Expressões Lambda e Operações Agregadas no Java 8

```
public class Person {

      public enum Sex {
         MALE, FEMALE
      }

      private String name;
      private int age;
      private Sex gender;

      public int getAge() {
          return age;
      }

      public String getName() {
          return name;
      }

      public Sex getGender() {
          return gender;
      }
```

```
public void printPerson() {
   System.out.println(name);
}


public static List<Person>
                        createRoster() {
   List<Person> result = new
                   ArrayList<Person>();
   result.add(new
         Person("Joao",Sex.MALE,21));
   result.add(new
        Person("Pedro",Sex.MALE,19));
   result.add(new
      Person("Maria",Sex.FEMALE,20));
   result.add(new
       Person("Rita",Sex.FEMALE,22));
   result.add(new
          Person("Rui",Sex.MALE,23));
   return result;
 }
}
```

Expressões Lambda e Operações Agregadas no Java 8

Escolher e imprimir pessoas dado um critério:

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

Invocando:

```
List<Person> roster = Person.createRoster();
printPersonsOlderThan (roster, 20);
```

Expressões Lambda e Operações Agregadas no Java 8

Escolher e imprimir pessoas dado outro critério:

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

Invocando:

```
List<Person> roster = Person.createRoster();
printPersonsWithinAgeRange (roster, 14, 30);
```

Expressões Lambda e Operações Agregadas no Java 8

Critério de escolha definido por uma classe:

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

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

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {

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

Invocando:

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

Expressões Lambda e Operações Agregadas no Java 8

Critério de escolha definido por uma classe anónima:

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

A classe anónima usada como argumento da invocação de `printPersons` implementa o interface `CheckPerson`.

Expressões Lambda e Operações Agregadas no Java 8

Uma **expressão lambda** consiste de:

- Uma *lista de parâmetros formais* entre parentesis
  - podemos omitir os tipos;
  - podemos omitir os parentesis se temos só um parâmetro;
- Uma *seta* `->`
- Um *corpo*, que consiste de uma única expressão ou bloco de instruções
  - Se for uma única expressão, o sistema de execução avalia-a e retorna o seu valor;
  - Em alternativa podemos usar uma instrução `return` (bloco entre chavetas)

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

  - Uma expressão lambda pode também representar um método `void`

```
nome -> System.out.println(nome)
```

Podemos olhar para as expressões lambda como se fossem métodos sem nome.

Expressões Lambda e Operações Agregadas no Java 8

Critério de escolha definido por uma expressão Lambda:

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

Podemos usar uma expressão Lambda porque o interface `CheckPerson` é funcional, ou seja, declara uma única operação.

A expressão lambda que damos como argumento na invocação do método representa a implementação do único método declarado no interface funcional (`CheckPerson` neste exemplo).

Expressões Lambda e Operações Agregadas no Java 8

Critério de escolha definido usando interfaces funcionais do pacote `java.util.function` *juntamente com expressões Lambda:*

```
interface Predicate<T> {
   boolean test(T t);
 }
```

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

Invocando:

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

Expressões Lambda e Operações Agregadas no Java 8

Usando mais expressões Lambda:

```
public static void processPersons(List<Person> roster,
                                  Predicate<Person> tester,
                                  Consumer<Person> block) {

        for (Person p : roster)
            if (tester.test(p))
                block.accept(p);
 }
```

```
interface Consumer<T> {
  void accept(T t);
}
```

Invocando:

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

Expressões Lambda e Operações Agregadas no Java 8

E mais ainda:

```
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);
         }
}
```

```
interface Function<T,R> {
    R apply(T t)
}
```

Invocando:

```
        processPersonsWithFunction(roster,
                                   p -> p.getGender() == Person.Sex.MALE
                                        && p.getAge() >= 18
                                        && p.getAge() <= 25,
                                   p -> p.getName(),
                                   nome -> System.out.println(nome)
                                   );
```

Expressões Lambda e Operações Agregadas no Java 8

Usar genéricos:

```
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);
         }
}
```

Invocando:

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

Lembrar que `roster` é uma `List<Person>`

Expressões Lambda e Operações Agregadas no Java 8

---

*Usar **operações agregadas** que aceitam expressões Lambda como parâmetros:*

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

Operações Agregadas (alguns exemplos)

`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)`

Expressões Lambda e Operações Agregadas no Java 8

```
for (Person p : roster) {
     System.out.println(p.getName());
 }

roster
     .stream()
     .forEach(e -> System.out.println(e.getName());
```

- Um *pipeline* é uma sequência de operações agregadas.
- Uma *stream* é uma sequência de elementos.
  - Não é uma estrutura de dados que guarda elementos; ela "transporta" valores a partir de uma fonte através de um pipeline.
- Um *pipeline* contém os seguintes componentes:
  - Uma fonte ou origem: pode ser uma coleção, um array, uma função geradora ou um canal I/O. Neste exemplo, a fonte é a coleção roster.
  - Zero ou mais *operações intermédias*. Uma operação intermédia, como um filter, produz uma nova stream.
  - Uma *operação terminal*. Produz um resultado que não é uma stream, tal como um valor primitivo (p.ex. um `double`), uma coleção ou nenhum valor (por exemplo, a operação terminal `forEach`).

Expressões Lambda e Operações Agregadas no Java 8

A média das idades das pessoas do sexo masculino na lista `roster`:

```
double average = roster
     .stream()
     .filter(p -> p.getGender() == Person.Sex.MALE)
     .mapToInt(e -> e.getAge())
     .average()
     .getAsDouble();
```

`mapToInt` retorna uma nova stream do tipo `IntStream` (contém somente inteiros). A operação aplica a função representada pelo seu parâmetro a cada elemento duma stream.

Finalmente, no exemplo atrás:

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

Expressões Lambda e Operações Agregadas no Java 8

- `Stream<T>` é um interface genérica que define uma série de operações sobre *streams* (tanto operações terminais como intermédias). Alguns exemplos:
  - `long count()`
  - `static <T> Stream <T> concat(Stream<? extends T> a, Stream<? extends T> b)`
  - `Stream <T> filter(Predicate<? super T> predicate)`
  - `void foreach(Consumer<? super T> action)`
  - `<R> Stream <R> mao(Function<? super T, ? extends R> mapper)`
  - `T reduce(T identity, BynaryOperator<T> accumulator)`
  - `Stream<T> sorted(Comparator<? super T> comparator)`
- Podemos criar `Streams` a partir de coleções usando o método default Stream<E> stream() da interface `Collection`.
- Podemos criar Streams a partir de *arrays* usando o método Arrays.stream(T[] array) da classe `Arrays`.
- A interface `Iterable<T>` tem uma implementação *default* do método `foreach`.
- Num pipeline as operações intermédias só são avaliadas quando a operação terminal é invocada – lazy evaluation.

Referências para métodos

- Usamos expressões lambda para criar métodos anónimos. No entanto há casos em que a expressão lambda não faz mais que invocar um método existente.
- Nesses casos é mais claro referirmo-nos a esse método usando o seu nome. Conseguimos isso através de ***referências a métodos***. São expressões lambda mais compactas e fáceis de ler.
  - Por exemplo,

```
roster
    ...
    .map(p -> p.getName())
    .forEach(nome -> System.out.println(nome));
```

  - Podemos usar referências para métodos como argumentos na invocação:

```
roster
    ...
    .map(Person::getName)
    .forEach(System.out::println);
```

Referências para métodos

- O operador **::** separa o nome de um objeto ou classe do nome do método. Há três casos a considerar:

    1. *objecto::metodoInstancia*
    2. *Classe::metodoEstatico*
    3. *Classe::metodoInstancia*
    4. *Classe::new*

Exemplos:

1. *objecto::metodoInstancia* – `System.out::println` onde `System.out` é um objeto e `println` o nome de um método
2. *Classe::metodoEstatico* – `Math::pow` em vez de `(x,y) -> Math.pow(x,y)`. Neste caso `Math` é o nome de uma classe e `pow` o nome de um método estático.

Referências para métodos

3. *Classe::metodoInstancia* – `String::compareToIgnoreCase` onde `String` é o nome de uma classe e `compareToIgnoreCase` é o nome de um método de instância (não **static**).
   a. sendo método de instância, precisa de um objecto como alvo de chamada;
   b. tem apenas um parâmetro;
   c. como há que comparar dois objectos, o primeiro parâmetro da função torna-se o alvo da chamada e o segundo torna-se o (único) parâmetro do método;
   d. isto é, a expressão `String::compareToIgnoreCase` tem o mesmo significado do que

```
(x, y) -> x.compareToIgnoreCase(y)
```

4. *Classe::new* – `GoSquare::new` onde `GoSquare` é o nome de uma classe. Segue-se exemplo.

Exemplo

```
public abstract class Square {
    private String name;
    public Square (String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
    public abstract void imprime();
}
```

```
public class GoSquare extends Square {
    public GoSquare (String n) {
        super(n);
    }
    public abstract void imprime(){
        System.out.println("GoSquare"
                                + getName());
    }
}
```

```
public static void main(String[] args) {

    List<Function<String, Square>> constructors = Arrays.asList(GoSquare::new,
                                                    GoToJailSquare::new);

            int choice = new Random().nextInt(constructors.size());

            Square s = constructors.get(choice).apply("Olah");

            s.imprime();
    }
```