Types des données et classes standard

Christelle CAILLOUET
(christelle.caillouet@unice.fr)

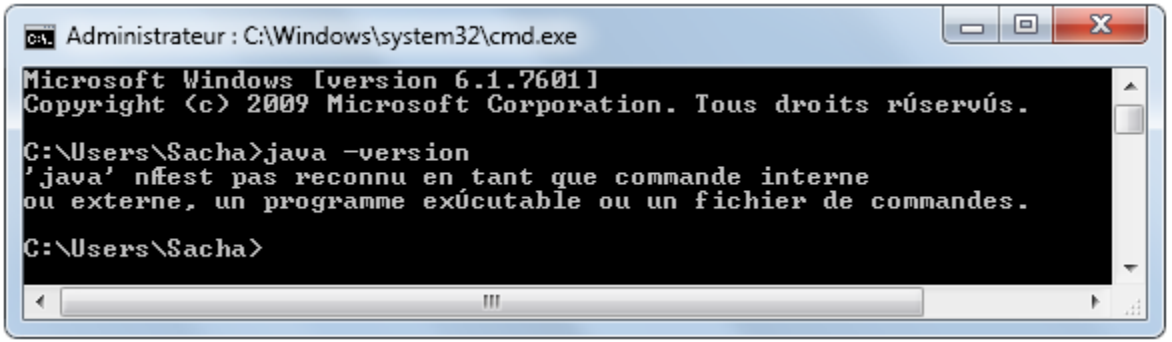
Retour sur le premier TD

Prendre vos clickers

Et brancher le récepteur…

Si l'on obtient ce message, que faut-il faire ?

20



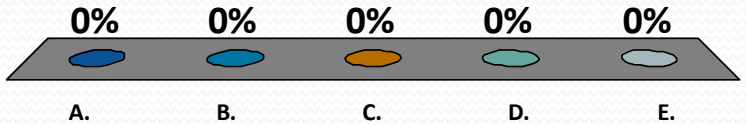
A. Arrêter de faire du java

B. Enlever l'option –version

C. Mettre une majuscule à java

✓ D. Mettre à jour une variable d'environnement

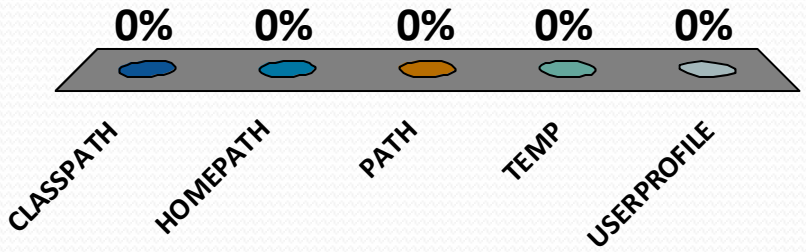
E. Redémarrer l'ordinateur



Et maintenant, quelle variable d'environnement ?

A. CLASSPATH
B. HOMEPATH
✓C. PATH
D. TEMP
E. USERPROFILE

20



Les variables d'environnement utiles

- **PATH** : contient une liste de répertoires dans lesquels vont être recherchés les fichiers exécutables

➔ Ainsi, afin de pouvoir utiliser *javac*, *java*,..., il peut être nécessaire de modifier le PATH afin d'y ajouter le répertoire bin du JRE/JDK

- **CLASSPATH** : permet de spécifier à la JVM les emplacements des fichiers compilés
  Indispensable pour les packages prédéfinis

➔Lorsque la JVM a besoin d'une ressource ou d'une classe, elle la recherche dans les divers éléments du CLASSPATH dans l'ordre de leur déclaration

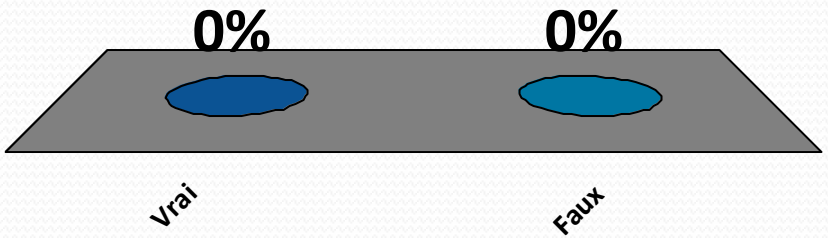
Ce constructeur est-il correct ?

```
public class Point {
    private double abscisse;
    private double ordonnee;

    /**
    * Constructeur par defaut
    **/
    public Point(){
        abscisse = null;
        ordonnee = null;
    }
```

A. Vrai

✓B. Faux

20



Types et manipulation

Nature des variables en java

1. Type **primitif**
   - La déclaration réserve la place mémoire pour stocker sa valeur (qui dépend de son type)

1. Type **objet**
   - La déclaration ne fait que réserver la place d'une **référence** (sorte d'adresse mémoire) qui permettra d'accéder à l'endroit en mémoire où est effectivement stocké l'objet
     (vaut **null** si référence inconnue)
     ➔ *toute variable désignant un objet est donc un pointeur !*

Types primitifs

Type	Taille (octets)	Valeur	Défaut
`boolean`	1	`true` ou `false`	`false`
`byte`	1	Entier signé	`0`
`short`	2	Entier signé	`0`
`int`	4	Entier signé	`0`
`long`	8	Entier signé	`0`
`float`	4	Réel signé	`0.0`
`double`	8	Réel signé	`0.0`
`char`	2	Caractère Unicode	`\u0000`

Attention aux nombres à virgule flottante

- Ils ne sont que des approximations des valeurs !
- Leur égalité au sens de l'opérateur == **n'a aucun sens**
- Qu'est-ce que cela signifie en pratique ?

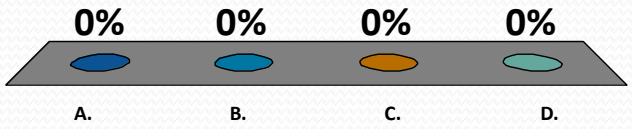
Quel résultat obtient-on ?

```
double d = 0.0;
int nb = 0;
while (d != 1.0 /* && nb < 10 */){
    d += 0.1;
    nb++;
}
System.out.println("nb = "+nb+", d = "+d);
```

En laissant la 2ème condition en commentaire

A. `nb = 1, d = 0.1`
   …
   `nb = 10, d = 1.0`
B. Erreur de compilation
✓C. Boucle infinie
D. Aucune idée

20



Attention aux nombres à virgule flottante

```
double d = 0.0;
int nb = 0;
while (d != 1.0 /* && nb < 10 */){
    d += 0.1;
    nb++;
}
System.out.println("nb = "+nb+", d = "+d);
```

// boucle infinie !!

➔ Si on décommente, affiche

```
nb = 10, d = 0.9999999999999999
```

- Il faut tester leur *proximité* modulo un epsilon donné

```
static final double EPSILON = 0.00001;
...
while (Math.abs(1.0 - d) < EPSILON)
```

Equivalence type primitif/classe

- Chacun des types primitifs peut être *enveloppé* dans un objet provenant d'une classe prévue à cet effet et appelée ***Wrapper*** (mot anglais signifiant *enveloppeur*). Les enveloppeurs sont donc des objets représentant un type primitif.

- **Avantages :**
  - Les *Wrapper* peuvent être utilisés comme n'importe quel objet, ils ont donc leurs propres méthodes.
- **Inconvénients :**
  - Plus d'espace mémoire que le type primitif
    Par exemple: `int` = 4 octets en mémoire mais `Integer` = 32 octets sur une VM en 64 bits (20 octets en 32 bits).
  - L'objet enveloppant est immuable (ne peut pas être modifié)
    Toute modification de sa valeur nécessite de créer un nouvel objet et de détruire l'ancien, ce qui augmente le temps de calcul.

Type primitif/classe

Type	Classe équivalente (*Wrapper*)	Min	Max
boolean	Boolean		
byte	Byte	-128 (Byte.MIN_VALUE)	127 (Byte.MAX_VALUE)
short	Short	-32 768 (Short.MIN_VALUE)	32 767 (Short.MAX_VALUE)
int	Integer	-2 147 483 648 (Integer.MIN_VALUE)	2 417 483 647 (Integer.MAX_VALUE)
long	Long	-9 223 372 036 854 775 808 (Long.MIN_VALUE)	9 223 372 036 854 775 807 (Long.MAX_VALUE)
float	Float	1.40239846E-45 (Float.MIN_VALUE)	3.40282347E38 (Float.MAX_VALUE)
double	Double	4.9406564584124654E-324 (Double.MIN_VALUE)	1.7976931348623l6E308 (Double.MAX_VALUE)
char	Character	\u0000	\uFFFF

Exemples

- Création d'un objet à partir d'un primitif

```
int a = 4;
Integer i = new Integer(a);
```

- Création d'un primitif à partir d'un objet

```
int j = i.intValue();
```

- Création d'un primitif à partir d'une chaîne

int	Integer.parseInt(String s)	*convert s to an int value*
double	Double.parseDouble(String s)	*convert s to a double value*
long	Long.parseLong(String s)	*convert s to a long value*

Conversion de type de données (Transtypage)

- Conversion implicite :
  - Modification du type de donnée effectuée automatiquement par le compilateur (par exemple entre les types primitifs et leur *Wrapper*)

```
int n = 5;
Integer m = n;
```

- Conversion explicite (*cast*) :
  - Modification du type de donnée forcée

```
double x = 8.234;
int n = (int) x; // donne n = 8
```

Passage de paramètre

1. Type primitif
   - Passage par **valeur** :
     - La valeur de l'argument est recopiée dans le paramètre de la méthode
     - Les modifications sur le paramètre (i.e. dans la méthode) *sont sans effet* sur l'argument (après l'appel)

Quel est le résultat ?

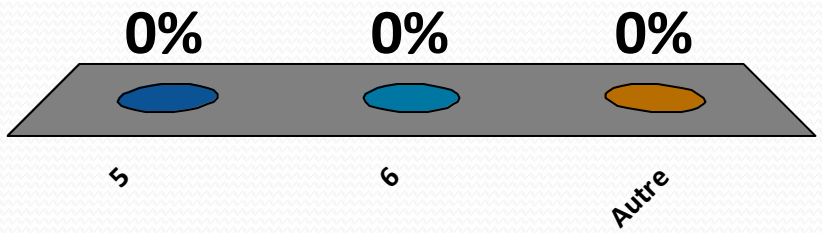
```
public static void m1(int i){
    i++;
}

public static void main(String[] args) {
    int entier = 5;
    m1(entier);
    System.out.println(entier);
}
```

✓A. 5

B. 6

C. Autre

30

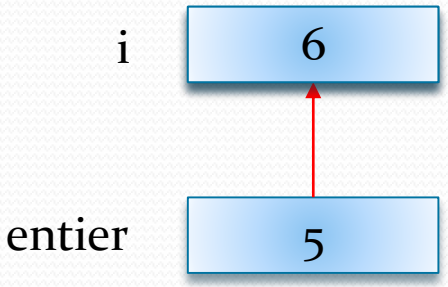


Passage de paramètre

1. Type primitif

```
public static void m1(int i){
    i++;
}

public static void main(String[] args) {
    int entier = 5;
    m1(entier);
    System.out.println(entier);
}
```

➔ 5



Passage de paramètre

2. Type objet
   - Passage par **référence** :
     - La référence est recopiée dans le paramètre de la méthode
     - Les modifications effectuées en suivant cette référence (des champs de l'objet) *sont répercutées* dans la mémoire et donc sur l'argument
     - En revanche, la modification de la référence elle-même est *sans effet* sur l'argument
       ➔ copie locale d'un objet ayant la même référence

Quel est le résultat ?

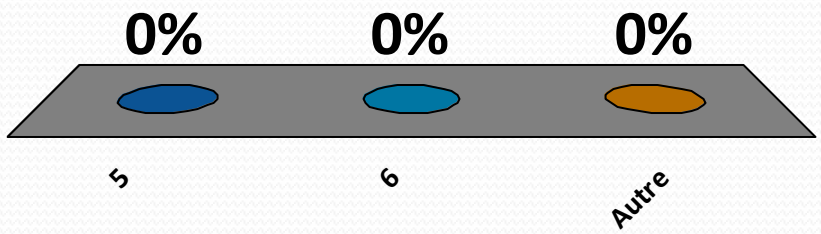
```
public class Box{
    private int field;

    public static void m2(Box b){
        b.field++;
    }

    public static void main(String[] args) {
        Box box = new Box();
        box.field = 5;
        m2(box);
        System.out.println(box.field);
    }
}
```

A. 5

✓ B. 6

C. Autre

60

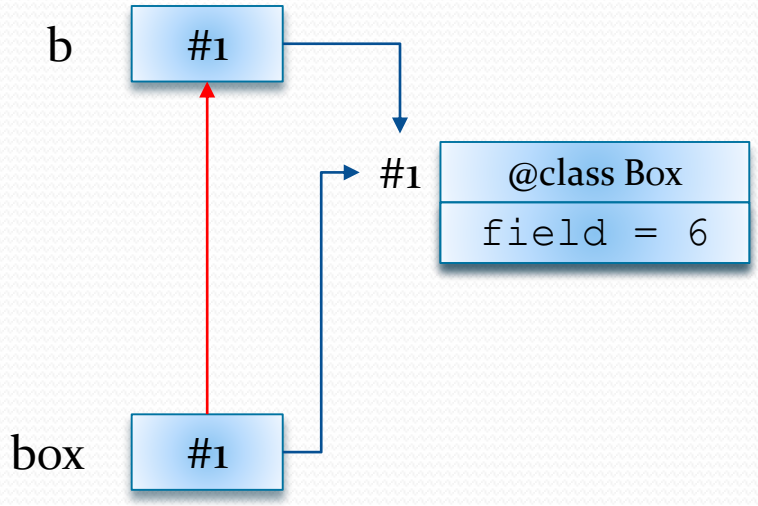


Passage de paramètre

2. Type objet

```
public class Box{
    private int field;

    public static void m2(Box b){
        b.field++;
    }

    public static void main(String[] args) {
        Box box = new Box();
        box.field = 5;
        m2(box);
        System.out.println(box.field);
    }
}
```

➔ 6



Quel est le résultat ?

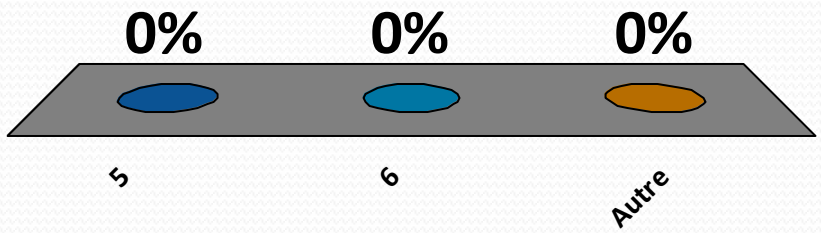
```
public static void m3(Box b){
    Box tmp = new Box();
    tmp.field = b.field+1;
    b = tmp;
}

public static void main(String[] args) {
    Box box = new Box();
    box.field = 5;
    m3(box);
    System.out.println(box.field);
}
```

✓A. 5

B. 6

C. Autre

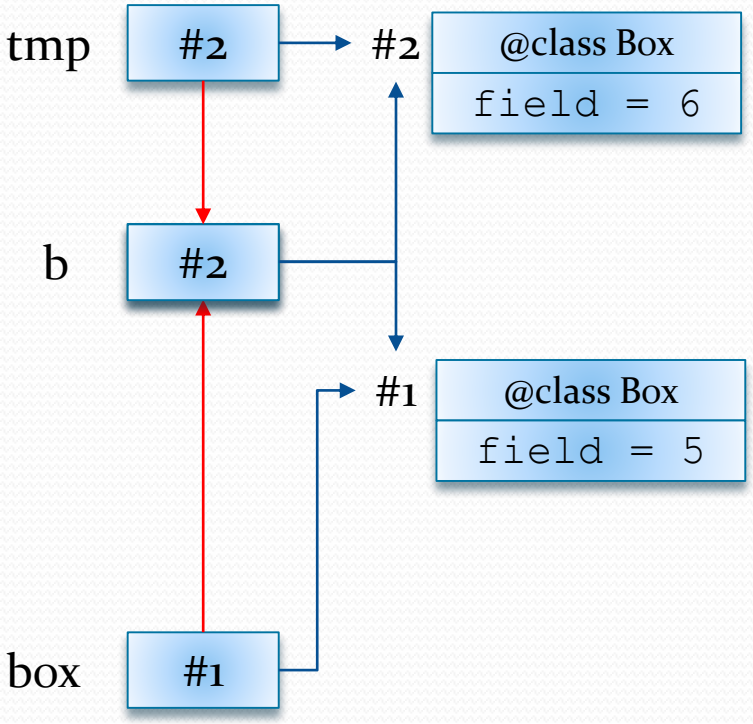
60



Passage de paramètre

2. Type objet

```
public static void m3(Box b){
    Box tmp = new Box();
    tmp.field = b.field+1;
    b = tmp;
}

public static void main(String[] args) {
    Box box = new Box();
    box.field = 5;
    m3(box);
    System.out.println(box.field); ➔ 5
}
```



Allocation mémoire

- Un objet non alloué a pour valeur `null` (aucune référence)

- Pour qu'une variable objet prenne une autre valeur que `null`, il faut lui affecter une référence (à l'aide de l'opérateur d'allocation **`new`**)
  - La place de la référence est alors réservée en mémoire
  - La zone allouée dépend de la taille de l'objet

```
new Box(); // besoin de stocker 1 int (field)
new int[10]; // stocker 10 int + la taille du tableau
new Point(1,3); // zone mémoire initialisée à l'aide
                   d'un constructeur
```

Désallocation mémoire

- Non gérée par le programmeur mais par le **Garbage Collector**

- Les objets qui ne sont plus référencés (par aucune variable !) sont *récupérés* par le garbage collector pour *recycler* l'espace mémoire

- Les variables cessent de référencer un objet :
  - Quand on leur affecte un autre objet, ou `null`
  - Quand on quitte le bloc où elles sont définies

Types et Classes standard

Les tableaux en Java

- En Java, les tableaux sont des **objets**
- Déclaration à l'aide des crochets []

```
int[] tab1 = {1,2,3,4,5,6,7,8,9};
int tab2[] = new int[10];
```

- Tableaux à plusieurs dimensions (tableaux de tableaux) :
  - Tableaux à 2 dimensions = matrice [][]
  - Tableaux à 3 dimensions [][][]
  - …

- Accès à la taille par l'attribut publique **length**

```
int n = tab2.length;
```

Exemples

```
public static void main(String[] args){
    int[] tabInt = {0,1,2,3,4,5,6,7,8,9};
    double[] tabDouble = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
    char[] tabChar = {'a','b','c','d','e','f','g','h','i','j','k','l','m'};
}
```

- Tableaux d'objets

```
String[] tabString = {"lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"};
```

```
public class Livre{
    private String titre;
    private Lecteur emprunteur;
```

```
public class Lecteur{
    private String nom;
    private Livre[] emprunts;
    private int nbEmprunts;
```

Package java.lang

http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html

- Ensemble de classes prédéfinies qui proposent des services

- Principales classes (importation implicite)
  - La classe **Math**
  - la classe **String**
  - La classe **System**
  - La classe **Object**
  - Les classes enveloppes (*Wrappers*)
  - La classe **Class**
  - La classe **RunTime**
  - …

La classe **String**

http://docs.oracle.com/javase/8/docs/api/java/lang/String.html

- Représente les chaînes de caractères
- En Java, les chaînes de caractères sont des **objets**
  /!\ ce ne sont pas des tableaux !

- Création d'une chaîne

```
String s = new String("bonjour");
String t = "bonjour";
```

- Taille d'une chaîne

```
int taille = s.length();
```

- Accès à un caractère

```
char c = s.charAt(3);
```
➔ j

Les chaînes de caractères

- Concaténation de chaînes

```
String s = t + " toi";
t += " toi";
```

- Transtypage implicite

```
int i = 12;
String s = "" + i;
```

- Comparaison de deux chaînes
  - Opérateur == compare les références des objets
  - Méthode `equals` compare les contenus des références

```
if (s.equals("bonjour"))…
```

- Recherche d'une sous-chaîne

```
int position = s.indexOf("toi");
```

- Extraction d'une sous-chaîne

```
String t = s.substring(position, position+3);
```

La classe **Math**

http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

public class Math	
double abs(double a)	*absolute value of a*
double max(double a, double b)	*maximum of a and b*
double min(double a, double b)	*minimum of a and b*
double sin(double theta)	*sine of theta*
double cos(double theta)	*cosine of theta*
double tan(double theta)	*tangent of theta*
double toRadians(double degrees)	*convert angle from degrees to radians*
double toDegrees(double radians)	*convert angle from radians to degrees*
double exp(double a)	*exponential* (e^a)
double log(double a)	*natural log* ($\log_e a$, *or* $\ln a$)
double pow(double a, double b)	*raise a to the bth power* (a^b)
long round(double a)	*round a to the nearest integer*
double random()	*random number in* [0, 1)
double sqrt(double a)	*square root of a*
double E	*value of e (constant)*
double PI	*value of π (constant)*

La classe **System**

http://docs.oracle.com/javase/8/docs/api/java/lang/System.html

- Centralise l'accès :
  - Aux trois flux de base ***in, out, err***
  - À l'horloge du système d'exploitation
  - Aux fonctions utilitaires de la JVM

- Principales méthodes de classe :
  - Méthodes ***currentTimeMillis***
  - Méthode ***exit*** `(System.exit(0);)`
  - Méthodes ***getProperty, setProperty***
  - …

- Attributs de classe associés aux flux standards

```
System.out          Classe PrintStream
System.in           Classe PrintStream
System.err          Classe PrintStream
```

Hiérarchie de classes

- Un package contient un ensemble de classes
- Toutes les classes d'un package sont organisées en hiérarchie

- Dans le package `java.lang`, toutes les classes sont dérivées de la classe `Object`, base de la hiérarchie

http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html

Object	Class `Object` is the root of the class hierarchy.

La classe **Object**

- Cette classe contient (sous forme de méthodes), les servitudes de base pour la gestion des objets

- **Transmet implicitement toutes ses méthodes à toute classe Java**
  - Relation d'héritage (cf. cours suivants)
  - Induit la nécessité de **redéfinir** ces méthodes dans toute classe Java
  - **Transtypage implicite** possible de toute référence sur un objet d'une classe quelconque, dans une variable de type `Object` (analogie avec le type void* du langage C)

La classe **Object**

- Met à disposition un constructeur par défaut

- Principales méthodes (d'instance)
  - `toString` : retourne un descriptif de l'objet cible
  - `equals` : prédicat d'égalité de 2 objets
  - `clone` : crée et retourne une copie de l'objet cible
  - `getClass` : retourne la classe de l'objet cible
  - …

- Dans toute classe, on peut redéfinir les méthodes de la classe **Object** (cf. cours Polymorphisme plus tard)

http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Découvrir d'autres packages et classes existantes

- API Specification

  - Les packages

http://docs.oracle.com/javase/8/docs/api/overview-summary.html

  - Les classes

http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html