Expressions lambda en Java

Plan

Plan

Pourquoi les *lambdas*

Les limites du *tout objet* à la Java

- Impossible de définir une fonction en dehors d'une classe
- Impossible de passer une fonction en paramètre d'une méthode
- ⇒ utilisation des classes internes anonymes (*e.g.* GUI)
- ⇒ beaucoup de lignes pour peu de choses

Introduction des lambdas en Java 8

- ⇒ Simplifier / clarifier le code
- ⇒ syntaxe proche de la programmation fonctionnelle
- appelée aussi *closure* ▸ plus d'information

Plan

Exemple de use case
Traitements sur collections : `List<Person>`

Person.java

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

Approche 1 : créer une méthode par traitement

Exemple dans `RosterTest.java`

```
static List<Person> roster;

. . .

public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Défauts

- fortement lié à l'API de `Person`
- ne considère qu'un seul cas : plusVieuxQue

Approche 2 : créer une méthode plus générale

RosterTest.java

```
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Défauts

- toujours fortement liée à l'API de `Person`
- plus générique mais toujours liée à une recherche spécifique

Approche 3 : séparer traitement / requête

RosterTest.java

```
public static void printPersons(List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

CheckPerson.java

```
interface CheckPerson {
    boolean test(Person p);
}
```

Approche 3 : séparer traitement / requête

CheckPersonEligibleForSelectiveService.java

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

RosterTest.java

```
printPersons(roster, new CheckPersonEligibleForSelectiveService());
```

Défaut

- non liéé à l'API de `Person` mais nécessite une classe de plus

Approche 4 : utiliser une class interne anonyme

RosterTest.java

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

Défaut

- pas de nouvelle classe mais un code assez lourd

Approche 5 : utiliser une lambda comme paramètre

RosterTest.java

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
                  && p.getAge() >= 18
                  && p.getAge() <= 25
);
```

Remarque

- Nécessite que la méthode (signature) de l'approche 4 existe : `RosterTest.printPersons(List<Person>, CheckPerson)`

Approche 6 : réutiliser le JDK

à propos de `CheckPerson.java`

```
interface CheckPerson {
    boolean test(Person p);
}
```

Remarques

- Interface très simple : **une seule méthode** abstraite
- $\Rightarrow$ Interface dite ***fonctionnelle***
- plusieurs interfaces fonctionnelles existent déjà : ▸ java.util.fonction
- `java.util.function.Predicate<T>` propose un trairement équivalent à `CheckPerson` : un test sur un type

Approche 6 : réutiliser le JDK

`Predicate<T>` à la place de `CheckPerson` ⇒ `RosterTest.java`

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

utilisation ⇒

```
printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Approche 7 : plus de lambdas

Remplacer print par un autre traitement object -> void ?

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

utilisation de `java.util.function.Consumer<Y>`

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }}
```

Approche 7 :

`processPersons(List<Person>, Predicate<Person>, Consumer<Person>)`

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }}
```

Utilisation de `processPersons` avec `print`

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

Approche 7 suite
`Consumer<Y>` sur le résultat de `Function<T,R>`

processPersonsWithFunction(List<Person>, Predicate<Person>, Function<Person, String>, Consumer<String>)

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Approche 7 suite

Utilisation

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
. . .
processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Approche 8 : version générique

Même chose sur n'importe quels types :

```
public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function <X, Y> mapper,
    Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}

. . .

processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Approche 9 : Utilisation d'opérations qui acceptent des lambdas

Ces opérations existaient déjà :

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

processElements Action	Aggregate Operation
Obtain a source of objects	`Stream<E> stream()`
Filter objects that match a `Predicate` object	`Stream<T> filter(Predicate<? super T> predicate)`
Map objects to another value as specified by a `Function` object	`<R> Stream<R> map(Function<? super T,? extends R> mapper)`
Perform an action as specified by a `Consumer` object	`void forEach(Consumer<? super T> action)`

Plan

Syntaxe des lambdas

Une lambda expression est formée de la façon suivante :

- une liste de paramètres séparés par une virgule et encadrés par des parenthèses (si plus que deux ou 0) : `(a,b)` ou `a` ou `()`
- le symbole `->`
- un corps constitué d'une seule instruction ou d'un bloc encadré d'accolades. Si l'instruction est unique, sa valeur est retournée. On peut utiliser un `return` dans le cas contraire.

exemples équivalents

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
. . .
p -> {
    return p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25;
}
```

Plan

Soit le code suivant

Person.java et Roster.java

```
    public Calendar getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }

. . .

class PersonAgeComparator implements Comparator<Person> {
    public int compare(Person a, Person b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}

. . .

Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);

. . .

Arrays.sort(rosterAsArray, new PersonAgeComparator());
```

Référencement d'une méthode existante

Sachant que `Comparator` est une interface fonctionnelle

Roster.java

```
Arrays.sort(rosterAsArray, new PersonAgeComparator());

. . . //replaced by

Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);

. . . //replaced by

Arrays.sort(rosterAsArray, (a, b) -> Person.compareByAge(a, b) );

. . . //replaced by

Arrays.sort(rosterAsArray, Person::compareByAge);
```

Différents types de référencement

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Conclusion

Sur les lambdas

- Simplifient le code et apporte de la clarté (lorsqu'on les maîtrise)
- $\Rightarrow$ avec modération et avec une indentation bien pensée

Ressources Web

- ▸ *Développons en Java* sur les lambdas
- ▸ *10 Best Java Tutorials, Courses, and Books to learn Lambda*

Ce cours reprend le tutoriel d'Oracle sur ▸ l'usage des lambdas