- TABLEAUX

Un tableau est un ensemble indexé de données d'un même type.
L'utilisation d'un tableau se décompose en trois parties :

- Création du tableau ;
- Remplissage du tableau ;
- Lecture du tableau.

# TABLEAUX

- Création d'un tableau

Un tableau se déclare et s'instancie comme une classe :

```
int monTableau[ ] = new int[10]; ou  int [ ] monTableau = new int[10];
```

L'opérateur [ ] permet d'indiquer qu'on est en train de déclarer un tableau.

Dans l'instruction précédente, nous déclarons un tableau d'entiers (int, integer) de taille 10, c'est-à-dire que nous pourrons stocker 10 entiers dans ce tableau.

Si [ ] suit le type, toutes les variables déclarées seront des tableaux, alors que si [ ] suit le nom de la variable, seule celle-ci est un tableau :

```
int [] premierTableau, deuxiemeTableau;
float troisiemeTableau[], variable;
```

# TABLEAUX

- ## Remplissage d'un tableau

Une fois le tableau déclaré et instancié, nous pouvons le remplir :

```
int [] monTableau = new int[10];

monTableau[5] = 23;
```

L'indexation démarre à partir de 0, ce qui veut dire que, pour un tableau de N éléments, la numérotation va de 0 à N-1.

Dans l'exemple ci-dessus, la 6ème case contient donc la valeur 23.

Nous pouvons également créer un tableau en énumérant son contenu :

```
int [] monTableau = {5,8,6,0,7};
```

Ce tableau contient 5 éléments.

Lorsque la variable est déjà déclarée, nous pouvons lui assigner d'autres valeurs en utilisant l'opérateur `new` :

```
monTableau = new int[]{11,13,17,19,23,29};
```

# TABLEAUX

## Lecture d'un tableau

Pour lire ou écrire les valeurs d'un tableau, il faut ajouter l'indice entre crochets (`[` et `]`) à la suite du nom du tableau :

```
int [] monTableau = {2,3,5,7,11,23,17};
int nb;

monTableau[5] = 23;  // ->  2 3 5 7 11 23 17
nb = monTableau[4];  // 11
```

L'indice 0 désigne le premier élément du tableau.

L'attribut `length` d'un tableau donne sa longueur (le nombre d'éléments). Donc pour un tableau nommé `monTableau` l'indice du dernier élément est `monTableau.length-1`.

Ceci est particulièrement utile lorsque nous voulons parcourir les éléments d'un tableau.

```
for (int i = 0; i < monTableau.length; i++)
{    int élément = monTableau[i];
    // traitement
}
```

# TABLEAUX

- Parcours des tableaux

```
int[] monTableau = {150, 200, 250};
for (int élément : monTableau)
{ // traitement
}
```

Attention néanmoins, la variable `element` contient une copie de `monTableau[i]`. Avec des tableaux contenant des variables primitives, toute modification de `élément` n'aura aucun effet sur le contenu du tableau.

```
// Vaine tentative de remplir tous les éléments du tableau avec la valeur 10
for(int élément : monTableau)
{ élément = 10;
}
// La bonne méthode :
for(int i=0 ; i<monTableau.length ; i++)
{  monTableau[i] = 10;}
```

```
// Ou plus court :
Arrays.fill(monTableau, 10);
```

Pour éviter de modifier la variable, utilisez le mot-clé `final` :

```
for (final int élément : monTableau)
{
```

# TABLEAUX

- Tableaux à plusieurs dimensions

```
int[][] matrice = new int[5][];
for (int i=0 ; i<matrice.length ; i++)
    matrice[i] = new int[6];
```

Java permet de résumer l'opération précédente en :

```
int[][] matrice=new int[5][6];
```

La première version montre qu'il est possible de créer un tableau de tableaux n'ayant pas forcément tous la même dimension.

On peut également remplir le tableau à la déclaration et laisser le compilateur déterminer les dimensions des tableaux, en imbriquant les accolades :

```
int[][] matrice =
    {
        { 0, 1, 4, 3 } , // tableau [0] de int
        { 5, 7, 9, 11, 13, 15, 17 } // tableau [1] de int
    };
```

# TABLEAUX

- Tableaux à plusieurs dimensions

Pour déterminer la longueur des tableaux, on utilise également l'attribut `length` :

```
matrice.length      //  2
matrice[0].length   //  4
matrice[1].length   //  7
```

De la même manière que précédement, on peut facilement parcourir tous les éléments d'un tableau :

```
for (int i=0 ; i<matrice.length ; i++)
{
      for (int j=0 ; j<matrice[i].length ; j++)
      {
            //Action sur matrice[i][j]
      }
  }
```

# TABLEAUX

- Tableaux à plusieurs dimensions

Pour déterminer la longueur des tableaux, on utilise également l'attribut `length` :

```
matrice.length      //  2
matrice[0].length   //  4
matrice[1].length   //  7
```

De la même manière que précédement, on peut facilement parcourir tous les éléments d'un tableau :

```
for (int i=0 ; i<matrice.length ; i++)
{
      for (int j=0 ; j<matrice[i].length ; j++)
      {
            //Action sur matrice[i][j]
      }
  }
```

# Analyse d'un problème avec l'approche OO

Exemple: Une société de location de voiture

Chaque voiture possède les caractéristiques suivantes:

- un identifiant : matricule
- un couleur: noir, gris, rouge
- marque : Renault, Peugeot
- un prix de location / jour
- une description: type du carburant, nombre de portes, etc
- nombre de jours de location

C'est possible d'ajouter ou de diminuer le nombre de jours de location et de calculer le prix total de location.

# Identification de l'objet

Objet

Attributs

Méthodes

| Voiture |
| --- |
| ID<br>Prix<br>Couleur<br>Description<br>Nombre jours location<br>  |
| Augmenter nombre jours location<br>Diminuer nombre jours location<br>Calculer Prix Total Location |

# Classe et objet JAVA

Une classe n'est pas un objet. Une classe est un patron d'objet.

**Classe** « Voiture »

**Objet 1**
Id: 125Tunisie160
Prix: 100 DT
Couleur: Gris
NbreJL: 10

**Objet 2**
Id: 653Tunisie123
Prix: 70 DT
Couleur: Rouge
NbreJL: 5

# Classe et objet JAVA

• classe : structure d'un objet, la déclaration de l'ensemble des entités qui composeront un objet.

•Un objet est une **instanciation** d'une classe **objet = instance**

•Une classe est composée de deux parties :

- Les attributs (appelés aussi *données membres*) : il s'agit des données représentant l'état de l'objet
- Les méthodes (appelées aussi *fonctions membres*): il s'agit des opérations applicables aux objets

# Déclaration d'une classe

```
public class Voiture
{
        /*Déclaration des attributs*/
        /*Déclaration des méthodes*/
        //commentaire sur une seule ligne
        /*commentaires sur
        plusieurs lignes*/
}
```

- Le nom de la classe doit commencer par une majiscule
- Exemple: **C**ompte**B**ancaire, **A**gence**V**oyage

# Déclaration des attributs

**Syntaxe:**

type nom_variable [=value];

**int id = 0;**

- Le nom de l'attribut doit commencer par une lettre miniscule
- Exemple: age, quantiteStock

# Types de données en JAVA

- Deux grands groupes de types de données :
  - types primitifs
  - types objets (instances de classe)
- Les types de données utilisés sont :
  - les nombres entiers
  - les nombres réels
  - les caractères et les chaînes de caractères
  - les booléens
  - les objets

# Déclaration des méthodes

**Syntaxe:**

```
Type_retour  nom_methode([arguments])
   {

   }
```

```
void afficherInfoVoiture (){

   }
```

- Le nom de la méthode doit commencer par un verbe

# Classe voiture

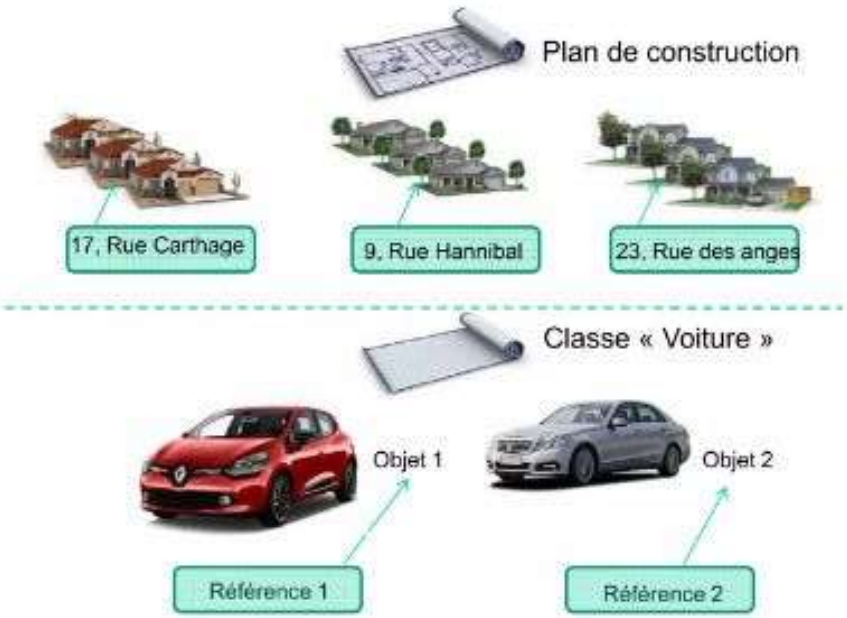
```
public class Voiture{
    int id;
    char couleur;
    float prix;
    String description;
    int nbreJourLocation;
    void augmenterNbJourLocation (int nombre) {
       nbreJourLocation += nombre;
     }

     void diminuerNbJourLocation (int nombre) {
       nbreJourLocation - = nombre;
      }

    void  afficherInfoVoiture() {
      System.out.println(id+ " " +couleur+ " " +prix+ " " +description);
     }
}
```

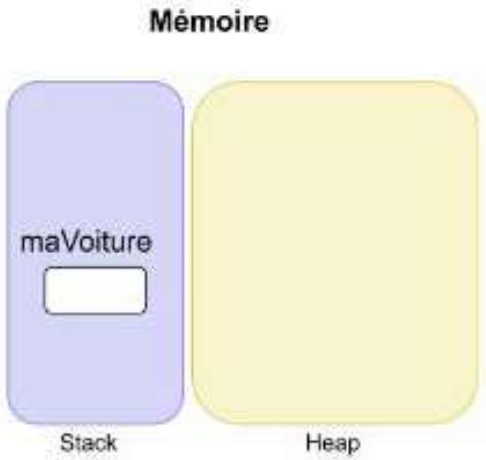
# POO

## Création des objets



# Notion de référence

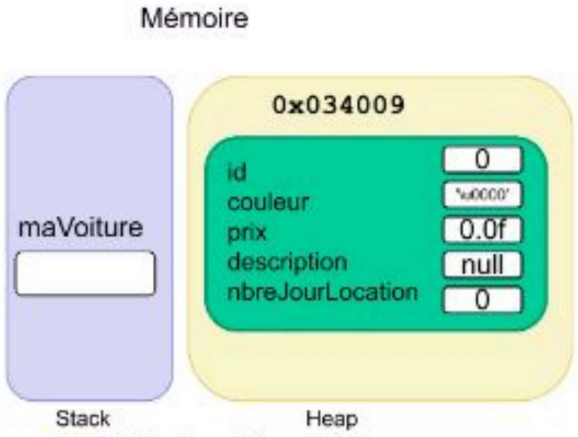
```
public class Test{
    public static void main(String[] args){
        Voiture maVoiture;
        maVoiture=new Voiture();
        maVoiture.couleur='R';
    }
}
```




- Création d'une variable *maVoiture* de type Voiture

# Notion de référence

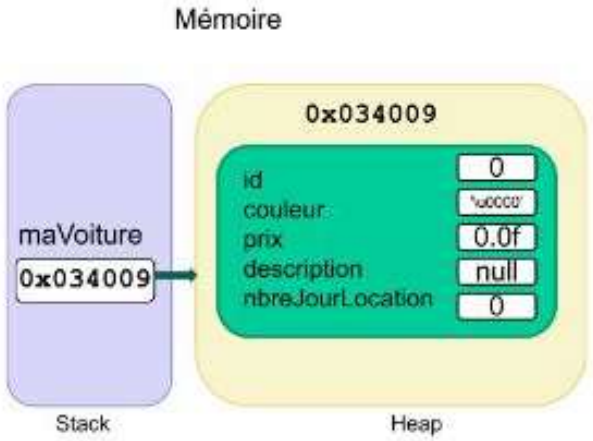
```
public class Test{
    public static void main(String[] args){
        Voiture maVoiture;
        maVoiture =new Voiture();
        maVoiture.couleur='R';
    }
}
```




- Instanciation de la classe Voiture ➡ Création d'un objet.
- Cet objet possède une adresse de son emplacement dans la mémoire (**0x034009**)

# Notion de référence

```
public class Test{
    public static void main(String[] args){
        Voiture maVoiture;
        maVoiture=new Voiture();
        maVoiture.couleur='R';
    }
}
```




- Lier l'objet créé et la variable *maVoiture*

➡ *maVoiture* est la référence de l'objet créé

# Les constructeurs

Pour créer un ***objet*** à partir d'une classe, on utilise l'opérateur ***new***.

```
public class Test{
    public static void main(String[] args){
        Voiture maVoiture;
        maVoiture=new Voiture();
    }
}
```

L'opérateur ***new*** fait appel au ***constructeur*** de la classe

```
class Voiture{
    Voiture() {}
}
```

- un constructeur porte le même nom que la classe dans laquelle il est défini
- un constructeur n'a pas de type de retour (même pas *void*)

# Les constructeurs

- **Constructeur par défaut**

```
Voiture(){}
```

Le constructeur par défaut initialise les attributs de la classe aux valeurs par défaut.

```
Voiture(){
    id=0;
    couleur='N';
    prix=75.5f;
}
```

- **Constructeur surchargé**

```
Voiture(int id, char couleur, float prix){
    this.id=id;
    this.couleur=couleur;
    this.prix=prix;
}
```

# Les constructeurs

- Absence de constructeur dans la classe ➡ le compilateur crée automatiquement un constructeur par défaut implicite

- Si le constructeur surchargé est créé, le constructeur par défaut implicite ne sera plus créé par le compilateur

- La plateforme java différencie entre les différents constructeurs déclarés au sein d'une même classe en se basant sur le nombre des paramètres et leurs types.

On ne peut pas créer deux constructeurs ayant le même nombre et types des paramètres.

```
Voiture (int id) {
      this.id=id
}
Voiture (int id) {
      this.id=id*2
}
```

**Erreur de compilation**

# Les constructeurs

Quel constructeur va choisir Java lors de création de l'objet ?

```
class Voiture{
      int id;
      char couleur;
      float prix;
      String description;
      int nbreJourLocation;

      Voiture() {}

      Voiture(int id) {
            this.id=id;
      }

      Voiture(int id, char couleur) {
            this(id);
            this.couleur=couleur;
      }
}
```

**Utilisation:**

```
Voiture v1=new Voiture();
Voiture v1=new Voiture(122);
Voiture v1=new Voiture(122, 'N');
```

# POO: this and super

**super()** est utilisé pour appeler le constructeur de la classe de base (classe parent).

```
class Animal { // Superclass (parent)
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Dog extends Animal { // Subclass (child)
  public void animalSound() {
    super.animalSound(); // Call the superclass method
    System.out.println("The dog says: bow wow");
  }
}

public class Main {
  public static void main(String args[]) {
    Animal myDog = new Dog(); // Create a Dog object
    myDog.animalSound(); // Call the method on the Dog object
  }

  }
```

# POO: this and super

**this()** est utilisé pour appeler le constructeur de la classe actuelle .

```
public class Main {
  int x;
  // Constructor with a parameter
  public Main(int x) {
    this.x = x;
  }
  // Call the constructor
  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println("Value of x = " + myObj.x);
  }
}
```

# POO: static

**static** signifie que quelque chose est directement lié à une classe :

- si un champ est static, il appartient à la classe → **Attributs de classe**
- si une méthode est static (méthode statique), elle appartient à la classe → méthode de classe.

Par conséquent, tu peux utiliser le nom de la classe pour appeler une méthode **static** ou **référencer un champ static**.

Par exemple,

si le champ count est static dans la classe `Counter`, tu peux référencer la variable avec l'expression suivante : `Counter.count`.

# POO: static

- Tu ne peux PAS accéder aux membres non static d'une classe dans un contexte static, comme une méthode ou un bloc static. La compilation du code ci-dessous générera une erreur :

```
public class Counter {
private int count;
public static void main(String args []) {
   System.out.println(count); //  Compile time error
}
}
```

- Les méthodes static ont un avantage pratique en cela qu'il n'y a pas besoin de créer un nouvel objet chaque fois que tu veux les appeler.

# POO: static

- Un autre point important est que vous ne pouvez pas remplacer (@Override) les méthodes statiques. Si vous déclarez une telle méthode dans une subclass, c'est-à-dire une méthode avec le même nom et la même signature, vous "cachez" simplement la méthode de la superclass au lieu de la remplacer. Ce phénomène est connu sous le nom method hiding. Cela signifie que si une méthode static est déclarée à la fois dans la classe parent et la classe enfant, la méthode appelée sera toujours celle du type de la variable au moment de la compilation. Contrairement à ce qui arrive avec le remplacement de méthode, ces méthodes ne seront pas exécutées lors de l'exécution du programme.

Prenons un exemple :

```
class Vehicle {
     public static void kmToMiles(int km) {
          System.out.println("Inside the parent class");
}}
class Car extends Vehicle {
     public static void kmToMiles(int km) {
          System.out.println("Inside the child class");
 }}
```

```
public class Demo {
   public static void main(String args []) {
      Vehicle v = new Car();
       v.kmToMiles(10);
  }}
```

# POO: final

Un attribut peut être déclaré comme **final**. Cela signifie qu'il n'est plus possible d'affecter une valeur à cet attribut une fois qu'il a été initialisé. Dans cas, le compilateur exige que l'attribut soit initialisé *explicitement*.

```
public class Voiture {

  public String marque;
  public float vitesse;
  public final int nombreDeRoues = 4;

  }
```

**final** porte sur l'attribut et empêche sa modification. Par contre si l'attribut est du type d'un objet, il est possible de modifier l'état de cet objet.

```
public class Facture {

  public final Voiture voiture = new
Voiture();

  }
```

```
Facture facture = new Facture();
facture.voiture.marque = "DeLorean"; // OK

  facture.voiture = new Voiture() // ERREUR
  DE COMPILATION
```

# POO:Attributs de classe finaux

Pour déclarer un attribut de classe, on utilise le mot-clé **static**.

```
public class Voiture {

  public static final int nombreDeRoues = 4;
  public String marque;
  public float vitesse;

  }
```