Types primitifs et non primitifs

Types de données primitifs vs non primitifs

- En Java les types de données sont divisés en deux groupes:
  - **Primitifs**: comprend **octet**, **court**, **int**, **long**, **float**, **double**, **boolean** et **char**
  - Types de données **non primitifs** – tels que **String**, **Arrays** et **Classes**

Types primitifs *vs* types référence : Autoboxing

- En Java pour chaque type primitif il existe un type objet correspondant appelée *type enveloppe* (ou *wrapper type*)

  `Integer` pour `int`, `Boolean` pour `boolean`, *etc.*

- Les collections Java n'utilisent que des types objets :

```
ArrayList<int> liste = new ArrayList<>(); // ne compile pas
ArrayList<Integer> liste = new ArrayList<>(); // correcte
```

- Pour jongler entre les types enveloppes et les types primitifs, Java utilise le déballage/emballage automatique :

```
ArrayList<Integer> liste = new ArrayList<Integer>();
int nombre = 5;
liste.add(new Integer(nombre)); //emballage (boxing)
liste.add(nombre); //emballage automatique (autoboxing)
int v1 = liste.get(0); // déballage automatique (auto-unboxing)
int v2 = liste.get(1); // déballage automatique (auto-unboxing)
Integer v3 = liste.get(0);
Integer v4 = liste.get(1);
int v5 = v4.intValue(); // déballage
```

Types référence (objet)

- Hormis les types primitifs, tous les types en Java sont des objets.

- Exemples des types objets :
  - `String`
  - Tableaux, listes (et les autres structures de données)
  - Toutes les autres classes pré-définies en Java ou créées par l'utilisateur

- La valeur par défaut des types objets est `null` :

```
Voiture v;
System.out.println(v); // affiche "null"
```

pourquoi java contient des types primitifs?

- La réponse est que les **types primitifs sont une exception** en Java, car ce **ne sont pas des objets**, ce qui rend Java (si on voulait être rigoureux) un langage non pas orienté objet au sens de **SmallTalk**, mais un langage appliquant une « philosophie » objet.

- **Creating object**, allocating heap is too costly and there is a **performance penalty** for it.

- in-case you are **ready to compromise** bit on **performance** but you **require more facility of OOP**. So in that case **you can use wrappers (enveloppes)**.

Les classes enveloppes

- Les **classes enveloppes** (appelées aussi ***wrappers***) sont nées d'un besoin de pouvoir **encapsuler des types primitifs dans des objets** afin de pouvoir par exemple les mettre dans une collection ou bien dans une base de données objet.

- ***Integer*** *objetEntier = new* ***Integer(2)***.
- **int** a = 6;

Integer b = new **Integer**(0); //**boxing**

b = a; //**autoboxing**

- ***Unboxing***: a = b;

Utilité des types primitifs

L'esprit de la programmation orientée objet est que *tout* est objet.

à quoi bon avoir créé des types primitifs ? ? ?

Que fait-il ?

```
class App {
    public static void main(String args[]) {
        Integer somme = 0;
        for (int i = 0; i < 100000; i++)
            somme += i;

        System.out.println(somme);
    }
}
```

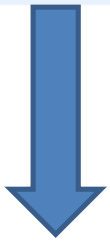
Inconvénients ?

Inconvenient des types non primitifs

```java
public class Class1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        long start_time =System. currentTimeMillis();
        int somme=3;
        for (int i=1; i<100000; i++) {somme+=i;}
        long final_time =System. currentTimeMillis();
        System.out.println("start_time="+start_time);
        System.out.println("final_time="+final_time);
        System.out.println("time="+(final_time-start_time));
    }

}
```



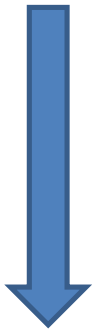
Problems @ Javadoc Declaration Console

<terminated> Class1 [Java Application] C:\Users\kahlo\.p2\pool\p

```
start_time=1694791150404
final_time=1694791150406
time=2
```

```java
public class Class1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        long start_time =System. currentTimeMillis();
        Integer somme=3;
        for (int i=1; i<100000; i++) {somme+=i;}
        long final_time =System. currentTimeMillis();
        System.out.println("start_time="+start_time);
        System.out.println("final_time="+final_time);
        System.out.println("time="+(final_time-start_time));
    }
```



Problems @ Javadoc Declaration Console

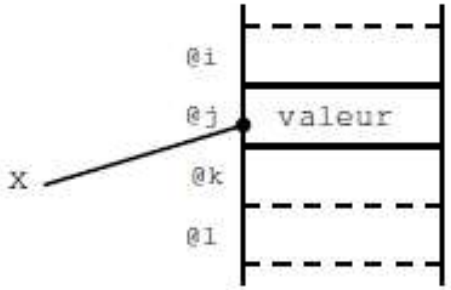
<terminated> Class1 [Java Application] C:\Users\kahlo\.p2\pool

```
start_time=1694791218798
final_time=1694791218806
time=8
```

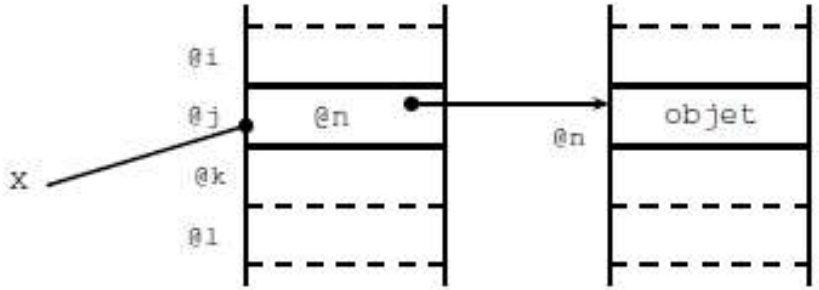
Typage Java : représentation en mémoire

Comment une variable x est représentée en mémoire ?

Cas 1 type primitif : `x = valeur`



Cas 2 type référence : `x = objet`



Accès par valeur vs accès par référence

```java
int x = 10, y = 20;
x = y;
System.out.println(x + ", "+y); // resultat ?
y++;
System.out.println(x + ", "+y); // resultat ?
```

```java
class Compte {

    private String nom, prenom;
    private double soldeCompte;

    public Compte(String n, String p) {
       nom = n; prenom = p;
       soldeCompte = 0;
    }

    public void rechargerCompte(double somme) {
       soldeCompte += somme;
    }


    // méthode spécifique Java pour l'affichage
    public String toString() {
       return "[" + prenom +
            ","+nom+","+soldeCompte +"]";
       }

}
```

```java
Compte c1 = new Compte("Tartempion", "Riri");
c1.rechargerCompte(100);
Compte c2 = new Compte("Barbanchu", "Fifi");
c2.rechargerCompte(392);
Compte c3 = new Compte("Duchmolle", "LouLou");

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?

c1 = c2;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?

c1.rechargerCompte(55);

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?

c2 = c3;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?

c3 = c1;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?
```

Accès par valeur vs accès par référence

```
Compte c1 = new Compte("Tartempion", "Riri");
c1.rechargerCompte(100);
Compte c2 = new Compte("Barbanchu", "Fifi");
c2.rechargerCompte(392);
Compte c3 = new Compte("Duchmolle", "LouLou");

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?
```

```
[Riri,Tartempion,100.0]
[Fifi,Barbanchu,392.0]
[LouLou,Duchmolle,0.0]
```

```
c1 = c2;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
```

```
[Fifi,Barbanchu,392.0]
[Fifi,Barbanchu,392.0]
```

```
c1.rechargerCompte(55);

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
```

```
[Fifi,Barbanchu,447.0]
[Fifi,Barbanchu,447.0]
```

```
c2 = c3;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?
```

```
[Fifi,Barbanchu,447.0]
[LouLou,Duchmolle,0.0]
[LouLou,Duchmolle,0.0]
```

```
c3 = c1;

System.out.println(c1); // resultat ?
System.out.println(c2); // resultat ?
System.out.println(c3); // resultat ?
```

```
[Fifi,Barbanchu,447.0]
[LouLou,Duchmolle,0.0]
[Fifi,Barbanchu,447.0]
```

Passage de paramètres

Dans les langages de programmation modernes, il existe essentiellement deux modes de passage de paramètres :

Passage par valeur

C'est *la valeur* de la variable au moment de l'appel qui est utilisée. Autrement dit, c'est une *copie* du contenu de cette variable au moment de l'appel qui est passée à la fonction.

⇒ La variable initiale n'est pas accessible (donc **non-modifiable**) par la fonction appelée.

Passage par adresse

C'est *l'adresse* de la variable placée en paramètre lors de l'appel qui est utilisée.

⇒ La variable initiale est accessible (et donc **modifiable**) par la fonction appelée

Passage de paramètres en Java

En Java, le passage de paramètres se fait toujours **par valeur**

Autrement dit, la méthode manipule une *copie* du contenu de la variable passée en paramètre.

1. si la variable est de type primitif : son contenu est une valeur effective, donc c'est en quelque sorte une "vraie" copie

```
public void foo(int v) {
   v = 30;
   // v est une copie de la variable passée en paramètre,
   // à la sortie de la fonction, les modifications
   // sur cette copie seront oubliées
}
```

```
public void toto() {
   int x = 10;
   System.out.println(x); // 10
   foo(x);
   System.out.println(x); // 10
}
```

2. si la variable est de type référence, alors la méthode manipule une copie de l'adresse (indiquant où est situé l'objet concerné). Donc : cette adresse est *non-modifiable* (car passage par valeur) **mais**

l'objet référencé, lui, est accessible, et donc modifiable à travers son interface publique.

Passage de paramètres en Java

Attention aux paramètres de type objet !

```
class Voiture {
   String modele;

   public Voiture(String m) {
      modele = m;
   }

   public void changerModele(String m) {
      modele = m;
   }

   public String toString() {
      return "La voiture est une "+modele;
   }
}
```

```
class Usine1 {
   public void customizer(Voiture v) {
      v = new Voiture ("berline");
   }
}
```

```
class Usine2 {
   public void customizer(Voiture v) {
      v.changerModele("berline")
   }
}
```

```
class App {
   /* Du code ici */
   public static void main(String args[]) {
      Voiture maCaisse = new Voiture("cabriolet");
      System.out.println(maCaisse); // resultat ?
      Usine1 u1 = new Usine1();
      u1.customizer(maCaisse);
      System.out.println(maCaisse); // resultat ?
   }
}
```

```
class App {
   /* Du code ici */
   public static void main(String args[]) {
      Voiture maCaisse = new Voiture("cabriolet");
      System.out.println(maCaisse); // resultat ?
      Usine2 u2 = new Usine2();
      u2.customizer(maCaisse);
      System.out.println(maCaisse); // resultat ?
   }
}
```

Passage de paramètres en Java

```
class App {
   /* Du code ici */
   public static void main(String args[]) {
      Voiture maCaisse = new Voiture("cabriolet");
      System.out.println(maCaisse); // resultat ?
      Usine1 u1 = new Usine1();
      u1.customizer(maCaisse);
      System.out.println(maCaisse); // resultat ?
   }
}
```

```
La voiture est une cabriolet
La voiture est une cabriolet
```

```
class App {
   /* Du code ici */
   public static void main(String args[]) {
      Voiture maCaisse = new Voiture("cabriolet");
      System.out.println(maCaisse); // resultat ?
      Usine2 u2 = new Usine2();
      u2.customizer(maCaisse);
      System.out.println(maCaisse); // resultat ?
   }
}
```

```
La voiture est une cabriolet
La voiture est une berline
```

Référence this

En Java, le mot clef `this` permet de référencer l'objet *courant*.

C'est l'**identité** de l'objet.

```
class Compte {

   private String nom, prenom;
   private double soldeCompte;

   public Compte(String n, String p) {
      nom = n;
      prenom = p;
      soldeCompte = 0;
   }

   public void rechargerCompte(double somme) {
      soldeCompte = somme;
   }

   // méthode spécifique Java pour l'affichage
   public String toString() {
      return "[" + prenom +
          ","+nom+","+soldeCompte +"]";
   }

}
```

```
class Compte {

   private String nom, prenom;
   private double soldeCompte;

   public Compte(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
      soldeCompte = 0;
   }

   public void rechargerCompte(double soldeCompte) {
      this.soldeCompte = soldeCompte;
   }

   // méthode spécifique Java pour l'affichage
   public String toString() {
      return "[" + prenom +
          ","+nom+","+soldeCompte +"]";
   }

}
```