Java 8
Lambdas e Streams

GUILHERME DE CLEVA FARTO

15º É Dia de Java
Universidade Federal de São Carlos – UFSCar
19-20 de Agosto de 2016

Instrutores

2

Guilherme de Cleva Farto

guilherme.farto@gmail.com

Ciência da Computação
Engenharia de Componentes Java
Mestrado em Comp. Aplicada

TOTVS
Analista de sistemas
Arquitetura Java e inovação

Tamires Alves da Silva

ta_alvess@yahoo.com.br

Lic. em Matemática
Análise e Desenv. de Sistemas
Sistemas para Internet
MBA em Análise e Intelig. de Negócios

TOTVS
Analista de sistemas
BA/BI e inovação

Instrutores

3

Adauto Matuzaki

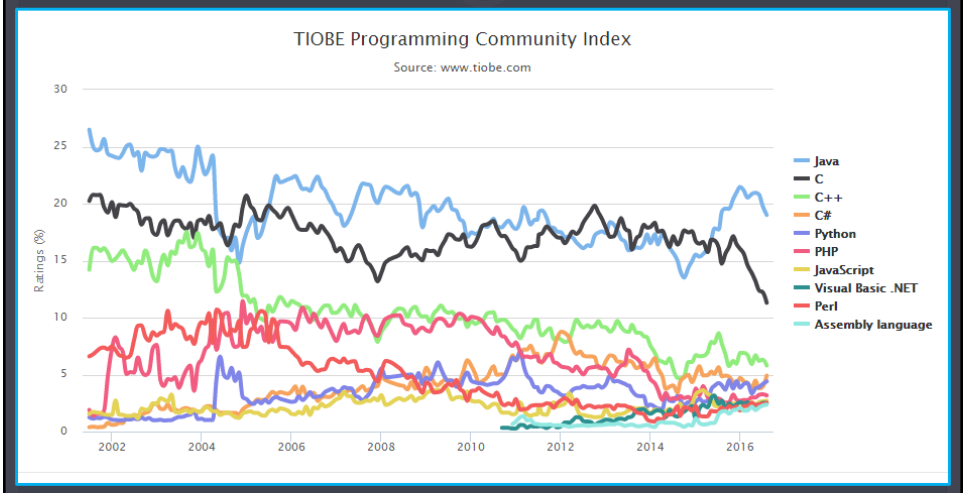
adautomatuzaki@gmail.com

Ciência da Computação
Engenharia de Componentes Java

TOTVS
Analista de sistemas
Reescrita de SRC legado
Automação agroindustrial

Por que estudar Java?

4



http://www.tiobe.com/tiobe_index - Agosto de 2016

Por que estudar Java?

5

Aug 2016	Aug 2015	Change	Programming Language	Ratings	Change
1	1		Java	19.010%	-0.26%
2	2		C	11.303%	-3.43%
3	3		C++	5.800%	-1.94%
4	4		C#	4.907%	+0.07%
5	5		Python	4.404%	+0.34%
6	7	^	PHP	3.173%	+0.44%
7	9	^	JavaScript	2.705%	+0.54%
8	8		Visual Basic .NET	2.518%	-0.19%
9	10	^	Perl	2.511%	+0.39%
10	12	^	Assembly language	2.364%	+0.60%

http://www.tiobe.com/tiobe_index - Agosto de 2016

Java 8: Top-features

Java 8: Top-features

7

Java + Javascript

Novo engine de execução dinâmica - Nashorn

Nova API de dados temporais (data e hora) – *based on Joda-Time*

Classes Clock, Duration, Period, Instant, LocalDate, LocalTime, ...

Concorrência e assincronicidade

Classes ForkJoinPool, CompletableFuture, ...

JavaFX

Diversas melhorias

What's new in JDK 8:

http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html
https://leanpub.com/whatsnewinjava8/read

Agenda do treinamento

Agenda do treinamento

9

Introdução a Lambdas
- Default Method
- Functional Interface
- Method Reference

Introdução a Streams

Lambdas

Introdução a Lambdas

11

Concepção baseada em λ-calculus

- Formalismo proposto por Alonzo Church em ~1930
- Lógica matemática para expressar computação a partir da abstração de funções
- Modelo universal de computação
  - Considerada "menor ling. de programação universal"

"A tutorial introduction to the Lambda Calculus" - Raúl Rojas
http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

Introdução a Lambdas

12

Concepção baseada em λ-calculus

- Ling. de programação abstrata em que funções podem ser combinadas para formar outras, de uma forma "pura"
- Funções são utilizadas como argumentos e retornadas como valores de outras funções
- Principal fundamento é a expressão definida por

  expr → λ var . expr | expr expr | var | (expr)

Introdução a Lambdas

13

Concepção baseada em λ-calculus

As funções no λ-calculus são escritas no formato prefixo

A avaliação da expressão Lambda procede por redução

(+ (* 5 6) (* 4 3)) =
(+ (30) (* 4 3)) =
(+ 30 (12)) =
(+ 30 12) =
42

Introdução a Lambdas

14

Impressão de Fibonacci sem Lambda

```
List<Integer> numbers =
 Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89);

for (int i = 0; i < numbers.size(); i++) {
   int number = numbers.get(i);

   System.out.println(number);
}
```

Introdução a Lambdas

15

Impressão de Fibonacci sem Lambda (Java 1.2)

```
List<Integer> numbers =
 Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89);

Iterator<Integer> iNumbers = numbers.iterator();

while (iNumbers.hasNext()) {
   Integer number = iNumbers.next();

   System.out.println(number);
}
```

Introdução a Lambdas

16

Impressão de Fibonacci sem Lambda (Java 1.5)

```
List<Integer> numbers =
 Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89);

for (Integer number : numbers) {
  System.out.println(number);
}
```

Introdução a Lambdas

17

Impressão de Fibonacci com Lambda (Java 1.8)

```
List<Integer> numbers =
 Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89);

numbers.forEach( (Integer n) -> System.out.println(n) );
```

ou

```
numbers.forEach( n -> System.out.println(n) );
```

ou

```
numbers.forEach( System.out::println );
```

Introdução a Lambdas

18

Impressão de Fibonacci com Lambda (Java 1.8)

```
List<Integer> numbers = ...

public interface List<E> extends Collection<E> {

public interface Collection<E> extends Iterable<E> {
```

Introdução a Lambdas

19

Impressão de Fibonacci com Lambda (Java 1.8)

```
public interface Iterable<T> {

   default void forEach(Consumer<? super T> action) {
      Objects.requireNonNull(action);

      for (T t : this) {
         action.accept(t);
      }
   }
```

Palavra default ?

Método em interface ?

Consumer ?

Introdução a Lambdas

20

Impressão de Fibonacci com Lambda (Java 1.8)

Classe "consumidora" de um Integer

```
public class Mostrador implements Consumer<Integer> {

   @Override
   public void accept(Integer number) {
      System.out.println(number);
   }

}
```

Introdução a Lambdas

21

Impressão de Fibonacci com Lambda (Java 1.8)

Classe "consumidora" de um Integer

```
List<Integer> numbers =
 Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89);

numbers.forEach(new Mostrador());
```

Introdução a Lambdas

22

Impressão de Fibonacci com Lambda (Java 1.8)

```
List<Integer> numbers =
 Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89);

numbers.forEach(new Consumer<Integer>() {
   @Override
   public void accept(Integer number) {
      System.out.println(number);
   }
});
```

Introdução a Lambdas

23

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe "consumidora" com Lambda ?

```
Consumer<Integer> mostrador = new Consumer<Integer>() {
    @Override
    public void accept(Integer number) {
        System.out.println(number);
    }
};
```

O que é relevante ?

Introdução a Lambdas

24

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe "consumidora" com Lambda ?

```
Consumer<Integer> mostrador = new Consumer<Integer>() {
    @Override
    public void accept(Integer number) {
        System.out.println(number);
    }
};
```

Fragmentos relevantes

Introdução a Lambdas

25

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe “consumidora” com Lambda ?

```
Consumer<Integer> mostrador = (Integer number) -> {
   System.out.println(number);
};
```

Injeção de Lambda no único método da interface Consumer

Introdução a Lambdas

26

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe “consumidora” com Lambda ?

```
Consumer<Integer> mostrador = (Integer number) -> {
   System.out.println(number);
};
```

Inferência pelo compilador ?

Introdução a Lambdas

27

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe “consumidora” com Lambda ?

```
Consumer<Integer> mostrador = number -> {
    System.out.println(number);
};
```

Introdução a Lambdas

28

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe “consumidora” com Lambda ?

```
Consumer<Integer> mostrador = number -> {
    System.out.println(number);
};
```

Remoção de { e } para blocos com uma única instrução

Introdução a Lambdas

29

Impressão de Fibonacci com Lambda (Java 1.8)

Como refatorar a classe "consumidora" com Lambda ?

```
Consumer<Integer> mostrador =
   number -> System.out.println(number);
```

então

```
numbers.forEach( number -> System.out.println(number) );
```

Lambda = expr → λ var . expr | expr expr | var | (expr)

Introdução a Lambdas

30

Definição de Expressão Lambda

Funções anônimas como parâmetros de métodos ou atribuídas a variáveis

Programação funcional

Pode conter zero, um ou mais parâmetros

(n) → ... ou (n, x) → ... ou () → ...

O tipo do parâmetro pode ser explícito, e.g. `(Integer n)`, ou inferido pelo compilador

A expr. Lambda pode conter zero, um ou mais instruções

Introdução a Lambdas

31

Definição de Expressão Lambda

Outros exemplos

```
 x -> x + 1
(x) -> x + 1
(int x) -> x + 1
(int x, int y) -> x + y
(x, y) -> x + y
(x, y) -> { System.out.println(x + y); }
() -> { System.out.println("I am a Runnable"); }
```

Introdução a Lambdas

32

Ordenação/Comparação sem Lambda (< Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

Collections.sort(pokemons, new Comparator<PokemonVO>() {
   @Override
   public int compare(PokemonVO p1, PokemonVO p2) {
      return p1.getHp().compareTo(p2.getHp());
   }
});
```

Introdução a Lambdas

33

Ordenação/Comparação com Lambda (Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

pokemons.sort(
    (p1, p2) -> p1.getHp().compareTo(p2.getHp())
);
```

Introdução a Lambdas

34

Remoção de Object sem Lambda (< Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

for (PokemonVO pokemon : pokemons) {
    if (!pokemon.hasEvolution()) {
        pokemons.remove(pokemon);
    }
}
```


```
java.util.ConcurrentModificationException
    at java.util.LinkedList$ListItr.checkForComodification(
        LinkedList.java:966)
    at java.util.LinkedList$ListItr.next(LinkedList.java:888)
    at farto.cleva.guilherme.main.Main.main(Main.java:50)
```

Introdução a Lambdas

35

Remoção de Object sem Lambda (< Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

List<PokemonVO> pokemonsFiltrados =
   new LinkedList<PokemonVO>();

for (PokemonVO pokemon : pokemons) {
   if (pokemon.hasEvolution()) {
      pokemonsFiltrados.add(pokemon);
   }
}
```

Introdução a Lambdas

36

Remoção de Object sem Lambda (< Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

Iterator<PokemonVO> iPokemons = pokemons.iterator();

while (iPokemons.hasNext()) {
   PokemonVO pokemon = iPokemons.next();

   if (!pokemon.hasEvolution()) {
      iPokemons.remove();
   }
}
```

Introdução a Lambdas

37

Remoção de Object com Lambda (Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

pokemons.removeIf(p -> !p.hasEvolution());
```

Introdução a Lambdas

38

Remoção de Object com Lambda (Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

Predicate<PokemonVO> naoPossuiEvolucao =
   p -> !p.hasEvolution();

Predicate<PokemonVO> possuiAtaqueFraco =
   p -> p.getAttack().compareTo(
      new BigDecimal("30")) < 0;

pokemons.removeIf(
   naoPossuiEvolucao.or(possuiAtaqueFraco)
);
```

Introdução a Lambdas

39

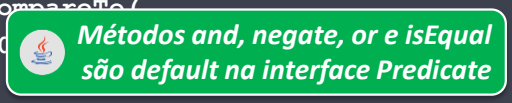
Remoção de Object com Lambda (Java 1.8)

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

Predicate<PokemonVO> naoPossuiEvolucao =
   p -> !p.hasEvolution();

Predicate<PokemonVO> possuiAtaqueFraco =
   p -> p.getAttack().compareTo(
      new BigDecimal("30

pokemons.removeIf(
   naoPossuiEvolucao.or(possuiAtaqueFraco)
);
```



Introdução a Lambdas: Default Method

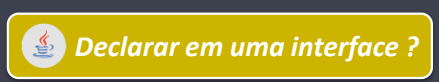
40

Revisitando Iterable<T>.forEach

```
public interface Iterable<T> {

   default void forEach(Consumer<? super T> action) {
      Objects.requireNonNull(action);

      for (T t : this) {
         action.accept(t);
      }
   }
```



Introdução a Lambdas: Default Method

41

Definição de um Default Method

- Método com uma implementação padrão em interface
- Oportuniza a adição de novas funcionalidades às interfaces de APIs e bibliotecas existentes
  - Garante a compatibilidade com versões anteriores
- Default methods podem ser sobrescritos (@Override) por uma nova interface ou implementação

Introdução a Lambdas: Functional Interface

42

```
numbers.forEach(new Consumer<Integer>() {

    @Override
    public void accept(Integer number) {
        System.out.println(number);
    }
});
```

Redução λ para

```
numbers.forEach( number -> System.out.println(number) );
```

Introdução a Lambdas: Functional Interface

43

Revisitando Consumer<T>

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t);

   default Consumer<T> andThen(
         Consumer<? super T> after) {
      Objects.requireNonNull(after);

      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

Introdução a Lambdas: Functional Interface

44

Definição de uma Functional Interface

- Interface que possui apenas um método abstrato
  - Independe da quantidade de métodos default
- Método abstrato pode ser instanciado por Lambda
- A anotação @FunctionalInterface é opcional, desde que apenas um método abstrato seja assinado (definido)
- A anotação @FunctionalInterface valida, se utilizada, a estrutura da interface funcional em tempo de compilação

Introdução a Lambdas: Functional Interface

45

Implementação de Functional Interface

```
@FunctionalInterface
public interface PokemonStats {

   public void powerUp(PokemonVO pokemon,
      BigDecimal value);

}
```

Engine de Damage/Points ?

Interface funcional válida !
Possui um único método abstrato

Anotação opcional

Introdução a Lambdas: Functional Interface

46

Implementação de Functional Interface

```
public interface PokemonStats {

   public void powerUp(PokemonVO pokemon,
      BigDecimal value);

   public void heal(PokemonVO pokemon);

}
```

Interface funcional inválida !
Possui dois métodos abstratos

λ para powerUp ou heal ?

Introdução a Lambdas: Functional Interface

47

Implementação de Functional Interface

```
@FunctionalInterface
public interface PokemonStats {

   public void powerUp(PokemonVO pokemon,
      BigDecimal value);

   public void heal(PokemonVO pokemon);

}
```

Exception !

```
Unexpected @FunctionalInterface annotation
  PokemonStats is not a functional interface
    multiple non-overriding abstract methods found
    in interface PokemonStats
```

Introdução a Lambdas: Functional Interface

48

Implementação de Functional Interface

```
@FunctionalInterface
public interface PokemonStats {

   public void powerUp(PokemonVO pokemon,
      BigDecimal value);

   default public void heal(PokemonVO pokemon) {
      pokemon.changeHp(new BigDecimal("10"));
   }

}
```

Interface funcional válida !
Possui um único método abstrato

Anotação opcional

Introdução a Lambdas: Functional Interface

49

```
PokemonStats psMinimalDamage = (p, v) -> p.changeHp(v);

PokemonStats psHugeDamage =
 (p, v) -> p.changeHp(v.multiply(new BigDecimal("2")));

PokemonTrainerVO guilherme = new PokemonTrainerVO(...

guilherme.powerUp(charmander, new BigDecimal("-10"),
   psMinimalDamage);    // -10

guilherme.powerUp(charmander, new BigDecimal("-10"),
   psHugeDamage);       // -20
```

Introdução a Lambdas: Functional Interface

50

Implementação de Functional Interface

```
@FunctionalInterface
public interface PokemonStats {

   public void powerUp...

   default public void heal...

   default public PokemonStats and(PokemonStats ps) {
      return (p, v) -> {
         powerUp(p, v); ps.powerUp(p, v);
      };
   }
}
```

Introdução a Lambdas: Functional Interface

51

```
PokemonStats psRestoreHp =
 (p, v) -> p.restoreHp(new BigDecimal("100"));

PokemonStats psIncreaseHp =
 (p, v) -> p.changeHp(new BigDecimal("25"));

guilherme.powerUp(charmander, new BigDecimal("-10"),
   psRestoreHp                // 100
   .and(psMinimalDamage)      // -10
   .and(psIncreaseHp)         // +25
   .and(psMinimalDamage)      // -10
   .and(psHugeDamage)         // -20
);
```

Introdução a Lambdas: Functional Interface

52

Functional Interfaces nativas

O pacote `java.util.Function` contém diversas interfaces funcionais nativas

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

```
BiConsumer<T, U>
Consumer<T>
Function<T, R>
Predicate<T>
Supplier<T>
UnaryOperator<T>
```

e outras

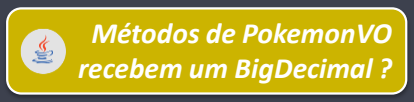
Introdução a Lambdas: Method Reference

53

Revisitando a Functional Interface implementada

```
@FunctionalInterface
public interface PokemonStats {

   public void powerUp(PokemonVO pokemon,
      BigDecimal value);
   ...
```




```
PokemonStats psMinimalDamage = (p, v) -> p.changeHp(v);

PokemonStats psMinimalDamage = PokemonVO::changeHp;
```

Introdução a Lambdas: Method Reference

54

Revisitando a Functional Interface implementada

```
@FunctionalInterface
public interface PokemonStats {

   public void powerUp(PokemonVO pokemon,
      BigDecimal value);
   ...

PokemonStats psRestoreHp = (p, v) -> p.restoreHp(v);

PokemonStats psRestoreHp = PokemonVO::restoreHp;
```

Introdução a Lambdas: Method Reference

55

Invocando Consumer<T> com Method Reference

```
pokemons.forEach((PokemonVO pokemon) -> pokemon.heal());

pokemons.forEach(pokemon -> pokemon.heal());

pokemons.forEach(PokemonVO::heal);
```

Introdução a Lambdas: Method Reference

56

Ordenação/Comparação com Method Reference

```
pokemons.sort(
    (p1, p2) -> p1.getHp().compareTo(p2.getHp())
);
```

refatorado

```
pokemons.sort(Comparator.comparing(PokemonVO::getHp));

pokemons.sort(Comparator.comparing(PokemonVO::getHp)
    .reversed());

pokemons.sort(Comparator.comparing(PokemonVO::getHp)
    .thenComparing(PokemonVO::getName));
```

Introdução a Lambdas: Method Reference

57

Manipulando List<T> com Method Reference

```
List<PokemonVO> pokemons = Pokedex.getPokemons();

List<PokemonVO> pokemonsFiltrados =
   new LinkedList<PokemonVO>();

pokemons.stream()
   .filter(p ->
      p.getHp().compareTo(new BigDecimal("30")) > 0)
   .forEach(pokemonsFiltrados::add);
```

ou

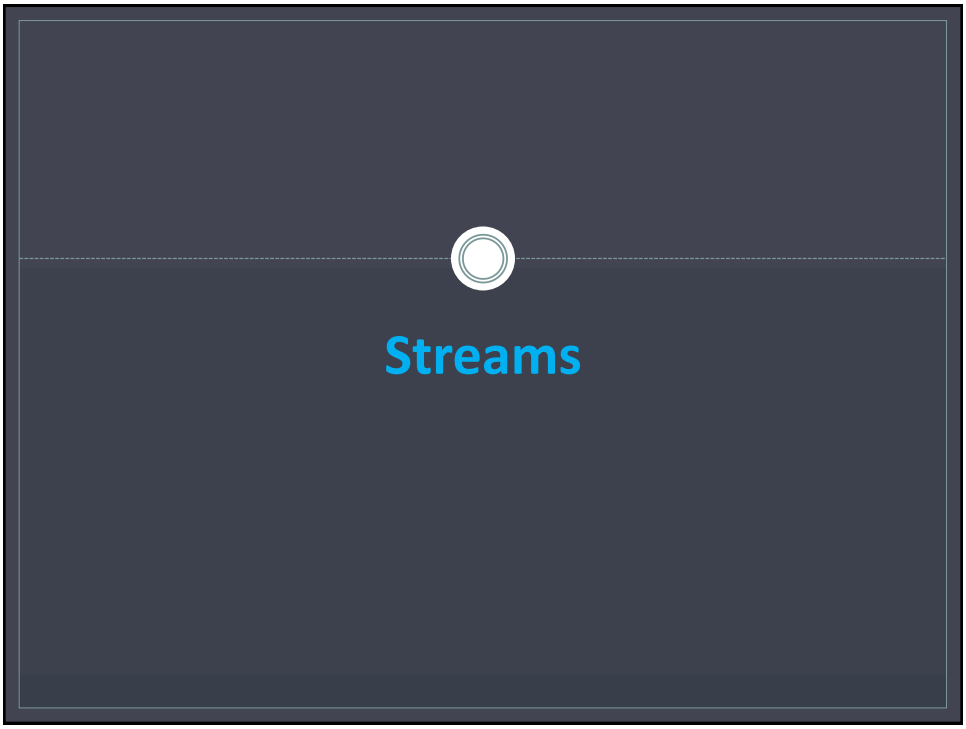
```
   .forEach(p -> pokemonsFiltrados.add(p));
```

Introdução a Lambdas: Method Reference

58

Definição de um Method Reference

- Simplifica a invocação de métodos definidos em λ
- Method Reference gera uma λ em tempo de compilação, que é "traduzida" para uma Functional Interface
- Não há uso de reflection
  - Performático: sem uso de overhead



Streams

Lambda

```
pokemons.forEach(pokemon -> pokemon.heal());
```

Lambda + Method Reference

```
pokemons.forEach(PokemonVO::heal);
```

Operações com List<E> e Collection<E>

`forEach`, `sort`, `removeIf` ...

O que é relevante ?

Streams

61

Filtrar os Pokémons que possuem evolução

```
List<PokemonVO> pokemonsFiltrados =
   new LinkedList<PokemonVO>();

for (PokemonVO pokemon : pokemons) {
   if (pokemon.hasEvolution()) {
      pokemonsFiltrados.add(pokemon);
   }
}
```

Streams

62

Ordenar os Pokémons filtrados pelo HP em ordem decresc.

```
Collections.sort(pokemonsFiltrados,
      new Comparator<PokemonVO>() {

   @Override
   public int compare(PokemonVO p1, PokemonVO p2) {
      return p1.getHp().compareTo(p2.getHp());
   }
});

Collections.reverse(pokemonsFiltrados);
```

Streams

63

Executar o método para evoluir os Pokémons

```
PokemonTrainerVO guilherme = new PokemonTrainerVO(...

for (PokemonVO pokemon : pokemonsFiltrados) {
   guilherme.evolvePokemon(pokemon);
}
```

Streams

64

Lambda + Default Method + Functional Interface
+ Method Reference + Stream

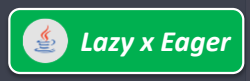
```
pokemons.stream()
    .filter(PokemonVO::hasEvolution)
    .sorted(Comparator.comparing(PokemonVO::getHp)
       .reversed())
    .forEach(p -> guilherme.evolvePokemon(p));
```

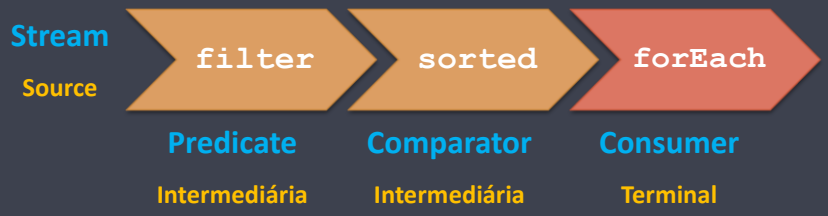
Streams

65

Uma Stream representa uma sequência de operações

- Encadeamento de operações/transações = *pipeline*
- Iniciada com um *source*
- Possui operações intermediárias e terminais





Streams

66

Execução paralelizada de uma Stream

```
pokemons.stream().parallel()
    .filter(PokemonVO::hasEvolution)
    ...
```

ou

```
pokemons.parallelStream()
    .filter(PokemonVO::hasEvolution)
    ...
```

Streams

67

Execução sequencial de uma Stream

```
pokemons.stream().sequential()
    .filter(PokemonVO::hasEvolution)
    ...
```

Streams

68

Operação terminal **Max**

```
Optional<PokemonVO> pokemonMaiorAtaque =
   pokemons.stream()
    .filter(PokemonVO::hasEvolution)
    .max(Comparator.comparing(PokemonVO::getAttack));

if (pokemonMaiorAtaque.isPresent()) {
   PokemonVO pokemon = pokemonMaiorAtaque.get();
   guilherme.evolvePokemon(pokemon);
}
```

Streams

69

Operação terminal **Max**

ou

```
pokemons.stream()
    .filter(PokemonVO::hasEvolution)
    .max(Comparator.comparing(PokemonVO::getAttack))
    .ifPresent(p -> guilherme.evolvePokemon(p));
```

Streams

70

Operação terminal **Collect**

Coletar um **List<PokemonVO>**

```
List<PokemonVO> pokemonsColetados = pokemons.stream()
    .filter(p ->
       p.getHp().compareTo(new BigDecimal("50")) > 0)
    .collect(Collectors.toList());
```

Streams

71

Operação terminal `Collect`

Coletar um `Map<Long, PokemonVO>`

```
Map<Long, PokemonVO> pokemonsColetados =
      pokemons.stream()
   .filter(p ->
      p.getHp().compareTo(new BigDecimal("50")) > 0)
   .collect(Collectors.toMap(PokemonVO::getId, p -> p));
```

ou

```
   ...
   .collect(Collectors.toMap(
      PokemonVO::getId, Function.identity()));
```

Streams

72

Operação terminal `Reduce`

Executar um `Map-Reduce` em um `List<PokemonVO>`

```
BigDecimal ataqueTotal = pokemons.stream()
   .filter(PokemonVO::hasEvolution)
   .map(PokemonVO::getAttack)
   .reduce(BigDecimal.ZERO, (x, y) -> x.add(y));
```

ou

```
   ...
   .reduce(BigDecimal.ZERO, BigDecimal::add);
```

Streams

73

Outras operações intermediárias

`flatMap`, `peek`, `distinct`, `limit`, `skip`, ...

Outras operações terminais

`find`, `match`, `count`, `min`, ...

Streams

74

Definição de Stream

- API para Collection<E> disponibilizada em `java.util.Stream`
  - *Pipeline* de operações para transformar dados
  - Não é uma estrutura de dados
- Oportuniza uma abordagem mais funcional para manipular coleções e listas
  - Menos descritivo e/ou imperativo
- Stream<T> JavaDoc

http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Dúvidas?

GUILHERME DE CLEVA FARTO

guilherme.farto@gmail.com

15º É Dia de Java
Universidade Federal de São Carlos – UFSCar
19-20 de Agosto de 2016