2i002 - Programmation par objets en Java

Christophe Marsala

(support réalisé par Vincent Guigue)

Cours 3

Programme du jour

Informations

- Christophe Marsala (email : `Christophe.Marsala@lip6.fr`)
- Page web de l'UE sur le site de la licence :

https ://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2016/ue/2I002-2017fev/

Plan du cours

String (suite)

2 choses à retenir sur les `String`

1. **Les chaînes sont immutables** : modifier une chaîne existante est impossible, il faut créer une nouvelle chaîne qui est une modification de l'ancienne. Cela rend la classe peu efficiente dans certain cas... Et il faut alors se tourner vers des objets plus évolués (StringBuffer notamment)
2. **Ne pas utiliser `==` avec les String** mais toujours la méthode .equals. Les deux versions compilent mais la première donnera régulièrement des résultats faux (que nous expliquerons plus tard).

```
String s1 = "Leia";
String s2 = "Luke";
if( s1.equals(s2) )
    System.out.println("les chaînes sont identiques");
else
    System.out.println("les chaînes sont différentes");
```


Classes enveloppes

Les types de base en JAVA sont doublés de wrappers ou classes enveloppes pour :

- utiliser les classes génériques (cf cours `ArrayList`)
- fournir quelques outils fort utiles

`int, double, boolean, char, byte, short, long, float` ⇒ `Integer, Double...`

Outils

```
Double d1 = MAX_VALUE; // valeur maximum possible
Double d2 = Double.POSITIVE_INFINITY; // valeur spécifique
                                      // gérée dans les opérations
Double d3  = Double.valueof("3.5"); // String => double
// Double.isNaN(double d), Double.isInfinite(double d)...
```

Documentation : http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html

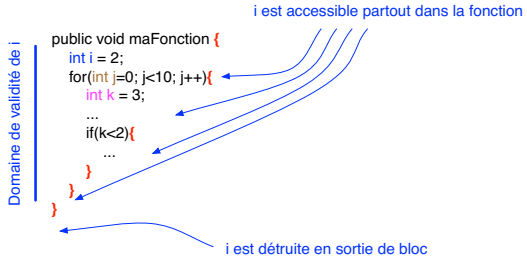
```
// conversions implicites = (un)boxing (depuis JAVA 5)
double d4 = d1;
Double d5 = d4;
```

Durée de vie

Logique de bloc

- une fonction est un bloc,
- une boucle ou une conditionnelle forme également un bloc,
- les blocs sont repérés par des accolades : **{...}**

Les variables **déclarées** dans un bloc sont détruites en sortant du bloc.



Conversions entre types

Java, un langage typé

Les types sont très importants en Java : le compilateur vérifie toujours les types des différentes variables

- Certaines conversions sont implicites

```
double d = 42; double d2 = i; // avec i un int existant
```

Tout type de base peut se convertir en String

```
String s = "mon message "+1.5+" "+d;
```

- Certaines conversions doivent être données explicitement

```
int i = (int) 2.4;
```

Perte d'information liée à la conversion ; Java ne tolère pas la conversion implicitement, il faut que le programmeur la demande explicitement (pour être sûr que la perte d'information est souhaitée).

- Conversions impossibles

Opérateurs classiques (par ordre de priorité)

opérateurs postfixés	[ ] . expr++ expr--
opérateurs unaires	++expr --expr +expr -expr ~ !
création ou cast	new (type) expr
opérateurs multiplicatifs	* / %
opérateurs additifs	+ -
décalages	<< >> >>>
opérateurs relationnels	< > <= >=
opérateurs d'égalité	== !=
et bit à bit	&
ou exclusif bit à bit	^
ou (inclusif) bit à bit	|
et logique	&&
ou logique	||
opérateur conditionnel	? :
affectations	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Conditionnelles

- Syntaxe de l'alternative

```
int i=11;
if(i > 38){
    // code à effectuer dans ce cas
}
else{ // le else est facultatif
    // Code à effectuer sinon
}
```

- En cas de clauses multiples :

```
switch(i){
case 1:
  // Code à effectuer si i == 1
  break; // sinon le reste du code est AUSSI effectué
case 2: //
   // Code à effectuer si i == 2
  break;
default : // Si on n'est passé nulle part ailleurs
}
```


Structures itératives

Même définition des boucles qu'en C/C++

- Syntaxes : 2 options (principales)
  Pour i allant de 0 à 9, faire...

```
int i;
for(i=0; i<10; i++){// i prend les valeurs 0 à 9
                    // ==> 10 itérations
// code a effectuer 10 fois
}
```

Tant que i inférieur à 10, faire...

```
int i = 0;
while(i<10){// i prend les valeurs 0 à 9 =
                    // 10 itérations
    // code a effectuer 10 fois
    i++; // ne pas oublier, sinon boucle infinie !
}
```

- D'autres syntaxes sont possibles : do...while etc...

Interruptions de fonctions/boucles (1/3)

Trois types d'interruptions de boucles

- return : l'interruption **la plus forte**. Retour anticipé de la fonction (**sort de la fonction**, pas seulement de la boucle).

```
// le modulo par 5 peut-il retourner un entier >=5?
public void maFonction(){
    for(int i=0; i<10; i++){
        if(i%5>4){
            System.out.println("C'est très étrange");
            return;
        }
    }
    System.out.println("l'opération modulo 5 retourne"+
       "toujours un entier inférieur à 5");
}
```

Interruptions de fonctions/boucles (2/3)

3 types d'interruptions de boucles

- return
- break : sortie anticipée de la boucle

```
1 // 6 fait-il parti des multiples de 2?
2 public void maFonction(){
3     boolean found = true;
4     for(int i=0; i<10; i++){
5         if(i * 2 == 6){
6             found  = true;
7             break; // pas besoin d'aller plus loin
8         }
9     }
10    if(found)
11        System.out.println("6 est un des multiples de 2");
12 }
```


Interruptions de fonctions/boucles (3/3)

3 types d'interruptions de boucles

- return
- break
- continue : passer à l'itération suivante

```
1 // afficher 3./i pour i variant de -10 à 10
2 // il faut penser à sauter le cas 0 qui provoque un problème
3 public void maFonction(){
4     for(int i=-10; i<=10; i++){// -10 et 10 inclus
5         if(i == 0)
6             continue;
7         System.out.println("3./"+i+" = "+(3./i) );
8     }
9 }
```

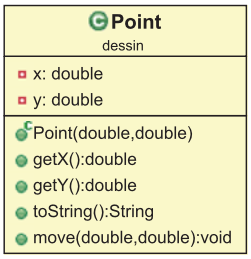
Ces instructions rendent le code plus lisible en limitant notamment le nombre de blocs imbriqués.

Plan du cours

Philosophie

On ne programme pas pour soi-même... Mais pour les autres :

- Respecter les codes syntaxiques : majuscules, minuscules...
- Donner des noms explicites (classes, méthodes, attributs)
- Développer une documentation du code (cf cours javadoc)
- ... Et proposer une vision synthétique d'un ensemble de classes : ⇒ **UML** : Unified Modeling Language



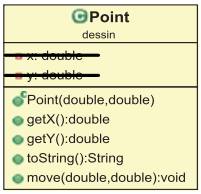
- nom de la classe
- attributs
- méthodes (et constructeurs)

\+ code pour visualiser `public`/`private`
\+ liens entre classes pour les dépendances (cf cours sur la composition)

UML Client *vs* Développeur

Plusieurs types de diagrammes pour plusieurs usages :

- Vue développeurs : représentation complète
- Vue clients : représentation `public` uniquement



Idée :

Le code doit être pensé pour les autres :

- tous les noms doivent être aussi clairs que possible
- un diagramme plus limité est plus facile à lire

Philosophie (suite)

Deux manières de voir l'UML :

1. Outil pour une visualisation globale d'un code complexe
2. Outil de conception / développement indépendant du langage

Dans le cadre de 2i002 : seulement l'approche 1

Limites de l'UML :

- Vision architecte...
- Mais pas d'analyse de l'exécution du code

Que se passe-t-il lors de l'exécution du programme :

Nouveau type de représentation :

Diagramme mémoire

Coté JVM : exécution du code

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

Comment décrire cette ligne de code ?

La variable p, de type Point, référence une instance dont les attributs x et y ont pour valeur respectivement 1 et 2.

Comment représenter cette ligne de code ?

- Représentation des classes sans les méthodes
- Valeurs des attributs
- Types & noms des variables
- Liens de référencement

Types de base *vs* Objet : signification de =

Les types de base et les objets ne se comportent pas de la même façon avec =

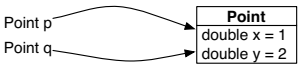
- Liste des types de base :
  `int, double, boolean, char, byte, short, long, float`

```
1 double a, b;
2 a = 1;
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

- et pour un Objet :

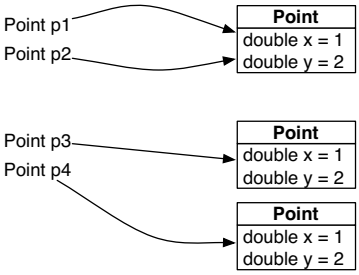
```
4 Point p = new Point(1,2);
5 Point q = p; // duplication de la référence...
6             // 1 seule instance !
```



2 variables, 2 références, mais 1 seule instance

Clonage *vs* copie de surface

```
1 public static void main(String[] args) {
2   Point p1 = new Point(1, 2);
3   Point p2 = p1;
4
5   Point p3 = new Point(1, 2);
6   Point p4 = new Point(1, 2);
7 }
```



- Les variables p1 et p2 référencent la même instance
- p3 et p4 référencent des instances différentes

Références & arguments de fonctions

- Passer un argument à une fonction revient à utiliser un signe =
- ... Objets et types de base se comportent différemment !

```
1 // classe UnObjet,
2 // (classe sans importance)
3 ...
4 public void maFonction1(Point p){
5   ...
6   p.move(1., 1.);
7   ...
8 }
9
10 public void maFonction2(double d){
11   ...
12   d = 3.; // syntaxe correcte
13           // mais très moche !
14   ...
15 }
```

```
1 // dans le main
2 UnObjet obj = new UnObjet();
3
4 Point p = new Point(1.,2.);
5 double d = 2.;
6
7 obj.maFonction1(p);
8 obj.maFonction2(d);
9
10 // p a pour attributs (x=2.,y=3.)
11 // d vaut 2
```

- Quand un objet est passé en argument :
  il n'y a pas duplication de l'instance (simplement 2 références vers 1 instance)
- Quand un type de base est passé en argument : duplication.

Types de base *vs* Objet : signification de ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un `boolean`
- Type de base : vérification de l'égalité **des valeurs**
- Objet : vérification de l'égalité **des références**
- ⚠ ATTENTION aux classes enveloppes (qui sont des objets)

```
1   double d1 = 1.;
2   double d2 = 1.;
3   System.out.println(d1==d2); // affichage de true
4                               //dans la console
5   Point p1 = new Point(1, 2);
6   Point p2 = p1;
7   System.out.println(p1==p2); // affichage de true
8
9   Point p3 = new Point(1, 2);
10  Point p4 = new Point(1, 2);
11  System.out.println(p3==p4); // affichage de false
12  Double d3  = 1.; // classe enveloppe Double = objet
13  Double d4 = 1.;
14  System.out.println(d3==d4); // affichage de false
```


Référence null

Que se passe-t-il quand on déclare une variable (sans l'instancier) ?

```
1 Point p;
```

- p vaut `null`.
- On peut écrire de manière équivalente

```
1 Point p = null;
```

- On ne peut pas invoquer de méthode

```
1 p.move(1., 2.); // => CRASH de l'exécution:
2                 // NullPointerException
```

- N'importe quel objet peut être `null` et réciproquement, on peut donner `null` à n'importe quel endroit où un objet est attendu... Même si ça provoque parfois des crashs.

```
1 // classe UnObjet,
2 // (classe sans importance)
3 ...
4 public void maFonction(Point p){
5   ...
6   p.move(1., 1.);
7   ...
8 }
```

```
1 // dans le main
2 UnObjet obj = new UnObjet();
3
4 obj.maFonction(null);
```