## Constructeurs et polymorphisme

Un constructeur est une méthode spécifiquement dédiée à la construction de l'**instance courante** d'une classe, il n'est pas prévu qu'il ait un comportement polymorphique.

Il est cependant possible d'invoquer une méthode polymorphique dans le corps d'un constructeur

☞ Ceci est cependant déconseillé : la méthode agit sur un objet qui n'est peut-être alors que partiellement initialisé !

Un exemple ...

## Constructeurs et polymorphisme (2)

```
abstract class A
{
   public abstract void m();
   public A() {
      m(); // méthode invocable de manière polymorphique
   }
}
class B extends A {
   private int b;
   public B() {
      b = 1; // A() est invoquée implicitement juste avant
   }
   public void m() { // définition de m pour la classe B
      System.out.println("b vaut : " + b);
   }
}
// .... dans le programme principal :
B b = new B();
```

☞ affiche : `b vaut 0`

## La super-classe `Object`

Il existe en Java une super-classe commune à toutes les classes : la classe `Object` qui constitue **le sommet de la hiérarchie**

Toute classe que vous définissez, si elle n'hérite d'aucune classe explicitement, dérive de `Object`

Il est donc possible d'affecter une instance de n'importe quelle classe à une variable de type `Object` :

```
Object v = new UneClasse (...); // OK
```

## La super-classe `Object` (2)

La classe `Object` définit, entre autres, les méthodes :

- `toString` : qui affiche juste une représentation de l'adresse de l'objet;
- `equals` : qui fait une comparaison au moyen de `==` (comparaison des références);
- `clone` : qui permet de copier l'instance courante

Dans la plupart des cas, ces définitions par défaut ne sont pas satisfaisantes quand vous définissez vos propres classes

☞ Vous êtes amenés à les **redéfinir** pour permettre un affichage, une comparaison ou une copie corrects de vos objets

☞ c'est ce que nous avons fait dans une séquence précédente avec `toString` !

☞ La classe `String` aussi par exemple redéfinit ces méthodes

## Exemple : redéfinition de `equals` héritée de `Object`

L'entête proposée pour la méthode `equals` dans une séquence précédente était :

```
public boolean equals(UneClasse arg)
```

or l'entête de la méthode `equals` dans `Object` est :

```
public boolean equals(Object arg)
```

☞ Nos définitions de `equals` constituaient jusqu'ici des **surcharges** et non pas des **redéfinitions** de la méthode `equals` de `Object` !

Dans la plupart des cas, utiliser une surcharge fonctionne sans problème, mais il est recommandé de **toujours procéder par redéfinition**.

## Surcharge, redéfinition (rappels)

On redéfinit (« override ») une méthode d'instance si les paramètres et leurs types sont identiques et les types de retour compatibles :

```
public boolean equals(Object o)
```

Si c'est le même nom de méthode seulement, on surcharge (« overload ») :

```
public boolean equals(UneClasse c)
```

## Redéfinition usuelle de `equals`

**Attention !** si l'on redéfinit `equals` pour la classe `Rectangle`, on doit pouvoir comparer un `Rectangle` avec n'importe quel autre objet : `unRectangle.equals("toto")` devrait retourner `false`.

```
class Rectangle {
  //...
  public boolean equals(Object autreObjet) {
   if (autreObjet == null)
    { return false; }
   else {
    if (autreObjet.getClass() != getClass())
     { return false; }
    else {
     Rectangle r = (Rectangle)autreObjet;
     return (largeur == r.largeur &&
             hauteur == r.hauteur);
    }
   }
  }
}
```