Compilation, encapsulation, types primitifs

Arnaud Labourel arnaud.labourel@univ-amu.fr

8 février 2022

Section 1

Compilation

Compilation/interprétation

- En Java, le code est généralement compilé puis ensuite exécuté.
- En Python, le code est généralement directement interprété.

Compilation

code source (langage de programmation) $\rightarrow$ code exécutable (langage machine)

Plusieurs phases :

- analyse lexicale (découpe le code en *tokens*)
- analyse syntaxique (vérifie la syntaxe)
- analyse sémantique (vérifie le typage)

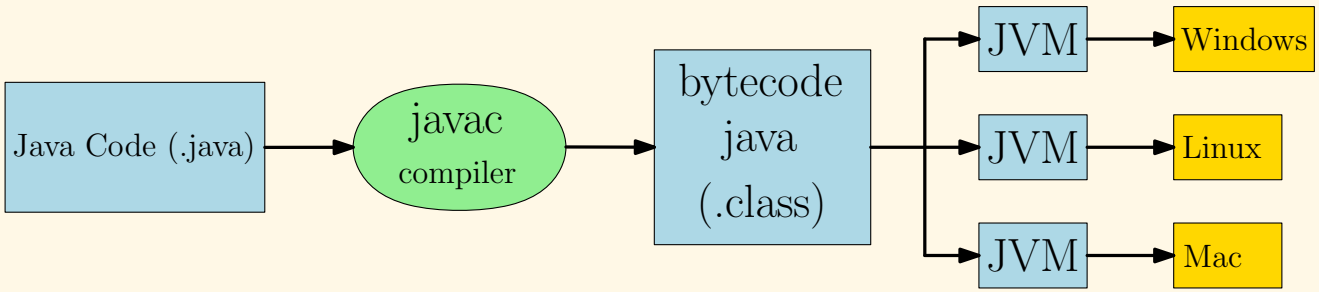
Commandes

- `javac MonFichier.java` pour compiler le fichier `MonFichier.java`
- `java MonFichier` pour exécuter le code de `MonFichier.java`

Compilation et machine virtuelle

Compilation Java

- le compilateur java génère du **bytecode java** = langage machine virtuel
- le code java s'exécute dans une **Java Virtual Machine (JVM)**



Machine virtuelle Java

- la JVM interprète le *bytecode* pour l'exécuter
- la JVM rend le code java indépendant de l'OS et de la machine hôte

$\Rightarrow$ *Compile once, run everywhere*

Exemple compilation → exécution

Pour être exécuté un code Java doit définir une méthode `main`.

```
public class App {
  public static void main(String[] args) {
    System.out.println("Hello world !");
  }
}
```

- la fichier doit avoir le même nom que la classe (ici `App` et donc fichier `App.java`)
- La commande `javac App.java` compile le fichier et crée l'exécutable `App.class`.
- La commande `java App` exécute `App.class` et affiche donc `Hello world !`.

Exemple compilation → exécution avec arguments

```
public class App2 {
  public static void main(String[] args){
    for(String arg : args)
      System.out.println(arg);
  }
}
```

- `String[] args` correspond aux arguments de l'appel d'exécution.
- La commande `java App2 to ta ti` exécute `App2.class` avec les arguments `to`, `ta` et `ti` et affiche donc :

```
to
ta
ti
```

Section 2

Encapsulation

Exemple classe `Client`

Pour le moment tout code ayant accès à un `Item`, a accès à son attribut `price` en lecture et écriture.

```
public class Client{
  void orderItem(Order order, String id) {
    Catalogue cata = order.getCatalogue();
    Item item = cata.getItem(id);
    item.price = 0; // l'item devient gratuit
    order.addItem(item);
  }
}
```

Comment « empêcher » cela ?

Comment interdire la modification des attributs depuis l'extérieur ?

Contrôle d'accès

- **Restreindre** la visibilité des attributs ou méthodes d'une classe.
- En Java : **modificateurs** d'accès précisés lors de la définition d'attributs ou méthodes.

Mot-clés `private` et `public`

Utilisables à la déclaration des attributs, méthodes et constructeurs.
Deux modificateurs d'accès possibles :

- `private` : accessible uniquement pour les instances de la classe c'est-à-dire uniquement depuis le code des méthodes de la classe
- `public` : accessible pour tout le monde c'est-à-dire dans le code de n'importe quelle méthode

La classe Item

```
public class Item {
  private float price;
  private String id;
  public float getPrice() {
    return this.price;
  }
  public float getId() {
    return this.id;
  }
  public boolean moreExpensiveThan(Item otherItem) {
    return this.price > otherItem.price;
  }
  public Item(float p, String id) {
    this.price = p;
    this.id = id;
  }
```

Méthode getTotalPrice dans Order

```
public class Order {
  List<Item> items;
  public float getTotalPrice() {
    float total = 0;
    // cumuler les prix de tous les articles
    for(Item item : this.items) {
      total += item.price; // interdit private !
    }
  return total;
  }
}
```

Erreur à la compilation : `price` étant un attribut privé, il n'est pas possible d'y accéder depuis une autre classe (ici `Order`).

Méthode getTotalPrice dans Order

```
public class Order {
    List<Item> items;
  public float getTotalPrice() {
    float total = 0;
    // cumuler les prix de tous les articles
    for(Item item : this.items) {
      total += item.getPrice();
      // autorisé public !
    }
  return total;
  }
}
```

On peut accéder à la valeur de price grâce à un getter qui lui est public.

Règle d'encapsulation

Règle

Rendre privés les attributs caractérisant l'état de l'objet et fournir si besoin des méthodes publiques permettant de modifier/accéder à l'attribut

`get` et `set`

accesseur/mutateur = getter/setter

Pour un attribut `float price` :

- `getPrice()` : accesseur
- `setPrice(float newPrice)` : mutateur/modificateur

Exemple getter/setter

```
private float price;

public float getPrice() {
  // accès en lecture
  return this.price;
}

public void setPrice(float newPrice) {
  // accès en écriture
  this.price = newPrice;
}
```

Principe d'encapsulation

Ne pas laisser l'état d'un objet en libre accès en modification

Intérêt de l'encapsulation

- **masquer l'implémentation**

  → toute la décomposition du problème n'a pas besoin d'être connue du « programmeur utilisateur »

- **protéger**

  → l'objet a le contrôle sur son état (contrôle sur les modifications)

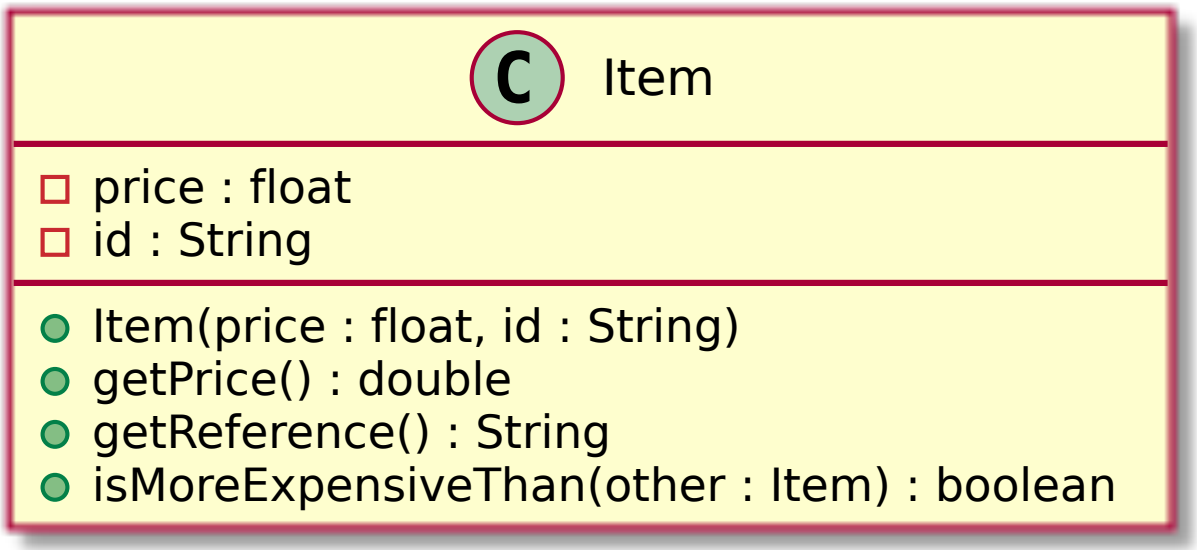
  → préserver l'intégrité des objets

  → le « programmeur créateur » contrôle (et est responsable) de son interface par rapport au « programmeur utilisateur »

- **permettre l'évolutivité**

  → il est possible de modifier tout ce qui n'est pas public sans impact pour le « programmeur utilisateur »

`private` et `public` dans les diagrammes de classes



Règle

- ○ = `public` (autre symbole possible +)
- □ = `private` (autre symbole possible -)

Encapsulation et contrôle de l'état

Toutes les valeurs autorisées par le type des attributs ne sont pas forcément correctes pour l'objet :

- attribut `int month` dans `Date` doit être entre 1 et 12
- attribut `double celsiusValue` dans `Temperature` doit être supérieur à `-273.15` (zéro absolu)
- attribut `double price` dans `Item` doit être positif

Toutes les valeurs autorisées par le type des paramètres d'une méthode ne sont pas forcément correctes :

- paramètre `double amount` dans `withdraw` doit être supérieur ou égal à 0 (pas de retrait d'argent d'un montant négatif)
- paramètre `int divisor` dans `divide` ne doit pas être égal à 0 (pas de division par zéro)

Contrôle avec mot clé assert

En Java, il est possible d'utiliser le mot-clé `assert` pour vérifier que certains prédicats sont vrais.

```
public class Temperature {
  private double valueInCelsius;
  public Temperature(double valueInCelsius) {
    assert valueInCelsius >= -273.15 :
            (valueInCelsius + " temp not allowed");
    this.valueInCelsius = valueInCelsius;
  }
}
```

$\Rightarrow$ vérifie que `valueInCelsius` est supérieur à `-273.15`.

Exemple de fonctionnement d'assert

```
public class Temperature {
  private double valueInCelsius;
  public Temperature(double valueInCelsius) {
    assert valueInCelsius >= -273.15 :
            (valueInCelsius + " temp not allowed");
    this.valueInCelsius = valueInCelsius;
  }
  public static void main(String[] args){
    Temperature t = new Temperature(-300.);
  }
}
```

```
Exception in thread "main"
  java.lang.AssertionError: -300.0 temp not allowed
    at Temperature.<init>(Temperature.java:4)
    at Temperature.main(Temperature.java:10)
```

Règles d'assert

Syntaxe d'assert

```
assert condition : message;
```

Fonctionnement d'assert

- Si la condition est fausse, l'exécution s'arête (fin du programme) et le `message` est affiché via une `Exception` de type `AssertionError`.
- Si la condition est vraie, rien ne se passe.

Activation d'assert

De base, les `assert` ne sont pas activés, il faut utiliser l'option `-ea` dans la commande `java` pour les activer :

```
java -ea Main
```

Exemple assert

```
public class Date {
  private int month;
  private int year;
  public Date(int month, int year) {
    this.setMonth(month);
    this.year = year;
  }
  void setMonth(int month){
    assert 1 <= month && month <= 12 :
           (month + " month number not allowed");
    this.month = month;
  }
}
```

$\Rightarrow$ vérifie que `month` est compris entre 1 et 12.

Section 3

Les types primitifs

Les types primitifs

En java, il existe des types **primitifs** qui ne sont pas des objets :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	`false` ou `true`	`false`

Types primitifs vs objets

- Les noms des types primitifs commencent par un minuscule (vs majuscule pour les types objets).
- Il n'y a pas de classe associée aux types primitifs et ils ne peuvent pas être utilisé pour appeler une méthode.
- Il n'y a pas de constructeurs pour les types primitifs (pas d'instanciation).
- Les variables de type primitif contiennent directement la valeur et **pas** une référence comme c'est le cas pour les objets.
- On affecte les variables de type primitifs avec :
  - des littéraux : `'a'`, `12`, `42.0`, `42.0f`, ...
  - des résultats d'opérations : ‘`2. + x,2 / 5,3 * 4`‘, ...

Comportement types primitifs

Lors d'un appel de méthode les arguments sont passés par valeur : une copie de la valeur de l'argument est créé lors de l'appel.

Cela un impact différent suivant que l'argument soit un objet ou un type primitif :

- Pour les objets, cela signifie passer une copie de la **référence** : il est donc possible de modifier l'état de l'objet.
- Pour les types primitifs, cela signifie que l'argument est une **copie** uniquement créée pour l'appel et toute modification de sa valeur n'aura pas d'impact en dehors de l'appel.

Exemple comportement objet

```
public class Point {
  private double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() { return x; }
  public double getY() { return y; }
  public void setX(double x) { this.x = x; }
  public void setY(double y) { this.y = y; }
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

Exemple comportement objet receveur

```
public class App{
  public static void main(String[] args){
    Point p = new Point(0., 0.);
    p.setX(1.);
    p.setY(2.);
    System.out.println(point.toString());
    // => affiche (1.0, 2.0)
  }
}
```

L'objet passé en notation pointée est modifié.

Exemple comportement objet en argument

```
public class Point {
  public void copyInto(Point p){
    p.setX(getX());
    p.setY(getY());
  }

  public static void main(String[] args){
    Point p1 = new Point(1, 2);
    Point p2 = new Point(0, 0);
    p1.copyInto(p2);
    System.out.println(p2.toString());
    // => affiche (1.0, 2.0)
  }
```

L'objet passé en argument est modifié.

Exemple comportement type primitif en argument

```java
public class Point {
  public double distanceTo(Point p){
    return Math.hypot(p.x - x, p.y - y);
  }
  public void addDistanceTo(double d, Point p){
    d += this.distanceTo(p);
  }
  public static void main(String[] args){
    double d = 0;
    Point p1 = new Point(1, 2);
    Point p2 = new Point(0, 0);
    p1.addDistanceTo(d, p2);
    System.out.println(d); // => affiche 0.0
  }
```

Le contenu de la variable passé en argument n'est pas modifié.

Égalité de type primitif

Pour tester l'égalité de types primitifs *entiers* (`byte`, `short`, `int`, `long`, `char` et `boolean`), il suffit d'utiliser ==.

Pour tester l'égalité de types primitifs *flottants* (`float`, `double`), il faut faire attention aux erreurs d'approximation.

Bonne comparaison entre 2 doubles d1 et d2

Fixer un ϵ petit et vérifier que $|d1 - d2| < \epsilon$.

Exemple :

```
double d1 = 0.1;
double d2 = 0.1 + 0.1 + 0.1;
double epsilon = 0.000001d;
d1 == d2// false : mauvaise comparaison
Math.abs(d1 - d2) < epsilon;// true : bonne comparaison
```