Programmation orientée objet. Cours 2

Marie-Pierre Béal
BUT 1

Listes
Types primitifs et leurs enveloppes
Records avancés
Packages et import
Méthodes d'instance vs. méthodes statiques

Listes

Un objet `List` définit une liste d'éléments

- indexée par un entier
- ordonnée par l'ordre d'insertion

L'API Java fournit plusieurs types de listes différentes.

- Elles sont paramétrées par le type des objets qu'elles contiennent.
- Chacune a des implantations modifiables et non modifiables

java.util.List

List.of vs ArrayList

Il y a deux implantations principales

- `List.of()` pour les listes non modifiables

```
var list = List.of("hello", "collection");
```

- `ArrayList` qui s'agrandit dynamiquement

```
var list = new ArrayList<String>();
list.add("hello");
list.add("collection");
```

java.util.List.of()

Liste non modifiable qui n'accepte pas null

- `List.of(E ...)`

```
var empty = List.<String>of(); // on doit indiquer le type
var one = List.of(42);
var three = List.of(1, 5, 8);
```

list.isEmpty(), list.size()

`list.isEmpty()` renvoie true si la liste est vide

```
List.of("hello").isEmpty()  // false
```

`list.size()` renvoie le nombre d'éléments

```
List.of("hello").size()  // 1
```

list.get(), list.getFirst(), list.getLast()

`list.get(index)` renvoie le nième élément (à partir de 0)

```
List.of("hello", "list", "map").get(1)  // list
```

- Lève une exception `IndexOutOfBoundsException` si on sort des bornes de la liste

Les listes sont dans des classes qui implémentent l'interface `SequencedCollection`.

`list.getFirst()` renvoie le premier élément
`list.getLast()` renvoie le dernier élément

```
List.of("hello", "list").getFirst()  // hello
List.of("hello", "list").getLast()  // list
```

- Lèvent une exception NoSuchElementException si la liste est vide

toString() / equals() / hashCode()

`list.toString()` renvoie une version textuelle

```
List.of(1, 2, 3).toString()  // [1, 2, 3]
```

`list.equals(list2)` permet de comparer des listes même si ce n'est pas la même implantation

```
var list = List.of(1, 2, 3);
IO.println(list.equals(list));  // true
var list2 = new ArrayList<Integer>();
list2.add(1);
list2.add(2);
list2.add(3);
IO.println(list.equals(list2));  // true
```

`list.hashCode()` renvoie une valeur résumée du contenu de la liste

```
List.of(1, 2, 3).hashCode()  // 30817
```

Parcours avec `for(:)`

On peut utiliser la syntaxe `for(:)` sur les listes

```
var list = List.of(1, 2, 3);
for(var element: list) {
  IO.println(element);
  // 1
  // 2
  // 3
}
```

C'est la même syntaxe que pour les tableaux mais le code généré par le compilateur n'utilise pas des indices (mais un Iterator, voir plus tard).

list.set()

`list.set(index, element)` change la valeur de l'élément à la nième case

```
var list = new ArrayList<String>();
list.add("hello");
list.set(0, "bonjour");
IO.println(list.getFirst());  // bonjour
```

list.add() / list.remove()

`list.add(element)` ajoute un élément à la fin
`list.addLast(element)` ajoute un élément à la fin (appelle add)
`list.addFirst(element)` ajoute un élément au début

- `list.add("hello")` // sur une liste où l'opération est supportée

`list.remove(element)` supprime le premier élément (en utilisant element.equals())

- `list.remove("hello")` // sur une liste où l'opération est supportée

`list.remove(index)` supprime à l'index et renvoie l'élément
`list.removeFirst()` supprime le premier élément et renvoie l'élément
`list.removeLast()` supprime le dernier élément et renvoie l'élément

- `list.removeFirst()` // les éléments sont décalés à gauche, sur une liste où l'opération est supportée
- lèvent une exception `NoSuchElementException` si la liste est vide

Listes non modifiables

Si la liste est non modifiable, les méthodes `set()`, `add()` et `remove(index)/remove(element)/removeFirst()/removeLast()` lèvent UnsupportedOperationException

List.of() construit une liste non modifiable

- `List.of("hello").set(0, "list")` // lève UnsupportedOperationException
- `List.of("hello").add("list")` // lève UnsupportedOperationException
- `List.of("hello").remove("hello")` // lève UnsupportedOperationException
- `List.of("hello").removeFirst()` // lève UnsupportedOperationException

conversion liste modifiable / non modifiable

`List.copyOf(collection)`

- Prend une collection en paramètre et copie tous les éléments dans une `List` non modifiable (la même que `List.of()`)

`new ArrayList<>(collection)`

- Prend une collection en paramètre et copie tous les éléments dans une `ArrayList` (modifiable)

`contains()` / `indexOf()` / `lastIndexOf()`

`list.contains(element)` renvoie si l'élément est contenu (en utilisant element.equals())

- `List.of("foo", "bar").contains("bar")` // true

`list.indexOf(element)` renvoie l'index du premier élément égal (en utilisant element.equals()) ou -1

- `List.of("foo", "foo").indexOf("foo")` // 0

`list.lastIndexOf(element)` renvoie l'index du dernier élément égal (en utilisant element.equals()) ou -1

- `List.of("foo", "foo").lastIndexOf("foo")` // 1

Lenteur de `contains/remove/indexOf/lastIndex`

Toutes ces méthodes ont une complexité en $O(n)$, elles nécessitent dans le pire cas de parcourir tous les éléments.

Il peut être plus efficace d'utiliser un dictionnaire (Map) pour avoir une recherche plus rapide (voir cours suivants).

Types paramétrés

En Java, le type entre < > doit être un type objet.
Pour une liste d'entiers, le type des éléments de la liste doit être `Integer` et pas `int`.

```
IO.println(List.of(1,2,3)); // [1, 2, 3]
```

C'est une `List<Integer>`.

Types primitifs et types enveloppes

Nom	Taille en bits	Taille en octets	Exemples	Enveloppe
`byte`	8	1	1, -128, 127	`java.lang.Byte`
`short`	16	2	2, 300	`java.lang.Short`
`int`	32	4	234569876	`java.lang.Integer`
`long`	64	8	2L	`java.lang.Long`
`float`	32	4	3.14, 3.1E12, 2e12	`java.lang.Float`
`double`	64	8	0.5d	`java.lang.Double`
`boolean`	8	1	`true` ou `false`	`java.lang.Boolean`
`char`	16	2	`'a'`, `'\n'`, `'\u0000'`	`java.lang.Character`

- Les caractères sont codés sur deux octets en Unicode.
- Les types sont indépendants du compilateur et de la plate-forme.
- Tous les types numériques sont signés sauf les caractères.
- Un booléen n'est pas un nombre.
- Les opérations sur les entiers se font modulo, et sans erreur :

```
byte b = 127; b += 1; // b = -128
```

Enveloppes des types primitifs

- Une instance de la classe enveloppe encapsule une valeur du type de base correspondant.
- Chaque classe enveloppe possède des méthodes pour extraire la valeur d'un objet (par exemple `o.intValue()` appliquée sur un objet `o` de la classe `Integer` renvoie une valeur de type `int`).
- Une méthode statique de chaque classe enveloppe renvoie un objet enveloppant le primitif correspondant – par exemple `Integer.valueOf(int p)`.
- Un objet enveloppant est non mutable : la valeur contenue dedans ne peut pas être modifiée.
- On transforme souvent les valeurs primitives en objets pour les mettre dans les collections.

Conversions automatiques

Auto-boxing

```
Integer i = 3; // int --> Integer
Long l = 3L;   // long --> Long
Long l = 3;    // erreur, int --> Integer -/-> Long
// alors que long l = 3; fonctionne.
```

Auto-unboxing

```
Integer i = Integer.valueOf(3);
int x = i; // Integer --> int
Long lo = null;
long l = lo; // erreur : java.lang.NullPointerException :
             // Cannot invoke "java.lang.Long.longValue()"
             // because "lo" is null
```

Les conversions se font aussi pour les paramètres lors d'appels de méthodes et pour les valeurs de retour des méthodes.

Record compléments

Un record public

```java
// dans le fichier Point.java
public record Point(int x, int y) {}
```

```java
// dans Test.java
class Test.java {
  static void main() {
    var point = new Point(3, 4);
    IO.println(point.x()); // 3
  }
}
```

Ici on a ajouté un modificateur de visibilité `public` qui signifie que le record sera visible dans tout code situé dans n'importe quel répertoire.

Accesseurs publics

Dans un record, le compilateur ajoute des méthodes publiques de même noms que les champs appelées accesseurs (ici `x()` et `y()`), avec comme code, le code des méthodes `x()` et `y()` généré par le compilateur.
C'est comme si on écrivait

```
public record Point(int x, int y) {
  public int x() {  // il est inutile de l'écrire
    return x;
  }
}
```

inutile signifie qu'il ne faut pas l'écrire (points en moins aux examens si vous l'écrivez).

Remplacer / Redéfinir les méthodes existantes

On peut changer l'implantation des accesseurs ou des méthodes `toString/equals/hashCode`.

```
public record Pair(String first, String second) {
  @Override
  public String toString() {
    return "Pair(" + first + ", " + second + ")";
  }
}
```

```
// dans Test.java
class Test.java {
  static void main() {
    IO.println(new Pair("toto", "titi")); // Pair(toto, titi)
  }
}
```

On utilise l'annotation `@Override` pour aider à la lecture, faire la différence entre une nouvelle méthode et le remplacement d'une méthode existante.

Constructeur canonique

Le constructeur est une méthode spéciale appelée lors du new pour initialiser les champs.

Dans un record, le compilateur génère automatiquement le constructeur canonique (celui qui initialise les champs).

```
public record Person(String name, int age) {
  // généré automatiquement
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Redéfinir le constructeur

Il est souvent nécessaire de remplacer le constructeur car on veut empêcher de créer des objets avec des valeurs erronées

```
public record Person(String name, int age) {
  public Person(String name, int age) {
    Objects.requireNonNull(name);
                    // plante (lève une NullPointerException)
                    // si name est null
    if (age < 0) {
      throw new IllegalArgumentException("age < 0");
    }
    this.name = name;
    this.age = age;
  }
}
```

On vérifie les pré-conditions.

Redéfinir le constructeur compact

Le constructeur canonique a une version “compacte”

```
public record Person(String name, int age) {
  public Person { // pas de parenthèses
    Objects.requireNonNull(name, "name is null");
    if (age < 0) {
      throw new IllegalArgumentException("age < 0");
    }
  }
}
```

qui ne laisse apparaître que les pré-conditions.
Le compilateur ajoute les `this.name = name;` etc à la fin du constructeur compact.

equals/hashCode/toString

Dans un record, le compilateur génère aussi les méthodes

- `equals()` : indique si deux objets de type `Point` ont les mêmes valeurs

```
static void main(String[] args) {
  var point = new Point(2, 3);
  var point2 = new Point(2, 3);
  println(point.equals(point2));  // true
  var point3 = new Point(4, 7);
  IO.println(point.equals(point3));  // false
  IO.println(point.hashCode()) // 65
}
```

- `hashCode()` : renvoie un entier "résumé", cf cours suivants

equals/hashCode/toString

- `toString()` : renvoie une représentation textuelle

```
static void main(String[] args) {
  var point = new Point(2, 3);
  IO.println(point.toString()); // Point[x = 2, y = 3]
  IO.println(point);  // Point[x = 2, y = 3]
}
```

Module, Package et import

Package

Une librarie Java (appelé un module) est composée de plusieurs packages.

La librarie par défaut du langage Java est appelée `java.base` et contient les packages :

Pour le JDK (la librairie par défaut de Java)

- java.lang : classes de base du langage
- java.util : classes utilitaires, structures de données
- java.util.regex : expression régulière (cf cours 2)
- java.io : pour faire des entrées/sorties
- java.nio.file : entrées/sorties sur les fichiers
- etc.

La directive import en Java

On peut faire une importation de tous les packages d'un module avec `import module` (JEP 476).

En début de fichier, on écrit :

```
import module java.base;  // ne pas oublier le ';'
```

`import` n'importe pas de fichier au sens de Python/C mais dit que l'on peut utiliser ArrayList à la place de java.util.ArrayList dans le code. Le mot-clef devrait s'appeler "alias" pas "import".

La directive import en Java

Ou bien, on spécifie des classes/records appartenant à des packages que l'on veut utiliser

```
import java.util.ArrayList;
```

Exemple

```
import module java.base;
public class Hello {
  static void main() {
    var list = new ArrayList();
  }
}
```

- Le vrai nom de `ArrayList` est `java.util.ArrayList` et le package `java.util` est importé car il est dans `java.base`.

Le code ci-dessus est équivalent au code sans import

```
public class Hello {
  static void main() {
    var list = new java.util.ArrayList();
  }
}
```

Champs et méthodes statiques

Champ static final

Un champ static final est une constante

```
public record Asset(long price) {
  private static final long MAX_TAX = 1_000_000L;

  public long computeTax() {
    return Math.min(MAX_TAX, price / 10);
  }
}
```

Aide à la lecture du code en remplaçant une valeur par un nom

- Aide à la maintenance du code

Champ `static` pas `final` (à ne pas faire)

Un champ static qui n'est pas final est une sorte de variable globale partagée par la classe ou le record.

Il ne faut pas en mettre sauf cas exceptionnel.

Méthodes statiques

Méthodes d'instance et méthodes statiques

On appelle

- une méthode d'instance sur une instance (avec un `.` après l'objet)
- une méthode statique sans instance, sur la classe (avec un `.` après le nom de la classe)

Lors de la déclaration

- une méthode statique est précédée du mot `static`.

Donc une méthode statique est une méthode que l'on appelle sur la classe indépendamment d'une instance.

Méthodes d'instance et méthodes statiques

`java.io.IO.println` est une méthode statique

```
static void main(){
  IO.println("Hello");
}
```

Il existe aussi une méthode non statique `System.out.println()`

```
static void main(){
  System.out.println("Hello");
}
```

Méthodes d'instance et méthodes statiques

`IO.readln` et `Integer.parseInt` sont des méthodes statiques

```
static void main(){
  var s = IO.readln("Enter an integer \n");
  IO.println(Integer.parseInt(s));
}
```

```
$ java ParseIntTest.java
Enter an integer
234
234
```

Méthodes d'instance et méthodes statiques

```
public record Taxi(boolean uber) {
  String name() {
    return this.uber? "Uber": "Hubert?";
  }
  static String bar() {
    return "Hello Taxi";
  }
}
```

```
static void main() {
  IO.println(new Taxi(true).name()); // name méthode d'instance
  IO.println(Taxi.bar());   // bar méthode statique
}
```

```
$ java TaxiTest.java
Uber
Hello Taxi
```

Et le main ?

Si on veut un main static, il faut le déclarer `"static"`.

Dans un fichier `HelloWorld.java` on met :

```
static void main(){
  IO.println("Hello World!");
}
```

```
$ java HelloWorld.java
Hello World!
```

appelle `HelloWorld().main()`.

Exercices

Exercice 1 Calcul de maximum

1. Écrire une méthode statique `maxList` qui prend en argument une listes d'entiers et renvoie le maximum de la liste.
2. Écrire une méthode statique `maxListIndex` qui prend en argument une listes d'entiers et renvoie l'indice d'un élément maximal de la liste.

Exercice 2 Poissons

1. Écrire un record `Fish` pour représenter des poissons. Un poisson a une espèce représentée par une chaîne de caractères `species`, un booléen `seaFish` qui indique si c'est un poisson de mer ou pas et une longueur `length` en cm.
2. Modifier le constructeur canonique pour qu'il lève une exception si l'espèce est `null` ou si la longueur n'est pas comprise entre 0 et 300 (on renverra une `IllegalArgumentException` dans ce cas).
3. Ajouter un constructeur qui permet de créer un poisson de mer à partir de son espèce et de sa longueur.
4. Ajouter une méthode `ifIsSeaFish` qui renvoie le poisson auquel on l'applique si c'est un poisson de mer, et `null` sinon.

Exercice 3 Liste de poissons

1. Écrire une classe `FishTest` qui contient une méthode `main` qui crée 3 poissons.
2. Créer et afficher une liste non modifiable `list1` contenant ces 3 poissons.
3. Créer et afficher une liste modifiable `list2` contenant ces 3 poissons.
4. Est-il possible d'ajouter un `Fish` dans `list1`. Pourquoi ?
5. Est-il possible d'ajouter un `Bird` dans `list2`. Pourquoi ?
6. Ajouter une méthode `allSeaFishes` qui prend une `ArrayList` de poissons en paramètre et renvoie `true` si tous les poissons qu'elle contient sont des poissons de mer, et `false` sinon. Quelle est la particularité de cette méthode ?
   Remarque : vous pouvez consulter la documentation des `ArrayList` ici :
   `https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayList.html`.

Exercice 3 Liste de poissons suite

1. Écrire une méthode `allSeaFishes2` qui fait la même chose que la précédente mais qui utilise la boucle for-each aperçue vu en cours (c'est la même chose qu'en Python, sans oublier le typage).
2. Ajouter une méthode `removeSeaFishes` qui prend une liste de poissons en paramètre et supprime tous les poissons de mer qu'elle contient.
3. Ajouter une méthode `withoutSeaFishes` qui prend une liste de poissons en paramètre et renvoie une liste qui contient les mêmes poissons, mais sans les poissons de mer. À votre avis, quelle méthode est meilleure : celle-ci ou la précédente ?
4. Ajouter une méthode `sameFishes` qui prend deux listes de poissons en paramètre et renvoie `true` si les listes contiennent exactement les mêmes poissons, et `false` sinon.
5. À quel(s) endroit(s) aurait-on pu utiliser des listes non modifiables dans le code que vous venez d'écrire ?