L'héritage

1. Introduction

L'héritage est un mécanisme qui facilite **la réutilisation du code** et **la gestion de son évolution**. Grâce à l'héritage, les objets d'une classe ont **accès aux données et aux méthodes** de **la classe parent** et peuvent les étendre.
Les sous classes peuvent **redéfinir** (voir section polymorphisme) les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent.
Les méthodes sont **redéfinies** (voir section polymorphisme) avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une **surcharge** (voir section polymorphisme).
L'héritage successif de classes permet de définir une hiérarchie de classe qui ce 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.

2.principe de l'héritage

L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique. La relation d'héritage est unidirectionnelle et, si une classe B hérite d'une classe A, on dira que B est une **sous classe** de A. Cette notion de sous-classe signifie que la classe B est un cas particulier de la classe A et donc que les objets instanciant la classe B instancient également la classe A.
Prenons comme exemple des classes Carre, Rectangle et Cercle. La figure 1 propose une Organisation hiérarchique de ces classes telle que Carre hérite de Rectangle qui hérite, ainsi que Cercle, d'une classe Forme.

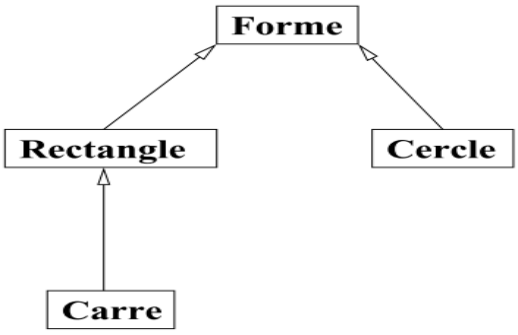

Figure1 :Exemple de relations d'héritage

3. Sous-classes et héritage

3.1. Les définitions :

- Definir une nouvelle classe a partir d'une autre ⇒ Dérivation
  - ✓ Nouvelle classe : classe derivee
  - ✓ Classe originale : superclasse ou classe de base.

- Une classe B qui **hérite** d'une classe A hérite des **attributs** et des **méthodes** de la classe A sans avoir à les **redéfinir**.
- B est une **sous-classe** de A ; ou encore A est **la super-classe** de B ou la **classe de base**.
- On dit encore que B est une **classe dérivée** de la classe A, ou que la classe B **étend** la classe A.
- Une classe ne peut avoir qu'une **seule super-classe** ; il n'y a pas **d'héritage multiple** en Java. Par contre, elle peut avoir plusieurs **sous-classes** (voir figure 2).
- La classe **Object** est la classe **parente** de toutes les classes en java. Toutes les variables et méthodes contenues dans la classe **Object** sont accessibles à partir de n'importe quelle classe car par **héritage successif** toutes les classes **héritent** de la classe **Object**.

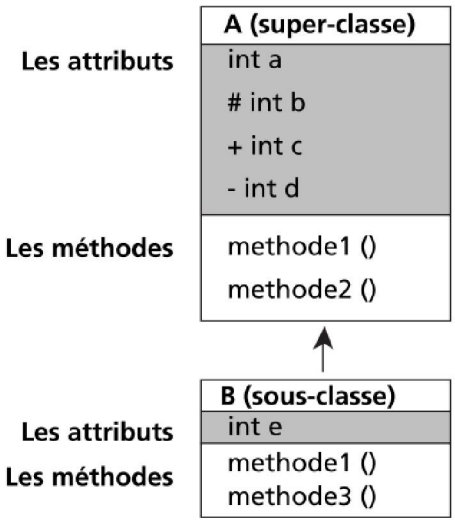

Figure 2 Le principe de l'héritage : la classe B **hérite** de la classe A.
Le symbole # indique un attribut **protected**.
Le symbole + indique un attribut **public**
Le symbole - indique un attribut **private**

- Toute instance de **B** est une instance de **A**.
- Toute instance de **B** possède tous les membres de **A** plus les membres definis dans **B**.
- Au niveau de la classe, tous les membres statiques de **A** sont des membres statiques de **B** (et **B** possède en plus les membres statiques definis dans **B**).
- On peut redefinir dans **B** les méthodes de **A**.

4. Héritage simple, héritage multiple

L'héritage permet à une sous-classe d'étendre les propriétés de la classe parente tout en héritant des champs (attributs) et des méthodes (comportement) de cette classe parente.il existe deux type d'héritages :

4.1. héritage simple : dans lequel une classe ne peut hériter que d'une seule super-classe. Voir figure suivante :

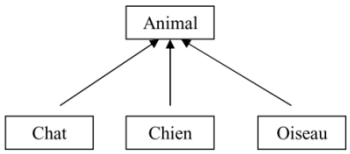

Figure 3 : héritage simple

4.1. héritage multiple: dans lequel une classe peut hériter de comportements et de fonctionnalités de plus d'une super-classe. Voir figure 4 :

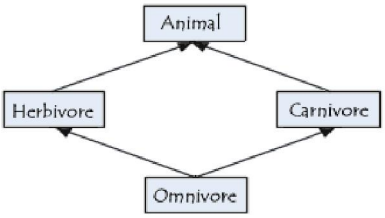

Figure 3 : héritage multiple

Il est supporté par certains langages de programmation, par exemple C++, Eiffel ou Python. Pouvant poser des problèmes, les développeurs de nombreux autres langages comme Ada, C#, Fortran, Java, Objective-C, Ruby, Swift ou Visual Basic ont préféré ne pas le proposer directement.

5. Hiérarchie de classes

Il est possible de représenter sous forme de hiérarchie de classes, parfois appelée *arborescence de classes*, la relation de parenté qui existe entre les différentes classes. L'arborescence commence par une classe générale appelée superclasse (parfois *classe de base*, *classe parent*, *classe ancêtre*, *classe mère* ou *classe père*, les métaphores généalogiques sont nombreuses).Puis les classes dérivées (*classe fille* ou *sous-classe*) deviennent de plus en plus spécialisées. Ainsi, on peut généralement exprimer la relation qui lie une classe fille à sa mère par la phrase "*est un*" (de l'anglais "*is a*"). La figure 4 présente une exemple de l'hiérarchie de classes

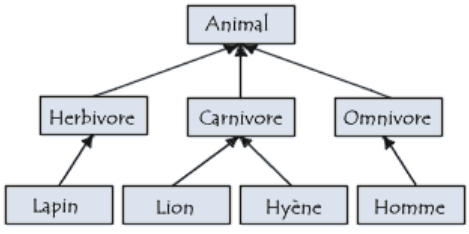

Figure 4 : l'hiérarchie de classes

5. Polymorphisme

Le polymorphisme est la faculté attribuée à un objet d'être une instance de plusieurs classes.

Le terme polymorphisme désigne la capacité d'une méthode à pouvoir prendre un comportement différent suivant le contexte: types des paramètres et type de l'objet courant.

Par exemple, si l'on veut créer une fonction calculant le maximum de deux nombres, il serait pratique qu'elle puisse porter le même nom quels que soient les paramètres utilisés. On voudrait créer ces fonctions de la manière suivante :

```
int     Max(int a, int b)               { ... }
float   Max(float a, float b)           { ... }
double  Max(double a, double b)         { ... }
...
k = Max(u,v) ;
```

Le compilateur connaît les types des variables u et v, il va donc chercher la fonction Max prenant ces types là en arguments. Les différentes fonctions Max ne traitent pas les mêmes éléments, pourtant conceptuellement, elles effectuent le même calcul. C'est un des aspects du polymorphisme

Ce concept de polymorphisme s'étend à l'héritage. Par exemple si vous faîtes un jeu de simulation affichant plusieurs types de véhicules à l'écran, il serait agréable d'avoir une unique fonction AfficheToi().

Ainsi pour dessiner l'écran du jeu, je demande à chaque objet de s'afficher en utilisant le même nom de fonction. Cette action est similaire pour une voiture, un avion, un vélo, chacun va s'afficher à l'écran, mais avec un comportement différent. Un objet A320 dessinera spécifiquement un Airbus A320, chaque voiture Peugeot 205 dessinera une Peugeot 205… Il est possible, mais plus lourd à gérer, de créer des fonctions différentes comme AfficheA320(), AfficheC3(), AffichePeugeot205()… Ainsi une fonction peut exister sous le même nom dans une hiérarchie mais avoir un comportement différent suivant le type de l'objet instancié. On parle alors de **polymorphisme d'héritage**.

6. Héritage et polymorphisme en Java

a. Héritage simple (extends)

a.1. Syntaxe :

```
class Fille extends Mere { ... }
```

On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe **Object** comme classe parent.

Exemple 1

```
//Classe de base
class graphique {
private int x, y;
graphique (int x, int y) {this.x = x ; this.y = y ;}
void affiche () {
```

```
System.out.println (`` Le centre de l'objet se trouve dans : `` + x + `` et `` + y) ;}
double surface () {return (0) ;}
} // fin de la classe graphique

//Classe dérivée 1
class Cercle extends graphique {
private double rayon =1 ;
void affiche () { Sustem.out.println (`` C'est un cercle de rayon `` + rayon) ;
Super.affiche () ;}
double surface () {return (rayon * 2* 3.14) ;} }

//Classe dérivée 2
class Rectangle extends graphique {
private int larg, longueur ;
Rectangle ( int x, int y, int l1, int l2)
{super (x,y) ; longueur = l1 ; larg = l2 ;}
double surface () {return (longueur*largeur) ;}
}// fin de la classe Rectangle

//Classe dérivée 3
class carré extends graphique {
private double côté ;
Carré (double c, int x, int y) { Super ( x,y) ; Côté = c ;}
double surface () {return (côté * côté) ;}
}// fin de la classe carré
```

a.2.Interprétation

1. Dans cet exemple, la classe *Cercle* hérite de la classe graphique les attributs et les méthodes, redéfinit les deux méthodes *surface* et *affiche* et ajoute l'attribut *rayon*.
2. La classe *Rectangle* redéfinit la méthode *surface* et ajoute les attributs *longueur* et larg
3. La classe *carré* ajoute l'attribut *côté* et redéfinit la méthode *surface*.

a.3.Remarques 1

Le concept d'héritage permet à la classe dérivée de :

1. Hériter les attributs et les méthodes de la classe de base.
2. Ajouter ses propres définitions des attributs et des méthodes.
3. Redéfinir et surcharger une ou plusieurs méthodes héritées.
4. Tout objet peut être vu comme instance de sa classe et de toutes les classes dérivées.
5. Toute classe en Java dérive de la classe Object.
6. Toute classe en Java peut hériter directement d'une seule classe de base. Il n'y a pas la notion d'héritage multiple en Java, mais il peut être implémenté via d'autres moyens tels que les interfaces (voir section interface).

Pour invoquer une méthode d'une classe parent, il suffit d'indiquer la méthode préfixée par **super**. Pour appeler le constructeur de la classe parent il suffit d'écrire **super** (paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe parent est géré par le langage : une évolution des règles de gestion de la classe parent conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

b. Encapsulation dans l'héritage

i. Protection des membres (protected)

i.1.L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès **public** restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur **private** est bien héritée mais elle n'est pas accessible directement, elle peut l'être via les méthodes héritées.

Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur **private**, il faut utiliser le modificateur **protected**. La variable ainsi définie sera hérité dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

Exemple 2 : La classe dérivée et la classe de base sont dans le même package

```
package Formes_geo;
class graphiques {
private int x, y ;
public String couleur ;
void affiche () {
System.out.println (`` Le centre de l'objet = (`` + x + `` , `` + y + ``) `` ) ; }
double surface ( ) {return (0) ;} } //fin de la classe graphiques

package Formes_geo;
class cercle extends graphiques{
double rayon ;
void changer_centre ( int x1, int y1, double r) {x = x1 ; y = y1 ;

// faux car x et y sont déclarés private dans la classe de base

rayon =r ;

public double surface ( ) {return (rayon * 2* 3.14) ;}
public void affichec () {affiche () ;

// juste car le modificateur de visibilité de ce membre est freindly (sans modificateur)

System.out.println (`` Le rayon = `` + rayon + `` La couleur = `` + couleur ) ;}}
```

i.1.1.Interprétation:

Dans le premier cas, la classe dérivée est la classe de base se trouvent dans le même package, donc la classe cercle a pu accéder aux membres (attributs et méthodes) publiques (l'attribut couleur), de même elle a pu accéder aux membres « freindly » c'est-à-dire qui n'ont pas un modificateur de visibilité (la méthode affiche), mais elle n'a pas pu accéder aux membres privés (x et y).

Exemple 3 : La classe dérivée et la classe de base ne sont pas dans le même package

```
package Formes_geo;
class graphiques {
private int x, y ;
public String couleur ;
protected String nom
void affiche () { System.out.println (`` Le centre de l'objet = (`` + x + `` , `` + y + `` ) `` ) ; }
double surface ( ) {return (0) ;} } //fin de la classe graphiques

package FormesCirc ;
class cercle extends graphiques{
double rayon ;
void changer_centre ( int x1, int y1, double r)
{x = x1 ; y = y1 ;
// faux car x et y sont déclarés private dans la classe de base
rayon =r ;
public double surface ( ) {return (rayon * 2* 3.14) ;}
public void affichec () {affiche () ;
// FAUX car le modificateur de visibilité de ce membre est freindly
Centre=15 ;
// Juste car le modificateur de visibilité de ce membre est protected
System.out.println (`` Le rayon = `` + rayon + `` La couleur = `` + couleur ) ;}}
```

i.1.2.Interprétation:

Dans le deuxième cas, la classe dérivée est la classe de base ne sont pas dans le même package, donc la classe cercle a pu accéder aux membres (attributs et méthodes) publiques (l'attribut couleur), mais, elle n'a pas pu accéder aux membres « freindly » c'est-à-dire qui n'ont pas un modificateur de visibilité (la méthode affiche) et aux membres privés (x et y).
Pour l'attribut protected nom de type String est claire qu'il est accessible par la classe fille cercle, car il respect la règle de l'encapsulation. (voir l'accès aux variable chapitre encpsulation)

- Nous avons déjà vu qu'il existe 3 types de modificateurs de visibilité

publiques (« public »), privés (private) et de paquetage ou freindly (sans mention).

- Il existe encore un quatrième droit d'accès dit protégé (mot clé « protected »)

ii. Constructeurs des classes (this(), super())

Le mot réservé super ermet d'invoquer un champ (attribut ou méthode) de la super-classe de l'objet sur lequel on est entrain de travailler (objet dit courant), et qui serait si non caché par un champ de même nom de la classe de l'objet courant (soit que le champ en question soit une méthode qui a été redéfinie, soit que le champ soit un attribut qui a été masqué).

- Lorsque, dans une méthode, on invoque un champ de l'objet courant, on ne précise en général pas le nom de l'objet, ce que l'on peut néanmoins faire avec le mot this. Lorsqu'on emploie super, ce mot vient remplacer this qui figure implicitement.
- Employer super revient à remplacer la classe de l'objet courant par la super- classe.
- Remarquons qu'il est interdit de "remonter de plusieurs étages" en écrivant super.super.etc.
- On utilise aussi super(paramètres) en première ligne d'un constructeur. On invoque ainsi le constructeur de la super-classe ayant les paramètres correspondants.

Pour l'objet this voir section accès à l'instance this chapitre 2 « Encapsulation »

ii.1.Construction et initialisation des objets dérivés

En java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet. 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. Le constructeur de la classe de base est désigné par le mot clé « **super** ».

Exemple 4 :

```
class graphiques {
private int x, y ;
public graphiques (int x1, int y1) {this.x = x1 ; this.y=y1 ;}
//autres méthodes
}
class cercle extends graphiques{
private double rayon ;
cercle ( int a, int b, double r) {
super (a, b) ;
//appelle le constructeur de la classe de base
rayon = r ; }
//autres méthodes
}
```

Remarque 2 :

Dans le cas général, une classe de base et une classe dérivée possèdent chacune au moins un constructeur. Le constructeur de la classe dérivée complète la construction de l'objet dérivé, et ce, en appelant en premier lieu le constructeur de la classe de base.

Toutefois, il existe des cas particuliers qui sont :

cas1: La classe de base ne possède aucun constructeur
Dans ce cas, si la classe dérivée veut définir son propre constructeur, elle peut optionnellement appeler dans la 1ère instruction de constructeur la clause « super () » pour désigner le constructeur par défaut de la classe de base.

Cas2: La classe dérivée ne possède pas un constructeur
Dans ce cas, le constructeur par défaut de la classe dérivée doit initialiser les attributs de la classe dérivée et appeler :

- Le constructeur par défaut de la classe de base si elle ne possède aucun constructeur.
- Le constructeur sans argument si elle possède au moins un constructeur.
- Dans le cas où il n'y a pas un constructeur sans argument et il y a d'autres constructeurs avec arguments, le compilateur génère des erreurs.

Exemple 5 (cas1)

```
Class A {
// pas de constructeur
}
class B extends A {
public B(….)
{super() ; //appel de constructeur par défaut de A
……..
}
B b = new B(…); // Appel de constructeur1 de A
```

Exemple 6 (cas2)

```
Class A {
public A() {….} //constructeur1
public A ( int n) {…..} //constructeur2
}
class B extends A {
// …..pas de constructeur
}
B b = new B(); // Appel de constructeur1 de A
```

Exemple 7 (cas 2)

```
Class A {
public A ( int n) {…..} //constructeur1
}
class B extends A {
// …..pas de constructeur
}
B b = new B();
/* On obtient une erreur de compilation car A ne dispose pas
un constructeurs sans argument et possède un autre constructeur*/
```

Exemple 8 (cas1 et2)

```
Class A {
//pas de constructeurs
}
class B extends A {
//pas de constructeurs
}
B b = new B ();
// Appel de constructeur par défaut de B qui Appelle le constructeur par
//défaut de A.
```

iii. Classe 'Object'

Toute classe dérive implicitement de la classe **Object**, membre du paquet **java.lang**. Autrement dit, les deux définitions suivantes sont équivalentes :

```
class A { ... }
class A extends Object { ... }
```

Object est la classe de base de Java. Tous les objets que l'on va créer par la suite héritent les méthodes de la classe **Object**

Toute classe est donc un sous-type d'`Object`. Par suite, toute méthode déclarée avec un paramètre de type **`Object`** accepte en argument une instance de n'importe quelle classe (mais pas une valeur d'un type primitif).

Tous les types de tableaux sont également des sous-types d'`Object` (ainsi que des interfaces `Cloneable` et `java.io.Serializable`). Ainsi, un tableau quelconque peut être affecté à une variable de type `Object`. Voici quelques unes des méthodes qui sont définies par la classe **`Object`** :

```
package java.lang;

public class Object {
  public String toString() { ... }
  public final Class getClass() { ... }
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
  protected Object clone()
    throws CloneNotSupportedException { ... }
  ...
}
```

- ✓ La méthode **`toString`** retourne une représentation de l'objet par une chaîne de caractères. C'est cette méthode qui est invoquée quand un objet figure en argument de la méthode **`print`** ou **`println`** ou dans une expression de concaténation de chaînes. Il est souvent utile de la redéfinir (voir section polymorphisme), pour obtenir une

chaîne de caractères plus lisible. Par exemple, la classe `Point` pourrait la redéfinir ainsi :

```
class Point {
 // ...
 public String toString() {
  return "(" + x + ", " + y + ")";
 }
}
```

- ✓ La méthode **`getClass`** retourne une référence à une instance de la classe `Class` qui représente la classe de l'objet. Cette instance permet d'obtenir diverses informations sur cette classe (son nom, ses membres, sa surclasse, etc.), par exemple :

```
 void printClassName() {
 System.out.println("La classe de " + this +
   " est " + this.getClass().getName());
 }
```

- ✓ La méthode **`equals`** permet de tester l'égalité de deux objets. La méthode définie dans la classe **`Object`** est la plus discriminante possible, ce qui signifie que **`x.equals(y)`** retourne **`true`** si et seulement si `x` et `y` sont des références au même objet, c'est-à-dire si **`x == y`**.

Par exemple, si la classe `Point` ne redéfinit pas **`equals`**, la valeur de l'expression

```
new Point().equals(new Point())
```

Qui compare deux instances différentes, retourne **`false`**. Il est donc utile de redéfinir **`equals`**. Dans le cas de la classe `Point`, un point est égal à un objet `o` si `o` est un point et si les champs correspondants sont égaux. L'expression `o instanceof Point`, de type `boolean`, permet de tester si le type de l'objet désigné par `o` est un sous-type de `Point` :

```
package geometrie;

class Point {
// ...
public boolean equals(Object o) {
return o instanceof Point &&
this.x == ((Point)o).x &&
this.y == ((Point)o).y;
}
}
```

Il serait tentant de définir une méthode qui compare seulement des instances de `Point` entre elles :

```
class Point {
// ...
boolean equals(Point p) { ... }
}
```

Cependant, ce ne serait pas une redéfinition de la méthode **`equals`** de la classe **`Object`**, car elle n'a pas le même profil. Dans la situation suivante, l'égalité des deux points serait donc testée à l'aide de l'**`equals`** d'**`Object`** et non par celle de `Point` :

```
Object o = new Point(), p = new Point(1, 2);
 boolean b = o.equals(p);
```

- ✓ La méthode clone permet de dupliquer un objet. Vous pouvez voir Cette opération importante sera examinée plus loin.Cette méthode a pour but de créer et de renvoyer une copie exacte de l'objet considère.

On souhaite que x.**clone**() != x et x.**clone**().**getClasse**() == x.**getClasse**()

iv. Transtypage (cast) implicite et explicite

Le transtypage est nécessaire quand il risque d'y avoir perte d'information, comme lors de l'affectation d'un entier **long** (64 bits) à un entier **int** (32 bits), ou d'un réel **double** vers un réel **float**. **On force alors le type**, indiquant ainsi au compilateur qu'on est conscient du problème de risque de perte d'information. C'est aussi le cas lorsqu'on veut prendre la partie entière d'un réel. Les affectations suivantes sont **implicitement** (**sans écrire l'instruction de transtypage : c'est-à-dire sans mettre les parenthèses : voir les exemples dans la présente section**) autorisées entre entiers et/ou réels ; on peut affecter un **byte** à un **short**, ou un **float** à un **double** :

Figure 5 : Les conversions implicites autorisées en Java.

Exemples iv.1 :

Quand les deux opérandes d'un opérateur ne sont pas du même type, celui qui est de type inférieur est converti dans le type de l'autre. Pour la variable l4 ci-après, on multiplie i3 (qui vaut 10) par la valeur réelle de 2.5f convertie en un entier long. Pour l5, i3 est converti en flottant ; le résultat en flottant est converti en entier long. Les résultats sont différents.

```
int i3 = 10;
long l4 = i3 * (long) 2.5f; // résultat : 20
long l5 = (long) (i3 * 2.5f); // résultat : 25
```

Remarque 3 :
Dans les expressions arithmétiques, les entiers de type **byte** ou **short** sont considérés comme des **int** (convertis d'office en **int**). Si b1 est un **byte**, b1 + 1 est considéré être un **int**. L'affectation b1 = b1 + 1 est illégale **sans cast** (de type **int** -> **byte**) alors que b1 = 5 est acceptée.

Exemple iv.2

```
// TransTypage.java les casts (conversions forcées)
class TransTypage
{ public static void main (String[] args) {
        byte b1 = 15;
        int i1 = 100;
        long l1 = 2000;
        float f1 = 2.5f;
// f pour float; par défaut de type double
double d1 = 2.5;  // ou 2.5d
//  b1 = b1 + 1;    // cast nécessaire : b1 + 1 est converti en int
b1 = (byte) (b1 + 1);  // OK avec cast
//   byte b2 = i1;       // cast nécessaire : de int -> byte
byte b2 = (byte) i1;    // OK avec cast
System.out.println ("b2 : " + b2);

long l2 = i1;             // OK sans cast : de int -> long
System.out.println ("l2 : " + l2);

//int i2 = l1;            // cast nécessaire : de long -> int
int i2 = (int) l1;        // OK avec cast
System.out.println ("i2 : " + i2);


float f2 = l1;        // OK sans cast : de long -> float
System.out.println ("f2 : " + f2);
//long l3 = f1;    // cast nécessaire : de float -> long
long l3 = (long) f1;              // OK avec cast
System.out.println ("l3 : " + l3);


double d2 = f1;        // OK sans cast : de float -> double
System.out.println ("d2 : " + d2);
//float f3 = d1;    // cast nécessaire : de double -> float
float f3 = (float) d1;  // OK avec cast
System.out.println ("f3 : " + f3);


int i3 = 10;
long l4 = i3 * (long) 2.5f;       // l4 vaut 20
long l5 = (long) (i3 * 2.5f);      // l5 vaut 25
System.out.println ("\nl4 : " + l4 + "\nl5 : " + l5);

// Dans une expression arithmétique, byte, short sont considérés comme des int
```

```
byte b3 = 10;
//short s4 = 2*b3;              // erreur : b3 est considéré comme un int
short s4 = (short) (2*b3);   // cast nécessaire : de int -> short
System.out.println ("s4 : " + s4);

int i4 = 10;
long l6 = 1000;
//int i5 = i4 + l6;    // erreur : le résultat est de type long
int i5 = (int) (i4 + l6);     // cast nécessaire : de long -> in

System.out.println ("i5 : " + i5);
} // main
} // class TransTypage
```

v. Limitation de l'héritage (final)

v.1.Le modificateur «final»

Cas1 : Le modificateur « **final** » placé devant une classe interdit sa dérivation
Exemple v.1.

```
final class A {
//Méthodes et attributs }
class B extends A              //Erreur car la classe A est déclarée finale
{//....}
```

Cas 2 : Le modificateur « **final** » placé devant une méthode permet d'interdire sa **redéfinition** (voir le cours polymorphisme)
Exemple v.2.

```
class A {
final Public void f(int x) {…..}
//…Autres méthodes et attributs
}
class B {
//Autres méthodes et attributs
public void f (int x) {…} // Erreur car la méthode f est déclarée finale. Pas de
redéfinition
}
```

Cas 3 : Pour une donnée, **final** indique qu'il s'agit d'une **constante**, d'instance s'il n'y a pas simultanément le modificateur static, et de classe si la donnée est **final static**. Une donnée **final** ne pourra être affectée qu'une seule fois.
Exemple v.2.

```
class c1 {final int i=5          // i est donnée constante}
```

Remarque 4 :

Les classes et les méthodes peuvent être déclarées finales pour des raisons de sécurité pour garantir que leur comportement ne soit modifié par une sous-classe.

Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

1. la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre **protected** et **private** ;
2. pour empêcher la redéfinition d'une méthode (surcharge) il faut la déclarer avec le modificateur final. Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivant :

- ✓ la méthode héritée convient à la classe fille : on ne doit pas la redéfinir ;
- ✓ la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voir la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (via **super**) pour garantir l'évolution du code ;
- ✓ la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition

pour la redéfinition et la surcharge : voir le cours de polymorphisme…….