Programmation orientée objet. Cours 2

Marie-Pierre Béal et Carine Pivoteau
BUT 1

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

Listes

L'API Java fournit plusieurs types de listes différentes (nous verrons les liens entre elles dans la suite du cours).
Elles sont paramétrées par le type des objets qu'elles contiennent (`E`).

Elles ont en commun certaines méthodes, dont celles-ci :

void	**add**(int index, **E** element)	Inserts the specified element at the specified position in this list (optional operation).
boolean	**add**(**E** e)	Appends the specified element to the end of this list (optional operation).
void	**clear**()	Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o)	Returns `true` if this list contains the specified element.
E	**get**(int index)	Returns the element at the specified position in this list.
int	**indexOf**(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**()	Returns `true` if this list contains no elements.
int	**lastIndexOf**(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**remove**(int index)	Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o)	Removes the first occurrence of the specified element from this list, if it is present (optional operation).
E	**set**(int index, **E** element)	Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**()	Returns the number of elements in this list.

Listes non modifiables

L'interface `java.util.List` fournit une méthode statique `of` qui permet de créer et renvoyer une liste non modifiable d'objets du même type `E`, spécifié avec la notation `<E>` :

```
static <E> List<E> of(E... elements)        Returns an unmodifiable list containing an arbitrary number of elements.
```

On peut choisir n'importe quel type d'objet pour les éléments :

```java
import java.util.List;
public class Test {
  public static void main(String[] args) {
    var animals = List.of("tortue", "lapin", "lion", "orignal");
    System.out.println(animals); // [tortue, lapin, lion, orignal]
    var pie = new Bird("pie", 10); // avec le record Bird, cf TD 1
    var heron = new Bird("héron", 115);
    var birds = List.of(pie, heron, new Bird("corbeau",70));
    System.out.println(birds);
    // [Bird[name=pie, wingspan=10], Bird[name=héron, wingspan=115],
    //  Bird[name=corbeau, wingspan=70]]
```

Listes non modifiables

On peut utiliser uniquement les méthodes qui ne modifient pas la liste :

```
    ...
    var animals = List.of("tortue", "lapin", "lion", "orignal");
    var birds = List.of(pie, heron, new Bird("corbeau",70));
    System.out.println(birds.get(0)); // Bird[name=pie, wingspan=10]
    System.out.println(birds.getFirst()); // Bird[name=pie, wingspan=10]
    System.out.println(animals.get(1).charAt(0)); // l
    var sum = 0;
    for (var bird : birds) {  // boucle foreach
      sum += bird.wingspan();
    }
    System.out.println(sum); // 195
    birds.add(heron);
    // Exception in thread "main"
    // java.lang.UnsupportedOperationException
  }
}
```

Listes modifiables : `ArrayList`

Dans le package `java.util`, il existe une classe `ArrayList` qui permet de créer des listes (implantées sous la forme de tableaux) que l'on peut ensuite modifier :

Exemple d'une liste de `Point`.

```
import java.util.ArrayList;
public class Test {
  public static void main(String[] args) {
    var list = new ArrayList<Point>();
    System.out.println(list); // []
    var p1 = new Point(0,0);
    var p2 = new Point(1,1);
    list.add(p1);
    list.add(p2);
    list.add(new Point(1,1));
    System.out.println(list);
     // [Point[x=0, y=0], Point[x=1, y=1], Point[x=1, y=1]]
    list.remove(p2);
    System.out.println(list);  // [Point[x=0, y=0], Point[x=1, y=1]]
  }
```

Et si on veut une liste d'entiers ?

... Ou plus généralement une liste de primitifs ?

Pas de problème :

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

Mais quel est le type de `List.of(1,2,3)` ?

- ▷ `List<int>` ? NON !

Ce n'est pas possible en java, le type entre < > doit être un type objet.

C'est une `List<Integer>`. On doit utiliser l'enveloppe du type primitif.

Types primitifs

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'appel de méthodes et pour les valeurs de retour des méthodes.

Le problème avec la valeur `null`

On peut créer un `Segment` en indiquant ses extrémités :

```
public record Segment(Point start, Point end){
  public boolean isHorizontal(){
    return start.y() == end.y();
  }
}
```

Qu'affiche le code suivant ?

```
public class SegmentTest {
  public static void main(String[] args) {
    var segment = new Segment(new Point(3,4), null);
    System.out.println(segment.start()); // Point[x=3, y=4]
    System.out.println(segment.end());   // null
    System.out.println(segment.start().x()); // 3
    System.out.println(segment.isHorizontal());
   // Exception in thread "main" java.lang.NullPointerException
  }
}
```

Empêcher les exceptions "non prévues"

- Le programmeur doit pouvoir savoir quand une exception risque de se produire (pour l'éviter).
- Les exceptions doivent être réservées à des comportements anormaux.
- Par exemple, un accesseur sur un objet non null n'est pas censé causer une exception.

On souhaite s'assurer, à la création, que les segments sont bien formés de deux objets `Point` (pas de référence `null`).

On peut utiliser la méthode statique `requireNonNull`, qui se trouve dans la classe `java.util.Objects`

Record : constructeur canonique

Il faut aussi pouvoir agir lors de la construction du record. Pour cela, on peut modifier le **constructeur canonique** :

```
import java.util.Objects;

public record Segment(Point start, Point end) {
  // constructeur canonique
  public Segment(Point start, Point end) {
    this.start = start;
    this.end = end;
  }
}
```

this

Le mot clé `this` est utilisé pour désigner une référence sur **l'objet courant** dans le code du record.

Remarque : on ne l'utilise que lorsqu'il y a une ambiguïté.

Empêcher les champs null

On peut maintenant rajouter de quoi tester les champs :

```
import java.util.Objects;

public record Segment(Point start, Point end) {
 // constructeur canonique
  public Segment(Point start, Point end) {
    Objects.requireNonNull(start);
    Objects.requireNonNull(end);
    this.start = start;
    this.end = end;
  }
}
```

Ce qui équivaut à :

```
public Segment(Point start, Point end) {
  this.start = Objects.requireNonNull(start);
  this.end = Objects.requireNonNull(end);
}
```

Exception à la création

L'erreur a désormais lieu à la création du segment.

```
public class SegmentTest {
  public static void main(String[] args) {
    var p1 = new Point(0,0);
    Point p2 = null;
    var s1  = new Segment(p1,p2); // erreur ici
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException
  at java.util.Objects.requireNonNull(Objects.java:201)
  at Segment.<init>(Segment.java:8)
  at SegmentTest.main(SegmentTest.java:5)
```

Record : constructeur compact

Il existe aussi une version plus courte : le constructeur compact.

Les paramètres ont obligatoirement les même noms et types que les champs, on peut donc les omettre.

Les affectations sont imposées, elles peuvent donc être implicites.

Le constructeur précédent est équivalent à :

```
import java.util.Objects;

public record Segment(Point start, Point end) {
  // constructeur compact
  public Segment {
    Objects.requireNonNull(start);
    Objects.requireNonNull(end);
  }
}
```

D'autres constructeurs ?

On peut construire un record avec des paramètres différents de ses champs. Pour cela, on utilise le constructeur canonique : `this`.

Par exemple, on peut utiliser les cordonnées des extrémités :

```
public record Segment(Point start, Point end) {
  public Segment(int xStart, int yStart, int xEnd, int yEnd) {
    this(new Point(xStart, yStart), new Point(xEnd, yEnd));
  }
}
```

```
public class SegmentTest {
  public static void main(String[] args) {
    var s1 = new Segment(new Point(0, 0), new Point(2, 2));
    var s2 = new Segment(0, 0, 2, 2);
    // Segment[start=Point[x=0, y=0], end=Point[x=2, y=2]]
  }
}
```

Changer les méthodes fournies ?

L'annotation `@Override` sera expliquée plus tard :

```
public record Point(int x, int y) {
  @Override
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

```
public record Segment(Point start, Point end) {
  @Override
  public String toString() {
    return "Segment " + end + " -- " + start;
  }
}
```

```
public class SegmentTest {
  public static void main(String[] args) {
    var s1 = new Segment(new Point(0, 0), new Point(2, 2));
    // Segment (2,2) -- (0,0)
}
```

D'autres méthodes

```
public record Segment(Point start, Point end) {
  public double length() {
    return start.distance(end);
  }
  // renvoie vrai si le segment courant est plus court que le paramètre
  public boolean isShorter(Segment other) {
    return length() < other.length();
  }
}
```

```
    ...
    var s1 = new Segment(new Point(0, 0), new Point(2, 2));
    System.out.println(s1.length()); // 2.828427124746l903
    var s2 = new Segment(new Point(10, 10), new Point(2, 2));
    System.out.println(s1.isShorter(s2)); // true
    ...
```

Et si on veut renvoyer le segment le plus court ?

L'objet courant : `this` ?

Rappel : le mot clé `this` est utilisé pour désigner une référence sur l'objet courant.

```
public record Segment(Point start, Point end) {
  ...
  public Segment shortest(Segment other) {
    if (length() < other.length()) {
      return this;
    }
    return other;
  }
}
```

```
    ...
    var s1 = new Segment(new Point(0, 0), new Point(2, 2));
    var s2 = new Segment(new Point(10, 10), new Point(2, 2));
    System.out.println(s1.shortest(s2)); // Segment (2,2) -- (0,0)
    ...
```

Méthodes d'instance

Les méthodes vues précédemment (`toString`, `equals`, `println`, ...) sont des **méthodes d'instance** : ce sont des méthodes qui sont **propres à un objet (`this`)** et doivent être appelées **sur** cet objet.

Par exemple, pour générer un nombre aléatoire, on peut utiliser une méthode d'instance sur un objet `Random` :

```
import java.util.Random;

public class Test {
  public static void main(String[] args) {
    var random = new Random();
    for (var i = 0; i < 10; i++) {
      System.out.println(random.nextDouble());
      // double dans [0.0,1.0[
    }
  }
}
```

Fonctions : les méthodes statiques

Pour générer un nombre aléatoire, on peut aussi utiliser la méthode statique `random` de la classe `Math` :

```
public class Test {
  public static void main(String[] args) {
    var randomValue = Math.random(); // double dans [0.0,1.0[
    System.out.println(randomValue);
  }
}
```

Les méthodes statiques, elles ne sont pas liées aux objets :

- On le repère grâce au mot-clé `static` dans leur en-tête.
- Une méthode statique est appelée en la préfixant par le nom de la classe où elle est définie (peut être omis dans la même classe).
- Elles correspondent aux fonctions du C, ou à celles vues en Python.

Remarque : la méthode `main` est forcément statique...

Méthodes statiques

On peut ajouter des méthodes statiques à un record. Par exemple, pour aider au calcul de la longueur :

```
public record Point(int x, int y) {
   private static int diffSquare(int a, int b) {
    return (a - b) * (a - b);
  }

  public double distance(Point other) {
    return Math.sqrt(diffSquare(x, other.x) +
                     diffSquare(y, other.y));
  }
}
```