Héritage

1. Nouveaux mots clés

- **extends**
- **super**

2. Nouveaux concepts

- **redéfinition d'une méthode**
  la méthode dans la classe fille a même signature que dans la classe parente. Elle "remplace" celle de la classe parente qui était héritée. La méthode de la classe parente reste accessible en utilisant `super`.
- **polymorphisme** (lien dynamique) : le type d'une variable évolue en cours d'exécution en fonction de ses affectations (cf suite).

3. Règles :

3.1 Règle pour les constructeurs :

Tout constructeur appelle directement ou indirectement un constructeur de sa super-classe.
Si la 1ère instruction d'un constructeur n'est pas
super(...) avec ou sans arguments ou
this(...) avec ou sans arguments
un appel super() -sans argument- est ajouté automatiquement avant la 1ère instruction.

→ il faut qu'un tel constructeur existe dans sa super-classe.

Une classe a toujours au moins un constructeur.
Si une classe n'a aucun constructeur explicitement défini, elle dispose automatiquement du constructeur par défaut : constructeur sans argument et de corps vide.
A partir du moment où un constructeur a été défini, le constructeur par défaut n'existe plus. S'il est nécessaire, il faut le rajouter explicitement.

Exemple :

```
class MaClasse {
}
```

`MaClasse` possède le constructeur par défaut : `public MaClasse(){}`

```
class MaClasse {
   public MaClasse(...){...}
}
```

Le constructeur `public MaClasse(){}` n'existe plus.

On peut le rajouter explicitement

```
class MaClasse {
    public MaClasse(...){…}
    public MaClasse(){}
}
```

3.2 Typage statique et dynamique (polymorphisme)

Si un objet est déclaré de type T, on peut lui affecter un objet de la classe T ou d'une sous-classe (descendant) quelconque de T. L'inverse n'est pas admis.

Exemple :

```
class Parent {…}
class Descendant extends Parent {…}
class Test {
    public static void main (String[] args) {
          Parent p ;
          // p = new Parent(…) ;
          p = new Descendant(…) ;
    }
}
```

Compilation

`p` sera toujours et uniquement de type `Parent` (à cause de la déclaration `Parent p ;`).

→ le compilateur accepte uniquement les références aux attributs et méthodes connues au niveau de `Parent`

→ Pour appeler un attribut ou une méthode spécifique à `Descendant` faire un **cast** :

`((Descendant) p).methodeDescendant() ;`

Vous garantissez qu'à cette ligne de code, `p` est de type `Descendant` et que le programme peut se dérouler sans erreur.

Exécution

Le type effectif de `p` peut varier en fonction de ses affectations. Dans notre exemple `p` devient de type `Descendant`. Ce sont donc les méthodes définies au niveau de `Descendant` qui seront appelées (important pour les méthodes redéfinies).

Instructions pour connaître le type effectif de `p`

- **`getClass`** : `if (p.getClass() == (Class) Descendant.class)`
- **`instanceOf`** : `if (p instanceOf Descendant)`
  vrai pour la classe effective et toutes ses classes parentes

A utiliser avec parcimonie.