LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 1 – 16 septembre 2022

Programme du jour

Informations

- Christophe Marsala (email : `Christophe.Marsala@lip6.fr`)
- Site Moodle de l'UE : LU2IN002 - S1-22
  - planning et salles de TD/TME (page licence)
  - support de cours
  - version PDF du recueil d'exercices (ne pas imprimer ! !)
  - quizzes d'auto-évaluation en ligne
  - forum pour poser des questions
  - rendu des devoirs (TME, projet)

Fonctionnement

- Organisation de l'UE
  - 1h45 Cours
  - 1h45 TD : connaître le cours *n* avant d'aller en TD *n*
  - 1h45 TME (en binôme)
- Evaluation
  - contrôle de mi-semestre (semaine du 7 novembre) = 25%
  - TME solo (séance 9) = 15%
  - mini-projet (rendu & soutenance lors du dernier TME) = 10%
  - examen final (janvier) = 50 % de la note finale

Rem : aucun document n'est autorisé lors des épreuves

Plan du cours

Java : le choix d'une architecture dynamique

Le langage Java

- langage moderne qui puise son inspiration de sources diverses
  - syntaxe très proche du C/C++
  - architecture dynamique avec un compilateur et une machine virtuelle JVM qui exécute le code Java compilé
- évolution régulière
  - composants additionnels et fonctionnalités
  - la syntaxe n'a, elle, que peu évolué
- trés utilisé dans l'industrie
  - API (Application Programming Interface) de programmation pour de multiples applications
  - rapidité d'exécution

Le langage Java

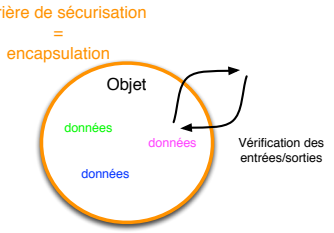
- Langage orienté objet, syntaxe simple, traits impératifs
- Langage robuste et sûr
- Pour créer des applications indépendantes de la machine
- Langage compilé pour créer des applications performantes

Philosophie Objet

Pourquoi faire de la programmation objet ?

- Pour développer des systèmes complexes... *Sans se planter*

- **diviser** le système complexe en une multitude de systèmes simples : les objets
- **sécuriser** l'accès aux données sensibles

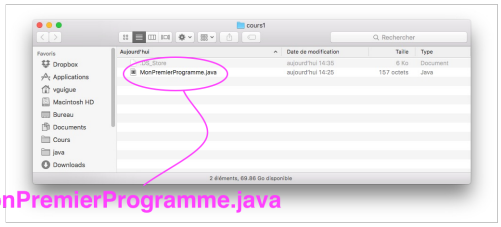


- [corollaire] Travailler à plusieurs... *Avec un minimum de bugs*
  - toujours penser son programme pour les autres : **sécuriser, simplifier, compartimenter**
  - double vision : fournisseur / client

Plan du cours

Une classe, un main()

1. En Java, tout code doit être encapsulé dans une classe
   - **1 classe** est mise dans **1 fichier** du même nom que la classe
   - écriture du fichier : éditeur de texte (gedit, emacs,...)
   - règle : les noms de classe commencent par une majuscule



2. Programme principal = un point d'entrée dans un système avec de nombreuses classes
   - ce programme est exécutable après compilation

Une classe, un main : syntaxe

La syntaxe des **signatures de classe** et de la **signature du main** est à apprendre par coeur (explications dans les cours suivants).

1. Signature d'une classe
   - `public`
   - `class`
   - suivi du nom de la classe
2. Signature d'un main
   - `public static void`
   - `main`
   - toujours le même argument `String[] args`
3. Puis les instructions dans le main
   - par exemple, affichage de la chaîne `"Bonjour !"`

```
1 // dans le fichier MonPremierProgramme.java
2 public class MonPremierProgramme{
3         public static void main(String[] args) {
4             System.out.println("Bonjour !");
5     }
6 }
```


Compilation/Exécution

- Pré-requis :
  - JDK (Java Dev. Kit) installé sur la machine
  - être dans le bon répertoire (!)

1. Compilation
   - vérification de la syntaxe, droits d'accès...
   - création d'un exécutable en bytecode : `MonPremierProgramme.class`

```
»javac MonPremierProgramme.java
```

2. Exécution
   - exécution dans la console

```
»java MonPremierProgramme (pas d'extension)
```

⇒ résultat :

```
» Bonjour !
```

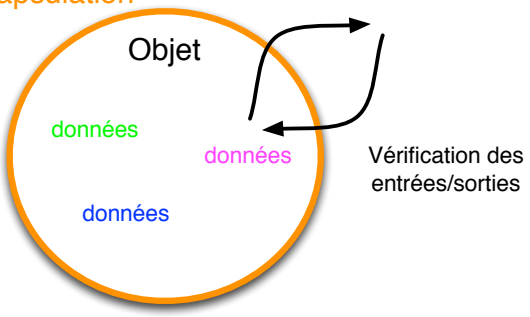
Plan du cours

Approche orientée-objet

- Diviser un programme complexe en **objets**
  - objet autonome : réutilisable dans plusieurs projets
    - Vecteurs, Personne, DisplaySimulation,...
  - objet sécurisé : garantie de bon usage par d'autre
    - un objet intègre des données et des méthodes pour les manipuler proprement,
    - les transactions bancaires sont journalisées, les éléments d'une simulation physique ne se téléportent pas...
  - objet simple et intuitif
- Enjeux
  - réfléchir en amont au découpage et à la sécurisation
  - documenter le code (en premier lieu, respecter les conventions pour faciliter la compréhension)

Approche orientée-objet

Barrière de sécurisation
=
encapsulation



Barrière de sécurisation

Syntaxe : signature et attributs

1. Fichier : 1 classe correspond à 1 fichier
   - nom de la classe + .java = nom du fichier
   - marquer l'encapsulation, faciliter la ré-utilisation
   - le nom de classe commence par une majuscule
2. Signature : public class
3. Déclaration des classes attributs, variables d'instance, champs
   - répondre à : De quoi est composé notre objet ?
   - les attributs sont presque toujours private (cf plus loin)
   - nom des attributs en minuscules
4. Définir des méthodes
   - comment construire un objet ? on définit un constructeur
   - quelles opérations effectuer sur l'objet ?
   - nom des méthodes en minuscules

```java
// Création du fichier Point.java
public class Point{ // classe publique
    private double x,y; // attributs privés
```


Syntaxe : constructeur

Comment construire un `Point` ?
⚠ Attention à considérer le problème des 2 points de vues

Fournisseur

- Besoin : 2 coordonnées fournies en argument
- Action : initialisation des valeurs des attributs

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
}
```

Client

ie : utilisateur d'une classe Point existante...

- Besoin : créer une instance dans la mémoire avec les bonnes valeurs

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p = new Point(2., 3.1);
  }
}
```

Un constructeur porte le nom de la classe et il ne rend rien !

Création d'objets

```java
Point p1 = new Point(1,2);
```

Description de cette ligne de code (terminologie)

À l'issue de l'exécution de cette ligne, la variable p1, de type Point, contient la référence d'un objet, instance de la classe Point, dont les attributs x et y ont pour valeur respectivement 1 et 2.

Représentation mémoire à l'issue de l'exécution

On indique :

- la classe sans les méthodes
- les valeurs des attributs prises pour l'objet correspondant
- le type et le nom des variables
- le lien de référencement (flèche)

Créer un objet : instanciation

Coté fournisseur :
mise en route de l'objet

Constructeur = contrat d'initialisation des attributs

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2,double y2){
    x = x2;
    y = y2;
  }
}
```

Coté client :
création d'une instance de classe

Instanciation = création d'une zone mémoire réservée à l'objet

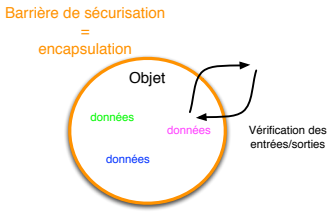
```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p1 = new Point(1., 2.);
  }
}
```

La variable p1, de type Point, référence une instance de la classe Point dont les attributs ont pour valeur 1 et 2.

Syntaxe : méthodes

Comment manipuler un Point ?

1. **définir des méthodes**
   eg : accéder aux attributs (en lecture)
2. les **invoquer** depuis l'extérieur



Fournisseur

```java
//Fichier Point.java
public class Point{
 private double x,y;
 public Point(double x2, double y2){
    x = x2;
    y = y2;
 }

 public double getX(){
    return x;
 }

}
```

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p = new Point(2., 3.1);

    double px = p.getX();

    System.out.println(
            "coord x de p:"+px);
  }
}
```


Autorisations d'accès

- `public` : accessible / visible depuis l'extérieur de l'objet (eg : un main, un autre objet...)
- `private` : protégé / invisible depuis l'extérieur de l'objet

- Les constructeurs sont en général `public`
  - ils ont vocation à être appelés depuis l'extérieur
- Les attributs sont en général `private`
  - ils sont protégés et non accessibles depuis l'extérieur

Fournisseur

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
}
```

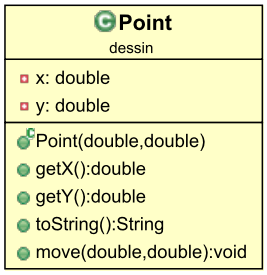
Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // opération autorisée
    Point p = new Point(2., 3.1);

   // opération impossible :
   // ERREUR DE COMPILATION
   double d = p.x;
  }
}
```


Représentation UML

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



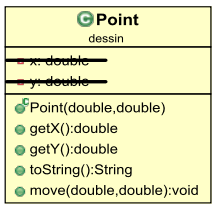
- nom de la classe
- attributs
- méthodes (et constructeurs)

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

UML Client *vs* Fournisseur

Plusieurs types de diagrammes pour plusieurs usages :

- Vue fournisseur : représentation complète
- Vue client : 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

Représentation UML (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 LU2IN002 : seulement l'approche 1

Limites de l'UML :

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

Reflexion sur la syntaxe objet

Exemple type : addition de 2 `Point`

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p  = new Point(2., 3.1);
    Point p2 = new Point(0.5, 1);

    Point p3 = p.add(p2);
  }
}
```

Fournisseur

```java
//Fichier Point.java
public class Point{
 private double x,y;
 public Point(double x2, double y2){
    x = x2;
    y = y2;
 }

 public Point add(Point p){
   return new Point(x+p.x, y+p.y);
 }
```

Syntaxe objet = il faut penser objet... Pas évident au début !

Syntaxe : redéfinition de méthodes standards

- Des méthodes standards existent et sont utilisables pour tous les objets (cf cours héritage)...
  - mais avec un comportement pas toujours satisfaisant
- Exemple : conversion d'un objet en chaîne de caractères : `public String toString()`

Fournisseur

```java
//Fichier Point.java
public class Point{
 ...
 public String toString(){
    // redéfinition
    return "["+ x +","+ y +"]";
 }
}
```

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p = new Point(2., 3.1);

    String str = p.toString();

    System.out.println("p:" + str);
  }
}
```

```
» p: Point@8764152
```

```
» p: [2, 3.1]
```


Compilation/exécution

Nous avions vu précédemment comment compiler et executer UNE classe... Comment faire maintenant qu'il y en a plusieurs ?

```
» javac Point.java
» javac TestPoint.java
» java TestPoint
```

Syntaxe réduite :

```
» javac Point.java TestPoint.java OU javac *.java
» java TestPoint
```

Remarque : il peut y avoir plusieurs `main` (mais pas plus de 1 par classe)

- Compilation de tous les `main` d'un coup
- Execution d'un seul (appel à la classe correspondante)

Plan du cours

Types des variables de base en Java

- Entier, réel, booléen, caractère : ces types sont disponibles de base en Java avec les opérateurs les plus courants.
  `int, double, boolean, char, byte, short, long, float`
  ⚠ La plupart des types et syntaxes associées sont comparables au C/C++... Sauf le booléen.

Le booléen vaut `true`/`false` et n'est pas convertible en entier

- Déclaration

```java
int i; // déclaration de i
System.out.println(i); // => 0
double d = 2.6;
boolean b = true; // ou false
char c = 'a';
```

```java
// operations de base: + - / * ...
int j = i+2;
int k = 1/2; //=0 Attention a la division entiere
```


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	`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=` `>>>=`

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

```
int i = (int) true; // conversion impossible des booléens
```


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

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


Interruptions de fonctions/boucles (3/3)

3 types d'interruptions de boucles

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

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

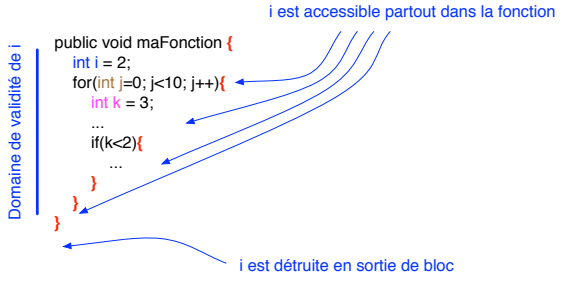
Ces instructions rendent le code plus lisible en limitant notamment le nombre de blocs imbriqués.

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.



Classe String

Gestion des chaînes de caractères

String n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à Java et son caractère immutable la rapproche très nettement d'un type de base.

```
String s = "Luke"; // création d'une chaîne de caractères
s = s + " est le frère de Leia";
System.out.println(s); // affichage de s dans la console
```

⚠ Ne pas confondre l'**objet** `String` et l'affichage dans la console.

Les **possibilités sont nombreuses** : extraction de sous-chaînes (substring), division en plusieurs chaînes (split), recherche de caractères, construction de nouvelles chaînes à partir d'expressions régulières (replace)... Toute la documentation sur : `http://docs.oracle.com/javase/7/docs/api/java/lang/String.html`

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");
```


Exemples de formatages

```
double[] tab = new double[10];
for(int i=0; i<10; i++)
    tab[i] = Math.random()*1000;
for(int i=0; i<10; i++)
    System.out.println(tab[i]);
```

```
510.4306229034564
775.6503067597263
15.528224029893511
...
```

```
for(int i=0; i<10; i++)
    System.out.println(
      String.format("%12f", tab[i]));
```

```
  510,430623
  775,650307
  15,528224
...
```

```
for(int i=0; i<10; i++)
    System.out.println(
      String.format("%10.3f", tab[i]));
```

```
  510,431
  775,650
   15,528
...
```

```
for(int i=0; i<10; i++)
    System.out.println(
      String.format("%010.3f", tab[i]));
```

```
000510,431
000775,650
000015,528
...
```

Ca marche aussi avec les entiers (%d) et les String (%s)