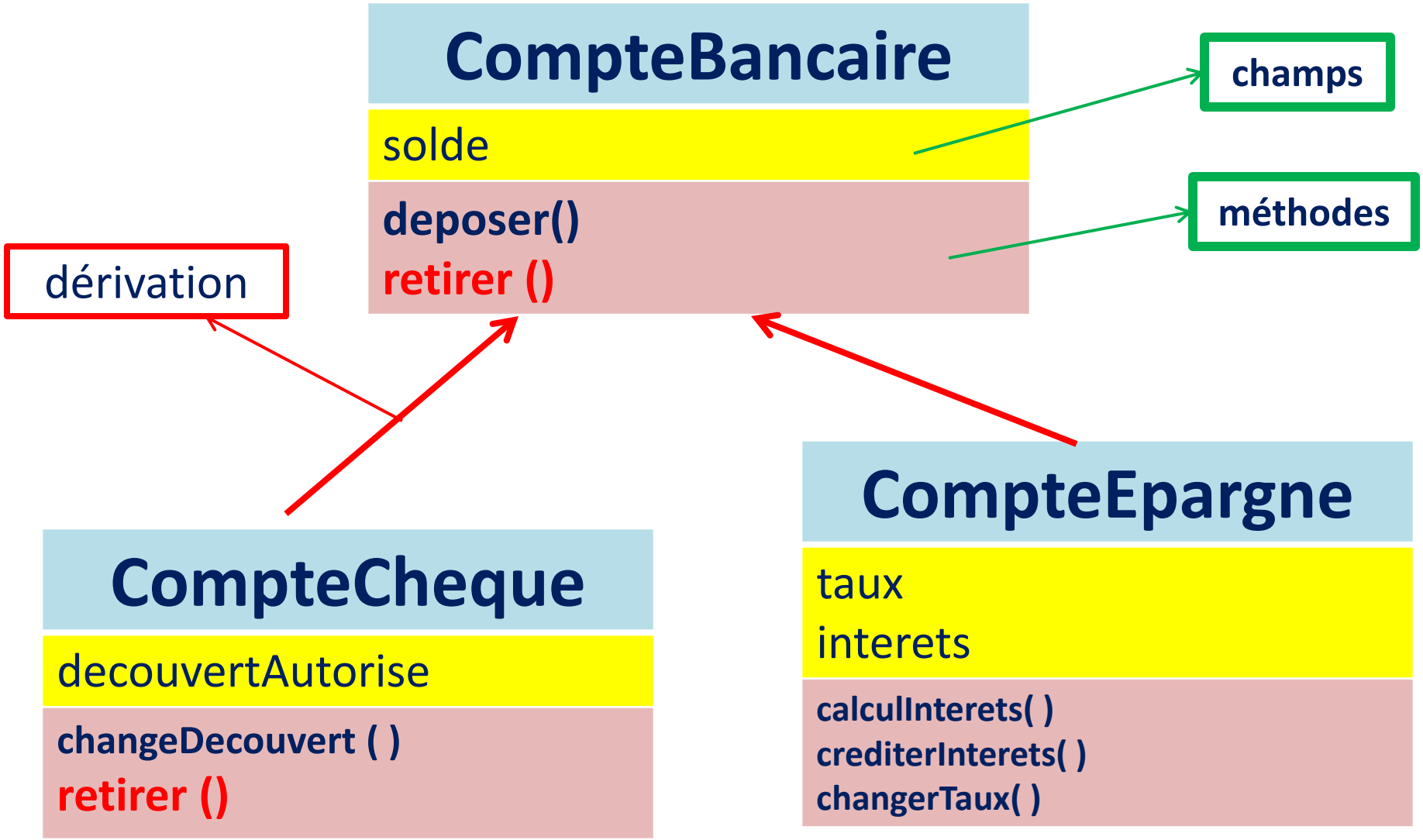
Supposons disposer d'une classe CompteBancaire:

```
package sow.compte;

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

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**.

Le concept d'héritage



Le concept d'héritage

```
package sow.compte;

public class CompteCheque extends CompteBancaire {

        double decouvertAutorise;

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

En Java, on utilise la mention **extends** pour signaler au compilateur que la classe **CompteCheque** dérive de la classe **CompteBancaire**.

Ici, on rajoute un nouveau champ **decouvertAutorise**. Et la méthode **retirer** est redéfinie.

Le concept d'héritage

```
package sow.compte;
public class CompteEpargne extends CompteBancaire {
                double   taux;
 CompteEpargne (double solde, double taux)
        { super (solde);
          this. taux = taux;
        }
       // pas de méthode retirer
//……
}
```

Ici, on rajoute un nouveau champ **taux**.

Accès aux membres {champs et méthodes} de la classe de base.

Avec l'instruction :

CompteEpargne ce = new CompteEpargne (20000, 0.05) ;

On peut bien faire:

ce.retirer (10000) ;

malgré que la méthode retirer n'est pas définie dans la classe CompteEpargne.

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 decouvertAutorise (déjà membre de *CompteCheque);***
- **appelle le constructeur de CompteBancaire pour initialiser le champ solde (hérité)**

dans l'initialisation de 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**.

Quelques remarques

Nous avons mentionné au **chapitre 5** qu'il est possible d'appeler dans un constructeur un autre constructeur **de la même classe**, en utilisant le mot clé ***this*** comme nom de méthode . Comme celui effectué par ***super***, cet appel doit correspondre à la première instruction du constructeur .

Dans ces conditions, on voit bien qu'il n'est pas possible d'exploiter les deux possibilités en même temps . **Autrement dit, dans un constructeur d'une classe dérivée il n'est pas possible d'appeler en même temps un constructeur de la même classe et un constructeur d'une classe de base.**

L'appel par ***super*** *ne concerne que le constructeur de la classe de base de niveau* immédiatement supérieur (vu qu'1 classe peut dériver d'une classe qui aussi dérive d'1autre.

Redéfinition de membres (1/4)

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

```
package sow.compte;
class CompteCheque extends CompteBancaire {
double decouvertAutorise;
// méthode redéfinie
  void retirer (double montant)
      { if (solde + decouvertAutorise >=montant ;
         solde -= montant ;
      }
void imprimeHistoriqueCheque( )
{ System.out.print (" solde =" +solde + " " +
"decouvertAutorise="
+decouvertAutorise);
  }
}
```

Redéfinition de membres (2/4)

Avec :
CompteBancaire cb; CompteCheque cc;
l'appel : cb.retirer (20000);
appelle la méthode retirer de CompteBancaire.
l'appel : 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 decouvertAutorise.
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 de membres (3/4)

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

```
package sow.compte;
class CompteCheque extends CompteBancaire {
double decouvertAutorise;
// methode redefinie
  void retirer (double montant)
      { if (solde + decouvertAutorise >=montant ;
         solde -= montant ;
      }
 void imprimeHistoriqueCheque( )
   {    imprimeHistorique( )
 System.out.print (" et le decouvertAutorise
="+decouvertAutorise);
   }
 }
```

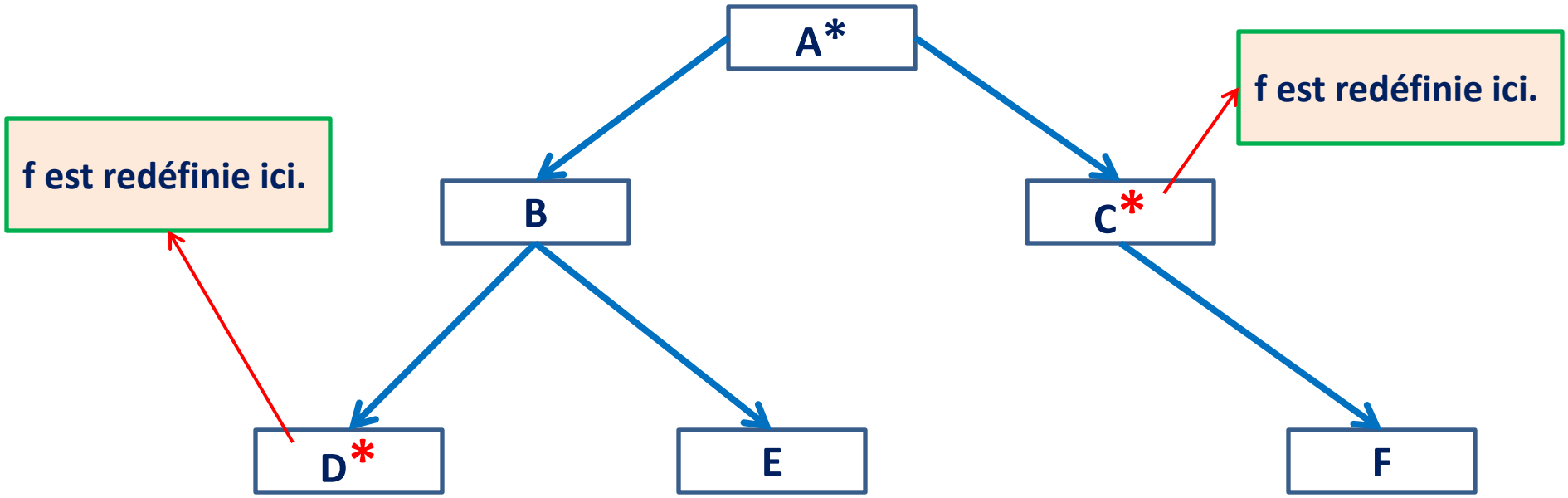
Redéfinition de membres (4/4)

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

```
package sow.compte;
class CompteCheque extends CompteBancaire {
double decouvertAutorise;
// methode redefinie
  void retirer (double montant)
      { if (solde + decouvertAutorise >=montant ;
         solde -= montant ;
      }
  void imprimeHistoriqueCheque( )
    {   super.imprimeHistorique( )
 System.out.print (" et le decouvertAutorise
="+decouvertAutorise);
    }
 }
```

super obligatoire.

Redéfinition et dérivation successive



L'appel de f conduira, pour chaque classe, à l'appel de la méthode indiquée en regard :

class A : méthode f de A	class D: méthode f de D
class B: méthode f de A	class E: méthode f de A
class C: méthode f de C	class F: méthode f de C

Surdéfinition et héritage

```
package sow.compte;

class Calcul {
  public double division ( int a) // 1
    { // instructions
    }
}

 class CalculXXX extends Calcul {
    public float division ( float a) // 2
      { // instructions
      }
}
```

```
Calcul c; CalculXXX cx;

int n; float p;

c.division (n);  // appel de 1

c.division (p); // erreur de compilation

cx.division (n); // appel de 1

cx.division (p); // appel de 2
```

La recherche d'une méthode acceptable ne se fait **qu'en remontant la hiérarchie d'héritage, jamais en la descendant** .C'est pourquoi l'appel **c.division (p);** ne peut être satisfait.

Contraintes sur la redéfinition (1/2)

En cas de redéfinition, Java impose seulement **l'identité des signatures** mais aussi du **type de la valeur de retour**

Valeurs de retour identiques

```
package sow.compte;
class CompteBancaire {
   double solde ;
   // ……..
        void   retirer (double  montant)
         { if (solde >=montant ;
            solde -= montant ;
         } }
```

Signatures identiques

```
package sow.compte;
class CompteCheque extends  CompteBancaire {
double decouvertAutorise;
// méthode redéfinie
     void   retirer (double  montant)
        { if (solde + decouvertAutorise >=montant )
          solde  -= montant ;
        }
}
```

Contraintes sur la redéfinition (2/2)

La redéfinition d'une méthode ne doit **pas diminuer les droits** d'accès à cette méthode.

Par contre, **elle peut les augmenter**.

```
class A
{ public void f ( int n){……}
}

class B extends A
{ // impossible de mettre private
private void f ( int n) {….}
}
```

```
class A
{ private void f ( int n){……}
}
class B extends A
{ // augmente les droits d accès: possible
public void f ( int n) {….}
}
```

Règles sur :
la Redéfinition et la Sur-définition

Si une méthode d'une classe 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.

Dans les autres cas, c'est-à-dire 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*** .

Quelques remarques

1. Une méthode de classe (***static***) *ne peut pas être redéfinie dans une classe dérivée.* Cette restriction va de soi puisque c'est le type de l'objet appelant une méthode qui permet de choisir entre la méthode de la classe de base et celle de la classe dérivé.
2. Les possibilités de redéfinition d'une méthode prendront tout leur intérêt lorsqu'elles seront associées au polymorphisme que nous allons étudié .

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.

Duplication de champs

```
class A
{ public int resultat;

  // instructions
}
```

```
class B extends A
{ /* le champ résultat est dupliqué */
public int resultat ;
   float calcul( int n)
      { return resultat + super.resultat +n;
      }
}
```

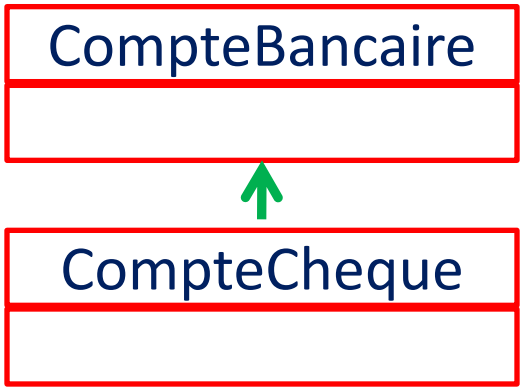
On utilise le mot clé **super** pour accéder à un champ de la super classe

Le Polymorphisme

Surclassement

La réutilisation de code est un aspect important de l'héritage, mais ce n'est peut être pas le plus important.
Un autre aspect fondamental est la relation qui relie une classe à sa super classe.

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.



Un CompteCheque est un CompteBancaire

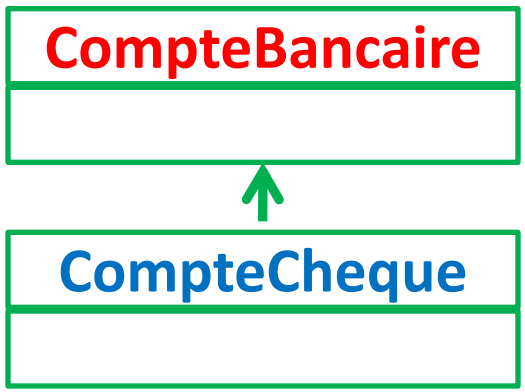
L'ensemble des comptes chèques est inclus dans l'ensemble des comptes bancaires

Surclassement

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.

« à 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) »

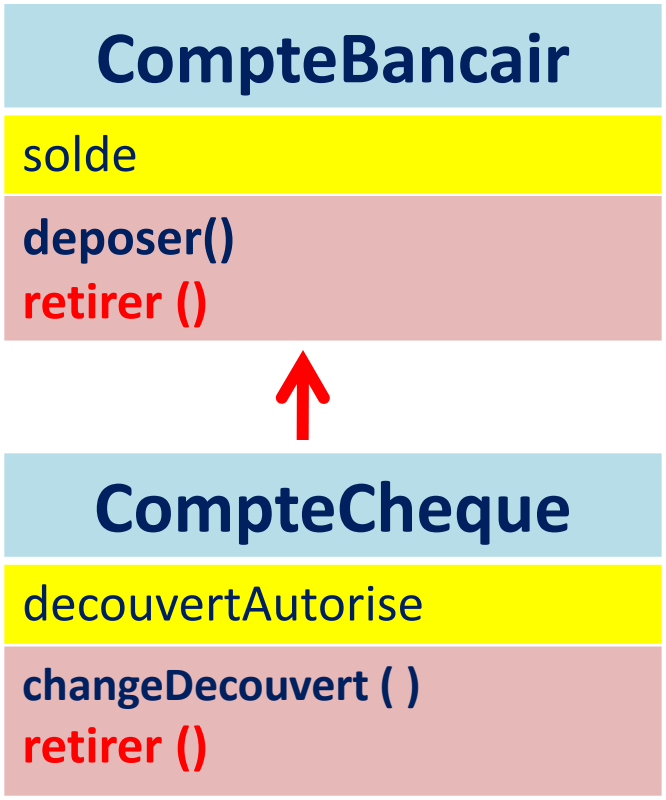



```
CompteBancaire cb;
cb = new CompteCheque( 100000, 50000 );
```

« 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 « sur-classé » 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.changeDecouvert ( );

cc.changeDecouvert( );
```

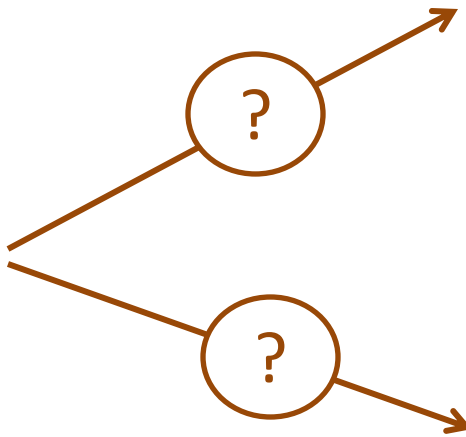
Liaison dynamique

Résolution des messages

« Que va donner cb.retirer (50) ? »

```
CompteBancaire cb = new
CompteCheque (500,100);
```

cb.retirer (50)




```
CompteBancaire
void retirer (double montant)
  { if (solde >=montant )
     solde  -= montant ;
   }
```

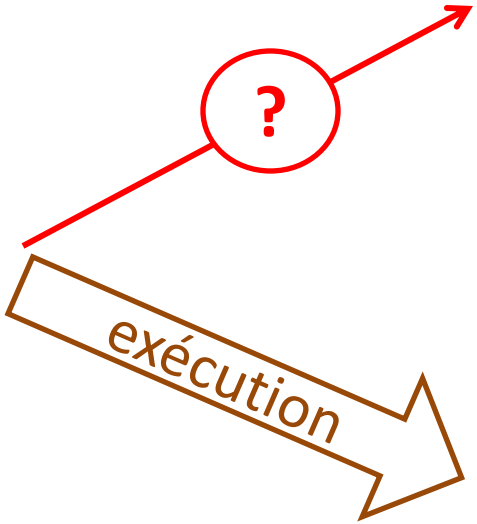
```
CompteCheque
void retirer (double montant)
{if (solde + decouvertAutorise >=montant )
solde -= montant ;
}
```

Liaison dynamique

Résolution des messages

```
CompteBancaire cb = new
CompteCheque (500,100);
```

cb.retirer (50)




```
CompteBancaire
void retirer (double montant)
  { if (solde >=montant )
     solde  -= montant ;
   }
```

```
CompteCheque
void retirer (double montant)
{if (solde + decouvertAutorise >=montant )
solde -= montant ;
}
```

Liaison dynamique

Résolution des messages

VRAIMENT A RETENIR

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.

Liaison dynamique

Mécanisme de résolution des messages

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 runtime binding)..

Liaison dynamique

Vérifications statiques

A la compilation: seules des vérifications statiques qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées.

La classe déclarée de l'objet recevant le message **doit posséder** une méthode dont la signature correspond à la méthode appelée.

A la compilation: il n'est pas possible de déterminer le type exact (réel) de l'objet récepteur du message.

Vérification statique : garantit dès la compilation que les messages pourront être résolus au moment de l'exécution. Elle permet de déterminer (figer) simplement la signature et le type de la valeur de retour de la méthode qui sera exécutée.

Liaison dynamique

Choix des méthodes, sélection du code

Le choix de la méthode à exécuter est effectuée statiquement *à la compilation* en fonction du type des paramètres.

La sélection du code à exécuter est effectuée dynamiquement *à l'exécution* en fonction du type effectif de l'objet récepteur du message.

Le Polymorphisme (1/4)

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

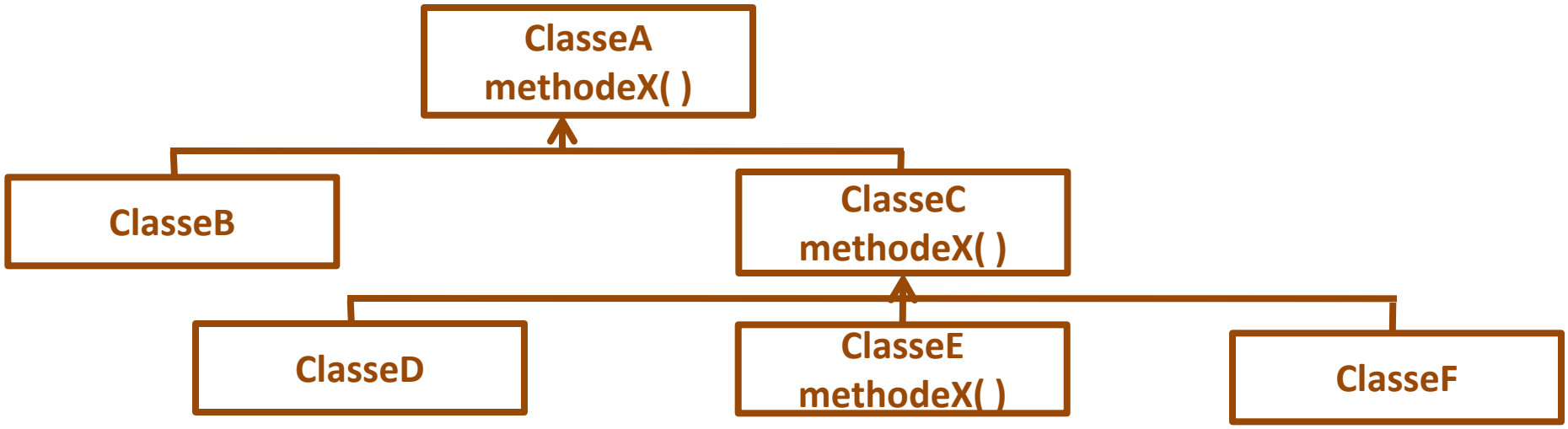
Le terme **polymorphisme** décrit la caractéristique d'un élément qui peut prendre plusieurs formes, à l'image de l'eau qui peut être à l'état liquide, solide ou gazeux.

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.*

Le **polymorphisme** est la troisième caractéristique essentielle d'un langage orienté Objet après l'abstraction des données (**encapsulation**) et l'**héritage**.

Le Polymorphisme (2/4)




```
classeA
objA ;
objA = …

objA.
methodeX( );
```

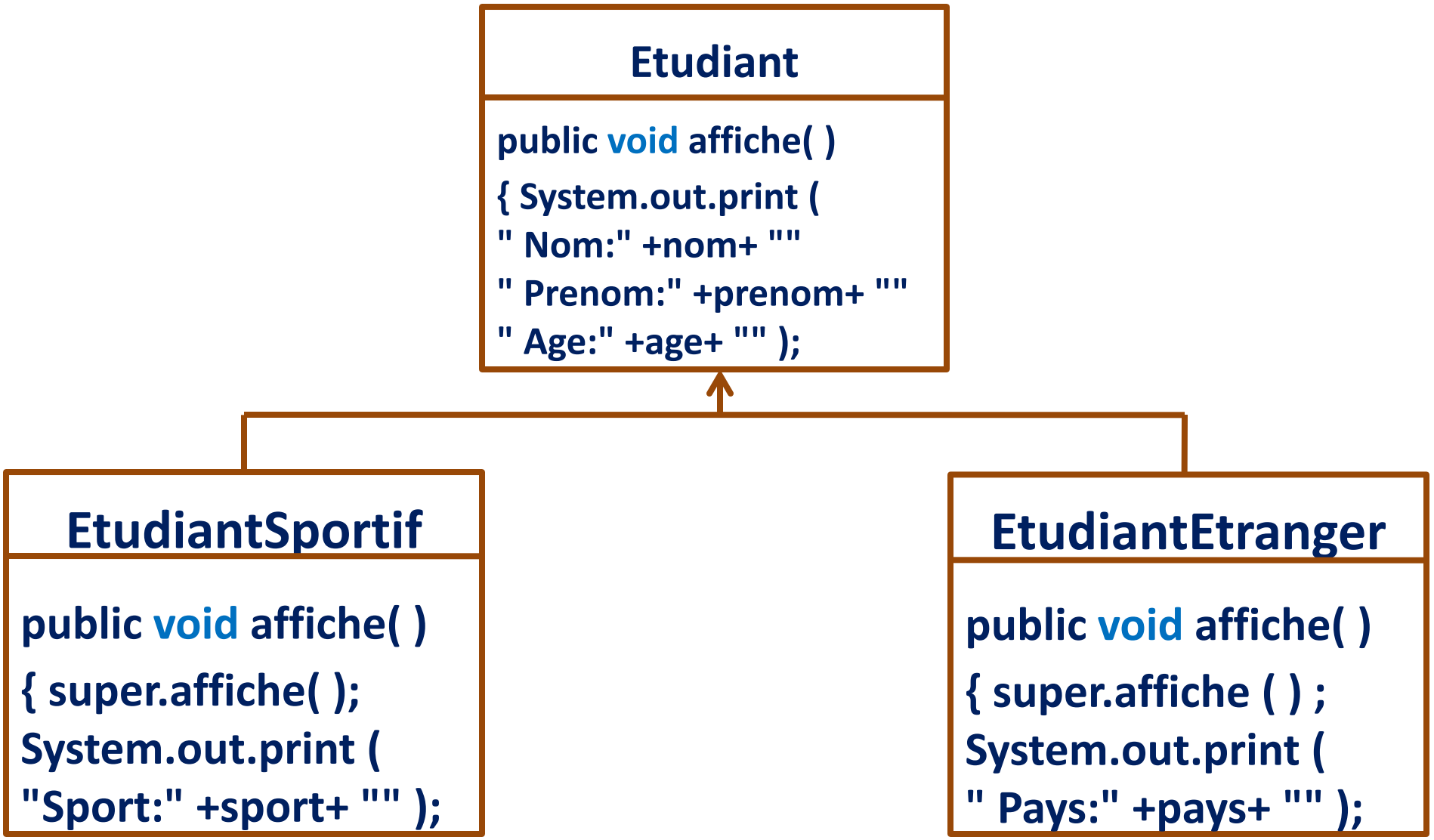
Surclassement :la référence peut désigner des objets de classes différentes (sousclasse de ClasseA)

+

Lien dynamique : le comportement est différent selon la classe effective de l'objet

Manipulation des objets de plusieurs classes par l' intermédiaire d'une classe de base commune.

Le Polymorphisme (3/4)



Le Polymorphisme (4/4)

```
public class GroupeTD {
 Etudiant liste [ ] = new Etudiant [30] ;
static int nbEtudiants ;
 public static void ajouter (Etudiant e)
  { if (nbEtudiants < liste.length)
     liste [nbEtudiants ++ ] = e ;
  }
 public static void afficherListe ( ) {
  for (int i = 0; i < nbEtudiants; i++)
   liste[i].affiche( );
 }
 public static void main (String args [ ] )
 { ajouter (new Etudiant (" Sene " , " Pierre " ,12));
   ajouter (new EtudiantSportif (" Fall " , " Fatou " ,5, " Natation "));
   ajouter (new EtudiantEtranger (" Ndiaye " , " Moussa " , 20 ," Senegal "));
   afficherListe ( ); } }
```

nbEtudiants = 3

À l'appel de afficherListe ( ) on a :

```
// appel la méthode de Etudiant
liste[0].affiche( );

// appel la méthode de EtudiantSportif
liste[1].affiche( );

// appel la méthode de EtudiantEtranger
liste[2].affiche( );
```

Polymorphisme : redéfinition et surdéfinition

```
public class ClasseXXX
{ public void methodeXXX ( double p ) // 1
   { …..
   }

public class ClasseYYY extends ClasseXXX
 {
   public void methodeXXX ( double k ) //redefinition 2
    { ……
    }
  public void methodeXXX ( float d ) //surdefinition 3
   { …….
   }
```

```
ClasseXXX objX = ….;
ClasseYYY objY =….;

float e;
// appel de 1
objX.methodeXXX ( e );
// appel de 3
objY.methodeXXX ( e );

objX = objY ;
// appel de 2 car poly…
objX.methodeXXX ( e );
```

La super classe Object

En Java, toute classe dérive *implicitement* de la classe Object. Elle est donc la classe de base de toute classe.

Une déclaration comme celle-ci :

```
class Point { …}
```

est équivalente à:

```
class Point extends Object { …}
```

Quelles en sont les conséquences ?

Référence de type Object

Une variable de type Object peut toujours être utilisée pour référencer n'importe quel objet d'un type quelconque.

« ceci peut être utile dans le cas où on manipule des objets sans connaître leur type exact ».

Object o;

Point p = new Point(….);

Pointcol pc = new Pointcol (…) ;

On peut faire:

o = p ;

o = pc ;

((Point)o).deplace (…); // OK

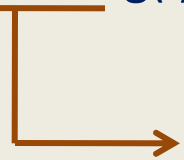
o.deplace (..) ; // erreur car deplace n'existe pas dans Object

Méthodes
equals et toString de Object

La méthode toString fournit une chaîne de caractères (ie un objet de la classe String) précisant:

- **le nom de la classe de l'objet concerné, suivi du signe @**
- **l'adresse de l'objet en hexadécimal,**

```
Point a = new Point (1,2) ;
System.out.println (" a = "+ a.toString( )) ;
```



a = Point@fc17aedf

NB: le plus souvent, vous aurez à redéfinir cette méthode.

Nous verrons cette méthode dans le module sur la classe String.

equals (1/3)

La méthode **equals** se contente de comparer les **adresses** de deux objets.
Avec :
Object o1 = new Point(1,2) ;
Object o2 = new Point(1,2) ;
o1.equals (o2) renvoie la valeur ***false***.
En effet cette méthode est définie dans **Object** comme ceci:

```
public boolean equals (Object o)
{ return this == o; // == teste les références
}
```

Vous pouvez aussi redéfinir cette méthode à votre convenance.

equals (2/3)

```
class Point
 {……
   boolean equals( Point p)
     { return ((p.x = = x) &&(p.y = = y));
     }
}
```

Avec :

Point a = new Point(1,2);

Point b = new Point(1,2) ;

a.equals (b); renvoie cette fois la valeur ***true***

Problème:

En revanche, avec :
Object o1 = new Point(1,2) ;
Object o2 = new Point(1,2) ;
l'expression *o1.equals(o2)* aura la valeur *false* car on aura utiliser la méthode *equals* de la classe *Object* et non celle de *Point (règles du polymorphisme).*

equals (3/3)

Il faut toujours prendre la peine de redéfinir convenablement une méthode. Pour résoudre le problème posé, on peut améliorer la redéfinition de la méthode comme ceci:

```
class Point
 {……
     boolean equals( Object o)
        { if ( ! o.instanceOf (Point))
          return false ;
          else Point p = (Point) o; // sousclassement (downcasting)
          return ((this.x = = p.x) &&(this.y = = p.y));
        }
}
```

Quelques définitions

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 ne peut instancier aucun objet.
Une telle classe ne peut servir qu'à une *dérivation (héritage).*

Dans une classe abstraite, on peut trouver :

- des **méthodes** et des **champs**, dont héritera toute classe dérivée.
- des **méthodes abstraites**,(avec *signature et type de valeur de retour).*

Le recours aux classes abstraites facilite 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.

Généralités

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 ; // déclaration d'une référence de type forme**
- **par contre on ne peut écrire : f = new Forme ( ); // INTERDIT**

Généralités

Maintenant si on se retrouve dans la situation suivante:

```
class Rectangle extends Forme
  {
        // ici on redéfinit TOUTES les méthodes héritées 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
```

Exemple

```
package sow;

abstract class Forme
  { public abstract double perimetre( ) ; }// fin de Forme
    class Circle extends Forme {private double r; //…constructeur à définir
                        public double perimetre ( ) { return 2 * Math.PI * r ; }
                    }//fin de Circle
    class Rectangle extends Forme { private double long, larg; //constructeur à définir
                        public double perimetre( ) { return 2 * (long + larg); }
                    }//fin de Rectangle
              /*  dans le main d'une classe de test    */
Forme [ ] formes = {new Circle(2), new Rectangle(2,3), new Circle(5)};
double somme_des_perimetres = 0;
for ( int i=0; i< formes.length; i++)
      somme_des_perimetres += formes[i].perimetre ( );
```

Quelques remarques importantes (1/2)

- 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.**
- **Une méthode abstraite ne doit jamais être déclarée private.**
- **Une méthode abstraite ne doit jamais être déclarée static.**

Quelques remarques importantes (2/2)

Dans l'entête, il faut obligatoirement mentionner le nom des paramètres formels (muets):

```
abstract class A
 {
   void g(int ) //nom d'argument muet obligatoire sinon erreur de compilation
 }
```

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.

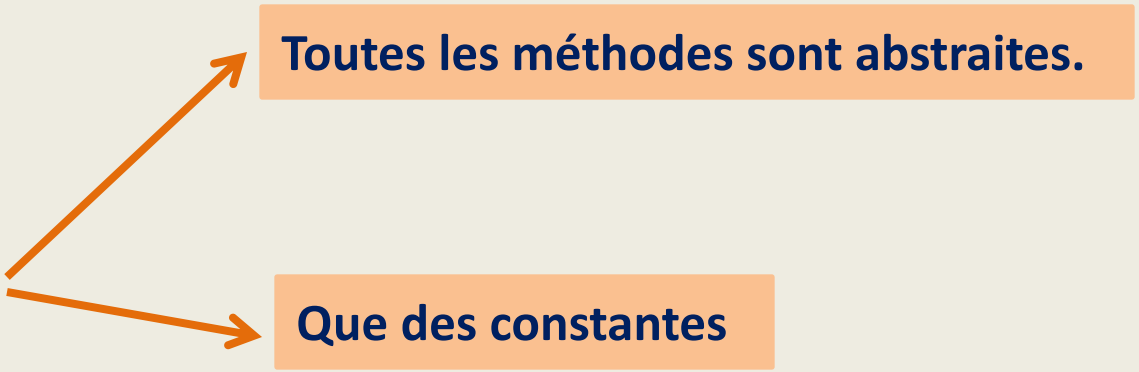
Interfaces

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 interface(s).

interface = classe



Généralités

```
package sow;

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
```

ici on ne peut pas mettre.

Les modificateurs private et protected sont interdits.
Toute variable déclarée ne peut être qu'une constante (donc ayant une valeur).

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

Utilisation d'une interface

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 que vous implémentez une interface, vous ne redéfinissez que les méthodes de l'interface, les constantes sont directement utilisables (vous n'avez pas besoin de les mentionner dans la classe qui implémente).

Exemple de mise en oeuvre

```
package sow;

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

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

Diverses situations avec les interfaces

On dispose de deux interfaces :

```
interface I1 {…}

interface I2 {…}
```

Le fait de pouvoir implémenter plusieurs interfaces peut résoudre le problème de la dérivation multiple connue dans les autres langages objets comme (C++)

Vous pouvez avoir:

```
interface I3 extends I1 { …..} //derivation dune interface

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

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

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

Les classes enveloppes

Il existe des classes nommées ***Boolean, Byte, Character, Short, Integer, Long, Float*** et ***Double*** , destinées à manipuler des valeurs de type primitif en les encapsulant dans une classe .

Cela permet de disposer de méthodes et de compenser le fait que les types primitifs ne soient pas des classes .

Toutes ces classes disposent d'un constructeur recevant un argument d'un type primitif :

```
Integer objInt = new Integer (5) ; // objInt contient la référence à un
                                   // objet de type Integer encapsulant la valeur 5
```

Elles disposent toutes d'une méthode de la forme ***xxxValue()*** (où **xxx** représentant le nom du type primitif) qui permet de retrouver la valeur dans le type primitif correspondant :

Exemple

Integer objet_n = new Integer (12) ;

Double objet_x = new Double (5.25) ;

int n = objet_n **.intValue ( ) ;** // objet_n contient 12

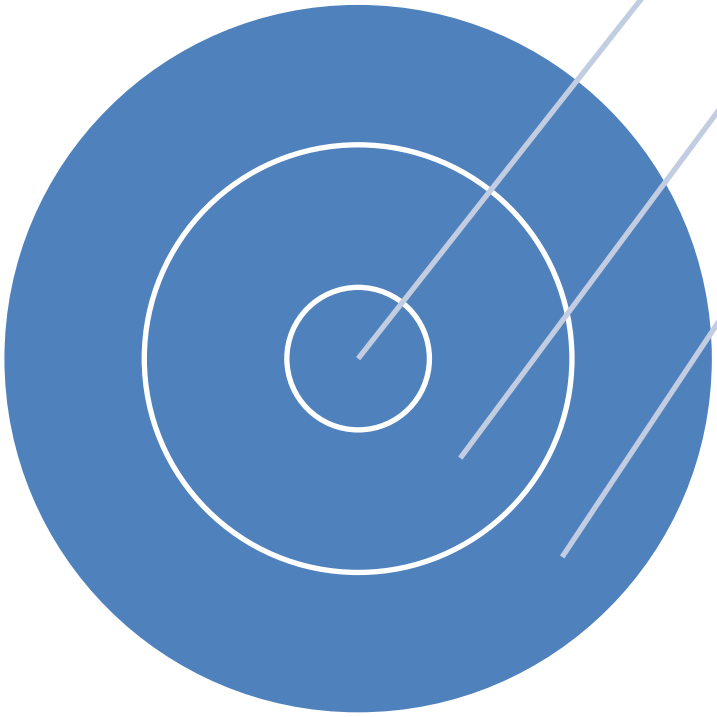
double x = objet_x **.doubleValue ( ) ;** // objet_x contient 5.25

Nous verrons aussi dans le chapitre consacré aux chaînes qu'elles disposent

d'une méthode ***toString*** effectuant la conversion de la valeur qu'elles

contiennent en une chaîne, ainsi que d'une méthode de la forme

parseXXX *permettant de* convertir une chaîne en un type primitif .



Fin

du

chapitre