Chapitre2

Déclarations et contrôle d'accès

1 Déclaration de classes

1.1 Règles de déclaration de fichiers source

- Un fichier code source contient une seule classe déclarée public et plus qu'une classe non déclarée public. Le nom du fichier (.java) doit être le nom de cette classe déclarée public
- Un fichier qui ne contient pas de classes déclarées public peut avoir un nom qui ne convient pas aux noms de classes du fichier
- Si la classe fait partie d'un package, une instruction package doit être la première ligne dans le fichier code source, avant les instructions import s'il y'en a.
- Il n'est pas possible de déclarer plusieurs classes de différents packages dans un même fichier

1.1.1 Les modificateurs (Modifiers)

La déclaration suivante est correcte :

```
class MyClass { }
```

Mais on peut ajouter des modificateurs avant la déclaration class.

- Modificateurs d'accès: public, protected, private.
- Modificateurs de Non-accès : strictfp, final, abstract

Il y'a quatre niveaux de contrôle d'accès et trois modificateurs d'accès. Le quatrième niveau de contrôle d'accès est obtenu lorsqu'il n'y a pas utilisation d'un modificateur d'accès dans ce cas le mode d'accès est dit : par défaut. Chaque, classe, méthode et variables d'instance déclarés ont un contrôle d'accès.

Les modificateurs d'accès utilisés pour une classe sont : **public** et **default**, les deux autres modificateurs d'accès n'ont pas de sens pour une classe.

1.1.2 Accès à une classe

Une classe A as un accès à une classe B veut dire elle peut:

- Créer une instance de la classe B
- Hériter *(extends) de la* classe B (devenir une sous classe de la classe B => class A extends B).
- *Accéder* en tenant compte du contrôle d'accès à certaines méthodes et variables de la classe B

Accès veux dire visibilité. Si une classe A ne peux pas voir une classe B, elle ne peut utiliser aucun membre de B.

Accès par défaut : *(pas de modificateur)*

- Le niveau d'accès est uniquement au niveau de celui du package
- La classe ne *peut* être vue que par une classe du même package

Si une classe A et une classe B sont dans des packages différents, et A a un accès par défaut, alors:

- la classe B ne pourrait ni importer la classe A ni créer une instance de la classe A.
  Exemple: A ob= new A ();
- la classe B ne peut pas déclarer une variable ou un type de fonction (return type) de classe A. Pour B la classe A n'existe même pas

  Example :

```
import packA.A;
class B extends A{
A claculer (){
A hh= new A();
return hh;
}
```

Erreur à la compilation: Can't access class packA.A. Class or interface must be public, in same package, or an accessible member class.import packA.A;

Accès public

Une classe d'accès public est accessible à partir de tous les packages. Cependant il faut tenir compte, si la classe public à utiliser n'est pas dans le même package de la classe en cours de développement, il faut l'importer.

Exemple:

```
package packA;
public class A{
}
```

Une classe B qui hérite de A, doit importer A comme suit :

```
package packB;
import packA.A;
classe B extends A{
A claculer (){
A hh= new A();
 return hh;
}
```

1.1.3 Modificateurs de non accès à la classe

Il est possible de modifier la déclaration de classe en ajoutant l'un des modificateurs : final ou abstract. Ces modificateurs peuvent être en addition au modificateur de contrôle d'accès utilisé, donc, par exemple, une classe peut être déclaré à la fois public et final.

Θ La combinaison des modificateurs n'est pas toujours possible,

Exemple : Ne **jamais** combiner final et abstract.

- **final :** le mot clé final signifie que la classe ne peut pas être héritée. Une tentation d'utiliser extends pour hériter d'une classe final, provoque une erreur de compilation.

Le mot clé final est utilisé lorsqu'il y a un besoin absolue pour garantir qu'aucune des méthodes dans la classe sera redéfinie (overridden).

Exemple:

```
final public class A{
}
public class B extends A {
}
```

- ***abstract :*** une classe abstract ne peut jamais être instanciée (créer des objets de son type). Sa seule mission est d'être héritée.

Exemple 1:

```
public abstract class A{
……;
A ob= new A();
}
```

Exemple 2: soit la classe abstract suivante:

```
abstract class Car {
   private double price;
   private String model;
   private String year;
   public abstract void goFast();
   public abstract void goUpHill();
   public abstract void impressNeighbors();
   public void doThis() { }     <-----     ……..
   // Additional, important, and serious code goes here
}
```

Soit une autre classe BMW dans le même package que Car et qui instancie la classe Car.

```
class BMW{
car C = new Car() ;
 }
```

Est-ce que les classe Car et BMW compilent?

La class Car compile alors que la compilation de la classe BMW produit une erreur de compilation du genre :

```
AnotherClass.java:7: class Car is an abstract
class. It can't be instantiated.
      Car x = new Car();
1 error
```

⇨ **Noter que les méthodes marquées abstract se terminent par un point-virgule (n'a pas de corps) au lieu des accolades**

⇨ Il est possible de mettre des méthodes non abstract dans une classe abstract.

⇨ Si une méthode dans une classe est déclarée abstract, la classe doit être déclarée abstract

⇨ une classe déclarée à la fois abstract et final ne compile pas (abstract et final sont contradictoires, une classe abstract doit être héritée alors qu'une classe final ne peut pas être héritée).

⇨ Une classe abstract peut ne pas avoir des méthodes abstract

2 Déclaration des interfaces

- La création des interfaces définie un contrat de ce qu'une classe peut faire sans décrire comment la classe va faire.
- une classe qui implémente une interface doit développer toutes les méthodes de cette interface
- L'intérêt des interfaces est de faire relation entre les classes qui n'ont aucune relation d'héritage
- Une interface peut avoir des méthodes déclarées abstract
- A partir de java 8, une interface peut avoir des méthodes déclarées **default** et des méthodes déclarées static
- Les méthodes **default** sont définies par l'interface. Elles ne nécessitent pas d'être redéfinie dans la class implémentant l'interface
- Les méthodes **static** sont définies par l'interface
- Dans l'interface, le compilateur ajoute automatiquement public à toutes ses méthodes, si le développeur ne les a pas fournit.
- Les variables définies dans une interface doivent être public, static et final c'est-à-dire des constantes,
- Les méthodes d'une interface ne doivent pas être final.
- Une interface peut hériter (*extends)* une ou plusieurs autres interfaces.
- une interface ne peut pas implémenter une autre interface ou une classe

Exemple : soit l'interface Fly et la classe Eagle suivants :

```
public interface Fly {
public int getWingSpan() throws Exception;
public static final int MAX_SPEED = 100;
public default void land() {
System.out.println("Animal is landing");
}
public static double calculateSpeed(float distance, double time) {
return distance/time;
}
}
public class Eagle implements Fly {
public int getWingSpan() {
return 15;
}
public void land() {
System.out.println("Eagle is diving fast");
}}
```

Exercice: mettre une croix devant les méthodes qui ne compilent pas.

```
void bounce();            public void bounce();      static void bounce();
abstract void bounce();  public abstract void bounce();  private void bounce();
abstract public void bounce();  final void bounce();   protected void bounce();
Static void bounce(){}     default int bounce();     public default int bounce(){}
```

2.1 Déclaration des constantes dans une interface

Ceci garantie que chaque classe qui implémente l'interface aura accès à la même constante. La règle à respecter pour déclarer les constantes dans une interface est qu'ils doivent être :

```
public static final
```

Toutes les constants d'interface sont déclarées public static final, implicitement.

Exemple: soient les deux codes suivants dans deux fichiers différents:

```
interface Foo {
  int BAR = 42;
  void go();}
class Zap implements Foo {
  public void go() {
     BAR = 27;
  }}
```

La valeur de BAR ne doit pas être modifiée, même dans l'interface elle-même, c'est pourquoi BAR=27 ; ne compile pas.
Les déclarations suivantes dans une interface sont tous identiques :

```
public int x = 1;         // Looks non-static and non-final, but isn't!
int x = 1;                // Looks default, non-final, non-static, but isn't!
static int x = 1;         // Doesn't show final or public
final int x = 1;          // Doesn't show static or public
public static int x = 1;        // Doesn't show final
public final int x = 1;         // Doesn't show static
static final int x = 1          // Doesn't show public
public static final int x = 1;  // what you get implicitly
```

3 Déclaration des membres de classes

Les méthodes et les variables d'instance (non locales) sont les membres de la classe. Il est possible de modifier un membre avec les modificateurs d'accès ou de non accès et aussi de combiner entre les modificateurs.

3.1 Les modificateurs d'accès

Il existe quatre modificateurs d'accès pour les variables et les méthodes : public, protected, default et private. Une classe A a un accès à un membre de classe B, signifie que les membres de la classe B sont visibles pour la classe A. Il faut comprendre deux différents états d'accès :

- *Le premier type d'accès :* si la méthode d'une classe peut accéder à un membre d'une autre classe, utilisant l'opérateur (.) pour invoquer la méthode ou utiliser une variable. Par exemple:

```
class Zoo {
  public String coolMethod() {
    return "Wow baby";  }}
class Moo {
  public void useAZoo() {
    Zoo z = new Zoo();
    System.out.println("A Zoo says, " + z. coolMethod());  }
}
```

- *Le second type d'accès :* quand un membre d'une sous-classe peut accéder à un membre de la superclasse à travers l'héritage. Si un membre est hérité dans une sous-classe, il sera comme un membre déclaré dans cette dernière. Par exemple :

```
class Zoo {
  public String coolMethod() {
    return "Wow baby"; }}
class Moo extends Zoo {
  public void useMyCoolMethod() {
    System.out.println("Moo says, " + this.coolMethod());
    Zoo z = new Zoo();
    System.out.println("Zoo says, " + z.coolMethod());}}
```

Il faut savoir l'effet de différentes combinaisons de mode d'accès d'une classe et d'un membre (exemple : une classe déclarée avec un accès par défaut ayant des variables déclarés public). Il faut au début, voir le niveau d'accès de la classe. Si la classe n'est pas visible à une autre classe, alors aucun de ses membres ne le sera, même si le membre est déclaré public.

3.1.1 Membres d'accès "public"

- Si la classe est visible, une méthode ou une variable déclarée public, signifie que toutes les autres classes peuvent y accéder indépendamment du package auquel elle appartient.

Exemple : Soient les deux fichiers suivants:

```
package mytests;
public class Sludge {
  public void testIt()
{ System.out.println("sludge"); }
}
```

```
package verifier;
import mytests.*;  // Importer toutes les classes du package mytest ;
class Goo {
  public static void main(String[] args) {
    Sludge o = new Sludge();
    o.testIt();}}
```

Est-ce que Goo peut invoquer la méthode testIt de Sludge?

=> Goo et Sludge sont dans des packages différents. Cependant, Goo peut invoquer la méthode dans Sludge sans problème car la classe Sludge et sa méthode testIt() sont marquées public.

- Une sous-classe (subclass) hérite les membres déclarés public de sa superclass peu importe si les deux classes sont dans le même package.

Exemple :

```
package mytests;
public class Roo {
  public String doRooThings() {
    return "fun";  }}
```

La classe Roo déclare public le membre doRooThings(). Une sous-classe de Roo peut appeler sa méthode possédée doRooThings().

```
package verifier;
import mytests.Roo;
class Cloo extends Roo {
  public void testCloo() {
    System.out.println(doRooThings());  }}
```

⇨ Dans la classe Cloo, la méthode doRooThings() est invoquée dans la méthode testCloo sans avoir besoin d'utiliser une référence. Dans une méthode d'instance, un membre est invoqué sans l'opérateur (.), signifie que la méthode ou la variable, appartient à la classe ou est écrit le code. Ça signifie aussi, que les membres sont implicitement accessibles utilisant la référence « this ». donc dans le code précédent, peut écrit comme this.doRooThings().

⇨ La référence « this » réfère toujours à l'objet courant.

3.1.2 Membres d'accès "private"

Les membres marqués "private" ne sont accessibles que par la classe dans laquelle ils sont créés.
Exemple: Si on Modifier l'accès de la méthode « doRoothings » dans la classe précédente « Roo » en le rendant « private », est-ce qu'elle sera accessible ?

```
package mytests;
public class Roo {
  private String doRooThings() {
    // imagine the fun code that goes here, but only the Roo class knows
    return "fun";}}
```

Soit le code suivant:

```
package verifier;
import mytests.Roo;
class  UseARoo {
  public void testIt() {
    Cloo r = new Cloo(); // class Cloo est public (peut être instanciée)
    System.out.println(r.doRooThings()); //Compiler error!
  }}
=> erreur de compilation
```

Les sous-classes ne peuvent pas hériter les membres privés (private).

Est-ce que la classe Cloo qui hérite de la classe Roo compile ?

=> à la compilation de Cloo, l'appel de la méthode private `doRooThings()` entraine une erreur de compilation.

3.1.3 Membres d'accès "default"

Un membre déclaré *default* peut être accessible uniquement si la classe qui lui accède appartient au même package.

Exemple: soient les classes suivantes, est ce que AccessClass compile ?

```
package mytests;
public class OtherClass {
  void testIt() {   // Pas de modificateur signifie; la méthode a un accès
                    //default
    System.out.println("OtherClass");  }}
```

```
package verifier;
import mytests.OtherClass;
class AccessClass {
  static public void main(String[] args) {
    OtherClass o = new OtherClass();
    o.testIt();  }}
```

=> erreur de compilation : l'accès de testIt () est par défaut la classe AccessClass est dans un package différent de celle de testIt donc AccessClass ne peut pas utiliser la méthode testIt

3.1.4 Membres d'accès "protected"

Un membre déclaré *protected* peut être accessible :

- Par une autre classe ou par héritage avec une sous-classe dans le même package
- Par héritage avec une sous classe dans un package différent. Dans ce cas la sous classe ne peut pas accéder au membre en utilisant une référence à la superclasse mais elle peut y accéder en utilisant une référence à la classe fille (sous-classe)

Exemple : Dans cette classe, une variable d'instance est déclarée protected.

```
package mytests;
public class Parent {
   protected int x = 9; // accès:  protected
}
```

Par quelles classes x peut être accessible ?

→La variable x est accessible par ……

……

……

Exemple : Soit une sous-classe child qui hérite de la classe Parent, existante dans un package différent et accède à la variable x :

```
package verifier; // package différent
import mytests.Parent;
class Child extends Parent {
   public void testIt() {
      System.out.println("x is " + x); // Pas de problème; Child hérite x =>
                                       //accès sans référence
   }}
```

On modifie la classe Child en ajoutant une instruction qui crée un objet Parent p et accède à x en utilisant la référence p:

```
package verifier;
import mytests.Parent;
class Child extends Parent {
   public void testIt() {
      System.out.println("x is " + x);
      Parent p = new Parent();
      System.out.println("X in parent is " + p.x); // erreur de compilation
   }}
```

Est-ce que le code compile ?

=>……..

Qu'est-ce qu'il faut pour que le code compile ?

…….

…….

Note: une fois la sous-classe –hors du package- hérite un membre « protected », il devient "private" pour les autres classes à l'exception des sous-classes de la sous-classe.

Le tableau suivant montre un résumé de tous les modes l'accès et visibilité.

Visibilité	Public	Protected	*Default*	Private
De la même classe	…..	……	…..	…..
De n'importe quelle classe dans le même package	…..	……	…..	…..
D'une sous-classe dans le meme package	…..	……..	……	….
D'une sous-classe dans un autre package	…..	……..	…..	….
De n'importe quelle classe qui ne soit pas une sous classe hors du package	……	…….	…..	….

4 Méthodes, variables et modificateurs d'accès

4.1 Modificateurs d'accès et variables locales

Les modificateurs d'accès ne sont pas applicables aux variables locales. Le seul modificateur qui peut être appliqué aux variables locales est « final ». Un modificateur associé à une variable locale provoque une erreur de compilation.

Exemple : Public int calculer(){
private int x=0; // provoque une erreur de compilation
…….
}

4.2 Modificateurs de non accès

4.2.1 Les variables d'instance

La déclaration des variables d'instance obéit aux règles suivantes:

- peut utiliser l'un de quatre niveaux de contrôle d'accès (peut avoir l'un de trois modificateurs d'accès)
- peuvent être marquées final
- ne peuvent pas être marquées abstract
- ne peuvent pas être marquées static, car elles deviennent des variables de classe.

4.2.2 Méthodes déclarées final

Le mot clé final prévient la méthode d'être modifiée (overridden) dans une sous-classe.

```
class SuperClass{
  public final void showSample() {
    System.out.println("One thing.");
  }}
```

```
class Subclass extends SuperClass{
  public void showSample() { // Try to override the final superclass method
    System.out.println("Another thing.");
  }}
```

=> erreur de compilation dans la classe Subclass au niveau de la méthode showSample qui ne peut pas être modifié (overridden) puisqu'elle est déclarée « final » dans SuperClass

4.2.3 Les arguments déclarés final

Les arguments sont les variables qui sont entre parenthèse après la déclaration de méthode. Ils ne peuvent avoir que le modificateur final.

Exemple : `public Record getRecord(int fileNumber, final int recNumber) {recNumber=8; return new Record() ;}`

recNumber est déclarée final, signifie que sa valeur ne doit pas être modifiée par la méthode et doit garder la même valeur attribuée lors d'appel de la méthode

4.2.4 Les méthodes déclarées abstract

Une classe qui hérite d'une classe abstract doit implémenter toutes les méthodes abstract de la superclasse sauf si la sous-classe est abstract. La règle est :

La première sous-classe concrète d'une classe abstract doit implémenter toutes les méthodes abstract de toutes les superclasses dans l'arbre d'héritage

Exemple 1: Les classes suivantes existent dans le même package. Les deux classes abstract « vehicle » et « car » avec « car » hérite de vehicle et la classe concrète Mini qui hérite de Car :

```
public abstract class Vehicle {
  private String type;
  public abstract void goUpHill();  // Abstract method
  public String getType() {         // Non-abstract method
    return type;
  }
}
public abstract class Car extends Vehicle {
  public abstract void goUpHill();  // Still abstract
  public abstract void doCarThings() ;
}

public class Mini extends Car {
  public void goUpHill() {    // Mini-specific going uphill code }
  public void doCarThings() {  }
}
```

=> La classe Mini implémente les méthodes abstract héritées de Car et de Vehicle

Exemple 2: Est-ce que les deux classes suivantes compilent ?

```
public abstract class A {
  abstract void foo();
}
class B extends A {
  void foo(int I) { }
}
```

=> La classe A compile alors que la classe B ne compile pas. La classe B n'implémente pas la méthode foo de sa superclasse abstract A. La méthode que B déclare est **overloaded** (une méthode utilisant le même identificateur de la méthode héritée mais des arguments différents)

A retenir les combinaisons des modificateurs utilisés dans les méthodes suivantes sont illégales et causent des erreurs de compilation:

abstract static void doStuff();
abstract final void doThings();
abstract private void doOthers();

Le tableau suivant résume les modificateurs qui peuvent être utilisés avec les variables locales, les variables d'instance et les méthodes

Variables locales	Variables non-locales	Méthodes
Final	final, public, protected, private, static	final, public, protected, private, static, abstract

4.2.5 Les méthodes avec liste de variables argument (var-args)

Les méthodes peuvent avoir un nombre variable d'arguments à l'aide de "var-args". Sachant que :

- **arguments :** spécifiés entre parenthèses au moment d'invoquer la méthode:

```
doStuff("a", 2);  // invoking doStuff, so a & 2 are arguments
```

- **paramètres :** déclarés dans la signature de la méthode:
  `void doStuff(String s, int a) { } // there are two parameters: String and int`
- **Type var-args** : il faut spécifier le type du paramètre var-args qui peut être un type primitive ou type d'objet
- **Syntax de base:** pour déclarer un paramètre var-arg, il faut taper : le_type… un espace, et le nom du vecteur
- **Autres paramètres:** il est légal que la méthode déclare d'autres paramètres mais il faut qu'ils soient déclarés avant var-args
- Une méthode ne peut déclarer qu'un seul paramètre var-args.

Ci-dessous des déclarations de var-args:
Légal:

```
void doStuff(int... x) { }  // expects from 0 to many ints as arguments
void doStuff2(char c, int... x)  { }  // expects first a char,then 0 to many ints
void doStuff3(Animal... animal) { }   // 0 to many Animals
```

Illégal:

```
void doStuff4(int x...) { }              // bad syntax
void doStuff5(int... x, char... y) { }  // too many var-args
void doStuff6(String... s, byte b) { }  // var-arg must be last
```

5 Déclarations des constructeurs

A la création d'un nouvel objet, au moins un constructeur est invoqué. Une classe peut contenir zéro ou plusieurs constructeurs. S'il n'y a pas un constructeur crée, le compilateur crée un.
Exemple:

```
class Foo {
  protected Foo() { }           // this is Foo's constructor
  protected void Foo() { }      // this is a badly named, but legal, method
}
```

- un constructeur ne peut jamais avoir un type de retour
- un constructeur peut avoir tous les modificateurs d'accès et peut avoir des paramètres y compris var-args
- un constructeur doit avoir le même nom dans la classe où il est déclaré
- les constructeurs ne peuvent pas être déclarés static, final ou abstract

Exemple: Cocher les constructeurs illégaux déclarés dans la classe suivante:

```
class Foo2 {
 static Foo2(float f) { }    ……..
 final Foo2 (long x) { }     ……..
 Foo2() { }                  ……..
 Foo2(int x, int... y) { }   ……..
 Foo2(short s);              ……..
 private Foo2(byte b) { }    ……..
 Foo2(int x) { }             ……..
 void Foo2() { }             ……..
 Foo() { }                   ……..
```

```
 abstract Foo2 (char c) { }  ……..
 Foo2 (int... x, int t) { } ……..
}
 }
```

6 Variables locales

Les variables locales sont déclarées et initialisées dans une méthode et elles seront détruites à la fin d'exécution de la méthode. Les variables locales sont toujours dans le **stack (pile)**, non dans le **heap (mémoire instantanée)**. Si la variable est une référence d'objet elle sera créée dans le **heap.** L'objet même est créé dans le heap.

Exemple 1: 1) soit la classe suivante

```
Class TestServer {
  public void logIn() {
   int count;
  System.out.println(count);
 }
}
```

→ Les variables locales n'ont pas une valeur par défaut, c'est pourquoi elles doivent être initialisées

2) On ajoute une méthode doSomething de paramètre int i et on assigne la valeur de count à i.

```
public void doSomething(int i)
{     i=count; //……
}
```

→ Les variables locales ne sont pas connues hors de la méthode où elles sont déclarées.

Exemple 2 : On modifie la classe TestServer comme suit:

```
class TestServer {
 int count = 9;
public void logIn() {
 int count = 10;
 System.out.println("count locale: " + count);   }
public void printCount () {
System.out.println("count instance: " + count);
  }
public static void main(String[] args) {
new TestServer().logIn();
new TestServer().count();
 }}
```

Quel est le résultat?

……

…….

→ Une variable d'instance peut avoir le même nom que la variable locale, mais ils désignent de variables distinctes.

Exemple: le code suivant essai d'assigner la valeur d'un paramètre à la variable d'instance

```
class Foo {
   int size = 27;
    public void setSize(int size) {
      size = size; // ??? which size equals which size???
   }}
```

=> Pour que le code soit plus lisible, il faut ajouter « this » qui signifie « l'objet courant » comme suit :

```
class Foo {
 int size = 27;
 public void setSize(int size) {
    this.size = size;  // this.size means the current object's instance variable,
                       //size. The size on the right is the parameter
    }}
```

7 Déclaration d'un vecteur (array)

En Java, les vecteurs sont des objets qui sauvegardent des variables multiples avec le même type ou des sous-classes du même type. Les vecteurs peuvent avoir des primitives ou des références d'objets. Un vecteur est un objet qui est toujours crée dans le **heap** même s'il contient des primitives uniquement.

Exemple: déclaration de vecteur de primitives

```
int[]  key;  // Square brackets before name (recommended)
int key  []; // Square brackets after name (legal but less readable)
```

Exemple: déclaration de vecteur de référence d'objet

```
Thread[] threads;   // Recommended
Thread threads  []; // Legal but less readable
```

Il est possible de déclarer des vecteurs multidimensionnels, qui sont des vecteurs de vecteurs. Voici un exemple :

```
String[][][] occupantName;// vecteur de trois dimensions
String[] ManagerName []; // vecteur de deux dimensions
```

Il ne faut pas préciser la taille du vecteur au moment de déclaration, ceci n'est possible qu'avec l'instanciation de l'objet vecteur. L'exemple suivant est faux :

```
int[5] scores;
```

8 Les variables déclarées final

- Une variable primitive déclarée final, s'il lui est assigné une valeur, il ne sera pas possible de la modifier.

- Une référence de variable marquée final ne peut pas être utilisée pour référencer un autre objet, seules les données de l'objet référé peuvent être modifiées.

Exemple: final Employee E=new Employee("Ali ", " Mr. ", " prof");
E= new Employee(" Med ", " Mr. ", "ing"); // erreur de compilation
E.name="Sameh" ; E.title="Ms"; //pas de problème, possible de modifier les valeurs de variabes

9 Variables et méthodes déclarés static

Le modificateur static est utilisé pour créer des variables et des méthodes indépendants des instances de la classe. Avant le mot static, l'un des modificateurs d'accées public, protected, private ou final peut être ajouté.

Les instances partagent la même valeur d'une variable static.

Exemple 1: Quel est le résultat du programme suivant ?

```
class Variete{
static int x;
double y;
static void  printS(){
x=3;
System.out.println(" static x="+x);
}
void printI(){
y=7;
System.out.println("instance y"+y+"  "+x);
}
public static void main(String [] args){
    printS();
    Variete obj = new Variete();
    obj.printI();
}
```

Résultat : ……

=> La variable x garde sa valeur affectée dans la méthode printS et est utilisée dans la méthode printI

=> Les variables et les méthodes static sont utilisés sans besoin de référence à un objet. Hors de leurs classes il doivent appelée en indiquant le nom de leurs classes exemple: static void m1(){} est déclaré dans la classe MaStatic, dans la classe MaClasse appeler m1() comme suit: MaStatic.m1();

=> Les membres static sont utilisés par des méthodes static et non static

Remarque: Les classes et les constructeurs ne peuvent pas être marqués static

Exemple 2: Quel est le résultat du programme suivant ?

```
class Variete1{
static int a;
int b;
Static{a=-9 ;
System.out.println("a="+a) ;
}
{b=15 ;
System.out.println("b="+b) ;
}
Variete1(){
System.out.println("une instance créée") ;
}
public static void main(String [] args){
Variete1 x= new Variete1();
Variete1 y= new Variete1();
}
```

……

……

……

=> les blocs static s'exécutent avant la méthode main

=> les blocs non static s'exécutent à chaque création d'une instance et avant l'exécution du constructeur

Ce qui peut être déclaré static :

- Les méthodes
- Les variables de classe
- Les blocs d'initialisation

10 Encapsulation

- Une classe permet d'envelopper les objets : Un objet est vu comme une entité opaque.
- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet
  - Accéder aux attributs d'un objet qu'à l'aide des méthodes: Utilisation des getters (accesseurs: méthode get()) et setters (modificateur) (méthode get() et set())
  - Les attributs doivent être déclarés privés

10.1 Méthodes d'accès aux valeurs des variables depuis l'extérieur

Comment peut-on accéder à la valeur d'une variable protégée ??

Exemple: Soit la classe Etudiant suivante:

```
class Etudiant {
private String nom, prenom;
public Etudiant(String st1, String st2){
nom=st1; prenom=st2;
}
}
public class UtiliserEtudiant{
public static void main(String[] argv) {
Etudiant e= new Etudiant("Mohammed","Ali");
System.out.println("Nom = "+e.nom);
}
```

Est-ce que la classe UtiliserEtudiant compile?

……..

Comment faire pour afficher le nom de l'étudiant e ?

On définit ***Un accesseur*** (méthode getNom) dans la classe Etudiant qui permet de retourner la valeur de Nom.

```
class Etudiant {
private String nom, prenom;
public Etudiant(String st1, String st2){
nom=st1; prenom=st2;
}
……..
……..
……..
……..
……..
……..
}
```

On modifie la classe UtiliserEtudiant comme suit:

```
public class UtiliserEtudiant{
public static void main(String[] argv) {
Etudiant e= new Etudiant("Mohammed","Ali");
System.out.println("Nom = "+e.getNom);
}
```

Pour modifier la valeur de l'attribut nom (champs private), on définit ***Un modificateur*** qui permet d'attribuer une valeur à nom.

```
class Etudiant {
private String nom, prenom;
public Etudiant(String st1, String st2){
nom=st1; prenom=st2;
}
public String getNom(){
Return nom;
}
public String getPrenom(){
Return prenom;
}
……..
……..
……..
……..
……..
……..
}
```