

# Rappels sur les types en Java

Programmation objet - I

L2 informatique

F. Bertrand

# Au programme d'aujourd'hui...

- Rappels sur les types
  - Types primitifs et types références
  - Comparaison d'objets
  - Gestion de la mémoire
- Les énumérations
- Les boucles *for-each*
- La surcharge de méthodes

# Rappels sur les types en Java

- Il existe deux catégories de types :
  - Types dits *primitifs* (8) (ex. `int`, `char`, `boolean`...) ne nécessitant pas une allocation mémoire explicite...
  - Types *référence* permettant de manipuler des objets appartenant à des classes (ou des énumérations) ou à des tableaux. Ils nécessitent une allocation mémoire explicite (`new`)

# Les types primitifs

- Au nombre de 8 :

Type	Nature	Valeur
<code>boolean</code>	booléen	<code>true</code> , <code>false</code>
<code>char</code>	caractère Unicode (16 bits)	<code>[\u0000,\uffff]</code> $\cong$ <code>[0,65535]</code>
<code>byte</code>	entier signé (8 bits)	<code>[-128,+127]</code>
<code>short</code>	entier signé (16 bits)	<code>[-32768,+32767]</code>
<code>int</code>	entier signé (32 bits)	<code>[-2147483648, +2147483647]</code>
<code>long</code>	entier signé (64 bits)	<code>[-9223372036854775808, +9223372036854775807]</code>
<code>float</code>	réel (32 bits = 24 + 8)	*
<code>double</code>	réel (64 bits = 53 + 11)	*

\* = <http://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.2.3>

# Les types référence

- Une variable d'un type référence  $T$  peut contenir :
  - soit `null`
  - soit la référence (adresse) d'une instance (objet) de  $T$
- $T$  peut être :
  - une classe, ex : `class Essai { }`
  - une énumération :  
`enum CouleurFeu { ROUGE, ORANGE, VERT }`
  - un tableau, ex : `int[] t;`

# Représentation des différents types

```
int a = 2
```

Adresse	Contenu
0x1000	2
...	

```
String s = new String("abc");
```

Adresse	Contenu
0x1000	0x2000
...	
0x2000	<i>data</i>
0x2001	a
0x2002	b
0x2003	c

Représentation simplifiée en mémoire des différents types

## Conversion types primitifs ↔ types référence

- Parfois il peut être nécessaire de transformer une variable de type primitif en objet (type référence) équivalent et inversement...
- Pour cela il existe un ensemble de classes représentant les différents types primitifs existants :

`boolean` ↔ `Boolean`, `byte` ↔ `Byte`, `char` ↔ `Character`,  
`short` ↔ `Short`, `int` ↔ `Integer`, `long` ↔ `Long`,  
`float` ↔ `Float`, `double` ↔ `Double`.

## Conversion types primitifs ↔ types référence (suite)

- À partir de la version 1.5 de Java, leur transformation en objet peut s'effectuer automatiquement si nécessaire (*autoboxing*) :

```
int i = 5;  
ArrayList<Integer> l = new ArrayList<Integer>();  
l.add(i);           // interdit en version 1.4 :  
                    // l.add(new Integer(i))
```

La méthode add prend un objet Integer en paramètre

```
int j = l.get(0); // interdit en version 1.4 :  
                // int j = l.get(0).intValue()
```



# Caractéristiques communes

- Le qualificateur `final` peut être appliqué à différents éléments (variables, paramètres, méthodes, classes).
- Lorsqu'il est appliqué à des variables (locales ou attributs), il les transforme en constantes (non modifiables) :

```
final String s = "abc";  
s = "def"; // ERREUR à la compilation
```

# Caractéristiques communes (suite)

- Lorsqu'il est appliqué à des paramètres d'une méthode, il empêche leur affectation dans le code de cette méthode :

```
void f(final int val) {  
    val = 6; // Erreur à la compilation  
    val++;  // Erreur à la compilation  
}
```

- Pour rappel, en Java, les paramètres sont toujours passés par valeur et c'est une copie du paramètre qui est manipulée dans la méthode...

# Caractéristiques communes (fin)

- Ainsi si on retire le qualificateur `final`, la valeur passée en paramètre (type primitif ou type référence) ne sera pas modifiée :

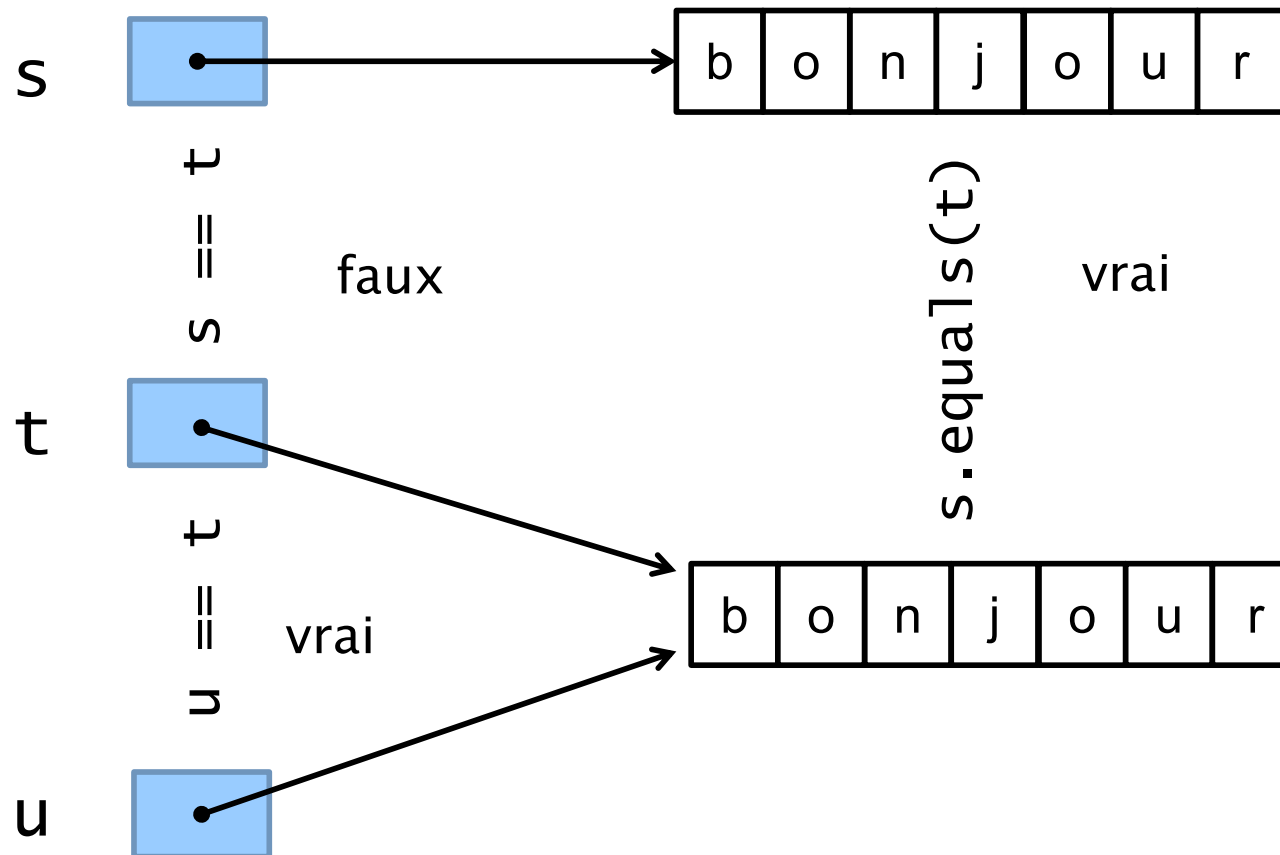
```
void f(int val, String s) {  
    val++;  
    s = "xyz";  
}  
int i = 5;  
String a = "abc";  
f(i,a);  
System.out.println("i = " + i + ", a = " + a);  
// affiche i = 5, a = abc
```

# Types référence : comparaison d'objets

- Avec les types primitifs, l'opérateur d'égalité (==) teste si deux valeurs sont **identiques**
- Avec les types référence, cet opérateur teste si deux références désignent le même objet (**même emplacement mémoire**) :

```
String s = "bonjour";  
String t = "bonjour";  
String u = t;  
if (s == t) // affichage ?  
    System.out.println("identité");
```

# Types référence : comparaison d'objets (suite)



## Types référence : comparaison d'objets (suite)

- Avec les types référence, si on souhaite tester l'**égalité** de deux objets, il est nécessaire d'utiliser la méthode `equals()` :

```
String s = "bonjour";  
String t = "bonjour";  
if (s.equals(t)) // affichage ?  
    System.out.println("égalité");
```

- Pour résumer :
  - **Identité** : opérateur "=="
  - **Égalité** : méthode `equals()`

## Types référence : comparaison d'objets (suite)

- Les classes de la bibliothèque Java possèdent généralement une méthode `equals` permettant de tester l'égalité de leurs instances (ex. `java.lang.String`, `java.awt.Point...`)
- Cependant pour les nouvelles classes créées, une version **par défaut** est fournie dont le **comportement est le même que celui de l'opérateur « == »**.
- Pour chaque nouvelle classe, il est donc important de créer une nouvelle version de la méthode `equals` sinon la version par défaut sera utilisée...

# Types référence : comparaison d'objets (suite)

- Exemple de définition de la méthode `equals` pour une classe représentant la notion de point dans un plan :

```
public class Point {  
    private double x,y;  
    public boolean equals(Point p) {  
        if (p == this) // si c'est le même objet  
            return true;  
        else  
            return (this.x == p.x && this.y == p.y);  
    }  
}
```



# Types référence : les objets immutables

- Certaines classes Java permettent de créer des objets non modifiables appelés **immutables**. Un exemple assez connu est la classe `String`.
- Cela a comme conséquence que l'opération de concaténation (+) **ne modifie pas un objet existant mais en crée un nouveau** à partir des deux opérandes:

```
String s1 = "bon";  
String s2 = "jour";  
String s3 = s1 + s2;  
// s1 n'est pas modifié par la concaténation
```

## Types référence : les objets immutables (suite)

- La classe `String` doit toujours être de préférence utilisée sauf s'il est nécessaire de concaténer un grand nombre de chaînes. La classe `StringBuilder` est alors plus performante.

```
StringBuilder sb = new StringBuilder();  
sb.append("Bonjour");
```

- Attention à ne pas confondre une chaîne vide avec l'absence de chaîne



`String s = "";`     $\neq$     `String s;`

## Types référence : allocation mémoire

- Une variable de type référence stocke l'adresse d'un objet (instance) qui a été alloué en mémoire.

```
Point p = new Point(1,2);
```

- Il n'est pas possible d'indiquer un emplacement mémoire particulier car la mémoire est gérée par la machine virtuelle.
- Java est conçu pour récupérer **automatiquement** la mémoire occupée par des objets « inutilisés »...

## Types référence : allocation mémoire (suite)

- **Exemple :**

```
Point p = new Point(1,2);  
p = new Point(2,3);  
// à ce stade le point (1,2) n'est plus référencé  
Point d = p;  
p = null;  
// ici le point (2,3) est toujours référencé (d)
```

- **Attention** : la mémoire d'un objet qui n'est plus référencé n'est pas immédiatement récupérée...
- Le processus qui s'occupe de la récupération mémoire des objets non référencés est appelé *collecteur de mémoire (garbage collector)*

## Types référence : allocation mémoire (suite)

- Cette gestion automatisée de la mémoire évite un certain nombre d'erreurs qui ne seront détectées qu'à l'exécution.
- Exemple d'erreur fréquent avec le langage C :

```
int *i = (int*) malloc (sizeof(int)); // allocation
int *j = i; // désignation du même espace mémoire
*i = 5;
free(i); // désallocation de la mémoire
*j = 2; // erreur à l'exécution
```

# Types référence : les énumérations

- Les types énumérés apportent beaucoup à la lisibilité et à la sûreté d'un programme
- Jusqu'à la version 1.4, Java ne permettait pas la définition de types énumérés. On utilisait généralement des constantes entières :

```
public static final int ROUGE = 0;  
public static final int ORANGE = 1;  
public static final int VERT = 2;  
public static final int BLEU = 10;
```

```
unFeuTricolore.changeCouleur(BLEU);  
// Aucun pb détecté !...
```

La méthode  
changeCouleur prend  
un int comme paramètre

## Types référence : les énumérations (suite)

- À partir de la version 1.5 Java a introduit la notion de type énuméré. L'exemple précédent revisité :

```
enum CouleurFeu { ROUGE, ORANGE, VERT; }  
class FeuTricolore {  
    private CouleurFeu couleur;  
    public void changeCouleur(CouleurFeu uneCouleur) {  
        this.couleur = uneCouleur;  
    }  
}  
...  
unFeuTricolore.changeCouleur(CouleurFeu.BLEU);  
// Erreur à la compilation car BLEU n'appartient pas  
// au type CouleurFeu !...  
unFeuTricolore.changeCouleur(CouleurFeu.VERT); // OK
```

# Types référence : les énumérations (suite)

- Il est possible d'accéder à l'ensemble des valeurs d'une énumération grâce à la méthode `values()` qui retourne un tableau contenant les différentes valeurs :

```
enum CouleurFeu { ROUGE, ORANGE, VERT; }  
class FeuTricolore {  
    public static void main(String[] args) {  
        for (CouleurFeu c : CouleurFeu.values()) {  
            System.out.println(c);  
        }  
    }  
}
```

Méthode fournie  
automatiquement  
pour une  
énumération

```
> java FeuTricolore  
ROUGE  
ORANGE  
VERT
```



# Types référence : les énumérations (suite)

- L'ordre des déclarations dans un enum est significatif, les éléments sont indexés par un entier débutant à 0. Cet index peut être récupéré via la méthode `ordinal()`.

```
public class FeuTricolore {  
    public static void main(String[] args) {  
        for (CouleurFeu c : CouleurFeu.values()) {  
            System.out.println(c + " " + c.ordinal());  
        }  
    }  
}
```

Méthode fournie  
automatiquement pour  
une énumération

```
> java FeuTricolore  
ROUGE 0  
ORANGE 1  
VERT 2
```

## Types référence : les énumérations (suite)

- En fait, une énumération est une classe avec des instances prédéfinies (pas d'ajout possible) et peut donc posséder attributs, constructeurs et méthodes :

```
public enum ErreurFichier {  
    FichierNonTrouve(-1), LectureImpossible(-2),  
    FichierDejaExistant(-4);  
    private int code; // attribut  
    // constructeur (non accessible)  
    ErreurFichier(int val) { this.code = val; }  
    // méthode  
    public int valCode() { return this.code; }  
}
```

# Parcours d'un ensemble d'éléments : les boucles *for-each*

- À partir de la version 1.5 de Java, le parcours de tableaux et de listes **en lecture** peut s'écrire de manière beaucoup simple...
- Ainsi le code suivant :

```
// Soit tabInt un tableau d'entier  
for(int i=0; i < tabInt.length; i++)  
    System.out.println(tabInt[i]);
```

- Peut s'écrire de la manière suivante :

```
for(int elt : tabInt)  
    System.out.println(elt);
```

Avantage : pas de  
gestion d'index !...

# Parcours d'un ensemble d'éléments : les boucles *for-each* (suite)

- En fait **l'index est géré automatiquement** mais en revanche **il n'apparaît plus** et ce type de boucle ne peut donc accéder à un ensemble d'éléments **uniquement en lecture** et aucune modification (ajout, suppression) n'est possible sur l'ensemble parcouru...

- La syntaxe est :

```
for(TypeElementDeListe nomElt : nomEnsemble) { ... }
```

- `nomEnsemble` peut être soit un tableau, soit une liste (en fait une collection au sens large)

# Parcours d'un ensemble d'éléments : les boucles *for-each* (suite)

- Cette syntaxe offre encore plus d'intérêt sur les listes parcourues à l'aide d'un index.
- Ainsi le code suivant :

Code peu efficace car `size()` est appelée à chaque itération

```
void affiche(ArrayList<String> mots) {  
    for(int i = 0; i < mots.size(); i++)  
        System.out.println(mots.get(i));  
}
```

- Peut s'écrire de la manière suivante :

```
void affiche(ArrayList<String> mots) {  
    for(String m : mots)  
        System.out.println(m);  
}
```

# La surcharge de méthodes

- Avec Java, il est possible de définir plusieurs méthodes ayant le même nom à **condition** que celles-ci aient des **paramètres différents**.
- Cette propriété s'appelle **surcharge de méthode** (*overloading*).
- Exemple :

```
public class Ecran {  
    void affiche(char c) { }  
    void affiche(int i) { }  
}
```

# La surcharge de méthodes (suite)

- Le choix de la méthode appelée est fait à la **compilation** en comparant les types des arguments réels avec ceux des arguments formels.

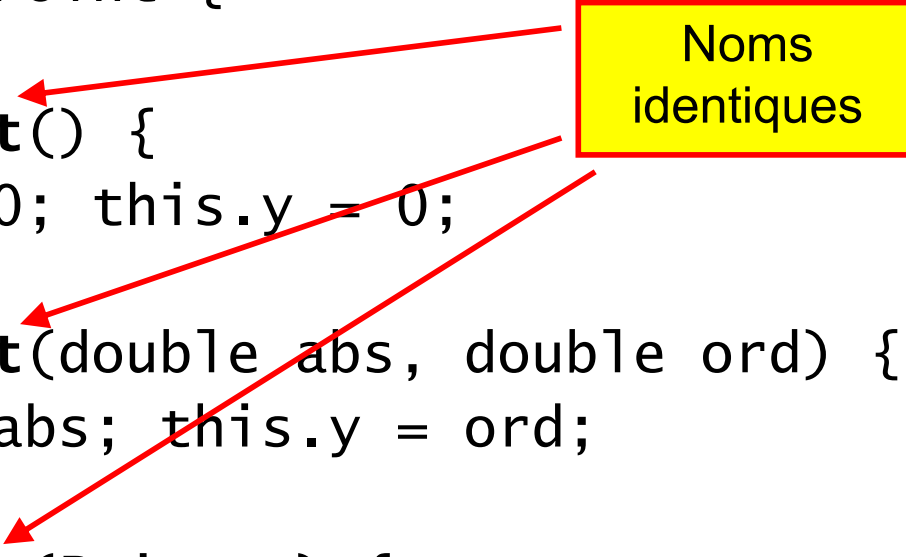
```
Ecran e = new Ecran();  
e.affiche('c');  
e.affiche(12);
```

- La surcharge sert généralement à conserver la **sémantique** (le sens) d'une méthode définie pour un type de paramètre mais à l'appliquer à d'autres types.

# La surcharge de méthodes (suite)

- Les constructeurs sont de bons exemples pour l'utilisation de la surcharge :

```
public class Point {  
    ...  
    public Point() {  
        this.x = 0; this.y = 0;  
    }  
    public Point(double abs, double ord) {  
        this.x = abs; this.y = ord;  
    }  
    public Point(Point p) {  
        this.x = p.x; this.y = p.y;  
    }  
}
```



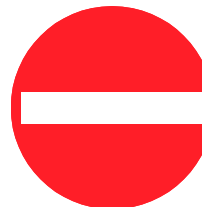
Noms identiques



# La surcharge de méthodes (suite)

- Contraintes :
  - La surcharge ne peut s'effectuer uniquement en attribuant des types de retour différents

```
classe Test {  
    int m() { ... }  
    char m() { ... }  
}
```



- En cas d'ambiguïté (plusieurs méthodes possibles), une erreur sera générée à la compilation...

## Pour résumer...

- Bien distinguer types primitifs/types références
- Attention à la différence entre `==` et `equals()`
- Penser à utiliser des types énumérés (augmente la lisibilité des programmes)
- Utiliser de préférence des boucles `for-each` lorsqu'on accède à un ensemble d'éléments sans le modifier...
- Si une méthode a besoin de prendre des paramètres différents, utiliser la surcharge plutôt que de créer plusieurs versions avec différents noms.