

Initiation génie logiciel : Notions programmation orientée objet et Java

Arnaud Labourel (arnaud.labourel@univ-amu.fr)



Section 1

Composition, agrégation et délégation

Exemple d'une classe Point

On considère la classe Point suivante :

```
public class Point {  
    public final double x, y;  
    public Point(double x, double y){this.x = x;  
                                     this.y = y;}  
    public double distanceTo(Point p){  
        double dx = this.x - p.x;  
        double dy = this.y - p.y;  
        return Math.hypot(dx, dy);  
    }  
    public double getX(){ return x; }  
    public double getY(){ return y; }  
}
```

Réutilisation de Point : composition

Afin d'implémenter ses services, une instance peut créer des instances d'autres classes et conserver leurs références dans ses attributs.

```
public class Circle {  
    private Point center;  
    private double radius;  
    public Circle(double x, double y, double radius){  
        this.center = new Point(x, y);  
        this.radius = radius;  
    }  
    public double getCenterX(){ return center.getX(); }  
    public double getCenterY(){ return center.getY(); }  
    public double getRadius(){ return radius; }  
}
```

Définition composition

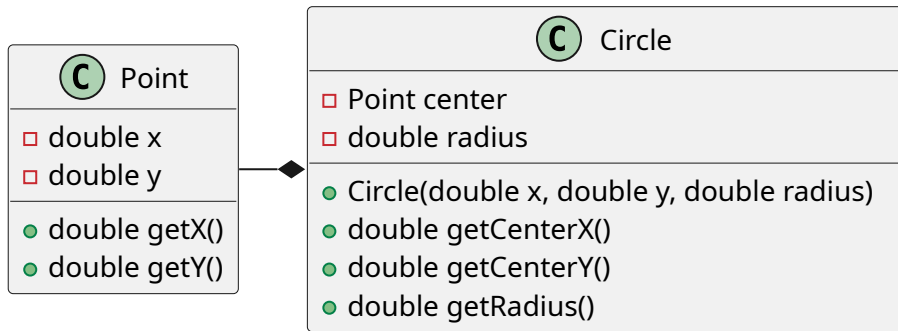
Composition d'une classe Partie dans une classe Classe

- Une (ou plus) instance(s) de Partie est stockée dans une instance de Classe
- Chaque instance de Partie appartient à une seule instance de Classe
- Chaque instance de Partie a sa durée de vie (construction) gérée par l'instance de Classe qui la contient
- Les instances de Partie n'ont pas connaissance de l'existence des instances de Classe

Exemple: Pièces d'une maison

Une maison contient une ou plusieurs pièces. La durée de vie d'une pièce est contrôlée par la maison, car elle n'existerait pas sans la maison.

Diagramme composition



Variante de Circle : agrégation

Une instance peut simplement posséder des références vers des instances d'une autre classe :

```
public class Circle {  
    private Point center, point;  
    public Circle(Point center, Point point){  
        this.center = center;  
        this.point = point;  
    }  
    public Point getCenter(){ return center; }  
    public double getRadius(){  
        double dx = center.x - point.x;  
        double dy = center.y - point.y;  
        return Math.hypot(dx, dy);  
    }  
}
```

Définition agrégation

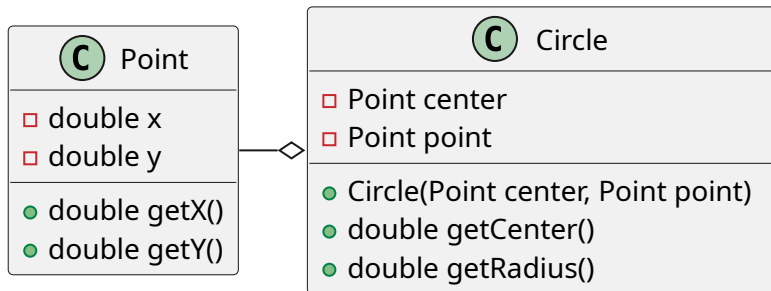
Agrégation d'une classe `Partie` dans une classe `Classe`

- une (ou plus) instance(s) de `Partie` est stockée dans une instance de `Classe`
- Chaque instance de `Partie` peut appartenir à plusieurs instances de `Classe`
- La durée de vie des instances de `Partie` n'est pas gérée par les instances de `Classe` qui la contient
- Les instances de `Partie` n'ont pas connaissance de l'existence des instances de `Classe`

Exemple: Bloc de construction d'une maison jouet

On a une maison jouet construite à partir de blocs. Vous pouvez la démonter, mais les blocs resteront.

Diagramme agrégation



Délégation

Délégation du calcul de la distance à l'instance center de la classe Point :

```
public class Circle {  
    private Point center, point;  
  
    public Circle(Point center, Point point){  
        this.center = center;  
        this.point = point;  
    }  
    public Point getCenter(){ return center; }  
    public double getRadius(){  
        return center.distanceTo(point);  
    }  
}
```

Définition délégation

Délégation d'une opération d'une classe *Classe* à une instance d'une classe *Delegate*

Une instance de *Classe* confie la résolution d'une de ses opérations (méthodes) à une instance de *Delegate*.

L'objet endossant cette responsabilité est nommé le *delegate* (ou délégué). Le rôle d'un *delegate* est de définir du comportement laissé à sa charge.

Exemple : confier le fait de faire un café à un stagiaire

Votre patron vous a demandé de lui faire un café, vous l'avez plutôt fait faire par un stagiaire.

Section 2

Arguments variables

Arguments variables : *varargs* (Java 1.5)

Permet de définir des fonctions prenant un nombre quelconque d'arguments.

Exemple :

```
public static int add(int... values) {  
    int sum = 0;  
    for (int value : values) total += val;  
    return sum;  
}
```

add s'appelle avec un nombre quelconque d'int ou même un tableau d'int :

```
add(); add(1); add(1,2); add(1,2,3);  
add(new int[]{1,2,3,4});
```

Dans le code de la méthode les arguments *varargs* sont un tableau.

Arguments variables : *varargs* (Java 1.5)

Syntaxe

```
ReturnType methodName(TypeArg1 a1, TypeArg2 a2, TypeVarArgs... varArgs)
```

Règles à respecter

- Un seul *varargs* par méthode.
- Le *varargs* doit être le dernier paramètre.
- Éviter les types paramétrés dans les *varargs*.
- Les *varargs* sont traités comme un tableau.

Section 3

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(Object object) {  
        stack[size] = object; size++;  
    }  
    public Object pop() {  
        size--;  
        Object object = stack[size];  
        stack[size] = null; // Pour le Garbage Collector.  
        return object;  
    }  
}
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();  
String string = "truc";  
stack.push(string);  
string = (String) stack.pop();  
// Transtypage obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();  
Integer intValue = new Integer(2);  
stack.push(intValue);  
String string = (String) stack.pop();  
// Erreur à l'exécution
```

La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();  
String string = "truc";  
stack.push(string); // Le paramètre doit être un String.  
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe Stack qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypes automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(T element) {  
        stack[size] = element; size++;  
    }  
    public T pop() {  
        size--;  
        T element = (T) stack[size];  
        stack[size] = null;  
        return element;  
    }  
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (*wrapper class*) `Integer` :

Interdit : ~~`Stack<int> stack = new Stack<>();`~~

Autorisé :

```
Stack<Integer> stack = new Stack<>();  
int intValue = 2;  
Integer integer = Integer.valueOf(intValue);  
// → emballage du int dans un Integer.  
stack.push(integer);  
Integer otherInteger = stack.pop();  
int otherIntValue = otherInteger.intValue();  
// → déballage du int présent dans le Integer.
```

Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31} - 1$
long	Long	64 bits	-2^{63} à $2^{63} - 1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	false ou true

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Classes d'emballage

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Character valueOf(char c)` (emballage)
- `public char charValue()` (déballage)
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();  
int intValue = 2;  
stack.push(intValue);  
// → emballage automatique du int dans un Integer.  
int otherIntValue = stack.pop();  
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des new soient présents dans le code.

Plusieurs paramètres de types

```
public class Pair<A, B> {  
    public final A first;  
    public final B second;  
    public Pair(A first, B second) {  
        this.first = first;  
        this.second = second;  
    }  
    public String toString() {  
        return "(" + first + ", " + second + ')';  
    }  
}
```

Utilisation:

```
Pair<String,Integer> pair = new Pair<>("toto", 12);
```

Méthodes paramétrées

Il est possible de mettre de définir des paramètres de types pour une méthode.

Il faut les déclarer avant le type de retour.

```
public class Pair<A, B> {  
    public static <C,D> Pair<C,D> makePair(C first, D second) {  
        return new Pair<C,D>(first, second);  
    }  
}
```

Utilisation:

```
Pair<String,Integer> pair = Pair.<String,Integer>makePair("toto",12);  
System.out.println(pair); // affiche (toto, 12)
```

Le *diamond* <>

Depuis Java 7, lors de l'appel d'un constructeur, il est possible de laisser les arguments de type vide (*diamond* <>) du moment que le compilateur peut déterminer (inférer) le type ou les types à partir du contexte.

```
Box<Integer> integerBox = new Box<>();  
public static <C, D> Pair<C,D> makePair(C first, D second) {  
    return new Pair<>(first, second);  
}
```

De même pour une méthode paramétrée en ne mettant même pas le *diamond*.

```
Pair<String,Integer> pair = Pair.makePair("tot",12);
```

Convention de nommage pour les types paramétrés

Par convention, les noms des paramètres de type sont des lettres majuscules.

C'est contraire bonnes pratiques visant à donner des noms détaillés aux éléments du code, mais indispensable pour différencier d'un coup d'œil les paramètres de type des classes.

Les noms les plus utilisés sont :

- E : *Element* (très utilisé pour les Collections de Java)
- K : *Key*
- N : *Number*
- T : *Type*
- V : *Value*

Section 4

Types paramétrés (notions avancées)

Condition sur les paramètres – Problématique

```
public interface Comparable<T> {  
    public int compareTo(T element);  
}  
  
class Greatest {  
    private String element;  
    public void add(String element) {  
        if (this.element==null || this.element.compareTo(element)<0)  
            this.element = element;  
    }  
    public String get() { return element; }  
}
```

Comment rendre la classe Greatest générique ?

Condition sur les paramètres

```
class Greatest<T extends Comparable<T>> {  
    private T element;  
    public void add(T element) {  
        if (this.element==null || element.compareTo(this.element)>0)  
            this.element = element;  
    }  
    public T get() {  
        return element;  
    }  
}
```

Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public T get() { return element; }  
}  
  
class Card implements Comparable<Card> { /* ... */ }  
class PrettyCard extends Card { /* ... */ }
```

Il n'est pas possible d'écrire les lignes suivantes, car PrettyCard n'implémente pas l'interface Comparable<PrettyCard> :

```
Greatest<PrettyCard> greatest = new Greatest<PrettyCard>();  
greatest.add(new PrettyCard(Card.diamond, 7));
```

Syntaxe ? super

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public T get() { return element; }  
}
```

```
class Card implements Comparable<Card> { /* ... */ }  
class PrettyCard extends Card { /* ... */ }
```

Il est possible d'écrire les lignes suivantes car PrettyCard implémente l'interface Comparable<Card> et Card super PrettyCard:

```
Greatest<PrettyCard> greatest = new Greatest<PrettyCard>();  
greatest.add(new PrettyCard(Card.diamond, 7));
```

Wildcard ?

Dans un paramètre de généricité, le symbole ? (appelé *wildcard*) dénote une variable de type anonyme.

On peut la contraindre avec les mot-clés `super` et `extends`.

Exemples :

- `List<?>` : une liste de type quelconque.
- `List<? extends Shape>` : une liste d'instances d'une sous-classe de `Shape`.
- `List<? super Disc>` : une liste d'instances d'une classe ancêtre de `Disc`.
- `E extends Comparable<? super E>` : un type `E` implémentant l'interface `Comparable<P>` pour `P` ancêtre de `E`.

? extends – Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public void addAll(List<T> list) {  
        for (T element : list) add(element);  
    }  
}
```

Il n'est pas possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();  
Greatest<Card> greatest = new Greatest<Card>();  
greatest.addAll(list);
```

? extends

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public void addAll(List<? extends T> list) {  
        for (T element : list) add(element);  
    }  
}
```

Il est maintenant possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();  
Greatest<Card> greatest = new Greatest<Card>();  
greatest.addAll(list);
```

Méthodes paramétrées et conditions sur les types

```
class Tools {  
    static <T extends Comparable<T>> boolean isSorted(T[] array) {  
        for (int i = 0; i < array.length-1; i++)  
            if (array[i].compareTo(array[i+1]) > 0)  
                return false;  
        return true;  
    }  
}
```

Exemple :

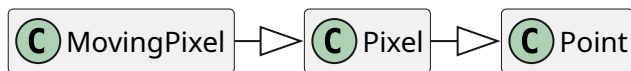
```
String[] array = { "ezjf", "aaz", "zz" };  
System.out.println(Tools.isSorted(array));
```

Méthodes paramétrées et conditions sur les types

Méthode pour copier une liste src vers une autre liste dest :

```
static <T> void copy(List<? super T> dest, List<? extends T> src)
```

On suppose qu'on a une classe MovingPixel qui étend Pixel qui elle-même étend Point.



On peut écrire :

```
List<MovingPixel> src = new ArrayList<>();  
List<Point> dest = new ArrayList<>();  
Collections.<Pixel>copy(dest, src);
```

Utilisation extends et super

Lorsqu'on a une collection d'objets de type T :

- En entrée/écriture, on veut donner des objets qui ont au moins tous les services des objets de type T.

On doit donc donner des objets dont la classe étend T : ? extends T

- En sortie/lecture, on veut récupérer des objets qui ont au plus tous les services des objets de type T.

On doit donc récupérer des objets qui sont étendu par la classe T : ? super T

Section 5

Record en Java

Mot-clé record (Java 14)

Nouvelle façon de définir des classes en Java.

```
record Rectangle(double length, double width) { }
```

Crée une classe avec :

- des attributs (*field*) `private final name` et `age` ;
- des *getters* `name()` et `age()` ;
- des méthodes `equals` et `hashCode` ;
- une méthode `toString`.

Pourquoi record ?

Pour définir une classe de données non mutables (réponses à des requêtes).

Équivalent à la classe suivante

```
public final class Rectangle {  
    private final double length;  
    private final double width;  
    public Rectangle(double length, double width) {  
        this.length = length; this.width = width;  
    }  
    double length() { return this.length; }  
    double width() { return this.width; }  
    public boolean equals(Object o) { /* ... */ }  
    public int hashCode() { /* ... */ }  
    public String toString() { /* ... */ }  
}
```

Équivalent à la classe suivante

```
public final class Rectangle {  
    // suite du transparent précédent  
    public boolean equals(Object obj) {  
        if (obj == this) return true;  
        if (obj == null || obj.getClass() != this.getClass())  
            return false;  
        var that = (Rectangle) obj;  
        return this.length == that.length &&  
            this.width == that.width;  
    }  
}
```

Équivalent à la classe suivante

```
public final class Rectangle {  
    // suite du transparent précédent  
    public int hashCode() {  
        return Objects.hash(length, width);  
    }  
    public String toString() {  
        return "Rectangle[" +  
            "length=" + length + ", " +  
            "width=" + width + ']';  
    }  
}
```

Section 6

Interfaces (notions avancées)

Les classes anonymes

Supposons que nous ayons l'interface suivante :

```
Interface ActionListener {  
    public void actionPerformed(ActionEvent event);  
}
```

Il est possible de :

- définir une classe anonyme qui implémente cette interface
- d'obtenir immédiatement une instance de cette classe

```
ActionListener listener = new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        counter++;  
    }  
};
```

Les classes anonymes

```
public class Window {  
    private int counter;  
    public Window() {  
        Button button = new Button("count");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                counter++;  
            }  
        });  
    }  
}
```

Les classes anonymes

Il est possible d'utiliser des attributs de la classe "externe" :

```
public class Window {  
    private Counter counter = new Counter();  
    public Window() {  
        Button button = new Button("count");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                counter.count();  
            }  
        });  
    }  
}
```

Les classes anonymes

Il est possible d'utiliser des variables finales de la méthode :

```
public class Window {  
    public Window() {  
        final Counter counter = new Counter();  
        Button button = new Button("count");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                counter.count();  
            }  
        });  
    }  
}
```

Lambda expressions (Java 8)

Avec Java 8, il est possible d'écrire directement :

```
public class Window {  
    public Window() {  
        Button button = new Button("button");  
        button.addActionListener(event -> System.out.println(event));  
    }  
}
```

Explication : ActionListener a une seule méthode donc on peut affecter une lambda expression (décrivant cette méthode) à une variable de type ActionListener.

```
public interface ActionListener {  
    public void actionPerformed(ActionEvent event);  
}
```

Interfaces fonctionnelles (Java 8)

Une interface n'ayant qu'une méthode abstraite est une interface fonctionnelle. Les quatre interfaces fonctionnelles suivantes (et plein d'autres) sont déjà définies :

```
public interface Predicate<T> {  
    public boolean test(T t);  
}  
  
public interface Function<T,R> {  
    public R apply(T t);  
}  
  
public interface Consumer<T> {  
    void accept(T t);  
}  
  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Syntaxe d'une lambda expression

Pour instancier une interface fonctionnelle, on peut utiliser une lambda expression :

L'interface suivante :

```
public interface MyFunctionalInterface{  
    public T myMethod(A arg1, B arg2, C arg2);  
}
```

peut être instanciée par :

```
MyFunctionalInterface fonc =  
    (arg1, arg2, arg2)  
    -> /* expression définissant le résultat de myMethod */
```

Si T est void alors l'expression peut être void comme un println.

Exemples de lambda expression

On considère une classe `Person` avec deux attributs `name` et `age` et les *getters* et *setters* associés.

On a le droit d'écrire les lambda expressions suivantes en Java :

- `person -> person.getAge() >= 18` de type `Predicate<Person>`
- `person -> person.getName()` de type `Function<Person,String>`
- `name -> System.out.println(name)` de type `Consumer<Person>`

Remarques

- Il n'est pas nécessaire de mettre le type des paramètres.
- On peut omettre les parenthèses dans le cas où il n'y a qu'un seul paramètre

Référence de méthodes

Dans un certain nombre de cas, une lambda expression se contente d'appeler une méthode ou un constructeur.

Il est plus clair dans ce cas de se référer directement à la méthode ou au constructeur.

Lambda expression	référence de méthode
<code>x -> Math.sqrt(x)</code>	<code>Math::sqrt</code>
<code>name -> System.out.println(name)</code>	<code>System.out::println</code>
<code>person -> person.getName()</code>	<code>Person::getName</code>
<code>name -> new Person(name)</code>	<code>Person::new</code>

`Stream` = Abstraction d'un flux d'éléments sur lequel on veut faire des calculs

Ce n'est pas une `Collection` d'élément, car un `Stream` ne contient pas d'élément

Création d'un `Stream` :

- À partir d'une collection comme une liste avec `list.stream()`
- À partir d'un fichier : `Files.lines(Path path)`
- À partir d'un intervalle : `IntStream.range(int start, int end)`

Exemples d'utilisation

```
public record Person(String name, int age) {}  
  
persons.stream()  
    .filter(person -> person.age() >= 18)  
    .map(Person::name)  
    .forEach(System.out::println);
```

Types des paramètres et retours des méthodes :

- `stream()` $\rightarrow$ `Stream<Person>`
- `filter(Predicate<Person>)` $\rightarrow$ `Stream<Person>`
- `map(Function<Person, String>)` $\rightarrow$ `Stream<String>`
- `forEach(Consumer<String>)` $\rightarrow$ `void`

Cycle de vie d'un Stream

Un Stream est toujours utilisé en trois phases :

- Création du Stream (à partir d'une collection, d'un fichier, ...),
- Opérations intermédiaires sur le Stream (suppression d'éléments, transformation de chaque élément, combinaison) qui prennent un Stream et renvoient un Stream,
- Une seule opération terminale du Stream (calcul de la somme, de la moyenne, application d'une fonction sans retour sur chaque élément, ...).

Opérations intermédiaires possibles sur un Stream

- `Stream<E> filter(Predicate<? super E>)` : sélectionne si un élément reste dans le Stream
- `<R> Stream<R> map(Function<? super E, ? extends R>)` : transforme les éléments du Stream en leur appliquant une fonction
- `Stream<E> sorted(Comparator<? super E>)` : trie les éléments

Opérations terminales possibles sur un Stream

- `long count()` : compte le nombre d'éléments
- `long sum()` et `double sum()` : somme les éléments (entiers ou double)
- `Stream<E> forEach(Consumer<? super E>)` : Appelle le consumer pour chaque élément
- `allMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour tous les éléments
- `anyMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour au moins un élément
- `toList()` : crée une liste avec les éléments du Stream