Part 4

HERITAGE ET POLYMORPHISME

Réutilisation de classes

- Comment utiliser une classe comme brique de base pour concevoir d'autres classes ?
- Dans une conception objet on définit des associations (relations) entre classes pour exprimer la réutilisation.
- UML (Unified Modelling Language) définie toute une typologie des associations possibles entre classes. Ici nous nous focaliserons sur deux formes d'association:
  - Un objet peut faire appel à un autre objet : **délégation**
  - Un objet peut être créé à partir du « moule » d'un autre objet : **héritage**

Délégation

Délégation

- Soit une classe A dont on a le code compilé
- Une classe C veut réutiliser la classe A
- Elle peut créer des instances de A et leur demander des services.

- On dit que :
  - la classe C est une **classe cliente** de la classe A
  - la classe A est une **classe serveuse** de la classe C

Délégation

- **Notation UML**

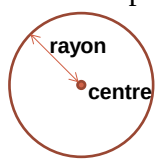
- **Traduction en JAVA**

```
public class C {
   private A refA;
      ......
}
```

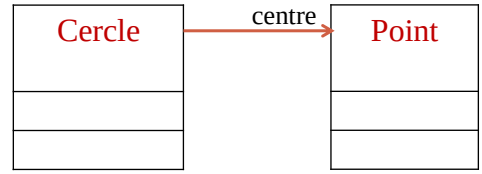
```
public class A {
      ......
}
```


Délégation : Exemple1

- Exemple la classe **Cercle**



  - **rayon:** double
  - **centre**: Point



- L'association entre les classes Cercle et Point exprime le fait qu'un cercle **possède (a un)** un centre.

```
public class Cercle{
   private Point centre ;  // centre du cercle
   private double r ;      // rayon du cercle
   ......
   public void deplace (int dx, int dy){
       centre.deplace(dx,dy);
   }
   ......
}
```

Délégation : Exemple1

```
public class Cercle{
   private Point centre ;  // centre du cercle
   private double r ;      // rayon du cercle
   ......

   public Cercle(Point centre, double r){
       this.centre = centre;
       this.r = r;
   }

   public void deplace(int dx, int dy){
       centre.deplace(dx,dy);
   }
   ......
}
```


Délégation : Exemple1

- Le point représentant le centre a une existence autonome (cycles de vie indépendants).
- Il peut être partagé : liée à plusieurs instances d'objets (éventuellement d'autres classes) .
- Il peut être utilisé en dehors du cercle dont il est le centre.

```
Point p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);

c2.deplace(10,0);
/* Affecte aussi c1: le centre des 2 cercles est (20,0) */

p1.deplace(0,20);
/* Affecte les deux cercles c1 et c2: le centre des 2
cercles est (20,20) */
```

Délégation : Exemple2

```
public class Cercle{
   private Point centre ;  // centre du cercle
   private double r ;      // rayon du cercle

   public Cercle(Point centre, double r){
       this.centre = new Point(centre);
       this.r = r;
   }

   public void deplace(int dx, int dy){
       centre.deplace(dx,dy);
   }
   public void affiche(){
       System.out.println ("Je suis un cercle de rayon " + r );
       System.out.print (" et de centre ") ;
       centre.affiche();
   }
}
```


Délégation : Exemple2

- Le Point représentant le centre n'est pas partagé (à un même moment, une instance de Point ne peut être liée qu'à un seul Cercle)
- Les cycles de vies du Point (centre) et du Cercle sont liés : si le cercle est détruit (ou copié), le centre l'est aussi.

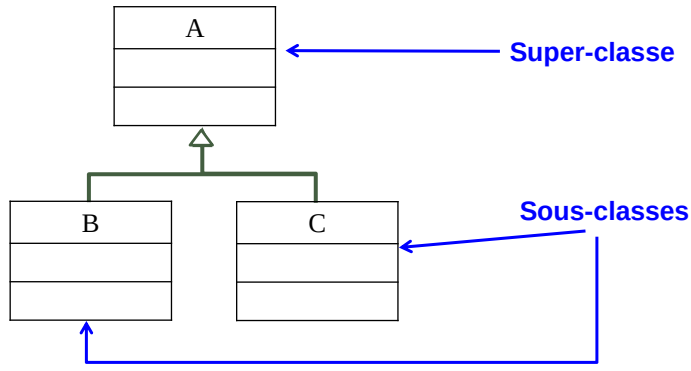
```
Point p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10)
Cercle c2 = new Cercle(p1,20);

c2.deplace(10,0);
// N'affecte que le cercle c2

p1.deplace(0,20);
// N'affecte pas les cercles c1 et c2
```

Héritage

Héritage

- C'est un mécanisme de dérivation entre classes, symbolisant la relation « **est un** ».
- L'héritage permet de reprendre les caractéristiques d'une classe A existante pour les étendre et définir ainsi une nouvelle classe B qui hérite de A.
- Les objets de B possèdent toutes les caractéristiques de A avec en plus celles définies dans B
- La relation d'héritage peut être vue comme une relation de "**généralisation/spécialisation"** entre une classe (la **super-classe**) et plusieurs classes plus spécialisées (ses **sous-classes**).

Héritage: Notation UML



- Les classes B et C **dérivent** / **héritent** de la classe A
- Un objet de la classe B ou C est un objet de la **super-classe** A.
- La classe A est appelée **classe mère**
- Les classes dérivées sont appelées **classes filles**

Héritage: Exemple

- Dans la classe Point définie précédemment, un point
  - a une position
  - peut être déplacé
  - peut calculer sa distance à l 'origine
  - …

Le problème:

- Une application a besoin de manipuler des points (comme le permet la classe Point) mais en plus de manipuler des **points colorés.**
  - PointGraphique = Point + une couleur + une opération d'affichage

La solution en POO:

- Définir une nouvelle classe qui **hérite** de la classe déjà existante

Héritage: syntaxe

- On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre: class Fille extends Mere { ... }

```
public class PointGraphique extends Point{
  private String couleur;

  public PointGraphique(int x, int y, String couleur) {
    super(x,y);
    this.couleur = couleur;
  }

  public void setCouleur(String couleur){
     this.couleur = couleur ;
  }
  public void affiche(){
     System.out.println ("Mes coordonnees " + x + " " + y
                         + " et ma couleur" + couleur);
   }
}
```

PointGraphique définit un nouvel attribut

PointGraphique définit une nouvelle méthode

On accède directement aux attributs hérités par ce qu'ils sont protected

```
public class Point{
   protected int x ; // abscisse
   protected int y ; // ordonnee

   //construteur
   public Point (int abs, int ord){
      x = abs ;
      y = ord ;
   }
   public Point (){
      this(0,0);
   }

   //méthodes
   public double distance(Point p){
    return Math.sqrt((x - p.x)*(x - p.x)+(y - p.y)*( y - p.y));
   }
   public void deplace (int dx, int dy){
     x += dx ;
     y += dy ;
   }
   public void affiche(){
     System.out.println ("Mes coordonnees " + x + " " + y);
   }
}
```

Classe mère de la classe PointGraphique

Ce que peut faire une classe fille

- La classe qui hérite peut
  - ajouter des attributs, des méthodes et des constructeurs
  - redéfinir des méthodes (même signature)
  - surcharger des méthodes (même nom mais pas même signature)
- Mais elle ne peut retirer aucune variable ou méthode!

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 mais via les **getters** et **setters**.
- 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 héritée dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

Relation entre constructeurs des classes mères/filles

- En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.
  - S'il est nécessaire d'initialiser certains champs de la classe de base et qu'ils sont **private**, il faudra disposer de **setters** ou bien recourir à un **constructeur de la classe mère**.
- Si un constructeur d'une classe fille appelle un constructeur de la classe mère, **il doit obligatoirement s'agir de la première instruction du constructeur** et ce dernier est désigné par le mot-clé **super** (utilisation analogue à celle de this).
- Sans précision du constructeur, c'est le constructeur par défaut qui est appelé (constructeur sans arguments).

Aucun constructeur explicite nulle part

```
public class Point {
	// Aucun constructeur explicite
}
```

```
public class PointGraphique extends Point {
	// Aucun constructeur explicite
}
```

```
PointGraphique p = new PointGraphique();
// Appel au constructeur par défaut de PointGraphique qui
appelle le constructeur par défaut de Point
```

Aucun constructeur explicite dans la classe mère

```
public class Point {
  protected int x,y;
  // Aucun constructeur explicite
}
```

```
public class PointGraphique extends Point {
  private String couleur;
  public PointGraphique(int x, int y, String c) {
     // Initialisation des attributs
    }
}
```

```
PointGraphique p = new PointGraphique(3,6, "green"); //
Appel au constructeur avec arguments de PointGraphique

PointGraphique p = new PointGraphique(); // Erreur vu que
PointGraphique dispose d'un constructeur (avec argument)
```


Aucun constructeur explicite dans la classe dérivée

```
public class Point {
  protected int x,y;
  public Point(){ …… } // constructeur sans arguments
  public Point(int x, int y){ …… } // avec arguments
}
```

```
public class PointGraphique extends Point {
       // Aucun constructeur explicite
}
```

```
PointGraphique p = new PointGraphique();
// Appelle le constructeur par défaut qui appelle le
constructeur sans arguments de Point
```

Aucun constructeur explicite dans la classe dérivée

```
public class Point {
  protected int x,y;
  public Point(int x, int y){ …… } // avec arguments
}
```

```
public class PointGraphique extends Point {
       // Aucun constructeur explicite.
       // Erreur à la compilation!!!!
}
```

- Le constructeur par défaut de **PointGraphique** cherche un constructeur sans argument de Point.
- Puisque **Point** dispose d'au moins un constructeur (avec argument), il remplace le constructeur sans argument, d'où erreur.

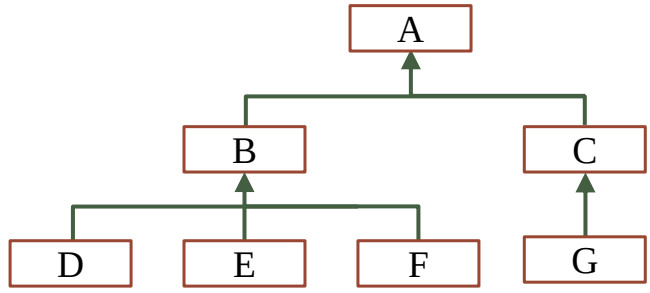
Plusieurs constructeurs dans les classes mère et fille

```
public class Point {
  protected int x,y;
  public Point(){ …… }
  public Point(int x, int y){ ……}
}
```

```
public class PointGraphique extends Point {
  private String couleur;
  public PointGraphique() {
    super(); // Appel au constructeur de Point

    }
  public PointGraphique(int x, int y, String c) {
    super(x,y); // Appel au constructeur de Point,
    obligatoirement comme première instruction
    couleur = c;
  }
}
```

Dérivations successives

- D'une même classe peuvent être dérivées plusieurs classes différentes,
- Une classe dérivée peut, à son tour, servir de classe de base pour une autre.



Héritage à travers tous les niveaux

```
public class A {
  public void hello() {
    System.out.println(«Hello»);
  }
}
```

```
public class B extends A {
  public void bye() {
    System.out.println(«Bye Bye»);
  }
}
```

```
public class C extends B {
  public void oups() {
    System.out.println(«Oups!»);
  }
}
```

- La hiérarchie des classes est parcourue de manière ascendante jusqu'à trouver la méthode correspondante.

```
C c = new C();

c.hello();
c.bye();
c.oups();
```

Redéfinition et surcharge des méthodes

- La redéfinition du comportement des méthodes de la classe mère peut prendre trois formes :
  - **Surcharge** de la signature d'une méthode héritée: overloading.
  - **Redéfinition** complète d'une méthode héritée sans surcharger sa signature : overriding.
  - **Spécialisation** d'une méthode héritée sans surcharger sa signature: specialization

Surcharge d'une méthode héritée

```
public class Point {
  Protected int x, y;
  public void deplace(int dx, int dy) {
    x += dx; y += dy;
  }
}
```

```
public class PointGraphique extends Point {
  public void deplace(float dx, float dy) {
    x += dx; y += dy;
  }
}
```

- La classe PointGraphique possède **deux méthodes** deplace, dont une héritée de Point.

Redéfinition d'une méthode héritée

```
public class Point {
  protected int x, y;
  public void deplace(int dx, int dy) {
    x += dx; y += dy;
  }
}
```

```
public class PointGraphique extends Point {
  @Override //Annotation Java pour indiquer la redéfinition
  public void deplace(int dx, int dy) {
    x += (dx + 0.1); y += (dy + 0.1);
  }
}
```

- La classe PointGraphique possède **une seule méthode** deplace(), qui **redéfinit** (override) le comportement de celle héritée de Point.

Redéfinition

- En cas de redéfinition, non seulement la **signature** de la méthode de base (celle de la classe mère) doit être **respectée**, mais également son **type de retour.**

```
public class Point {
  protected int x, y;
  public void deplace(int dx, int dy) { …… }
}
```

```
public class PointGraphique extends Point {
  @Override
  public boolean deplace(int dx, int dy) { …… } // Erreur!!
}
```

Comportement de @Override

- Utiliser cette annotation signifie :

  « Je suis en train de redéfinir une méthode de ma classe mère et je voudrais provoquer une erreur de compilation s'il n'existe pas de méthode correspondante dans la classe mère ».

- L'erreur peut être levée à cause d'une faute de frappe ou du changement de signature de la méthode dans la classe mère.
- En cas de changement de signature dans la classe mère, on se retrouvera avec une surcharge dans la classe fille, au lieu de la redéfinition ! !
  - il faut donc que le programmeur en soit averti, d'où erreur.

Changement de droits d'accès en cas de redéfinition

- Le droit d'accès de la méthode redéfinie dans la classe dérivée **ne doit pas être moins élevé** que celui dans la classe mère.
- private < protected < droit niveau package < public

```
public class Point {
  public void deplace(int dx, int dy) { ……}
  public void affiche() { ……}
}
```

```
public class PointGraphique extends Point {
  @Override
  public void deplace(int dx, int dy) { …… } // OK
  @Override
  private void affiche() { …… } // Erreur à la compilation!
}
```

Changement de droits d'accès en cas de surcharge

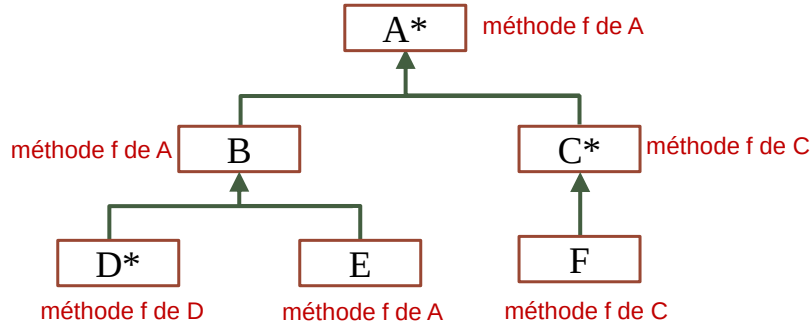
- Il n'y a pas de restriction sur le droit d'accès de la méthode surchargé de la classe dérivée, car il s'agit de méthodes différentes, leurs signatures étant différentes.

```
public class Point {
  public void deplace(int dx, int dy) { ……}
}
```

```
public class PointGraphique extends Point {
  private void deplace(float dx, float dy) { …… } // OK
}
```


Redéfinition de méthode et dérivations successives

- La redéfinition d'une méthode s'applique à une classe et à toutes ses descendantes jusqu'à ce que l'une d'entre elles redéfinisse la méthode.
- Dans l'arborescence suivante, l'appel de la méthode f conduira, pour chaque classe, à l'appel de la méthode indiquée en regard. **(*) signale la définition ou la redéfinition d'une méthode *f***



Spécialisation d'une méthode héritée

```
public class Point {
  protected int x,y;
  public void affiche() {
    System.out.println ("Mes coordonnees " + x + " " + y);
  }
}
```

```
public class PointGraphique extends Point {
  private String couleur;
  @Override
  public void affiche() {
    super.affiche(); // Réutilisation de affiche() de Point
    System.out.println ("Ma couleur " + couleur);
  }
}
```

- La classe PointGraphique possède **une méthode** affiche(), qui **spécialise** le comportement de celle héritée de Point.

Interdire la redéfinition d'une méthode

- On peut interdire la redéfinition d'une méthode en utilisant le mot-clé final dans sa signature.

```
public class Point {
  public final void deplace(int dx, int dy) { ……}
}
```

```
public class PointGraphique extends Point {
  /* La déclaration suivante renvoie une erreur, car la
  méthde 'deplace' est 'final' dans Point */
  @Override
  public void deplace(int dx, int dy) { …… }
}
```

Interdire la dérivation d'une classe

- Pour interdire la possibilité d'étendre une classe, on utilise aussi mot-clé **final** dans sa déclaration.

```
public final class Point {
    ……
}
```

```
/* La déclaration suivante renvoie une erreur, car Point
ne peut pas être étendue à cause du 'final'*/

public class PointGraphique extends Point {
   ……
}
```


Particularité de l'héritage en Java

- Héritage simple
  - Une classe ne peut hériter que d'une **seule autre classe.**
- En Java, toutes les classes héritent par défaut de la classe java.lang.Object
  - La déclaration :

```
public class Point { ... }
```

  - Est équivalente (implicitement) à:

```
public class Point extends Object { ... }
```

  - Lire http://docs.oracle.com/javase/8/docs/api/

La classe Object (1/2)

- Dans la classe Object, 7 méthodes sont d'accès public et 2 sont d'accès protected.
- Les classes définies par un programmeur hériteront de ces 9 méthodes.

Méthode	Fonctionnalité
(i) toString()	Retourne un objet de la classe String décrivant l'objet courant. Il s'agit du nom de la classe suivi de @ suivi d'une représentation hexadécimale de l'objet.
(ii) getClass()	Retourne un objet de type Class permettant d'identifier la classe de l'objet courant.

La classe Object (2/2)

(iii) finalize()	C'est la méthode qui est appelée lorsqu'un objet est détruit. On peut effectuer une surcharge de cette méthode à l'intérieur de sa propre classe en autant que l'on fasse appel à super.finalize().
(iv) clone()	permet de créer un nouvel objet du même type que l'objet courant où chaque variable du nouvel objet reçoit la même valeur que celle de l'objet courant. Cependant, lorsque des variables de la classe réfèrent à des objets, ceux-ci ne sont pas automatiquement dupliqués lors du clonage. Ainsi, 2 objets peuvent partager un même objet comme donnée membre.

(iii) et (iv) ont comme mode d'accès protected dans la classe Object.

Exemple Complet (1/3)

```
public class Point {
    private int x, y ;
    public Point (int x, int y){
        this.x = x ; this.y = y ;
    }
    public Point (Point p){
        this(p.x,p.y);
    }
    public void deplace (int dx, int dy){
        x += dx ;  y += dy ;
    }
    public void affiche (){
        System.out.println ("Je suis en " + x + " " + y) ;
    }
}
```


Exemple Complet (2/3)

```
public class PointGraphique extends Point{
    private String couleur ;
    public PointGraphique (int x, int y, String couleur){
        super (x, y) ; // obligatoirement comme première instruction
        this.couleur = couleur ;
    }
    @Override
    public void affiche (){
        super.affiche() ;
        System.out.println (" et ma couleur est : " + couleur) ;
    }
}
```

Exemple Complet (3/3)

```
public class TestPointGraphique{
    public static void main (String args[]){
        PointGraphique pg = new PointGraphique(3, 5, "rouge") ;
        pg.affiche() ; // ici, il s'agit de affiche de PointGraphique
        pg.deplace (1, -3) ;
        pg.affiche() ;
    }
}
```

```
Je suis en 3 5
 et ma couleur est : rouge
Je suis en 4 2
 et ma couleur est : rouge
```


Exercice 5

- On souhaite écrire une classe Ville destinée à modéliser une ville. Les attributs d'une ville sont : son nom (qui ne peut pas varier) et son nombre d'habitants (qui peut varier). Les villes sont de plus classées en trois catégories : **"grande"** pour les plus de 500 000 habitants, **"moyenne"** pour les plus de 100 000 mais moins de 500 000 habitants et enfin **"petite"** pour les autres.
  - Écrire une classe **Ville** conforme à la spécification.
    - La classe devra respecter le principe d'encapsulation.
    - Définir une méthode **toString()** qui renvoie les informations d'une ville, sous forme d'une chaine de caractère; par exemple *"El Jadida, 250000 habitants, moyenne ville"* .

Exercice 5 (suite)

- Une capitale est une ville contenant l'information du pays dans lequel elle se situe.
  - Écrire une classe **Capitale** destinée à modéliser une capitale.
  - La classe devra proposer une méthode **toString()** qui renvoie les informations d'une capitale, sous forme d'une chaine de caractère; par exemple *"Paris, 2101816 habitants, grande ville, capitale de France"*.
  - Écrire une classe **Principale** contenant seulement la méthode main où vous créerez deux objets villes de catégories différentes et un objet capitale, puis afficherez les informations de ces trois villes.

Upcasting & Liaison dynamique

Upcasting (1/2)

- L'héritage définit un cast implicite **de la classe fille vers la classe mère** (surclassement ou upcasting):
  - on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous classes.

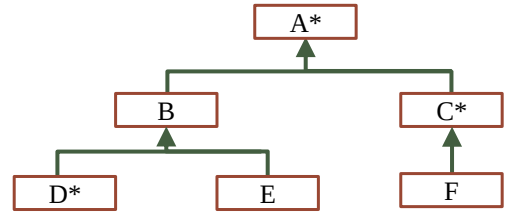
```
Point p = new Point(3,5);
……
p = new PointGraphique(10,20, "noir");
/* p de type Point contient la référence à un objet
de type PointGraphique */
```

- La conversion de types d'objets est aussi possible dans une structure hiérarchique de classes (voir slide suivant).

Upcasting (2/2)

- Avec ces déclarations :

```
A a ;
B b ;
C c ;
D d ;
E e ;
F f ;
```



- Les affectations suivantes sont légales :

```
a = b ; a = c ; a = d ; a = e ; a =f ;
b = d ; b = e ;
c = f ;
```

- En revanche, celles-ci ne le sont pas :

```
b = a ; // erreur : A ne descend pas de B
d = c ; // erreur : C ne descend pas de D
c = d ; // erreur : D ne descend pas de C
```

Liaison dynamique(1/2)

- Dans un appel de la forme x.f(...) où x est supposé de classe T, le choix de f est déterminé ainsi :
  - **à la compilation** : on détermine dans **la classe déclarée** ou ses ascendantes la signature de la meilleure méthode f convenant à l'appel, ce qui définit du même coup le type de la valeur de retour.
  - **à l'exécution**: on recherche la méthode f de signature et de type de retour voulus, à partir de la classe correspondant au **type effectif de l'objet référencé par x** (et non pas au type déclaré); si cette classe ne comporte pas de méthode appropriée, on remonte dans la hiérarchie jusqu'à ce qu'on en trouve une.
- Ce mécanisme est désigné sous le terme de **liaison dynamique** (dynamic binding, latebinding ou run-time binding)

Liaison dynamique(2/2)

```
public class A {
  public void m1() {
    System.out.println("m1 de A");
  }
}
```

```
public class B extends A {
  public void m1() {
    System.out.println("m1 de B");
  }
}
```

```
public class C extends B {
  public void m2() {
    System.out.println("m2 de C");
  }
}
```

Type déclaré

```
A obj = new C();
obj.m1(); //m1 de B
obj.m2(); //Erreur à
la compilation
```

Type effectif

Downcasting

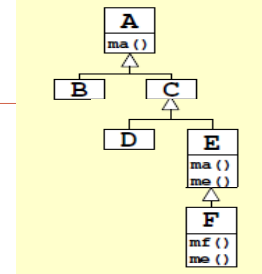
- La **conversions explicites** de la classe mère vers la classe fille (downcasting ou transtypage) permet de « forcer un type » **à la compilation**
  - C'est une « promesse » que l'on fait au moment de la compilation.
  - On utilise l'opérateur de cast déjà rencontré pour les types primitifs.

```
ClasseX obj = .....
ClasseA a = (ClasseA) obj;
```

- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de obj soit ClasseA ou n'importe quelle sous classe de ClasseA
- **Si la promesse n'est pas tenue une erreur d'exécution se produit.**

Upcasting/Downcasting

On parle de **upcast** et de **downcast** en faisant référence au fait que la classe mère est souvent dessinée au-dessus de ses classes filles dans les diagrammes UML.



	Compilation	Exécution
c.ma();	La classe C hérite la méthode ma()	Méthode ma() définie dans E
c.mf();	Pas de méthode mf() définie au niveau de la classe C	
B b = c;	Un C n'est pas un B	
E e = c;	Un C n'est pas forcément un E	
E e = (E) c; e.me();	Downcasting, le compilateur ne fait pas de vérification. E définit bien une méthode me()	Méthode me() définie dans F
D d = (D) c;	Downcasting, le compilateur ne fait pas de vérification.	Un F n'est pas un D

Polymorphisme

Polymorphisme

- Le polymorphisme est le troisième concept essentielle d'un langage orienté objet après l'encapsulation et l'héritage.
- On peut caractériser le polymorphisme en disant qu'il permet de manipuler des objets sans en connaître (tout à fait) le type.
- 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.

Polymorphisme en JAVA

- En résumé, le polymorphisme en Java se traduit par :
  - Le upcasting,
  - La liaison dynamique des méthodes.

- Exemples:
  - Exemple1
  - Exemple2

Exemple 1

```
public class TestPoly{
 public static void main (String args[]){
   Point p = new Point (3, 5) ;
   p.affiche() ; // appelle affiche de Point
   PointGraphique pc = new PointGraphique (4, 8, "vert");
   p = pc ; // p de type Point, reference un objet de type
   PointGraphique
   p.affiche() ; // on appelle affiche de PointGraphique
   p = new Point (5, 7) ; // p reference a nouveau un objet
   de type Point
   p.affiche() ; // on appelle affiche de Point
 }
}
```

Exemple 2

```
public class TabHeterogene{
  public static void main (String args[]){
    Point [] tabPts = new Point [4];
    tabPts [0] = new Point (0, 2) ;
    tabPts [1] = new Pointgraphique (1, 5, "noir");
    tabPts [2] = new Pointgraphique (2, 8, "bleu");
    tabPts [3] = new Point (1, 2) ;
    // affichage des points
    for (Point p : tabPts)
      p.affiche();
  }
}
```


Polymorphisme, redéfinition et surcharge

```
public class A{
  public void f (float x) { ..... }
  .....
}
public class B extends A{
  // redefinition de f de A
  public void f (float x) { ..... }
  // surcharge de f de B
  public void f (int n) { ..... }
  .....
}
```

```
A a = new A(...) ;
B b = new B(...) ;
int n = 3 ;
a.f(n) ; // appelle f (float) de A (ce qui est logique)
b.f(n) ; // appelle f(int) de B comme on s'y attend
a = b ;  // a contient une reference sur un objet de type B
a.f(n) ; // appelle f(float) de B et non f(int)
```

Exercice 6

- Soit les classes suivantes:

```
class A {
    public void a() {
        System.out.println("a de A");
    }

    public void b() {
        System.out.println("b de A");
    }
}

class B extends A {
    public void b() {
        System.out.println("b de B");
    }

    public void c() {
        System.out.println("c de B");
    }
}
```


Exercice 6 (suite)

```
class A {
    public void a() {
        System.out.println("a de A");
    }

    public void b() {
        System.out.println("b de A");
    }
}

class B extends A {
    public void b() {
        System.out.println("b de B");
    }

    public void c() {
        System.out.println("c de B");
    }
}
```

```
public class Correction{
    public static void main(String[] args) {
        A a1 = new A();
        A b1 = new B();
        B a2 = new A();
        B b2 = new B();
        a1.a();
        b1.a();
        a2.a();
        b2.a();
        a1.b();
        b1.b();
        a2.b();
        b2.b();
        a1.c();
        b1.c();
        a2.c();
        b2.c();
        ((B) a1).c();
        ((B) b1).c();
        ((B) a2).c();
        ((B) b2).c();
    }
}
```

Exercice 6 (suite)

- Supprimez, **dans la classe Correction**, les lignes qui provoquent une erreur et indiquez si l'erreur se produit à la compilation ou à l'exécution.

- Quel est le résultat de l'exécution qui sera affiché à l'écran après suppression des instructions à problème ?