

Programação Funcional em Java

Prof. Bernardo Copstein

PUCRS

Leitura adicional: <https://levelup.gitconnected.com/8-years-of-java-stream-api-understand-streams-through-8-questions-bd9e5d9d8bc>



```
mirror_mod = modifier_ob.  
set mirror object to mirror.  
mirror_mod.mirror_object  
operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.  
mirror_ob.select = 0  
= bpy.context.selected_object  
data.objects[one.name].select  
print("please select exactly  
-- OPERATOR CLASSES --  
types.Operator):  
on X mirror to the selected  
object.mirror_mirror_x"  
mirror X"  
context):  
context.active_object is not
```

Introdução

- Java sempre foi uma linguagem de programação orientada a objetos
- Desde a versão 8, porém, passou a dispor de recursos de programação funcional

O que é programação funcional?

- Em uma linguagem orientada à objetos podemos enviar ou receber objetos como parâmetro de e para os métodos e podemos atribuir objetos a variáveis
- Em uma linguagem funcional podemos enviar ou receber funções como parâmetro de e para os métodos e podemos atribuir funções a variáveis
- Isso nos permite escrever algoritmos de uma forma diferente, especificando o que deve ser feito e não como deve ser feito

Funcional x Estruturado

Mostra como
selecionar os
elementos de
interesse

POO

```
List<Pessoa> lst = new ...  
for(Pessoa p:lst){  
    if (p.getIdade()>15){  
        System.out.println(p)  
    }  
}
```

FUNCIONAL

```
List<Pessoa> lst = new ...  
lst.stream()  
    .filter(p->p.getIdade()>15)  
    .foreach(System.out::println)
```

Define a
característica
dos
elementos a
serem
mostrados

Uma linguagem híbrida

- O suporte à programação funcional transformou Java em uma linguagem híbrida:
- *Suporta o paradigma orientado a objetos*
- *Suporta o paradigma funcional*
- O suporte ao paradigma funcional em Java é feito através de expressões lambda
- Não existe programação funcional em Java sem as funções lambda

Expressões lambda em Java

- Em Java funções lambda são na verdade objetos
- Correspondem a implementações concretas de um tipo especial de interface conhecidas como “interfaces funcionais”
- Expressões lambda são então implementações de interfaces funcionais

O que são interfaces funcionais?

- Interfaces funcionais são interfaces que tem apenas um único método abstrato
- Podem ter mais de um método static
- Podem ter mais de um “default method
- Exemplo:

```
interface ICalculo{  
    int calcula(int valor);  
}
```

Interfaces podem ser parâmetros

```
public class PrecoFinal{  
    private int valor;  
  
    ...  
  
    public int calcPreco(ICalculo calc){  
        int preco = calc.calcula(valor)*1.1;  
        return preco;  
    }  
}
```


Interfaces pressupõem implementações concretas

```
public class Calcula implements ICalculo{  
    public int calcula(int valor){  
        return (valor+5)/2;  
    }  
}
```

Podem ser usadas de várias formas

INSTANCIANDO A CLASSE CONCRETA

```
PrecoFinal pf = new ...  
ICalculo calc = new Calcula ...  
int p = pf.calcPreco(calc)
```

USANDO UMA CLASSE ANINHADA ANÔNIMA

```
PrecoFinal pf = new ...  
int p = pf.calcPreco(new ICalculo(){  
    int calcula(int valor){  
        return (valor+5)/2;  
    });
```

Ou usando expressões lambda

INFORMANDO A EXPRESSÃO DIRETAMENTE NO PARÂMETRO

```
PrecoFinal pr = new ...  
int r = pr.calculaPreco(v->(v+5)/2);
```

ARMAZENANDO A EXPRESSÃO EM UMA VARIÁVEL

```
ICalcula calc = v->(v+5)/2;  
PrecoFinal pr = new ...  
int r = pr.calculaPreco(calc);
```

Expressões lambda podem assumir várias formas (um parâmetro)

```
interface Duplica { int dup(int n);}
```

```
// Classe aninhada anônima
```

```
Duplica d = new Duplica() {  
    @Override  
    public int dup(int n) {  
        return n1 * 2;  
    }  
}
```

```
// Um parâmetro e mais de uma linha de código
```

```
Duplica d = n -> {  
    int resp = n*2;  
    return resp;  
}
```

```
// Um parâmetro e uma linha de código
```

```
Duplica d = n -> n*2;
```

Note o mecanismo
de inferência de
tipos de Java em
ação

Expressões lambda podem assumir várias formas (mais de um parâmetro)

```
interface Adder { int add(int n1, int n2);}
```

```
// Classe aninhada anônima
```

```
Adder adder = new Adder() {  
    @Override  
    public int add(int n1, int n2) {  
        return n1 + n2;  
    }  
}
```

```
//Vários parâmetros e mais de uma linha de código
```

```
Adder adder = (n1, n2) -> {  
    return n1 + n2;  
}
```

```
// Vários parâmetros e uma linha de código
```

```
Adder adder = (n1, n2) -> n1 + n2;
```

A anotação:

@FunctionalInterface

Se quisermos que o compilador tenha condições de verificar se uma determinada interface é realmente uma interface funcional e nos avisar sobre isso, será necessário anotar a interface como segue:

@FunctionalInterface

interface ICalculo{

int calcula(int valor);

}

Um exemplo de aplicação

- Escrever um método genérico que executa o somatório dos elementos de uma coleção que atendem a uma certa condição.
- *A condição deve ser variável*
- *A operação que define o que deve ser somado também deve ser variável*
- Exemplos:
 - *Somar o salário dos funcionários que recebem insalubridade*
 - *Somar as idades dos funcionários que tem mais de dois dependentes*
 - *Somar o custo das mensalidades dos alunos aprovados por média*
 - *Somar todos os valores impares de uma lista de números*

No exemplo iremos considerar a classe funcionário

```
public class Funcionario {
    private int matricula;
    private String nome;
    private double salarioBase;
    private int nroDependentes;
    private boolean insalubridade;

    public int getMatricula() {
    public String getNome() {
    public double getSalarioBase() {
    public int getNroDependentes() {
    public boolean getInsalubridade() {
    public void aumentaSalBase(double taxa){
    public double inss() {
    public double irpf() {
    public double getSalarioBruto() {
    public double getSalarioLiquido() {
}
```

Definindo as interfaces funcionais

```
@FunctionalInterface  
  
public interface Condicao<T> {  
    boolean verifica(T obj);  
}
```

```
@FunctionalInterface  
  
public interface Operacao<T> {  
    Double calcula(T obj);  
}
```

Definindo o método genérico

```
public <T> Double somatorio(List<T> lst, Condicao<T> condicao, Operacao<T> oper){  
    Double somatorio = 0.0;  
    for(T obj:lst){  
        if (condicao.verifica(obj)){  
            somatorio += oper.calcula(obj);  
        }  
    }  
    return somatorio;  
}
```

Usando a função ...

```
// Calcula o total gasto com impostos com os funcionários insalubres  
Condicao<Funcionario> condicao = f->f.getInsalubridade();  
Operacao<Funcionario> impostos = f->f.inss()+f.irpf();
```

```
Double gastosComImpostos = somatorio(lstf,condicao,impostos);  
System.out.println("Impostos dos insalubres: "+gastosComImpostos);
```

```
// Calcula somatório dos salários dos funcionários com mais de dois dependentes  
Condicao<Funcionario> condicao = f->f.getNroDependentes() > 2;  
Operacao<Funcionario> salarios = f->f.getSalarioLiquido();
```

```
Double gastosComSalarios = somatorio(lstf,condicao,salarios);  
System.out.println("Gastos com salarios: "+gastosComSalarios);
```

As interfaces funcionais padrão

- O exemplo da função somatório é um exemplo de como podemos criar funções extremamente genéricas explorando as funções lambda.
- O maior inconveniente entretanto é ter de criar as interfaces separadamente.
- Como estas interfaces costumam ser muito parecidas, a API de Java incorporou uma série de interfaces pré-definidas.

Detalhando as interfaces padrão

Interface	Método	Descrição
Consumer<T>	void accept(T t)	Representa uma operação com um argumento que não retorna resultado
Predicate<T>	boolean test(T t)	Representa um predicado com um argumento. Um predicado é uma expressão booleana
Function<T,R>	R apply(T t)	Representa uma operação com um argumento que retorna um resultado
Supplier<T>	T get()	Representa uma fonte de dados
BiFunction<T,U,R>	R apply(T,U)	Representa uma operação com dois argumentos que retorna um resultado
BiConsumer<T,U>	void accept(T,U)	Representa uma operação com dois argumentos que retorna um resultado

A relação completa pode ser encontrada [<aqui>](#)

Reescrevendo o exemplo com as interfaces padrão

```
// Definindo a função usando as interfaces padrão
public <T> Double somatorio(List<T> lst, Predicate<T> condicao, Function<T, Double> operacao){
    Double somatorio = 0.0;
    for(T obj:lst){
        if (condicao.test(obj)){
            somatorio += operacao.apply(obj);
        }
    }
    return somatorio;
}
```

```
// Usando a função
Predicate<Funcionario> insalubres = f->f.getInsalubridade();
Function<Funcionario, Double> impostos = f->f.inss() + f.irpf();

Double gastosComImpostos = somatorio(lstf, insalubres, impostos);
System.out.println("Gastos com impostos dos insalubres: "+gastosComImpostos);
```

Usando as funções lambda diretamente

```
// Definindo a função usando as interfaces padrão
public <T> Double somatorio(List<T> lst, Predicate<T> condicao, Function<T, Double> operacao){
    Double somatorio = 0.0;
    for(T obj:lst){
        if (condicao.test(obj)){
            somatorio += operacao.apply(obj);
        }
    }
    return somatorio;
}

// Usando a função
Double gastosComImpostos = somatorio(lstf, f->f.getInsalubridade(), f->f.inss() + f.irpf());
System.out.println("Gastos com impostos dos insalubres: "+gastosComImpostos);
```



Exercícios

Veja a lista de exercícios no Moodle

A 3D rendering of a warehouse conveyor belt system. Several cardboard boxes are shown in motion along the belt. The boxes have various labels, including a barcode and a 'FRAGILE' warning. Red laser lines are projected onto the floor and the boxes, creating a grid pattern. The scene is lit with a cool blue light, and the perspective is from a low angle looking down the length of the conveyor.

Os novos recursos da API de coleções

A nova API de coleções

- A partir do momento que a linguagem Java incorporou o conceito de funções lambda, a interface da API de coleções foi expandida de maneira a explorar esses novos recursos visando a criação de código mais eficiente e mais legível.
- Toda a interface antiga foi mantida de maneira a manter a compatibilidade e permitir seu uso quando for a solução mais adequada.

*O método **ForEach***

- Em todas as coleções da API foi acrescentado o método *ForEach*
- O método *ForEach* recebe uma interface funcional ***Consumer<T>*** por parâmetro.
- É usado para executar uma operação qualquer sobre todos os elementos da coleção

Exemplos de uso de ForEach

```
List<Funcionario> lstf = new ...
```

```
// Aumenta o salário base dos funcionários em 10%  
lstf.forEach(f->f.aumentaSalBase(1.1));
```

```
// Imprime os dados de todos os funcionários  
lstf.forEach(f->System.out.println(f));
```

Encadeando operações: a função *andThen*

A função *andThen* é um método default da interface *Consumer*.

Exemplo:

```
Consumer<Funcionario> aumentaSalario = f->f.aumentaSalBase(1.1);
```

```
Consumer<Funcionario> imprimeFuncionario = f->System.out.println(f);
```

```
lstf.forEach(aumentaSalario.andThen(imprimeFuncionario));
```

Streams

Streams ***(fluxos)***

- Até o Java 8 a API de coleções era fortemente baseada em iteradores
- Uma das grandes inovações da API foi a introdução do conceito de Streams (fluxos)
- Todos os métodos da classe *Stream* utilizam interfaces funcionais como parâmetro

O que são Streams?

- Genericamente um stream é uma sequência contínua ou fluxo de elementos.
- Do ponto de vista técnico corresponde a interface:

```
interface Stream<T> extends BaseStream<T, Stream<T>>{  
    }
```

- Em Java, a interface *Stream* representa uma sequência, infinita ou não de elementos do tipo T.
- Sequências limitadas (bounded) representam sequências finitas e ilimitadas (unbounded) sequências infinitas

O que uma stream contém?

- Streams não contém dados
- Elas puxam dados de uma fonte e os processam
- Um stream age como um cano (pipe) entre uma fonte de água (source) e uma bacia (sink)



“Unbounded” não quer dizer infinita

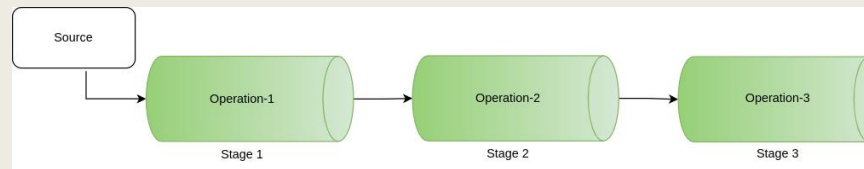
- Uma stream “unbounded” não é necessariamente infinita.
- Eventualmente pode ser infinita.
- Em outros casos pode significar apenas que no momento em que é criada não se conhece o tamanho

Quem são a “fonte” e a “bacia”?

- Em Java, a fonte e a bacia ou a **origem** e o **destino** podem ser streams, coleções ou agregadores como se verá mais adiante
- É importante entender que um stream nunca altera os dados obtidos da fonte.
- Um stream processa os dados da fonte e deposita os resultados em um destino.

Diferentes estágios em um pipeline

- Um “stream pipeline” equivale a várias operações encadeadas



- Cada estágio do pipeline processa uma operação
- Cada estágio do pipeline pode trocar o tipo do stream
- Cada estágio do pipeline serve de fonte para o próximo e de destino para o anterior

Dois tipos de operações em um pipeline

- **Operações intermediárias:** estágios intermediários que são passados para o próximo estágio
 - Mapeamentos: transformam os dados de um tipo em outro
 - Filtros: selecionam elementos a partir de predicados
- **Operações terminais:** representam a tarefa final de um pipeline. Servem para armazenar e/ou resumir os resultados.
 - Só é admitida uma operação terminal por pipeline



Em um pipeline

- Podem haver quantas operações intermediárias forem necessárias
- Só pode haver uma única operação terminal
- O processamento não se inicia até que a operação terminal seja invocada (lazy processing)
- O processamento tardio permite que os streams otimizem o gasto de memória

Obtendo as fontes de dados

(métodos básicos)

- **A partir de uma coleção:**

```
List<Pessoa> pessoas = new ArrayList<>()
```

```
Stream<Pessoa> = pessoas.stream();
```

- **A partir de elementos conhecidos**

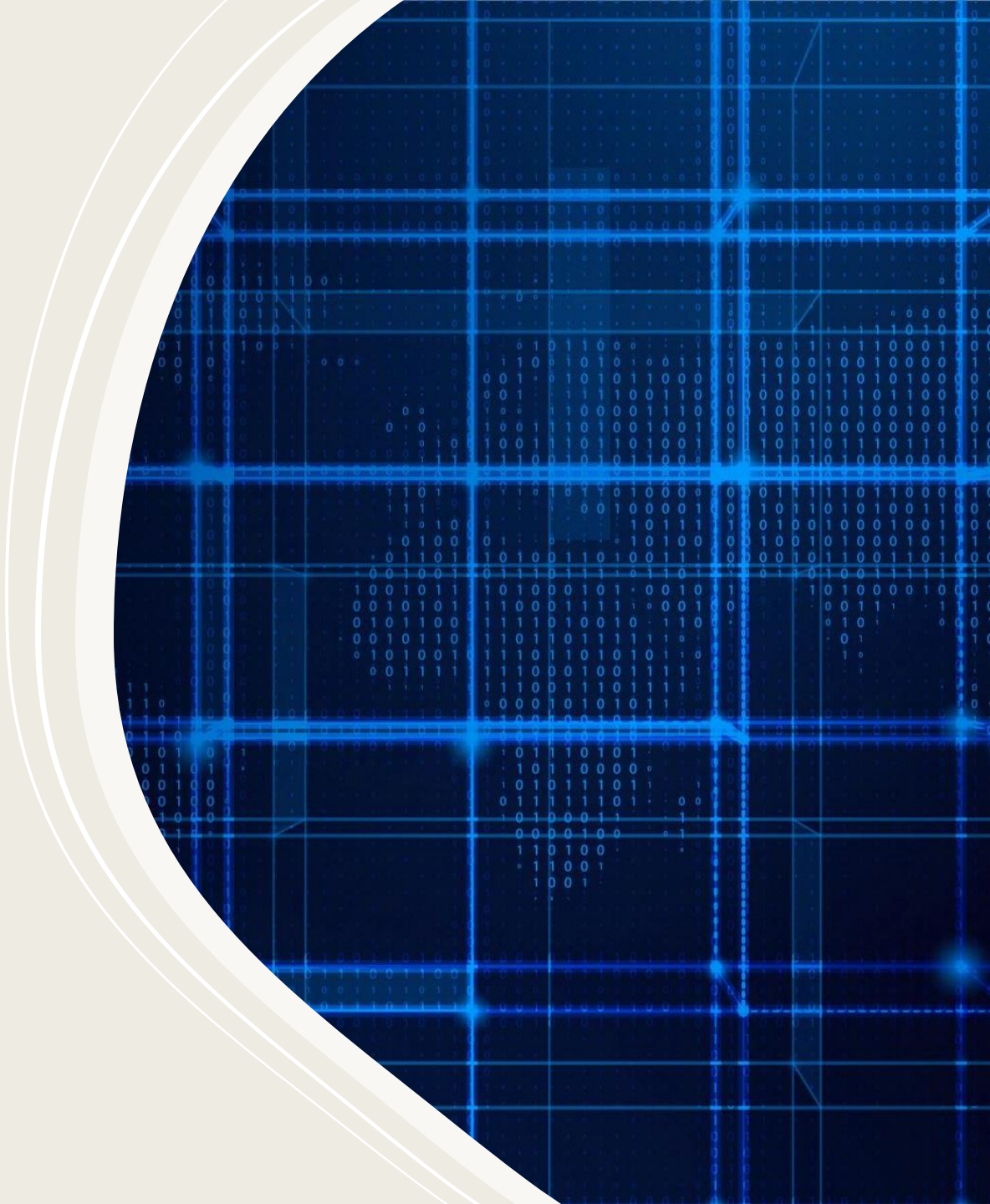
```
Stream<Integer> s = Stream.of(10,20,30,40);
```


O padrão map-filter-reduce

(operações intermediárias e finalização)

- O padrão de uso mais comum dos streams é o map-filter-reduce
- Nos exemplos que serão mostrados a seguir a fonte será uma coleção de funcionários (a classe funcionário foi apresentada anteriormente)

```
List<Funcionario> lst = new ArrayList<>();
```



Exemplo 1:

```
doublesalarioMedio = lst.stream()           // Stream<Funcionario>
    .filter(f->f.getInsalubridade())         // Stream<Funcionario>
    .mapToDouble(f->f.getSalarioLiquido())    // Stream<Double>
    .average()                               // Optional<Double>
    .getAsDouble();                          // Double
```

Exemplo 1: calcula média salarial dos funcionários insalubres

```
doublesalarioMedio = lst.stream()
```

```
.filter(f->f.getInsalubridade())
```

```
.mapToDouble(f->f.getSalarioLiquido())
```

} **Operações intermediárias**

```
.average() —————> Operação terminal (reduce)
```

```
.getAsDouble();
```

↓
→ Necessário para extrair o
"double" do *Optional*

Optionals representam valores que podem ser "null"

Exemplo 2: cria uma lista com os nomes dos funcionários que tem dependentes

```
List<String> nomes = lst.stream()           // Stream<Funcionario>
    .filter(f->f.getDependentes()>0)       // Stream<Funcionario>
    .map(f->f.getNome())                   // Stream<String>
    .collect(Collectors.toList());         // List<String>
```

Exemplo 3: selecionando partes do stream

```
List<String> nomes = lst.stream()           // Stream<Funcionario>
    .skip(1)                               // Stream<Funcionario>
    .limit(5)                              // Stream<Funcionario>
    .filter(f->f.getDependentes()>0)       // Stream<Funcionario>
    .map(f->f.getNome())                   // Stream<String>
    .toList();                             // List<String>
```

- skip(1): pula o primeiro funcionário.
- limit(5): considera apenas os próximos 5 funcionários da lista.
- toList(): terminador. Cria uma lista com o resultado.

Resumindo as operações intermediárias mais comuns

- `map`
- `mapToInt`, `mapToDouble`, `mapToLong`
- `filter`
- `skip`
- `limit`

Redutores terminadores tipo “match”

- São terminadores que retornam um booleano. Exemplos:

```
boolean temFamiliaNumerosa = lstf.stream().anyMatch(f->f.getNroDependentes()>5);
```

```
boolean naotemFamiliaNumerosa = lstf.stream().noneMatch(f->f.getNroDependentes()>5);
```

```
boolean todosGanhamMaisDe2000 = lstf.stream().allMatch((f->f.getSalarioLiquido()>2000.0));
```

Redutores terminadores tipo “find”

```
Funcionario nomeComD = lstf.stream()  
    .filter(f->f.getNome().charAt(0) == 'D')  
    .findFirst()  
    .orElse(null);
```

- Encontra a primeira ocorrência
- Retorna um *Optional*

```
Funcionario nomeComD = lstf.stream()  
    .filter(f->f.getNome().charAt(0) == 'D')  
    .findAny()  
    .orElse(null);
```

- Encontra qualquer ocorrência
- Retorna um optional

Redutores agregadores

```
double maiorSalario=lstf.stream().mapToDouble(f->f.getSalarioLiquido()).max().orElse(0.0);

double menorSalario=lstf.stream().mapToDouble(f->f.getSalarioLiquido()).min().orElse(0.0);

double mediaSalarial=lstf.stream().mapToDouble(f->f.getSalarioLiquido()).average().orElse(0.0);

double somatorio=lstf.stream().mapToDouble(f->f.getSalarioLiquido()).sum();
```




Exercícios

Veja lista de exercícios no Moodle

Outras formas de criar fontes de dados(1)

1. Stream vazio:

```
Stream<Integer> emptyStream = Stream.empty();
```

A expressão lambda corresponde ao método *get* da interface *Supplier<T>*

2. Streams infinitas

```
Stream<Integer> intStream = Stream.generate(() -> 7);  
intStream.forEach(System.out::println);
```

Stream infinito constante

```
Random random = new Random();  
Stream<Integer> randIntStream = Stream.generate(random::nextInt);  
randIntStream.forEach(System.out::println);
```

Stream infinito com valores aleatórios

Outras formas de criar fontes de dados(2)

3. Usando o método iterate

```
Stream<Integer> ints = Stream.iterate(1, i -> i <= 10, i-> i + 1);
```

Valor
inicial

Predicate<T>
(hasNext)

Function<T>
<next>

Outras formas de criar fontes de dados(3)

4. Usando a função *ints*:

// Stream ilimitado de inteiros

```
IntStream intStream = new Random().ints();
```

// Stream ilimitado de inteiros entre 1000 e 9999

```
IntStream intStream = new Random().ints( 1000, 9999);
```

// Stream limitado: 100 valores entre 1000 e 9999

```
IntStream intStream = new Random().ints(100, 1000, 9999);
```

Outras formas de criar fontes de dados(5)

5. Usando o método *chars*

```
IntStream chars = "Hey There!".chars();
```

6. Usando expressões regulares

```
Stream<String> words = Pattern.compile(" ").splitAsStream("Hey There Buddy");
```

7. Usando arquivos

```
Path booksFilePath = Path.of("src/main/resource/books.txt");  
Stream<String> bookStream = Files.lines(booksFilePath);  
bookStream.forEach(System.out::println)
```

Outras formas de criar fontes de dados(6)

8. Usando *StringBuilder*

```
Stream.Builder<String> cityStreamBuilder = Stream.builder();  
cityStreamBuilder.add("London").add("Edinburgh").add("Manchester");  
Stream<String> cityStream = cityStreamBuilder.build();
```

Terminadores para coleções (1)

Para os exemplos considere a *stream* que segue:

```
Stream<String> nameStream = Stream.of("NAME1", "NAME2", "NAME3",  
    "NAME4", "NAME5", "NAME6",  
    "NAME7", "NAME8", "NAME9",  
    "", "", " ");
```

Terminadores para coleções (2)

O método *collect* de *Stream* recebe por parâmetro o tipo de *Collector* que deve utilizar.

Exemplo 1: listas

```
List<String> names = nameStream  
.map(String::trim)  
.filter(Predicate.not(String::isEmpty))  
.collect(Collectors.toList());
```

Exemplo 2: conjuntos

```
Set<String> nameSet = nameStream  
.map(String::trim)  
.filter(Predicate.not(String::isEmpty))  
.collect(Collectors.toSet());
```



Terminadores para coleções (3)

Mapeia a própria string com seu comprimento

```
Map<String, Integer> map =  
nameStream.  
map(String::trim)  
.filter(Predicate.not(String::isEmpty))  
.collect(Collectors.toMap(s -> s,  
String::length));
```

```
Map<String, Integer> map =  
nameStream.  
map(String::trim)  
.filter(Predicate.not(String::isEmpty))  
.collect(Collectors.toMap(Function.identity(),  
String::length));
```

Ao invés de usar `s->s`, use a função identidade



Terminadores para coleções (4)

Coletando na coleção indicada:

```
List<String> names = nameStream  
    .map(s -> s.trim())  
    .filter(s -> !s.isEmpty())  
    .collect(Collectors.toCollection(LinkedList::new));
```

```
Set<String> nameSet = nameStream  
    .map(String::trim)  
    .filter(Predicate.not(String::isEmpty))  
    .collect(Collectors.toCollection(TreeSet::new));
```

Terminadores para coleções (5)

O método *joining* é útil para compor strings:

```
String s = Stream.of("one", "two", "three").  
    map(String::trim)  
    .filter(Predicate.not(String::isEmpty))  
    .collect(Collectors.joining(", "));  
System.out.println(s);
```

Resultado: one, two, three

Terminadores para coleções (6)

Use `groupBy` para organizar os valores por categoria em um dicionário.

Exemplo:

```
Map<Integer, List<Funcionario>> funcQtdDep = lstf
    .stream()
    .collect(Collectors.groupingBy(f -> f.getNroDependentes()));
```

Terminadores para coleções (7)

Neste exemplo queremos associar apenas o nome dos funcionários as quantidades de dependentes. Então usa-se uma versão com 2 parâmetros de *groupBy* que além da função de mapeamento admite outro *collector* como segundo parâmetro.

```
Map<Integer, List<String>> nomesQtdDep = lstf.stream()
    .collect(Collectors.groupingBy(f->f.getNroDependentes(),
        Collectors.mapping(f->f.getNome(), Collectors.toList())));
```

Se quisermos uma string com os nomes, ao invés de uma lista, usamos *joining*:

```
Map<Integer, String> strnomesQtdDep = lstf.stream()
    .collect(Collectors.groupingBy(f->f.getNroDependentes(),
        Collectors.mapping(f->f.getNome(), Collectors.joining(", "))));
```

Terminadores para coleções (8)

Neste exemplo ao invés de armazenarmos os nomes dos funcionários com cada quantidade de dependentes, armazenamos quantos funcionários temos para cada quantidade de dependentes:

```
Map<Integer, Long> QtdadePorQtdDep = lstf.stream()  
    .collect(Collectors.groupingBy(f->f.getNroDependentes(),  
        Collectors.mapping(f->f.getNome(), Collectors.counting())));
```

Terminadores para coleções (9)

Os métodos *groupingBy* e *mapping* recebem primeiro uma função e depois um coletor. E se quisermos o contrário: primeiro coletar e depois aplicar a função? No exemplo abaixo fazemos isso para transformar os resultados *Long* da função *counting* em *Integer*.

```
Map<Integer,Integer> QtdadeIntPorQtdDep = lstf.stream()  
    .collect(Collectors.groupingBy(f->f.getNroDependentes(),  
        Collectors.collectingAndThen(Collectors.counting(),l-> l.intValue()))));
```

FlatMap

FlatMap nos permite unir os conteúdos de dois streams. No exemplo temos um `Stream<Stream<String>>` e iremos ficar apenas com um `Stream<String>` com a concatenação dos conteúdos.

```
Stream<String> turma1 = Stream.of("Andre","Luiz","Carlos");  
Stream<String> turma2 = Stream.of("Pedro","Ana","Bernardo");  
Stream<Stream<String>> todasTurmas = Stream.of(turma1,turma2);  
Stream<String> todosAlunos = todasTurmas.flatMap(Stream->Stream);
```




Exercícios

Veja lista de exercícios no Moodle

Composição de predicados

- Supondo vários predicados, como gerar um predicado resultante da composição dos demais?

```
Predicate<Integer> positivo = n -> n >= 0;
Predicate<Integer> menorQue1000 = n -> n < 1000;
Predicate<Integer> maiorOuIgual1000 = n -> n >= 1000;
Predicate<Integer> par = n -> n % 2 == 0;
Predicate<Integer> impar = n -> ! par.test(n);
Predicate<Integer> paresPositivosMenoresQue1000 = positivo.and(par.and(menorQue1000));
Predicate<Integer> paresMenoresQue1000OuImparMaiorOuIgualQue1000 =
    (par.and(menorQue1000)).or(impar.and(maiorOuIgual1000));

List<Integer> nros = Stream.of(-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 1001, 1002, 1003, 1004, 1005).toList();
nros.stream().filter(paresPositivosMenoresQue1000).forEach(n -> System.out.print(n + ", "));
System.out.println();
nros.stream()
    .filter(paresMenoresQue1000OuImparMaiorOuIgualQue1000).forEach(n -> System.out.print(n + ", "));
System.out.println();
```

Composição de predicados armazenados em uma lista

```
List<Predicate<Integer>> filtros = new LinkedList<>();  
filtros.add(n -> n >= 0);  
filtros.add(n -> n < 1000);  
filtros.add(n -> n%2 == 0);
```

```
Predicate<Integer> combinandoComAnd = filtros.stream().reduce(Predicate::and).orElse(x -> true);  
Predicate<Integer> combinandoComOr = filtros.stream().reduce(Predicate::or).orElse(x -> false);
```

```
List<Integer> nros = Stream.of(-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 1001, 1002, 1003, 1004, 1005).toList();  
nros.stream().filter(combinandoComAnd).forEach(n -> System.out.print(n + ", "));
```



Exercícios

Veja lista de exercícios no Moodle