



# **P.O.O. (Programmation Orientée Objet)**

**CHOUTI Sidi Mohammed**

Cours pour L2 en Informatique

Département d'Informatique

Université de Tlemcen

*2019-2020*

1. Introduction à la Programmation Orientée Objet
2. Classes et Objets
3. Héritage, polymorphisme
4. Abstraction et déclaration finale
- 5. Interface, implémentation et Paquetage**
6. Classes Courantes en Java
7. Gestion des Exceptions
8. Interfaces graphiques

Il existe une autre technique pour introduire de l'abstraction

une **interface** est une classe **complètement abstraite**, c.-à-d. faite de :

- Méthodes **publiques abstraites**
- Variables **publiques statiques finales** (des constantes de classe)

- Toutes les **méthodes** sont **implicitement** déclarées **public abstract**
- Toutes les **variables** sont **implicitement** déclarées **public static final**

```
public interface Surfaceable {  
    double surface();  
    // équivaut à public abstract double surface();  
}  
  
public interface I {  
    int field = 10;  
    // équivaut à public final static int field = 10;  
}
```

**Toutes les méthodes déclarées (abstract) dans l'ensemble des interfaces dont on revendique l'implémentation doivent être implantées**

## Interface (exemple)

```
interface Pile {  
    boolean estVide();  
    void empiler(Object x);  
    Object depiler();  
}
```

---

```
class PileTab implements Pile {  
    Object[] tab = new Object[100];  
    int n = 0;  
  
    public boolean estVide() {return n == 0;}  
    public void empiler(Object val) {tab[n++] = val;}  
    public Object depiler() {return tab[--n];}  
}
```

```
public static void main(String[] args) {  
  
    Pile unePile = new PileTab();  
    ...  
    uneApplication(unePile);  
    ...  
}
```

```
interface I1 {  
    void m();  
}  
  
abstract class C1 {  
    abstract void g();  
}  
  
class C2 extends C1 implements I1 {  
    void m(){ // Le code de m }  
    void g(){ // Le code de g }  
}
```



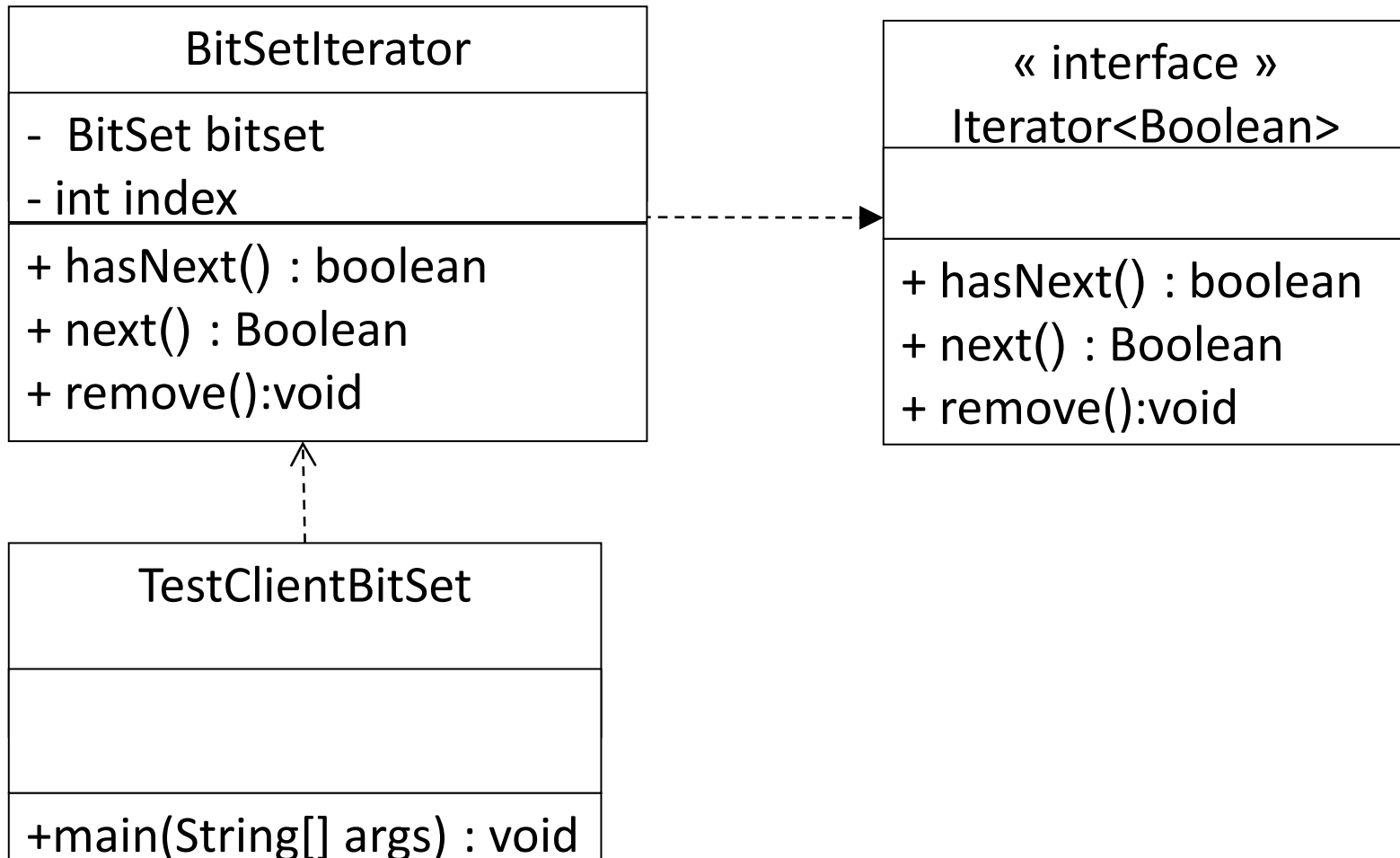
## Héritage entre interfaces

```
interface I2 extends I1 {  
    void n();  
}  
  
abstract class C3 implements I2 {  
    void m(){  
        // Le code de m();  
    }  
}
```

Héritage **simple** vs Héritage **multiple**

```
class MyClass
    extends MotherClass
        implements Interface1, Interface2 {
        ...
    }
```

## Exercice



## Exercice

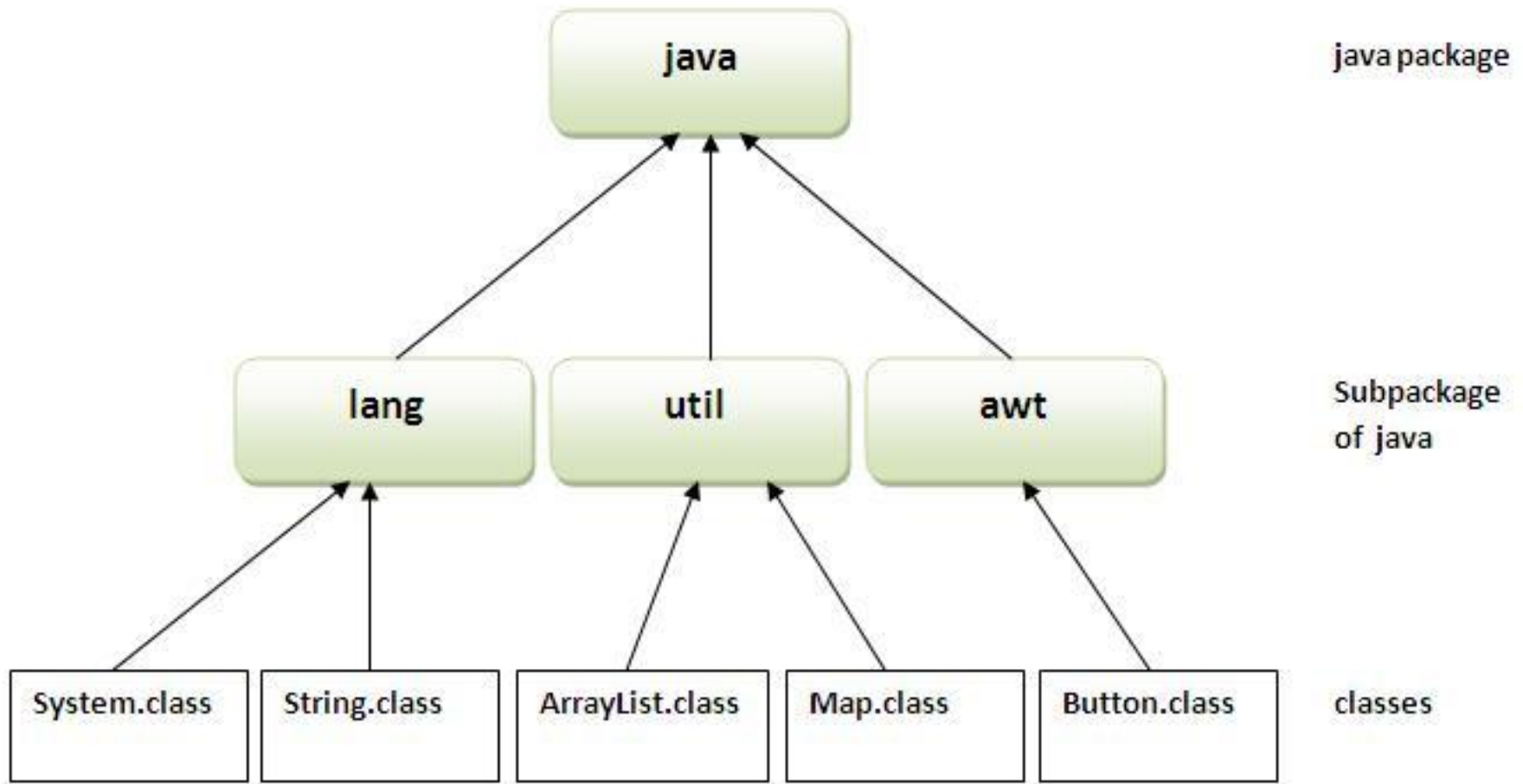
1- Ecrire une classe **BitSetIterator** qui implémente **Iterator**. **BitSet** est une classe qui implémente un vecteur de taille dynamique et dont les éléments sont des booléens. ses méthodes **set(i :int)** et **get(i :int)** permettent de modifier (par T) et de renvoyer la valeur du  $i^{\text{ème}}$  élément.

2- Ecrire le programme **TestClientBitSet** qui permettra d'initialiser un objet **BitSet**, d'accéder aux éléments de ce dernier à travers un objet **BitSetIterator** et de les afficher.

## Paquetage ou package

Un paquetage est un **regroupement** de classes. Les paquetages sont organisés **hiérarchiquement** comme des répertoires de classes.

## Arborescence des packages



## Arborescence des packages

```
java
```

```
|  applet
```

Le paquetage contenant les classes concernant les *applets*

```
|  awt
```

L'abstract Windowing Toolkit

```
|  beans
```

```
|  io
```

Le paquetage des classes gérant les entrées-sorties

## Arborescence des packages

`lang` Le paquetage contenant les classes de base de Java.

`ArithmeticException`

`...`

`Boolean`

`Class`

`Object`

`reflect`

`...`

`Method`

`...`

`String`

`Math`

`...`



## Utilisation

Pour utiliser dans un fichier java une classe **C d'un paquetage p** :

1- donner le nom de la classe in extenso :

```
class Truc{  
    ...  
    p.C variable = ...  
    ...  
}
```

2- ajouter une directive en tête du fichier :

```
import p.C;  
class Truc{  
    ...  
    C variable = ...  
    ...  
}
```

## Organisation

Pour **organiser ses propres classes** en paquetage :  
mettre en tête de fichier la directive **package** correspondante

```
package monpaquetage;  
class Truc{  
    ...  
}
```

```
package monpaquetage;  
class Machin{  
    ...  
}
```

## Paquetage

- Les paquetages représentent des **espaces de nommage** : deux paquetages peuvent contenir des classes de même nom.
- Les paquetages permettent l'organisation des classes par **thèmes, par applications.**

Exemples : **java.applet** contient les classes dédiées à la réalisation d'applications clientes pour pages web,  
**java.security** regroupe les classes dédiées à la gestion de la sécurité.

## Le concept

L'**encapsulation** est un mécanisme consistant à **cacher l'implémentation** de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.

L'encapsulation permet de définir des **niveaux de visibilité** des éléments d'une classe.

privé - protégé - public

## Principe

Une classe doit rendre visible ce qui est **nécessaire** pour manipuler ses instances et rien d'autre.

## Objectif

L'encapsulation permet donc de garantir **l'intégrité** des données contenues dans l'objet.

## Périmètre

En Java, il existe deux périmètres de visibilité :

**les classes et les paquetages.**

## Les modificateurs d'accès

Les 4 niveaux d'encapsulation sont par ordre de visibilité croissante :

- un membre privé (private) n'est visible que dans les instances directes de la **classe** où il est déclaré.
- un membre sans modificateur est visible uniquement dans les instances directes de la **classe** où il est déclaré et dans celles des **classes du même paquetage**.
- un membre protégé (protected) n'est visible que dans les instances, directes ou non, de la **classe** où il est déclaré (et donc dans les instances des **sous-classes**) ainsi que dans les **classes du même paquetage**.
- un membre public (public) est visible par **n'importe** quel objet

## En résumé

Modificateur	private	<i>aucun</i>	protected	public
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui



## Exemple

```
package p1;  
class X{  
    private int a;  
    int b;  
    protected int c;  
    public int d;  
}
```

```
package p1;  
class Z{  
    void m(){  
        X x=new X();  
        System.out.print(x.b+x.c);}  
}
```

```
package p2;}  
class Y extends X{  
    void m(){  
        System.out.print(a); // erreur  
        System.out.print(b); // erreur  
        System.out.print(c+d); // correct  
}
```

```
package p2;  
class W{  
    X x=new X();  
    void m(){ System.out.print(x.d); }  
}
```

# Encapsulation et paquetage

## Exemple : Trouvez les erreurs (1)

```
package p1;  
  
class A {  
    public int w = 0;  
    protected int x = 1;  
    private int m() { return 2; }  
}
```

## Encapsulation et paquetage

### Exemple : Trouvez les erreurs (2)

```
package p1;

public class B extends A {
    void testB() {
        A unA = new A();
        p2.C unC = new C();
        int x = unA.w + unA.x + unA.m() + w + x;
        m();
    }
}
```

## Encapsulation et paquetage

### Exemple : Trouvez les erreurs (3)

```
package p2;
import p1;

public class C extends B {
    void testC() {
        A unA = new A();
        B unB = new B();
        p1.A unAA = new p1.A();
        int y = unAA.x + x + unB.x;
    }
}
```

# Encapsulation et paquetage

## Exemple : erreurs trouvées (1)

```
package p1;  
  
public class A {  
    public int w = 0;  
    public int x = 1;  
    protected int m() { return 2; }  
}
```

## Exemple : erreurs trouvées (2)

```
package p1;

public class B extends A {
    void testB() {
        A unA = new A();
        p2.C unC = new p2.C();
        int x = unA.w + unA.x + unA.m() + w + this.x;
        m();
    }
}
```

## Encapsulation et paquetage

### Exemple : erreurs trouvées (3)

```
package p2;
import p1.*;

public class C extends B {
    void testC() {
        A unA = new A();
        B unB = new B();
        p1.A unAA = new p1.A();
        int y = unAA.x + x + unB.x;
    }
}
```

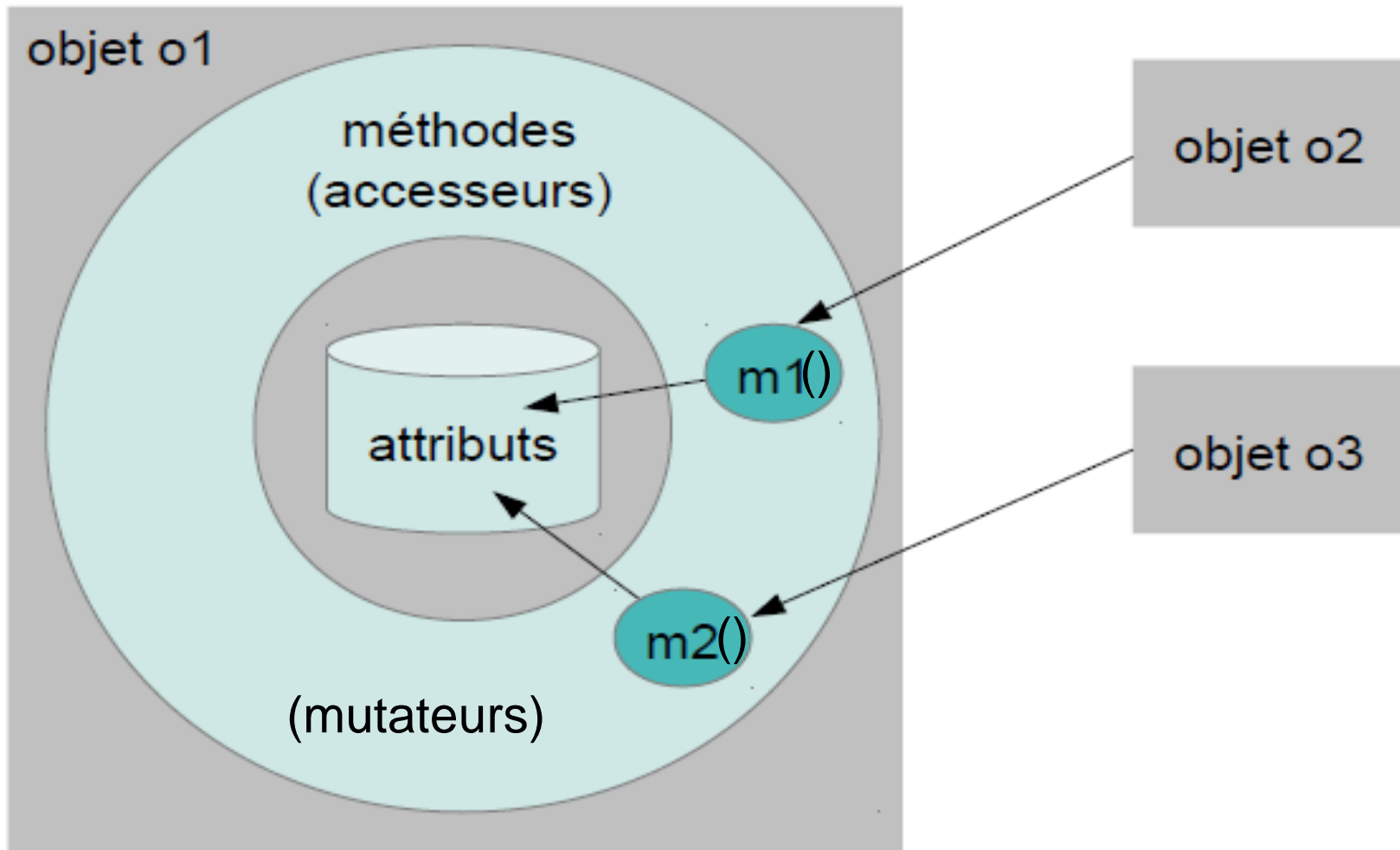
## Accesseur et Mutateur : Getter et Setter

Par défaut, **les attributs** doivent être **cachés**. Leurs valeurs ne doivent être **visibles et modifiables** qu'au travers **des méthodes**.

Les méthodes **intermédiaires** qui ne sont pas destinées à être utilisées à l'extérieur de la classe doivent être **cachées**.



## Accesseur et Mutateur : Getter et Setter



## Getter et Setter

```
class Classe {  
  
    private type attribut;  
  
    public type getAttribut(){  
        return attribut;  
    }  
  
    public void setAttribut(type a){  
        attribut = a;  
    }  
  
}
```

## Getter et Setter

```
class TestClasse {  
  
    public static void main(String args[]){  
  
        Classe c = new Classe();  
        c.setAttribut(type);  
        System.out.print(c.getAttribut);  
    }  
}
```

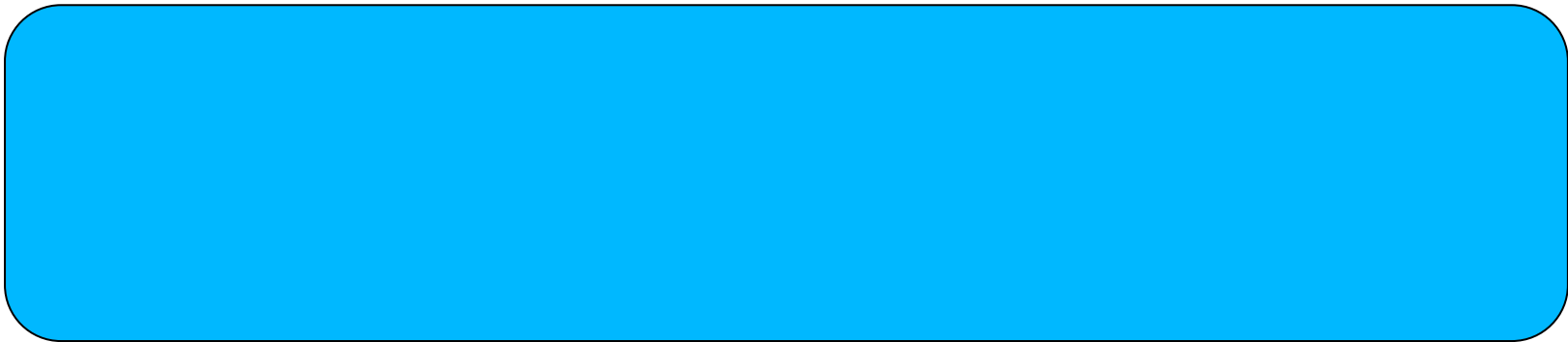
## Exercice

```
class Employe {  
    private int nSS;  
    private String nom;  
    private int age;  
    ...  
}
```

```
public class TestEmploye{  
    public static void main(String args[]){  
        Employe e= new Employe();  
        ...  
    }  
}
```

## Déductions

- Une méthode abstraite ne peut être privée car



- La redéfinition d'une méthode doit avoir une visibilité au moins égale à celle de la méthode de la super-classe

# Encapsulation et paquetage

## Exercice

Dans certains cas, on n'a besoin que **d'une seule instance d'une classe donnée.**

Écrire une classe qui **n'autorise qu'une seule instanciation.**