Module de mise à niveau en Java
Cours 1

Laure KAHLEM
laure.kahlem@univ-orleans.fr

Références

- Mickaël Kerboeuf. Fondements de la programmation orientée objet avec Java8. Ellipses, 2016.
- Kathy Sierra et Bert Bates. Java Tête la première. O'Reilly, 2007.
- David Flanagan. Java in a Nutshell. Manuel de référence. O'Reilly, 2002.
- Cyrille Herby. Apprenez à programmer en JAVA. Broché, 2012.
- José Paumard. « Java en ligne » [en ligne]. http://blog.paumard.org/cours/java/ (Consulté le 15/07/2015).

Références

- Tutos d'Oracle : https://docs.oracle.com/javase/tutorial/
- Thinking in Java (http://mindview.net/Books/TIJ/) / Penser en Java (http://penserenjava.free.fr/)
- Livre : Core Java, Prentice-Hall (vol. 1) / Au coeur de Java, Campus Press
- Le doudoux (http://www.jmdoudoux.fr/, 3413 pages)
- Internet :
  - developpez.com
  - dzone.com [refcardz]
  - slides de Richard Grin, Miage Nice
  - les cast codeurs
  - java code geek, java ranch, javaworld, InfoQ, … Stack Overflow !
  - Javarevisited (http://javarevisited.blogspot.fr)
  - twitter / youtube (devoxx FR)

Sommaire

- Pourquoi Java?
- Comment fonctionne Java?
- Structures fondamentales de Java
- Classes et objets
- Classes particulières



Pourquoi Java?

- **Langage de programmation orienté objet**
- **Avantages :**
  - Portabilité (JVM : *"Write Once, Run Anywhere"*).
  - Écosystème énorme (Spring, Hibernate, Android).
  - Performances
  - Sécurité
- **Inconvénients :**
  - Verbosité
  - Lenteur de démarrage (JVM)

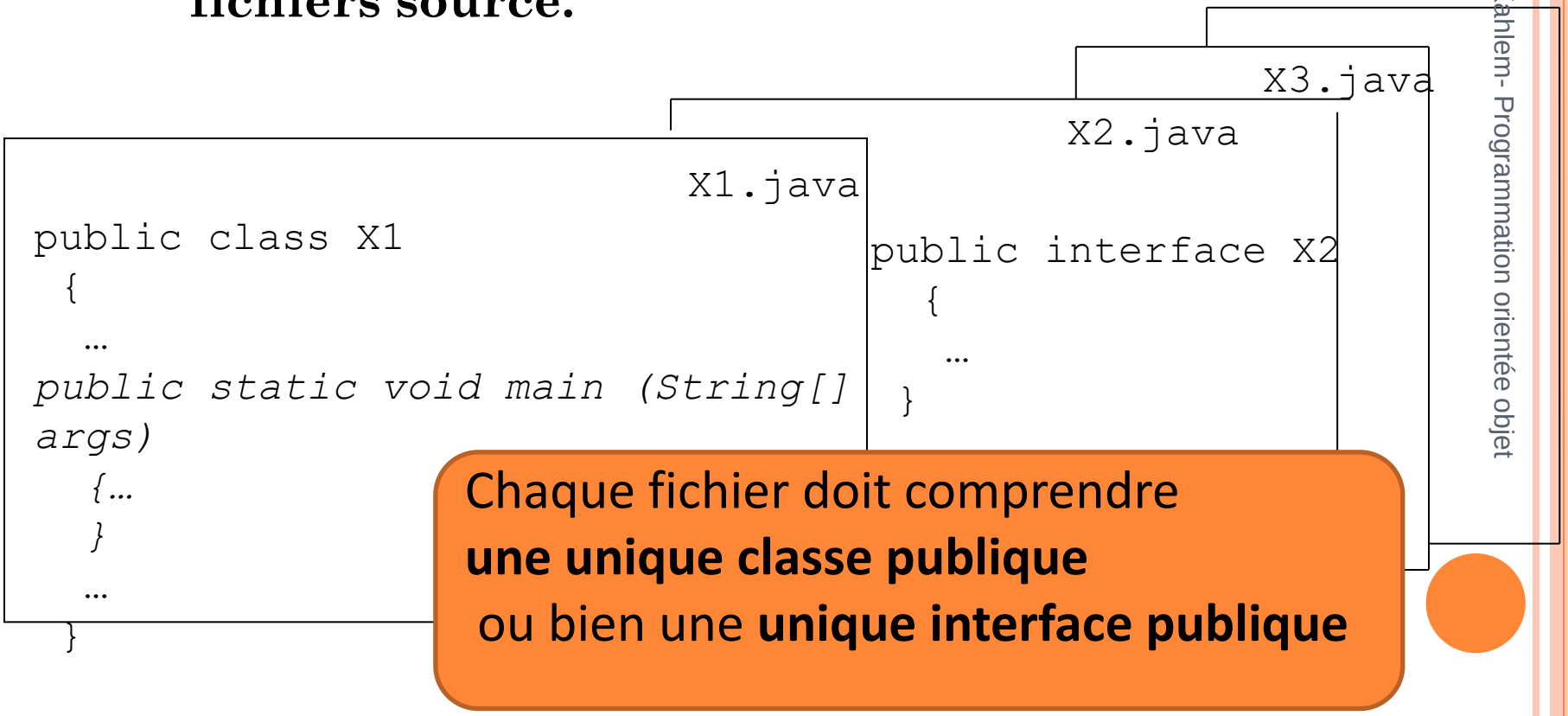


POURQUOI JAVA?

- **Java dispose d'une API (interface de programmation applicative) très riche (multimédia, interfaces graphiques, connectivité réseau, base de données…)**
  - Nombreuses classes structurées **en packages**
  - Consulter les versions postérieures à Java 8.
  - http://docs.oracle.com/javase/8/docs/api
- **Java est sûr**
  - Langage fortement typé
  - De nombreuses vérifications lors de l'exécution garantissent son bon fonctionnement

COMMENT FONCTIONNE JAVA?

- **Construction d'un programme JAVA :**
  - Un programme Java est composé **d'un ensemble de fichiers source.**

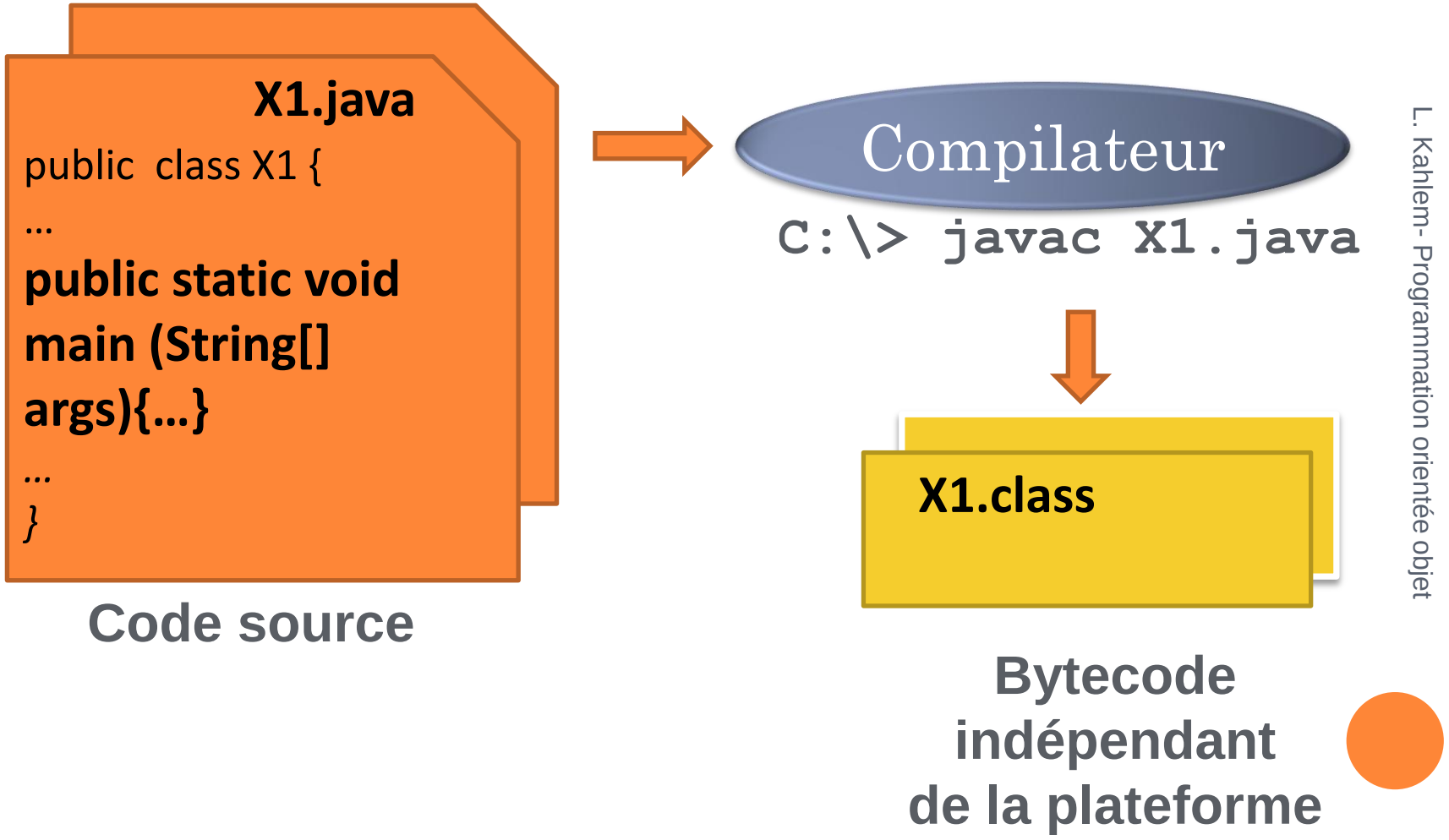


Comment fonctionne Java?

- **Compilation et exécution d'un programme JAVA**
  - Le code source doit être traduit en langage machine.
  - Deux étapes en Java:
    1. Le compilateur traduit le **code source** dans un langage intermédiaire , le **bytecode** .
       *Compilateur fourni par le JDK : **javac***
    2. Le bytecode est traduit dans le langage natif du processeur de l'ordinateur par la **JVM**.
       *Machine virtuelle fourni par le JDK : **java***

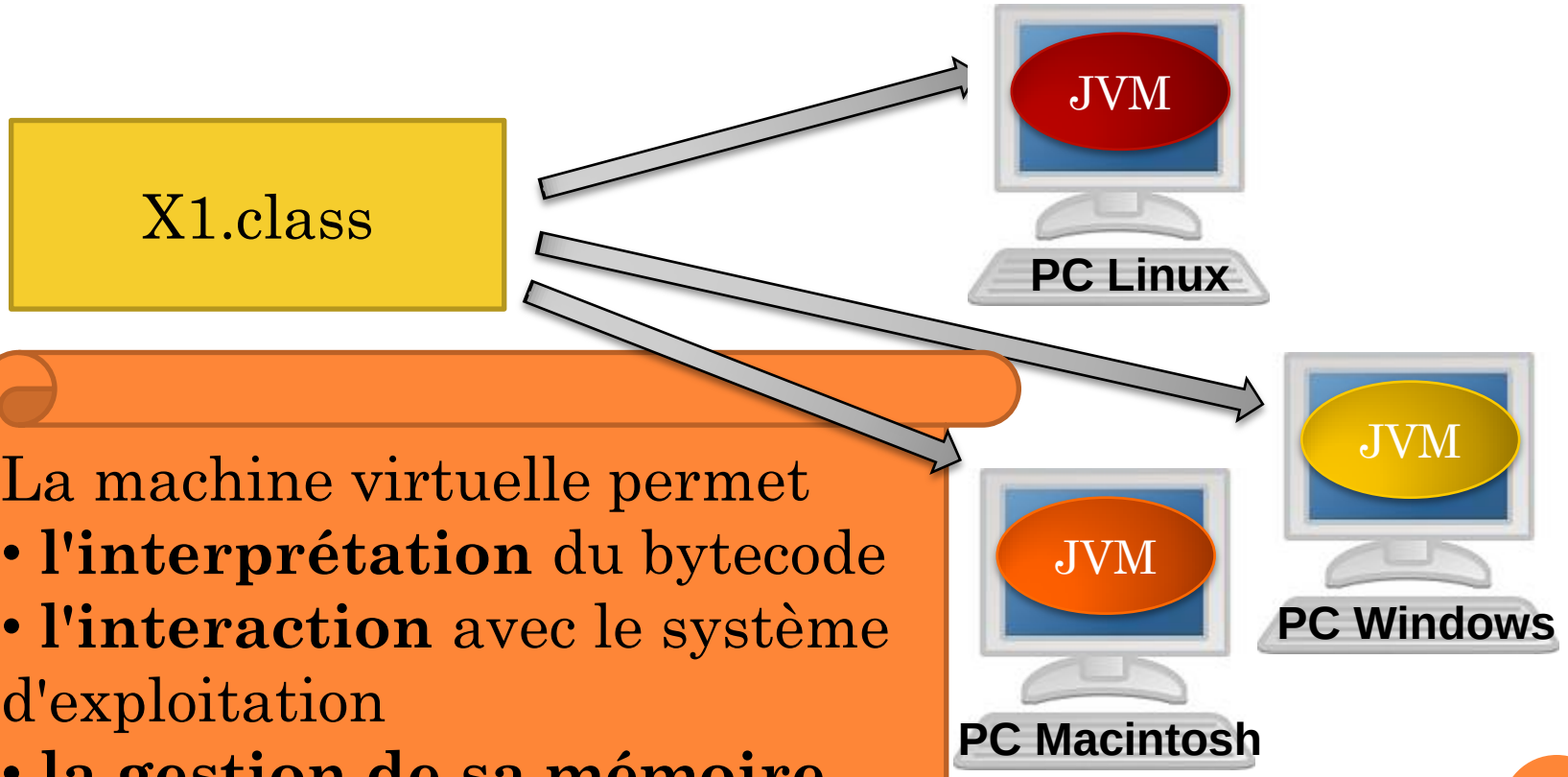
Comment fonctionne Java?

- **Compilation et exécution d'un programme JAVA**



Comment fonctionne Java?

- **Compilation et exécution d'un programme JAVA**



La machine virtuelle permet
- **l'interprétation** du bytecode
- **l'interaction** avec le système d'exploitation
- **la gestion de sa mémoire** grâce au ramasse miettes

```
Exécution :
C:\> java X1
```

Environnement de travail

- Plusieurs IDE :
  - **Eclipse IDE** for Java Developpers http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen3a
  - **IntelliJ** https://www.jetbrains.com/idea/
    - version gratuite [Community] et payante [Ultimate], licences accessibles dans les salles de TP de la fac
    - pour chez vous : https://www.jetbrains.com/shop/eform/students inscription avec votre mail de l'Université
- En TP, nous utiliserons **IntelliJ.**



Les structures fondamentales de Java

- **Les commentaires**
  - **Sur une ligne :** `//`
    - `int nb = 0; // nb représente le nombre d'étudiants`
  - **Sur plusieurs lignes :** `/* … */`
- **Commentaires JavaDoc**
  - Commencent par : `/**`
  - Se terminent par : `*/`
  - Les tags spécifiques sont introduits par :@
    - @author, @param, @return, @see, @throws
  - http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html (consulté le 04/09/2021)

Les structures fondamentales de Java

- **Les annotations**
  - **Instructions destinées au compilateur**
  - **Mots-clés précédé de @**
  - **Exemples**
    - **Déclaration et redéfinition de méthode**

      `@Override`
    - **Déclaration d'une interface fonctionnelle**

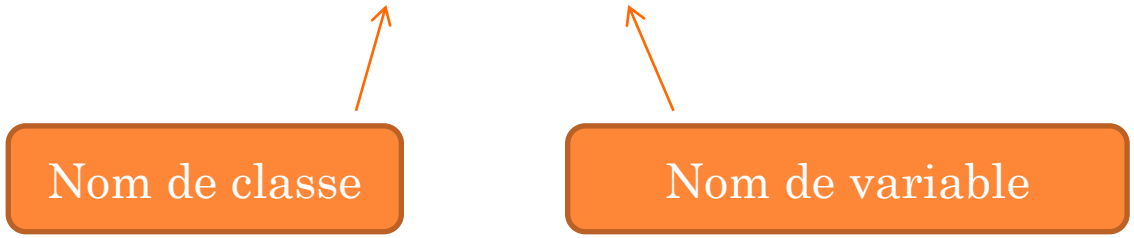
      `@FunctionalInterface`
    - **Suppression des avertissements de compilation**

      `@SuppressWarning`
    - **Annotation des méthodes dans Junit 4**

      `@Test, @Before, @After, @BeforeClass, @AfterClass`

Les structures fondamentales de Java

- **Choix d'un identifiant**
  - **Conventions de nommage**
    - **`Abc`** : nom de classe, d'interface ou de type énuméré
    - **`ABC`** : nom de constante
    - **`abcDef`**: nom de variable, d'attribut, de paramètre, de méthode, de package.
    - Exemple: `String nomEtudiant;`



Les structures fondamentales de Java

- **Les types de données**
  - **Les types primitifs**
  - **Les classes dont les objets sont les instances**



Les structures fondamentales de Java

- **Les types primitifs**
  - Les types primitifs se manipulent en Java uniquement **par valeur.**
  - Les valeurs que représentent le type primitif peuvent être :
    - stockées dans **des variables.**
    - ou nommées directement par des littéraux spécifiques du langage.

  Exemple :
  - 56 est un littéral du type primitif `int` qui représente la *valeur* 56.

  Cette valeur peut être stockée dans une variable : `int x;`

  `x = 56;`

L'affectation est une expression qui met à jour la variable et retourne la valeur affectée.

Les structures fondamentales de Java

- **Les types primitifs**
  - **Type booléen :** `boolean`
    - Deux littéraux : **`true`** et **`false`**.
  - **`Type caractère: char`** (Java utilise l'encodage **unicode** sur 2 octets)
    - `Littéraux délimités par '.`
    - `Caractères spéciaux introduits par \.`

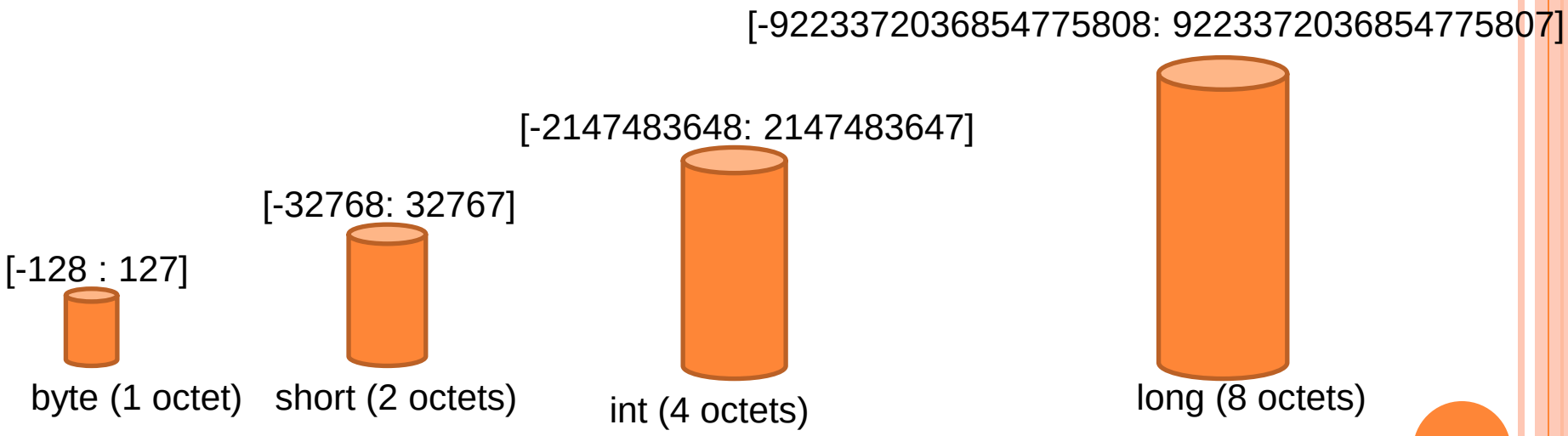
Exemples :

Caractère **u** : 'u'
Caractère **4** : '4'
Caractère \ : '\\'
Caractère ' : '\' '
Caractère @ de code unicode 0040: '\u0040'

Les structures fondamentales de Java

- **Types primitifs**
  - **Types numériques entiers**

    byte, short, int, long



Les structures fondamentales de Java

- **Types primitifs**
  - **Types numériques entiers**
    - **int est le type par défaut** pour 90% des cas.
    - **byte/short sont utiles pour l'optimisation mémoire** (tableaux, fichiers binaires).
    - **Exemple :**
      - `byte b = -5;`
      - `int j = 2000000000;`
      - `long = 12349L;` ← Suffixe L obligatoire
      - `int k = -0b11011011;`

Écriture binaire d'un littéral représentant la valeur -219

Les structures fondamentales de Java

- **Types primitifs**
  - **Types numériques flottants**

Types	**float**	**double**
Plus petite valeur absolue	$1,4.10^{-45}$	$4,9.10^{-324}$
Plus grande valeur absolue	$3,4028235.10^{38}$	$1,79769313486231 57.10^{308}$
Littéral correspondant la valeur : $1,456.10^{2}$ (=145,6)	1.456E2**f**	1.456E2 ou1.456E2d
Littéral correspondant la valeur : $-34,1.10^{-3}$ (=-0,0341)	-34.1E3**f**	-34.1E3 ou-34.1E3d

Les structures fondamentales de Java

- **Types primitifs**
  - **Types numériques flottants**
    - Il ne faut pas les utiliser pour des calculs financiers . Dans ce cas, utiliser la classe `BigDecimal`.
    - **Éviter float** sauf pour des raisons de mémoire (ex : tableaux de millions de valeurs).
    - Les types `float` et `double` incluent des valeurs spéciales `NaN` (Not a Number) et (+/-)`Infinity`.
      Exemple : `2.0/0=>Infinity`

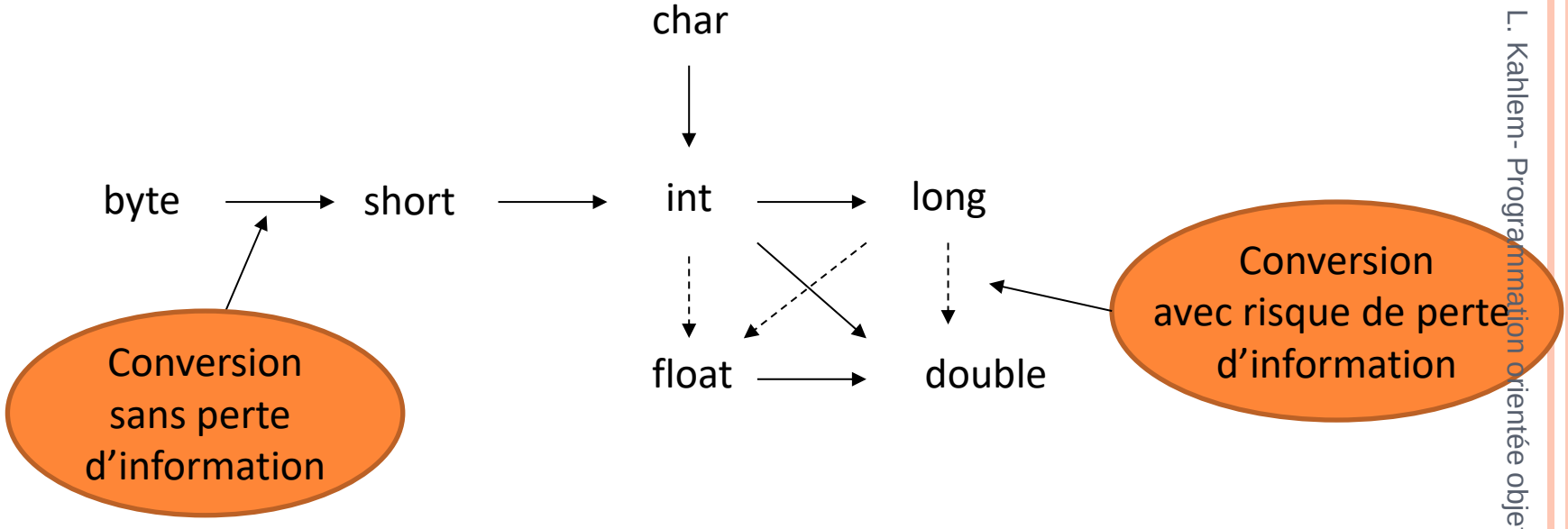
Les structures fondamentales de Java

- **Types primitifs**
- *Quel type utiliseriez-vous pour stocker :*
  - *L'âge d'une personne ?*
  - *Le nombre d'habitants d'un pays ?*
  - *La température en degrés Celsius ?*
  - *Un pixel RGB ?*



Les structures fondamentales de Java

- **Types primitifs**
  - **Conversions légales de types numériques**



Exemple :

```
int n= 123456789;
float f = n; //f vaut 1.23456792E8
int i = 'L'; //i vaut 76 (encodage unicode)
```

Les structures fondamentales de Java

- **Types primitifs**
  - **Transtypage (`cast`):** Permet de convertir un type plus grand en un type plus petit. Cette conversion pouvant entraîner des pertes d'information.

    Exemples :

```
double d = 5.6545;
int i = (int)d; // i = 5
```




Les structures fondamentales de Java

- **Types primitifs**
  - **Opérateurs usuels sur les types numériques:**
    - `= + - * / % ++ -- += -= *= /=`
    - `== != < > <= >=`
    - `& || !`
  - **`x++`** : la valeur actuelle de x est utilisée dans l'expression puis ensuite x est incrémentée.
  - **`++x`** : la valeur actuelle de x est incrémentée puis ensuite elle est utilisée dans l'expression.



Les structures fondamentales de Java

- **Types primitifs**
  - **Promotion numérique**
    - Tout calcul entre entiers (byte, short, int, long) donne un résultat de type `int` ou `long` (cas où au moins un des opérandes est de type `long`).
    - Tout calcul entre une opérande flottante et une entière donne un résultat de type flottant.
    - Tout calcul entre deux opérandes flottantes donne un résultat du type flottant le plus large.



Les structures fondamentales de Java

- **Exercice :**

On considère les déclarations suivantes :

```
int nbre1 = 3, nbre2 = 2;
```

Donner la valeur de la variable `resultat`:

```
double resultat= nbre1/nbre2;
```




Exemple : Gestion de points 2D



Les structures fondamentales de Java

- **Les classes et leurs instances**
  - **Une classe** définit un **type**.
  - Les valeurs associées sont des **objets**, instances de la classe.

Définition du corps de la classe :

```
class Point {
    double x;
    double y;
    …
}
```

Allocation dynamique par l'opérateur new

Instanciation de la classe :

new `Point()`

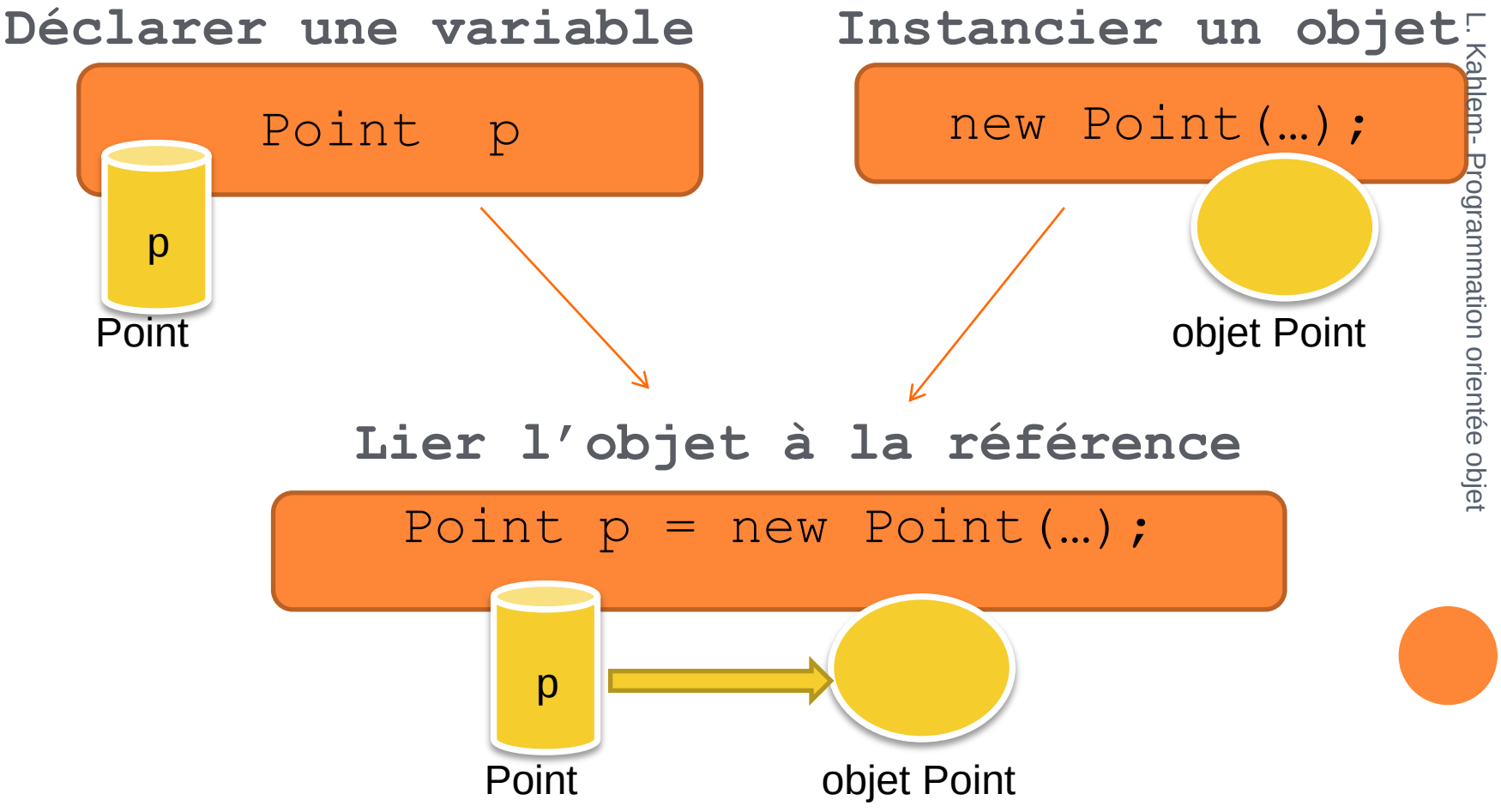
Les structures fondamentales de Java

- **Les classes et leurs instances**
  - Les objets sont manipulables **par référence** et non par valeur
  - **Les variables** de type objet contiennent **l'adresse mémoire** qui permet d'accéder à l'objet.
  - Valeur particulière qui ne désigne aucun objet: **`null`**.



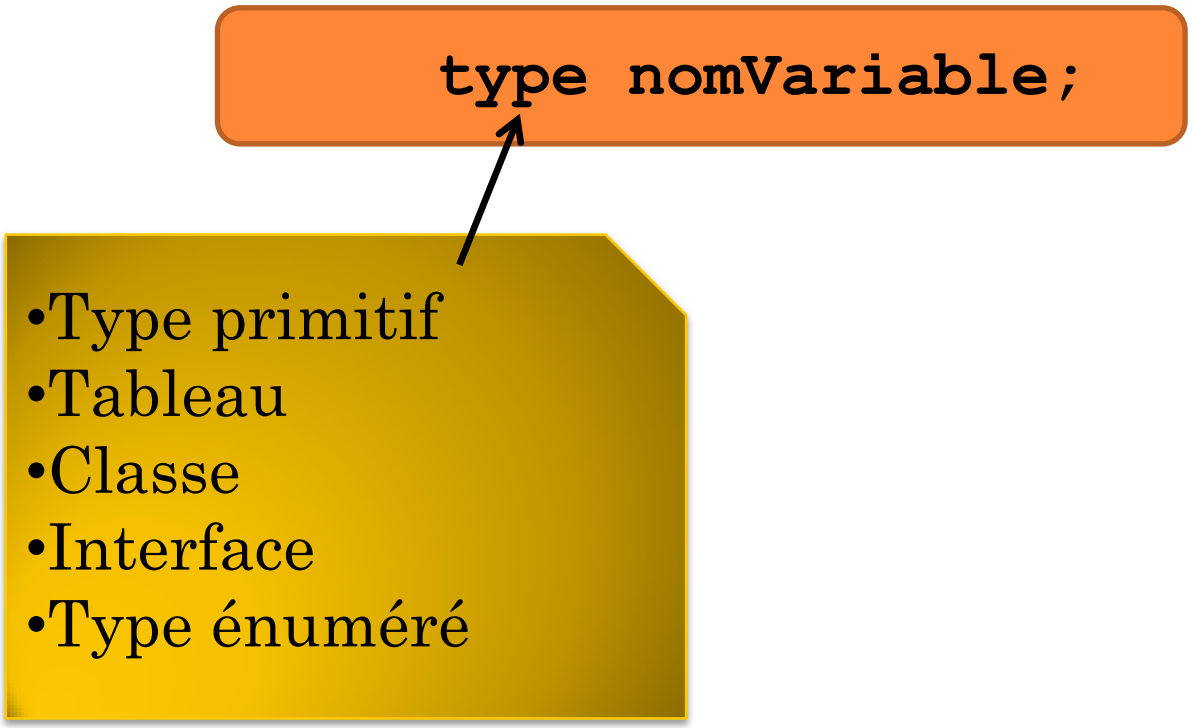
Les structures fondamentales de Java

- **Les classes et leurs instances**



Les structures fondamentales de Java

- **Les variables**
  - Déclaration de variable



Les structures fondamentales de Java

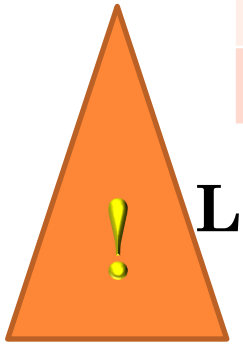
- **Les variables**
  - Une variable a une portée limitée au bloc de code dans lequel elle est déclarée (délimité par `{ . . . }`).
  - Une variable peut être déclarée au niveau d'une classe : **attribut** de la classe (portée peut être étendue à l'extérieur de la classe).
  - Dans tous les autres cas, une variable est **locale**.
  - **Valeur affectée à une variable:**
    - Littéral
    - Contenu d'une autre variable
    - Valeur d'une expression
    - Résultat de l'appel de fonction.

Les structures fondamentales de Java

- **Les variables**

  Les attributs sont initialisés par défaut

Type	Valeur par défaut
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`
`objet`	`null`



Les variables locales doivent être obligatoirement initialisées.

Les structures fondamentales de Java

- **La pile et le tas : deux espaces mémoire distincts**
  - L'espace mémoire alloué à une **variable locale lors de l'exécution d'une fonction ou d'un bloc de code** est situé **sur la pile.**
  - **La création d'un objet** avec l'opérateur **`new`** entraîne l'allocation dynamique d'un espace **sur le tas**.
  - Si une variable contient une référence à un objet, la valeur de cette référence est stockée dans la pile.
  - Si un objet n'est plus référencé, c'est le **ramasse-miettes** (garbage collector) qui se charge de libérer l'espace mémoire alloué.

Les structures fondamentales de Java

- **Instructions conditionnelles**
  - `if (condition) instruction`
  - `if (condition)`
    `  instruction1`
    `else`
    `  instruction2`

Les structures fondamentales de JAVA

- **Branchements conditionnels à choix multiples:**

```
if (condition1)
 instruction1
else if (condition2)
 instruction2
…
else if (condition3)
 instruction3
else
 instruction par défaut
```

Les structures fondamentales de Java

- **Branchements conditionnels à choix multiples:** le sélecteur `switch`
  - Type de la valeur énumérée pour sélectionner
    - Type primitif entier
    - Type primitif `char`
    - Chaîne de caractères : `String`
    - Objet de type `Enum`
  - L'instruction `break` en fin de clause `case` permet de sortir du `switch`. Sans break, le code de la clause suivante s'exécute en séquence.

Les structures fondamentales de Java

- **Instructions conditionnelles**
  - Opérateur conditionnel **`?:`** :

    l'expression ***`condition ? expression1 : expression2`*** est évaluée à **`expression1`** si la condition est vraie, à **`expression2`** sinon.

    Exemple : `min = (a < b )? a : b`

Les structures fondamentales de Java

- **Instructions itératives**
  - Boucles `while`
    - `while (condition) instruction;`
    - `do` instruction `while (condition);`



Les structures fondamentales de Java

- **Instructions itératives**
  - Boucles `for`
    - **`for ( expr1; expr2; expr3 ) boucle`**
      - `expr1` est l'expression d'initialisation,
      - `expr2` est la condition de continuation, si sa valeur booléenne est *false*, sortie de la boucle.
      - `expr3` est l'expression d'incrémentation.
    - **`for ( type variable : collection ) boucle`**

      Boucle de type `for each` permet de parcourir séquentiellement les éléments d'une collection.

Les structures fondamentales de Java

- **Tableaux à une dimension**
  - Un tableau est une collection de **valeurs indicées** (à partir de 0).

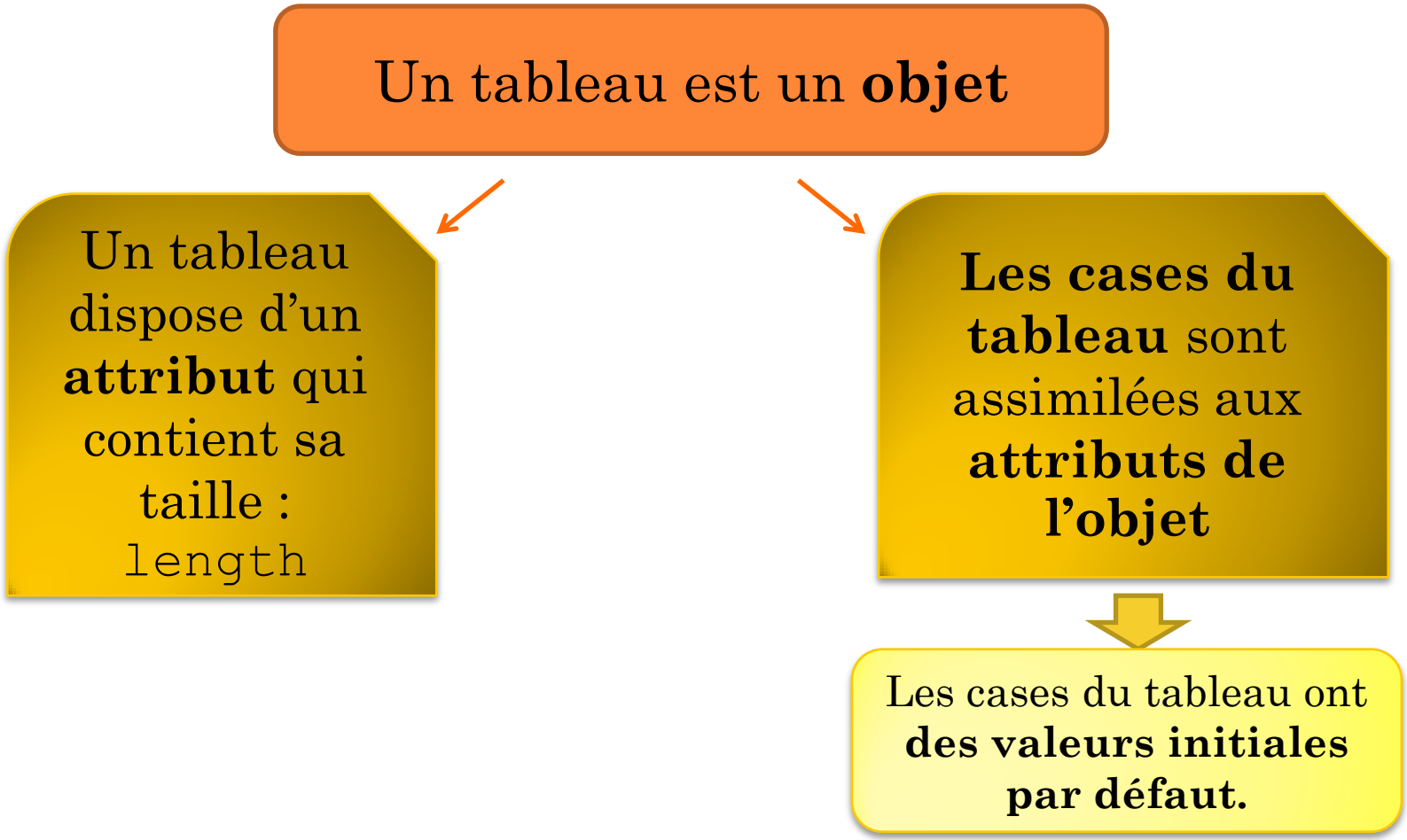
Un tableau est un **objet**

Une variable de type tableau **contient une référence**

Un tableau est **alloué dynamiquement** avec l'opérateur **new**

Les structures fondamentales de Java

- **Tableaux à une dimension**



Les structures fondamentales de Java

- **Tableaux à une dimension**
  - Exemple de déclaration d'une variable **`tab1`** de type tableau d'entiers:
    - **`int[] tab1;`**
  - Exemple de création du tableau d'entiers
    - **`int[] tab1 = new int[5];`**
  - Initialisation rapide d'un tableau d'entiers (pas besoin d'appeler *new*)
    - **`int[] tab1 = {23,45,6,7};`**

La dimension du tableau est fournie au moment de sa création.

Les structures fondamentales de Java

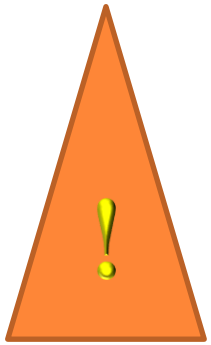
- **Tableaux à une dimension**
  - **Parcours du tableau tab1:**
    - ```
      for ( int i  =  0; i < tab1.length; i++)
          System.out.println(tab1[i]);
      ```
    - ```
      for ( int element : tab1)
          System.out.println( element );
      ```



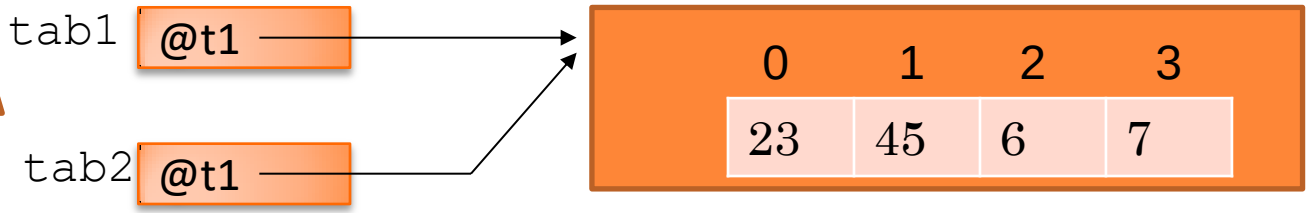

Les structures fondamentales de Java

- **Tableaux**
  - Copie d'une variable tableau dans une autre

```
int[] tab1 = {23, 45, 6, 7};
int[] tab2 = tab1;
```



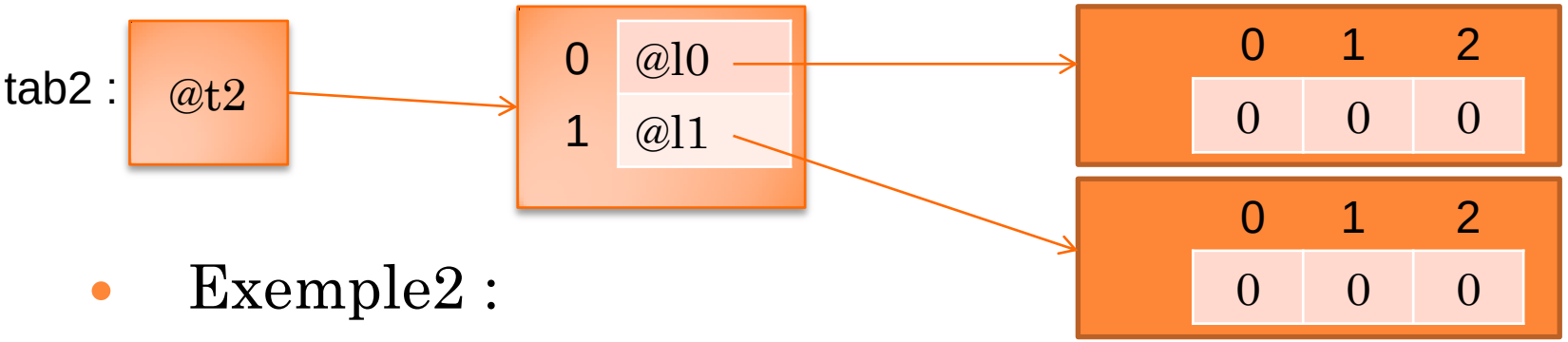
tab1 et tab2 font référence au même objet tableau.



Les structures fondamentales de Java

- **Tableaux à 2 dimensions**
  - **Un tableau à 2 dimensions est un tableau à 1 dimension de tableaux à 1 dimension**
  - Exemple1 :

```
int[][] tab2 = new int [2][3];
```



  - Exemple2 :

```
int[][] tab3 = new int[2][];
```

Les objets et les classes

- **Caractéristiques d'un objet**
  - **Une identité**
  - **Un état** : les données que l'objet porte en lui
  - **Un comportement** : déterminé par les messages qu'il reçoit.

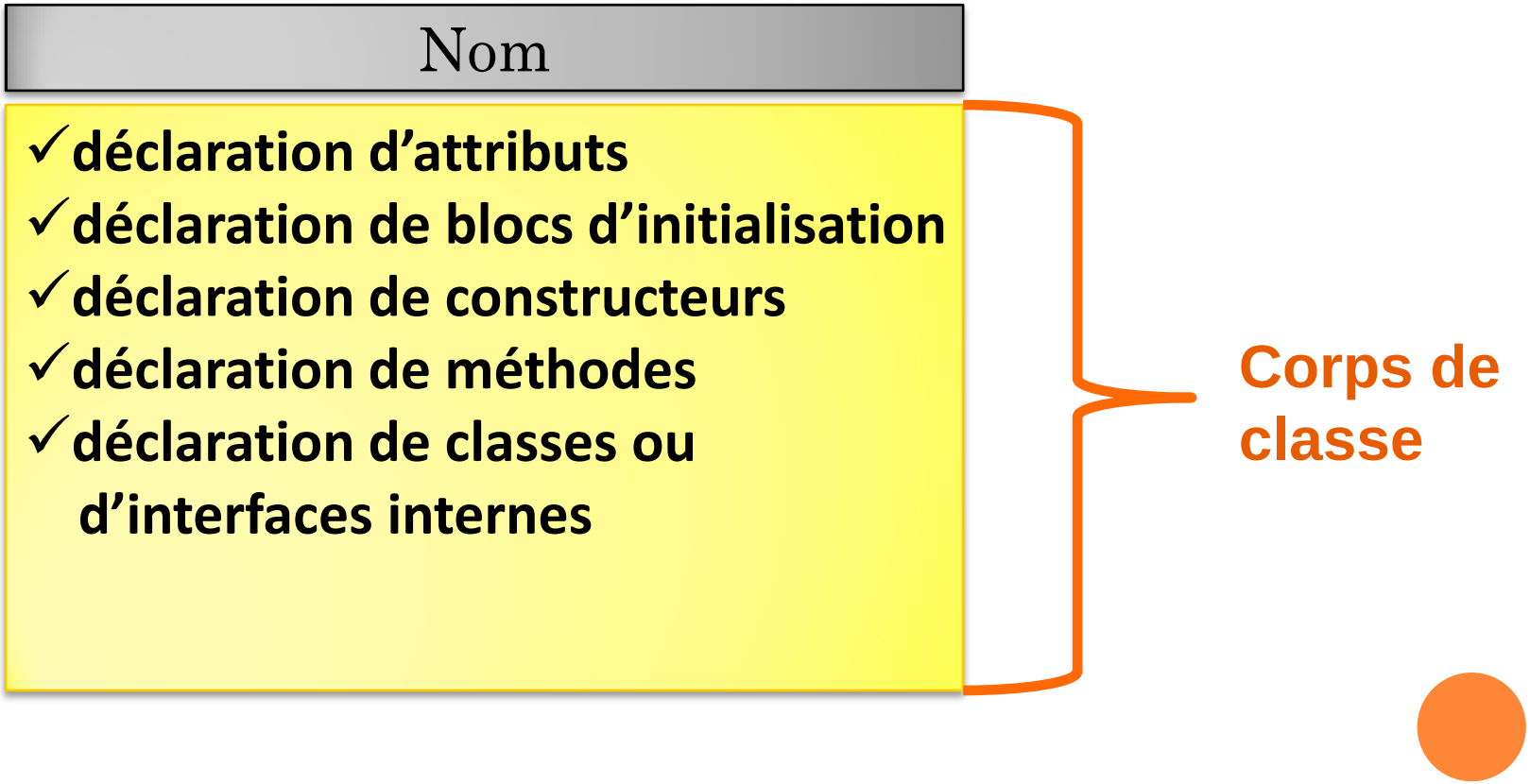
Principe d'encapsulation des données : Seul l'objet peut accéder à ses données. Pour qu'un objet extérieur y accède, il faut que l'objet ait été conçu pour proposer ce service.

Les objets et les classes

- **Concept de classe**
  - **modèle** décrivant les caractéristiques communes à plusieurs objets.
  - Une classe définit
    - les **attributs** ou **champs**
    - les **méthodes**.
  - Une classe sera **instanciée** pour créer un objet conforme au modèle.

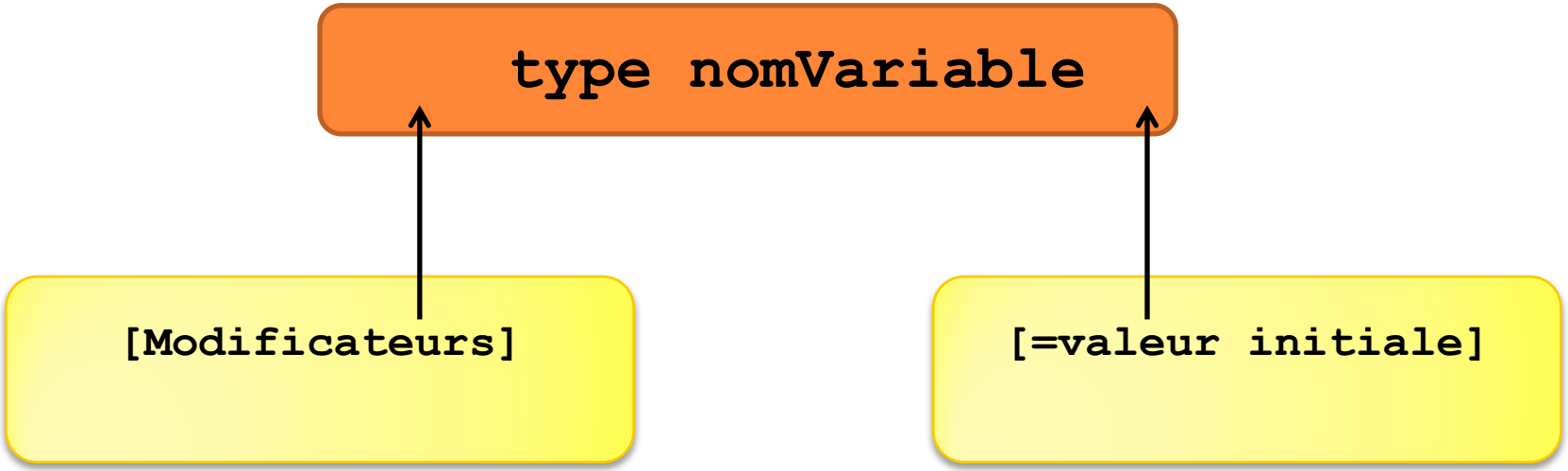
Les objets et les classes

- **Déclaration de classe**



Les objets et les classes

- **Déclaration d'attribut**
  - **Attribut = variable déclarée dans le corps de la classe**



Les objets et les classes

- **Exemple 1 :**
  - Création de la classe CompteBancaire:
    - **Variables d'instance (état) :**
      - Numéro du compte
      - Nom
      - Adresse
      - Solde

```
public class CompteBancaire {
   int numeroCompte ;
   String nom ;
   String adresse ;
   double solde ;
}
```

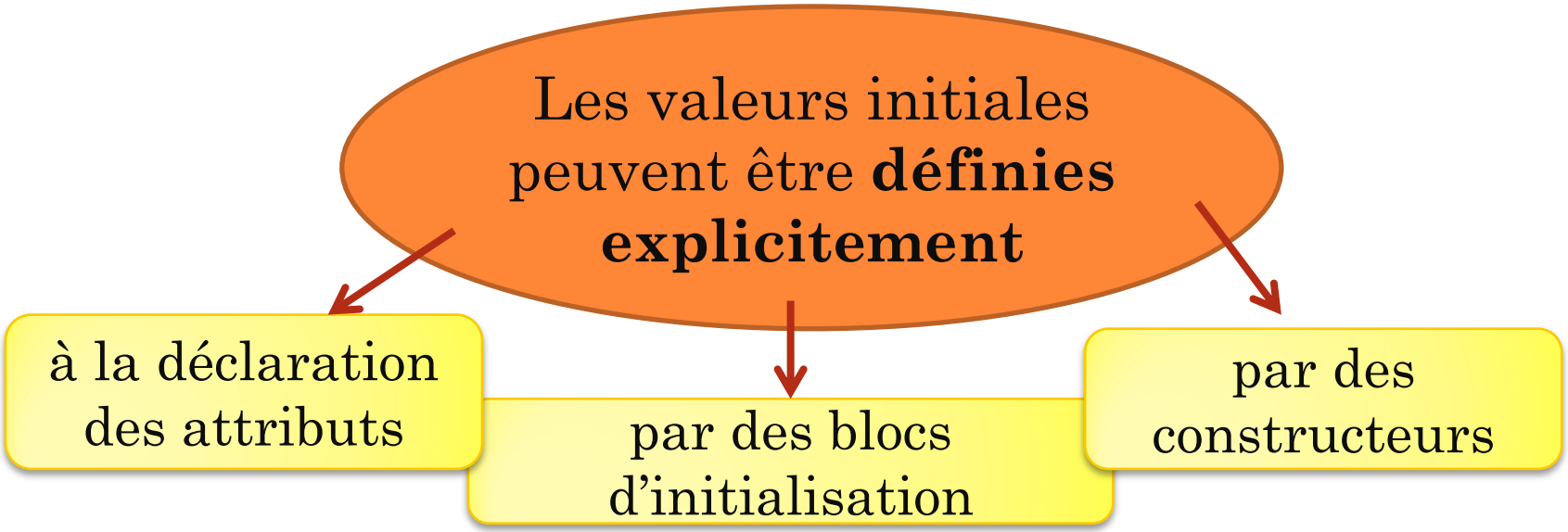
Les objets et les classes

- **Initialisation des objets**

À la création d'un objet, les attributs doivent recevoir des valeurs initiales.

Les objets et les classes

- **Initialisation des objets**



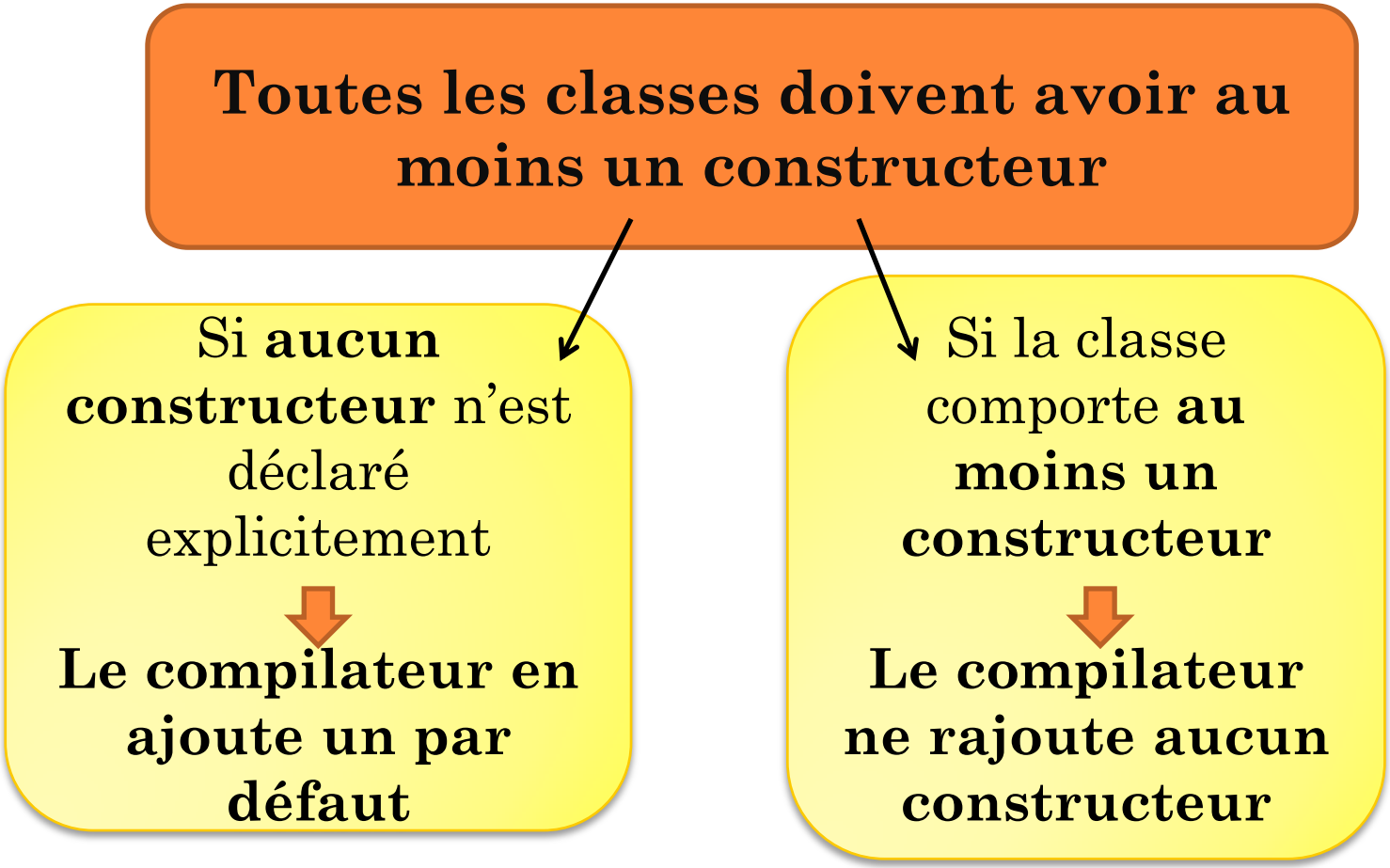
Sans initialisation explicite d'un attribut, le compilateur affecte une valeur par défaut

Les objets et les classes

- **Constructeurs**
  - Un constructeur est **une méthode particulière** qui sert à **initialiser les attributs de l'objet**.
  - **Il a les caractéristiques suivantes :**
    - Porte le **nom de la classe**
    - Ne comporte **pas de type de retour**
    - Est invoqué par l'opérateur **`new`**
  - **Surcharge de constructeurs**
    - On peut invoquer un constructeur dans un autre en le désignant par **`this`**. L'appel doit se faire en **première ligne** du constructeur.

Les objets et les classes

- **Constructeurs**



Les objets et les classes

- **Exemple 1 :**
  - Création de la classe CompteBancaire:
    - **Initialisation**
      - Créer un compte en passant le numéro, le nom et l'adresse en paramètres, solde=0.
      - Créer un compte en passant le numéro, le nom, l'adresse en paramètres et le solde



```
public class CompteBancaire {
    int numeroCompte ;
    String nom ;
    String adresse ;
    double solde ;

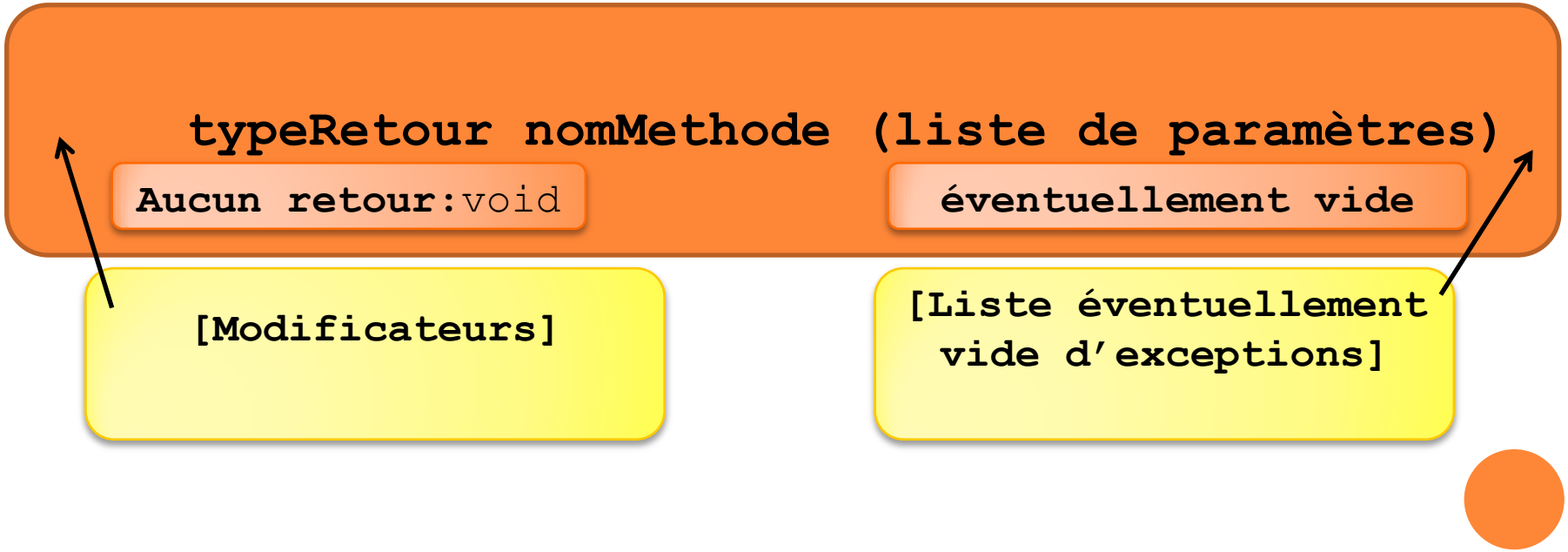
    CompteBancaire(int numeroCompte, String nom,
   String adresse, double solde){
                this.numeroCompte = numeroCompte;
                this.nom= nom;
                this.adresse=adresse;
                this.solde = solde;
    }
    CompteBancaire(int numeroCompte, String nom,
   String adresse){
                this (numeroCompte, nom, adresse, 0.0);

    }
}
```

Invocation du constructeur précédent

Les objets et les classes

- **Déclaration de méthode**
  - **Méthode = fonction déclarée dans le corps de la classe**



Les objets et les classes

- **Exemple 1 :**
  - Création de la classe CompteBancaire:
    - Ajout de services : **Méthodes (comportement)**
      - créditer
      - débiter

```
public class CompteBancaire {
    int numeroCompte ;
    String nom ;
    String adresse ;
    double solde ;
    CompteBancaire(int numeroCompte, String nom, String
   adresse, double solde){
                this.numeroCompte = numeroCompte;
                this.nom= nom;
                this.adresse=adresse;
                this.solde = solde;
    }
    CompteBancaire(int numeroCompte, String nom, String
   adresse){
                this (numeroCompte, nom, adresse, 0.0);

    }
     void crediter(double montant){
                solde = solde + montant;
     }
     void debiter(double montant){
                solde = solde - montant;
     }
}
```

Les objets et les classes

- **Paramètres de méthode**
  - **Les paramètres sont toujours passés par valeur** : La méthode reçoit une copie de toutes les valeurs de paramètre.
  - Une méthode ne peut pas modifier un paramètre de type primitif.
  - Une méthode peut modifier l'état d'un paramètre objet.

Les objets et les classes

- **Accès aux attributs et invocations de méthodes** : opérateur (.)
  - **Exemple :**

```
class A {
 int compteur = 0;
 void incremente() {
      compteur ++;
 }
}

class B {
 A a = new A();
 int c = a.compteur;

 void m (){a.incremente();}
}
```

Les objets et les classes

- **Surcharge de méthodes**
  - Possibilité dans le corps d'une classe de définir **plusieurs méthodes ayant le même nom**.
    - Elles doivent **différer en nombre et/ou en type de paramètres.**
    - On ne peut définir deux méthodes ayant même nom et ayant comme seule différence le type de retour.

Les objets et les classes

- **Nombre variable de paramètres dans une méthode**

```
typeRetour nomMethode (type param1, type param2,
type ... param3)
```

• Représente **une séquence éventuellement vide de valeurs du type déclaré**
• Il **y en a au plus une** dans une méthode
• Elle apparaît **en dernier** dans la liste des paramètres de la méthode

Les objets et les classes

- **Nombre variable de paramètres dans une méthode**
  - **Exemple**

```
class A {
    String m (int i, String... t){
          if (i < t.length && i>=0)
                return t[i];
          else return null;
    }
}
class B {
     A a = new A();
     String[] tab= {"a","b","c"};
     String s2 = a.m(2, tab);
     String s1 = a.m(0, "a","b","c");
    }
```

t correspond à un tableau

Deux appels possibles

Les objets et les classes

- **Modificateurs**
  - **final**

Attribut	Méthode	Classe
Attribut constant **Exemple :** `final int MIN = 3;`	Méthode non redéfinissable	Ne peut être décomposée en sous-classe

Les objets et les classes

- **Modificateurs**
  - **static**

Attribut statique (de classe)	Méthode statique
Commun à toutes les instances	**Fonction utilitaire de la classe**
Accès direct en utilisant le nom de la classe	Ne s'applique pas à un receveur. Accès direct en utilisant le nom de la classe
Stockage en mémoire dans une zone dédiée à la classe elle-même à laquelle toutes les instances sont liées.	Ne peut accéder aux attributs d'instance, seulement aux attributs statiques.

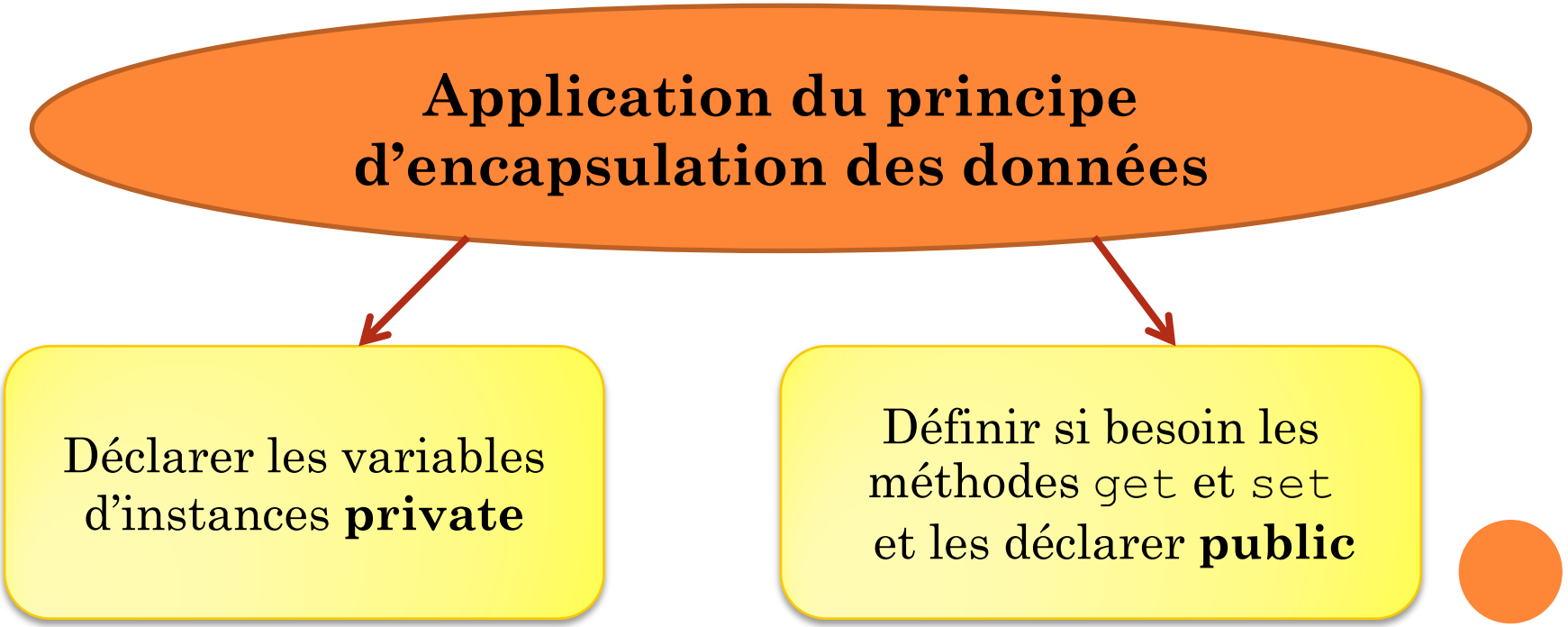
Les objets et les classes

- **Modificateurs définissant le niveau de visibilité d'un attribut ou d'une méthode ou d'une classe**

Déclaration	Classe	Package	Sous-classe	autre
`public m`	ok	ok	ok	ok
`protected m`	ok	ok	ok	
`m`	ok	ok		
`private m`	ok			

Les objets et les classes

- **Modificateurs définissant le niveau de visibilité d'un attribut ou d'une méthode ou d'une classe**



Les objets et les classes

- **Exemple 1 :**
  - Création de la classe CompteBancaire: Application du principe d'encapsulation

Application du principe d'encapsulation

```
public class CompteBancaire {
   private int numeroCompte ;
   private String nom ;
   private String adresse ;
   private double solde;

   public CompteBancaire(int numeroCompte, String nom, String
   adresse, double solde){
                this.numeroCompte = numeroCompte;
                this.nom= nom;
                this.adresse=adresse;
                this.solde = solde;
  }
   public CompteBancaire(int numeroCompte, String nom, String
   adresse){
                this (numeroCompte, nom, adresse, 0.0);
   }
   public String getNom(){
        return nom;
   }
   public void setNom(String nom){
        this.nom = nom;
   }
```

```
    public String getAdresse(){
          return adresse;
    }
    public void setAdresse(String adresse){
          this.adresse = adresse;
    }
    public int getNumeroCompte(){
          return numeroCompte;
    }
    //pas de méthode setNumeroCompte (int n)

    public int getSolde(){
          return solde;
    }
    //pas de méthode setSolde (double s)
    //le solde ne peut être modifié qu'en créditant ou débitant le compte

    public void crediter(double montant){
          solde = solde + montant;
    }
    public void debiter(double montant){
          solde = solde - montant;
    }
}
```

Les objets et les classes

- **On souhaite que le numéro du compte s'incrémente automatiquement à chaque création d'un nouveau compte.**
  Le numéro du compte ne sera plus passé en paramètre du constructeur.

```
public class CompteBancaire {
   private int numeroCompte ;
   private String nom ;
   private String adresse ;
   private double solde;
   //variable commune à toutes les instances
   private static int numeroSuivant =10;

   public CompteBancaire(String nom, String adresse, double
   solde){
                this.numeroCompte=numeroSuivant;
                numeroSuivant += 10;
                this.nom= nom;
                this.adresse=adresse;
                this.solde = solde;
                        }
   public CompteBancaire(String nom, String adresse){
                this (nom, adresse, 0.0);
   }
   …
}
```



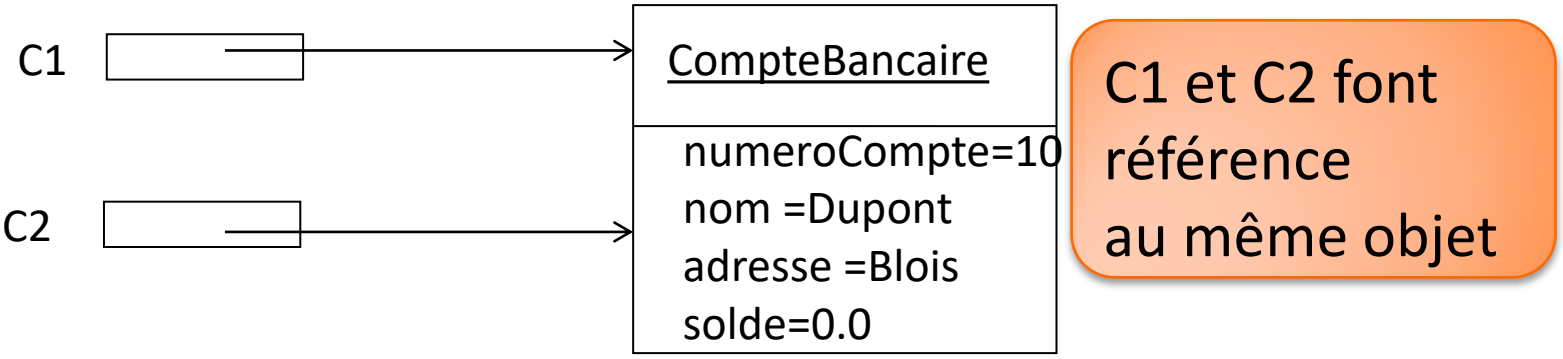

```
public class CompteBancaire {
   private int numeroCompte ;
   private String nom ;
   private String adresse ;
   private double solde;
   private static int numeroSuivant;
//bloc exécuté une seule fois au chargement dynamique de la classe
   static { numeroSuivant = 10; }
//bloc exécuté à chaque création d'un nouveau compte bancaire
   { numeroCompte = numeroSuivant; numeroSuivant += 10;}
   public CompteBancaire(String nom, String adresse, double
   solde){
                this.nom= nom;
                this.adresse=adresse;
                this.solde = solde;
                        }
   public CompteBancaire(String nom, String adresse){
               this (nom, adresse, 0.0);
   }
   …
}
```

Autre manière d'initialiser : utilisation de blocs d'initialisation

Les objets et les classes

- **Instanciation de la classe `CompteBancaire`**

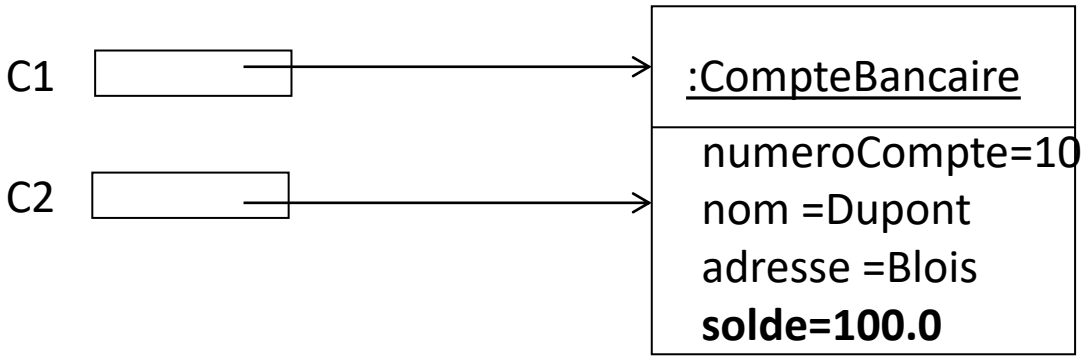
```
CompteBancaire c1 = new CompteBancaire ("Dupont",
    "Blois");
CompteBancaire c2 = c1;
```



Les objets et les classes

- **Appel de la méthode crediter**

```
c1.crediter (100);
```



Les objets et les classes

- **Packages**
  - Un package est une **structure logique** permettant de regrouper des classes (et des interfaces).
  - **Pas de notion de sous-package.**
  - Pour définir une classe (ou une interface) à l'intérieur d'un package **p**, il faut la définir dans un fichier Java dont la première ligne est : **`package p;`**

// Fichier **fichier1.java**

```
package p;
class A {/*…*/}
class B {/*…*/}
```

// Fichier **fichier2.java**

```
package p;
class E {/*…*/}
interface I {/*…*/}
```

Les objets et les classes

- **Packages**
  - **Conventions de nommage**
    - Toujours en minuscule
    - Caractères alphanumériques et des points « . »
    - Tout package devrait commencer par un nom de domaine à l'envers.
      Exemple : com.domaine.monpackage
    - Chaque nom de package correspond à une arborescence de dossier
      com/domaine/monpackage

Les objets et les classes

- **Packages et visibilité**
  - **Visibilité par défaut**

Par défaut, les classes et interfaces ne sont visibles et accessibles que par les membres du même package.

  - **Extension de visibilité avec le modificateur de classe `public`**

Une classe **publique** est **accessible à l'extérieur de son package.**

Les objets et les classes

- **Package et visibilité**

Si le fichier contient une classe publique, elle donne son nom au fichier.

```
// Fichier A.java
package p;
public class A {/*…*/}
class B {/*…*/}
```

Une seule classe ou interface publique par fichier

```
// Fichier C.java
package q;
import p.A;
public class C {
   A a;
      …
}
```

Classes particulières

- **La classe `String`**
  - Le **chaînes de caractères** en Java sont des **objets** instances de la classe **`String`**.
  - On ne peut pas ajouter, supprimer ou mettre à jour un caractère de la chaîne.

Les chaînes de caractères sont constantes



Leurs références peuvent être partagées sans risque

Classes particulières

- **La classe `String`**
  - **Différents constructeurs**
    - **Création d'une chaîne à partir d'un caractère**

```
String s1 = new String ('a'); // valeur "a"
```

    - **Création d'une chaîne à partir d'un tableau de caractères**

```
char[] charTab = {'b', 'o','n','j','o','u','r'};
String s2 = new String (charTab);
// valeur "bonjour"
```

    - **Création d'une chaîne à partir d'un littéral de type String**

```
String s3 = "bonjour" ;
```

Classes particulières

- **La classe `String`**
  - **Opérateur de concaténation : +**
  - **Fonctions de manipulation**
    - **Nombre de caractères de la chaîne**

      `int length()`
    - **Caractère à l'indice `index`**

      `char charAt (int index)`
    - **Tableau de caractères correspondant à la chaîne**

      `char[] toCharArray ()`

Classes particulières

- **La classe `String`**
  - **Fonctions de comparaison**
    - **Test d'égalité avec ou sans prise en compte de la casse**

      ```
      boolean equals(Object o)
      boolean equalsIgnoreCase(String str)
      ```

    - **Comparaison lexicographique avec ou sans prise en compte de la casse**

      ```
      int compareTo (String str)
      int compareToIgnoreCase(String str)
      ```

Classes particulières

- **La classe `String`**
  - **Fonctions de transformation**
    - **Création d'une chaîne par changement de casse**

      ```
      String toLowerCase()
      String toUpperCase()
      ```

    - **Création d'une chaîne en remplaçant un caractère**

      ```
      String replace (char oldChar, char newChar)
      ```

    - **Création d'une chaîne par suppression des espaces du début et de la fin**

      ```
      String trim ()
      ```




Classes particulières

- **La classe `String`**
  - **Fonctions d'extraction**
    - **Extraction d'une sous-chaîne de caractères**

      `String subString (int beginIndex, int endIndex)`
    - **Extraction de sous-chaînes de caractères séparées par les motifs correspondant à une expression régulière**

      `String[] split (String regex)`



Classes particulières

- **La classe `String`**
  - **Retourne la chaîne de caractère correspondant au type primitif**

    static `String valueOf (int i)`

    static `String valueOf (char c)`

    static `String valueOf (boolean b)`

Classes particulières

- **Les classes `StringBuffer` et `StringBuilder`**
  - Représentent des chaînes de caractères **modifiables**
  - Exemple des méthodes d'ajout , de suppression et de modification de caractères de la chaîne
    - `StringBuffer insert (int offset , String str)`
    - `StringBuffer deleteCharAt(int index)`
    - `StringBuffer replace(int start, int end , String str)`
  - Méthode permettant de concaténer une chaîne avec une autre : `append`

Classes particulières

- **Les classes enveloppes**
  - **Chaque type primitif a une classe correspondante, nommée classe enveloppe.**

Type primitif	`boolean`	`byte`	`char`	`short`
Classe enveloppe	`Boolean`	`Byte`	`Character`	`Short`
Type primitif	`int`	`long`	`float`	`double`
Classe enveloppe	`Integer`	`Long`	`Float`	`Double`

Classes particulières

- **Les classes enveloppes**
  - Les classes enveloppes fournissent les méthodes de base permettant de convertir des chaînes de caractères en nombre
    - Exemple :

```
int x = Integer.parseInt (s);
```

Classes particulières

- **La classe `java.util.Scanner`**
  - Donne la possibilité de balayer un texte et d'en extraire les éléments de type primitif ou les chaînes de caractères.
  - Un objet de type **`Scanner`** va décomposer son entrée en mots en utilisant un séparateur, qui par défaut est l'espace vide.



Classes particulières

- **La classe `java.util.Scanner`**
  - **Méthodes de la classe Scanner pour lire les entrées**
    - `public String nextLine()`
    - `public String next ()`
    - `public int nextInt()`
    - `public double nextDouble()`
  - **Méthodes de la classe Scanner pour tester l'existence d'un élément à lire**
    - `public boolean hasNextLine()`
    - `public boolean hasNext()`
    - `public boolean hasNextInt()`
    - `public boolean hasNextDouble()`



Classes particulières

- **La classe `java.util.Scanner` permet de lire les entrées**
  - **depuis un objet de type String (ex : `String leTexte`)**

```
Scanner sc = new Scanner (leTexte);
int i = sc.nextInt();
```

  - **depuis la fenêtre de la console**

On construit un objet Scanner attaché à l'unité « d'entrée standard » System.in :

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

  - **depuis un fichier contenant des entiers de type long**

```
Scanner sc = new Scanner(new File("DesNombres"));
while (sc.hasNextLong())
        { long aLong = sc.nextLong(); }
```

Classes particulières

- **La classe Arrays**
  - **Fournit de nombreuses méthodes statiques de manipulation des tableaux**
    - **Création d'une copie d'un tableau avec plus ou moins de cases**
      - **Méthode pour un tableau d'entiers:**

```
public static int[] copyOf (int[]
original, int newLength)
```

    - **Trier un tableau**
      - **Méthode pour un tableau d'entiers:**

```
public static void sort(int[] a)
```

Classes particulières

- **La classe Arrays**
  - **Fournit de nombreuses méthodes statiques de manipulation des tableaux**
    - **Test d'égalité de deux tableaux de tout type (même taille et même valeurs dans le même ordre)**
      - **Méthode pour deux tableaux d'entiers:**

        ```
        public static boolean equals(int[] a,
        int[] a2)
        ```

    - **Test d'égalité de deux tableaux d'objets en profondeur** (utile pour les tableaux à n dimensions)

```
public static boolean deepEquals(Object []
    a1, Object[] a2)
```

Classes particulières

- **La classe Arrays**
  - **Production d'une chaîne de caractères représentant les éléments du tableau**
    - **Méthode pour un tableau d'entiers:**

      `public static String` **`toString`**`(int[] a)`
    - **Méthode pour un tableau à n dimensions:**

      `public static String` **`deepToString`**`(Object[] a)`