Bases de la programmation orientée objet

Antoine Nongaillard

But Info – IUT – Université de Lille

R2.01 – Séance 01

Plan

Objectifs du premier semestre

Ressources `R1.01` - `S1.01` - `S1.02`

- Maîtriser les concepts fondamentaux de la programmation
- Déterminer les éléments pertinents d'un problème
- Savoir analyser et décomposer un problème
- ...

En d'autres termes :

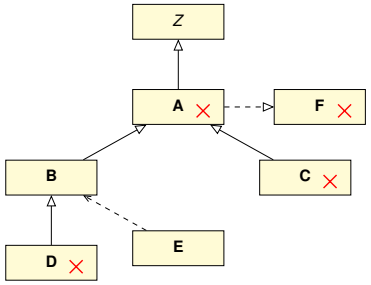
- décomposition d'une fonctionnalité en sous-fonction
- décomposition d'une fonction en séquence d'instructions combinée à des structures de contrôle comme les boucles (`for`, `while`) et des structures conditionnelles (`if`)

Objectifs du second semestre

Ressources `R2.01` - `R2.03` - `S2.01` + `S2.02`

- Maîtriser les concepts fondamentaux de la programmation orientée objet (encapsulation, composition, polymorphisme, héritage...)
- Lecture d'une conception orientée objet détaillée en UML (*Unified Modeling Language*)

La différence : la décomposition d'une fonctionnalité en instructions se combine aux mécanismes des langages orienté objet.



Une nouvelle fonctionnalité peut nécessiter quelques lignes de code dans différentes classes.

Plan

R1.X : programmation impérative

En `iJava`, une structure est formée de plusieurs champs éventuellement hétérogènes dans un premier fichier...

```
class Concert {
    String artist;
    int year, month, day;
}
```

... et l'écriture des fonctions permettant de manipuler ce type dans un autre fichier

```
if(c1.year<c2.year) return true;
return c.artist;
return "" + c.artist + ":" + c.year + "-" + c.month + "-" + c.day;
```

L'écriture impérative est indépendante de tout contexte. Tous les éléments requis doivent être pourvus en paramètres.

Exemple : les dates

On veut représenter plusieurs dates dans un programme. Comment faire ?

- Tableaux d'entiers `int day; int month; int year;`
- Tableau d'entiers à plusieurs dimensions `int dates[][];`
- Tableau de chaînes de caractères `String[] dates;`
- ...

Que choisir ? Comment garantir la cohérence entre données et traitements ?

- Les traitements sont spécifiques : calcul de la date du lendemain, du temps écoulé entre deux dates, validité d'une date...
- De nombreux cas sont particuliers existent : nombre de jours dans un mois, années bissextiles, nom des mois...

Est-ce à l'*utilisateur* de s'en préoccuper ?

R2.X : programmation orientée objet

Dans un **objet**, les traitements applicables à une structure `iJava` sont intégrés.

```
class Concert {
    String artist;
    int year, month, day;
    // ...
    boolean isBefore(Concert c) { // ...
    String getArtist() { // ...
    void printConcert() { // ...
```

Les traitements se rapportent alors à l'objet *courant*. Un des paramètres est implicite puisqu'il correspond à l'objet sur lequel est invoqué la méthode. Il définit un contexte d'exécution.

Différence d'utilisation

En iJava :

```
void algorithm() {
    Concert c1, c2;
    // ...
    println(isBefore(c1, c2));
    println("My favorite artist is "+ getArtist(c1));
    // ...
```

En Java :

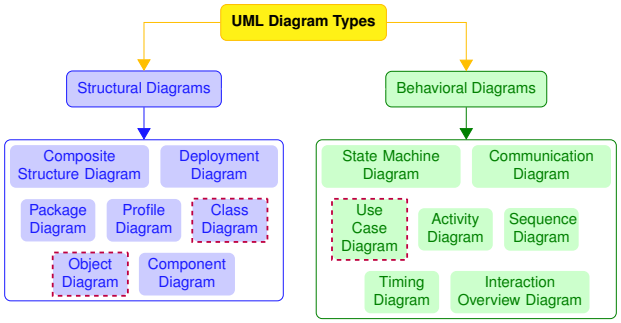
```
public static void main(String[] args) {
    Concert c1, c2;
    // ...
    System.out.println(c1.isBefore(c2));
    System.out.println("My favorite artist is " + c1.getArtist());
    // ...
```

On notera :

- le lancement du programme par la méthode : `public static void main(String[] args)`
- la méthode d'affichage d'une chaîne de caractères : `System.out.println(...)`

Unified Modelling Language

UML : Représentation graphique des éléments d'une application
Adopté en 1997 par *Object Management Group*, publié en 2005 comme standard *ISO*



Les classes

Une **classe** est un "type objet", et définit :

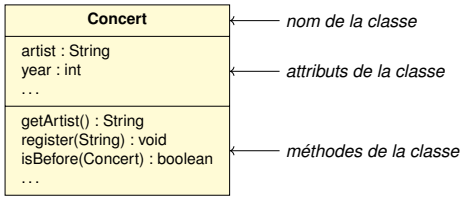
- la liste des **méthodes** définit le **comportement** de l'objet

  Une **méthode** est une fonction qui appartient à une classe et qui ne peut être **invoquée** que par les instances de cette classe : `isBefore(c1,c2)` $\neq$ `c1.isBefore(c2)`

- la liste des **attributs** définit la structure d'un objet

  Un **attribut** est une variable qui appartient à une classe et qui possède nécessairement un type. L'ensemble des valeurs d'attributs décrit l'**état** de l'objet.

Notation UML : représentation d'une classe dans un diagramme de classe



Plan

Références et mécanismes

Pour créer un objet à partir d'une classe, il faut **instancier** la classe. L'opérateur **`new`** permet de construire des objets (créer des instances de classes) par l'appel d'un **constructeur** :

```
Concert crt = new Concert();
```

La **notation pointée** donne accès aux attributs d'une instance et permet d'invoquer ses méthodes :

```
crt.artist = "bb42";             // assign a value
int aDay = crt.day;              // get a value
String aName = crt.getArtist();  // invoke a method
```

Le type de la référence définit ce qui est accessible. La validité n'est vérifiée qu'à la compilation (*late binding*).

En java, le mot-clé **`this`** est une **autoréférence** : il représente l'objet qui invoque la méthode et est toujours défini dans le contexte d'exécution d'une méthode.

Constructeurs et instanciation

Un **constructeur** est une méthode spéciale qui définit ce qui doit être fait pour créer une instance à partir d'une classe donnée, à savoir réserver l'espace mémoire pour les attributs de l'objet et leur affecte les valeurs initiales :

- il porte le **nom de la classe**
- il ne comporte pas de **type de retour**
- il est invoqué par le mot-clef **`new`**

En Java, **si** une classe ne définit pas un constructeur, alors il y a un constructeur par défaut (sans paramètre) ⇒ **ce dernier n'existe que si aucun constructeur n'est déclaré.**

Exemple d'invocation :

```
Book b0 = new Book();
Book b1 = new Book("Flatland", "Edwin A. Abbott");
Book b2 = new Book("The Hitchhiker's Guide to the Galaxy");
```


Un exemple

```
class Book {
  // attributes
  String author;
  String title;
  int publicationYear;
  String text;

  // constructor
  Book(String a, String t, int y, String contenu) {
    this.author = a;
    this.title = t;
    this.publicationYear = y;
    this.text = contenu;
  }

  // methods
  void printContent() {System.out.println(this.text);}
  String getAuthor() {return this.author;}
  String getTitle() {return this.title;}
  // ...
```


Initialisation des attributs

Par défaut, les attributs sont initialisés :

- à `0` pour les numériques
- à `\0` pour les caractères,
- à `false` pour les booléens,
- et à `null` pour les chaînes de caractères et les autres objets.

Ils peuvent être initialisés à des valeurs par défaut (identiques pour toutes les instances) en affectant une valeur ou une instance lors de la déclaration d'un attribut.

⚠ L'invocation d'une méthode sur une référence `null` provoque une erreur de type `NullPointerException` !

```
Concert c;
System.out.println(c.getArtist());  // lève un NullPointerException
```

Types primitifs et classes *Wrappers*

Vous les utilisiez, les **types primitifs** existent encore...
`boolean`, `byte`, `char`, `float`, `int`, `long`, `short`, `double`.

Mais il existe également des types objets correspondants : les **classes *Wrappers***.
`Boolean`, `Byte`, `Character`, `Float`, `Integer`, `Long`, `Short`, `Double`.

- Les types primitifs sont nécessaires lors, par exemple, d'opérations arithmétiques de base, qui n'acceptent qu'eux.
- Les classes *Wrappers* sont incontournables dès qu'on utilise des structures de données ou qu'on s'adonne à la programmation générique.
- Il existe des mécanismes dit **auto-boxing** et **unboxing** pour faire des conversions automatiquement.

Plan

Initialisation et affichage

```
class ColorRGB {
  int red, green, blue;
  ColorRGB() {
    this.red = 156;
    this.green = 27;
    this.blue = 104;
  }
  String print() {return "("+red+","+green+","+blue+")";}
}
```

Qu'affiche le code suivant ?

```
    ColorRGB c = new ColorRGB();
    System.out.println(c);
    System.out.println(c.print());
```

Réponse :

```
cours.s01.ColorRGB@6ff3c5b5
(156,27,104)
```


Affichage

En l'absence d'une méthode `public String toString()` :

```
class PairWithout {
  String label;
  int value;
  PairWithout(String s, int v) {
    this.label = s;
    this.value = v;
  }
  String print() {return this.label + "=" + this.value;}

  public static void main(String[] args) {
    PairWithout pwo = new PairWithout("aLabel", 42);
    System.out.println(pwo);
    System.out.println(pwo.print());
  }
}
```

On obtient l'affichage :

```
cours01.PairWithout@7960847b
aLabel=42
```

Affichage

En présence d'une méthode `public String toString()` :

```
class PairWith {
  String label;
  int value;
  PairWith(String s, int v) {
    this.label = s;
    this.value = v;
  }
  public String toString() {return this.label + "=" + this.value;}

  public static void main(String[] args) {
    PairWith pw = new PairWith("aLabel", 42);
    System.out.println(pw);
    System.out.println(pw.toString());
  }
}
```

On obtient l'affichage :

```
aLabel=42
aLabel=42
```


Plan

Convention à respecter

Il existe différentes conventions d'écriture à respecter selon les langages :

- *Pascal case* : Pascal, Php — `UserLoginCount`
- ***Camel case* : JS, C++, C#, Java** — **`userLoginCount`**
- *Kebab case* ou *Spinal case* : HTML ou CSS — `user-login-count`
- *Snake case* ou *Underscore case* : Python, Php, Ruby — `user_login_count`
- ...

Il faut veiller à respecter les conventions suivantes :

- `MyFirstObject` pour un objet
- `myFirstMethod(...)` pour une méthode
- `myFirstVariable` pour un attribut ou une variable
- `myfirstpackage` pour un paquetage

De plus, on écrit **une classe par fichier**, fichier qui doit porter le même nom que la classe.

Consignes liées aux modules

- Une arborescence à respecter :

```
devS2/
├── bin/          ← les fichiers binaires
├── doc/          ← la documentation
├── lib/          ← les librairies externes .jar
├── res/          ← les images, les fichiers de données...
├── src/          ← les fichiers sources
│   ├── tpXX/
│   │   ├── A.java
│   │   ├── B.java
│   │   └ ...
│   └ ...         ← une paquetage par semaine
└── test/         ← les classes de tests
```

- Les méta-données de projet et les fichiers sources ne sont pas mélangés (à spécifier lors de la création du projet) ;
- QCMs réguliers en TD avec correction en fin de séance