# R2.01 : cours n°7

I.Borne

1. Propriété « final »
2. Classe abstraite
3. Introduction au typage
4. Utilisation des types génériques



# 1 . La propriété « final » de java

La propriété « final » peut s'appliquer à différents éléments : attribut d'instance, de classe, méthode d'instance, de classe classe.

Un attribut d'instance de type primitif devient constant :

```
public final double PI = 3.14159;
```

La propriété UML associée dans une diagramme de classes est {frozen}

Un attribut de classe de type primitif devient constant :

```
public static final double PI = 3.14159;
```

La propriété UML associée dans une diagramme de classes est {frozen}

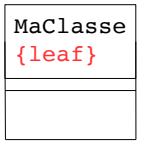


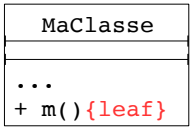
# Classe ou méthode « final »

- Une classe « final » ne pourra pas avoir de sous-classes.
- Une méthode d'instance « final » ne pourra pas être redéfinie dans une sous-classe

```
public final class Maclasse {
  …
}
```

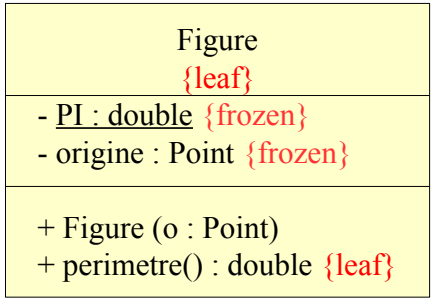
```
public class MaClasse {
   …
   public final void m(){
    …
   }
}
```






# Toutes les notations UML pour « final »



# 2 . Modélisation et héritage

- Dans une modélisation il est souvent souhaitable d'organiser ses classes avec un héritage et d'avoir une classe de base au sommet d'un héritage.
- Cette classe de base permet de factoriser ce qui est commun aux sous-classes.
- Par contre on peut ne pas souhaiter créer des objets de cette classe.
- Pour cela une classe peut être déclarée abstraite.



# Classes abstraites

Une **classe abstraite** est une classe qui n'a pas pour objet de créer des instances.
Son objectif est de servir de superclasse à d'autres classes.
Les classes abstraites peuvent contenir des méthodes abstraites.

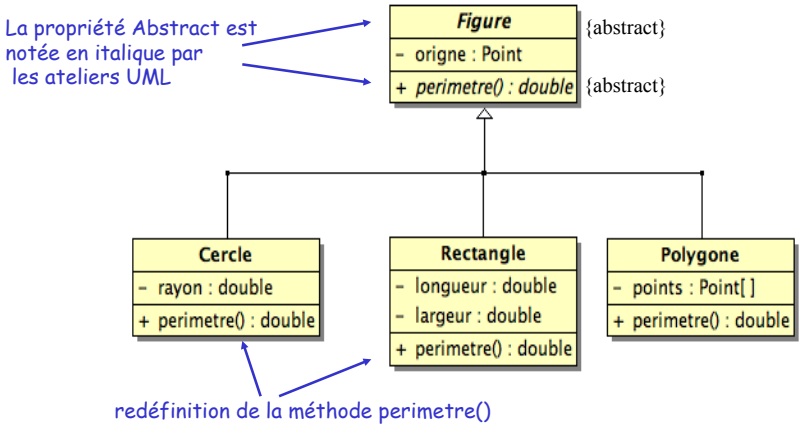
Déclarer une classes abstraite a plusieurs conséquences :

- Il est impossible de créer des instances à partir d'une classe abstraite, mais elle peut posséder des constructeurs.
- Seule une classes abstraite peut avoir des méthodes abstraites.
- Une classe abstraite avec des méthodes abstraites force ses sous-classes à redéfinir et à implémenter les méthodes abstraites.



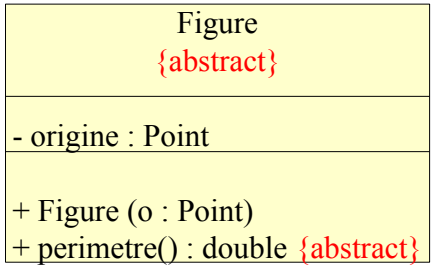
# Exemple de la classe **`Figure`**

Le rôle d'une classe abstraite est de regrouper des sous-classes concrètes






# Notation UML - notation Java




```
public abstract class Figure {
…
}
```

# Méthodes abstraites

- Les *méthodes abstraites* permettent de prévoir des opérations similaires sur des objets.
- Elles sont destinées à être définies ou redéfinies dans les sous-classes.
- Une définition de méthode abstraite est constituée d'une signature de méthode sans corps et marquée par le mot-clé abstract.

```
public abstract class Figure {
   private Point origine;

   public Figure(Point o) {
       this.origine = o;
   }

   public abstract double perimetre();
}
```



# Redéfinitions dans les sous-classes

```
public class Cercle extends Figure {
   private double rayon;

   public double perimetre () {
        return (2 * Math.PI * rayon);
   } . . .
}
```

```
public class Rectangle extends Figure {
   private double longueur, largeur;

   public double perimetre () {
       return (2*(longueur + largeur));
   }
  …
}
```



# Synthèse sur la propriété abstraite

- Une classe abstraite est une classe qui possède au moins une méthode abstraite.
- Une classe abstraite possède un constructeur qui sera appelé par les sous-classes lors du chaînage des constructeurs.
- Une classe abstraite peut posséder des variables d'instance ou de classe qui seront héritées par les sous-classes, et initialisées dans le constructeur de la classe abstraite



# 3 . Introduction au typage

- Le typage est une façon de restreindre l'ensemble des programmes syntaxiquement corrects.
- Le type défini par une classe est défini par :
  - la liste de ses attributs : type et visibilité
  - la liste des signatures de ses méthodes
- En Java le type défini par une classe porte le nom de la classe.
- *Dans ce cours nous ne faisons qu'une introduction simple du typage.*

# Les types en java

- Java fournit une sécurité de type (contrairement à C,C++) :
  - C'est un langage fortement typé
  - Il fournit une gestion de stockage automatique des objets
  - Il vérifie tous les accès aux tableaux pour garantir qu'ils sont dans les limites
- Une classe est la représentation d'un type de donnée.
- Les types sont organisés dans une hiérarchie.
- La relation de sous-type est transitive :
  - Si R est un sous-type de S et S un sous-type de T alors R est un sous-type de T

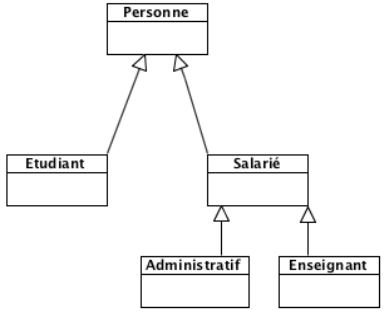


# Hiérarchie de type

- Java supporte la hiérarchie de types dans laquelle un type peut être le super-type des autres types.
- Tous les types d'objet sont sous-types d'Object.
- Le type apparent d'une variable est le type compris par le compilateur à partir de l'information disponible à la déclaration.
- Le type actuel d'un objet est son type réel : le type qu'il reçoit quand il est créé.
- Java garantit que le type apparent de toute expression est un super-type de son type actuel.



# Exemple de typage




Que peut-on dire des expressions suivantes (valides ou non) ?

```
Etudiant e1 = new Etudiant() ;
Personne e2, e3;
e2 = new Salarié() ;
e3 = new Enseignant() ;

Salarie e4 ;
e4 = new Etudiant() ;
e4 = new Administratif() ;
```



# 4 . Utilisation des types génériques

- Un code qui utilise les types génériques présente un certain nombre d'avantages sur ceux qui ne les utilisent pas :
  - Une vérification plus forte des types à la compilation
  - Elimination des cast : le typage peut être vérifié
  - Permet d'implémenter des algorithmes génériques

# Un premier exemple

- Un type générique est une classe ou une interface générique qui peut être paramétrée sur des types.
- Exemple :
  - Une classe Box qui opère sur des objets de tout type et qui fournit 2 méthodes :
    - **set** qui met un objet dans la boîte
    - **get** qui récupère l'objet

**1ère version : on type avec Object**

Quelles contraintes a-t-on ?

```
public class Box {
   private Object object ;
   public void set(Object object) {this.object = object;}
   public Object get() { return object;}
}
```



# Définition d'une classe générique

- Une classe générique rajoute dans son entête les paramètres des types.
- **2ème version** : On remplace toutes les occurrences de Object par T

```
public class Box <T> {
    private T t ;
    public void set(T t) {this.t = t;}
    public T get() { return t;}
}
```

- Le type peut être n'importe quel type non-primitif.
- Par convention les noms des types paramètres sont des lettres majuscules, on les retrouve dans les définitions des classes du framework Collection (E,K,N, T,V)



# Invocation et instantiation

- `Box<Integer> integerBox;`
- `Box<Integer> integerBox = new Box<Integer>();`
- `Box<String> stringBox = new Box<String>();`

```
integerBox.set(4 );
int n = integerBox.get()

stringBox.set("Bonjour");
System.out.println(stringBox.get());
```



# Types paramétrés multiples

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V>{
    private K key;
    private V value;
    . . .
}
```

# Exemple d'une classe Pile paramétrée

```
import java.util.ArrayList;

public class Pile<T> {
  private ArrayList<T> items;

 public Pile() {
  this.items=new ArrayList<T>();
 }

 public T empiler(T item) {
    this.items.add(item);
    return item;
 }

 public boolean estVide(){
  return(this.items.size()==0);
 }
```

```
public T depiler() {
 T item=null;
 if (!this.estVide()){
  item = this.items.get(
              this.items.size()-1);
  this.items.remove(item) ;
 }
 else {
  System.out.println("pilevide") ;
 }
 return item;
}

}
```



# Utilisation de la classe Pile

```
public class ClientPile{
   public static void main(String[]args) {
      Pile<String> p = new Pile<String>();
      System.out.println("empile :" + p.empiler("a"));
      System.out.println("empile :" + p.empiler("b"));
      System.out.println("depile :" + p.depiler());
      System.out.println("depile :" + p.depiler());
      System.out.println("depile :" + p.depiler());
  }
}
```

Résultat :

```
empile :a
empile :b
depile :b
depile :a
pile vide
depile :null
```



# Rappel : petits conseils de bonnes pratiques (1)

- `String` (revoir le TD de GL)
  - Ne pas faire : `String s = new String("silly");`
    Cela crée 2 instances de la classe `String`
  - Mais : **`String s = "No longer silly";`**

- Egalité des booléens
  - **Ne pas tester des égalités avec true ou false :**
    `if (untest == true)`
  - Utiliser : `if (test)`



# Petits conseils de bonnes pratiques (2)

1 - Rappel sur les boucles
- Exploration du 1er au dernier élément d'une collection ou d'un tableau
- Pas d'interruption au milieu de la boucle
- Utiliser **`for`** avec indice ou la forme **`for`** each

2 - Rappel sur le parcours d'une collectionou d'un tableau pour la recherche d'un élément et avec interruption possible
- On parcourt une collection avec un itérateur ou des indices pour un tableau
- La condition de sortie du **`while`** porte sur l'avancement des éléments et sur la condition de recherche

3 - Retour d'une méthode
- 1 seul ***return*** à la fin de la méthode