

Streams



Funções

No paradigma funcional, podemos passar funções como argumento para outras funções

```
public static void main(String[] args) {  
    Integer[] numeros = new Integer[] { 3, -7, 9, -2 };  
  
    Integer[] resultado1 =  
        transformaArray(numeros, Matematica::dobro);  
  
    Integer[] resultado2 =  
        transformaArray(numeros, Matematica::triplo);  
}
```

Matematica::dobro é uma referência para a função **dobro()** da classe **Matematica**

Lambdas

Quando as funções são simples, podemos passar lambdas

```
public static void main(String[] args) {
    Integer[] numeros = new Integer[] { 3, -7, 9, -2 };

    Integer[] resultado1 =
        transformaArray(numeros, (numero) -> numero * 2);

    Integer[] resultado2 =
        transformaArray(numeros, (numero) -> numero * 3);
}
```

(numero) -> numero * 3

parâmetros

corpo da função
(return)

Lambdas

Os lambdas podem dar imenso jeito para algumas coisas que aprendemos na parte de Programação por Objectos

Exemplo:

Ordenar listas de objectos

Solução - lambdas

A função `Collections.sort()` pode receber um lambda que define a regra de comparação

```
// vou ordenar por saldo  
Collections.sort(contas, (c1, c2) -> c1.getSaldo() - c2.getSaldo());
```

```
// agora vou ordenar por iban  
Collections.sort(contas, (c1, c2) -> c1.getIban().compareTo(c2.getIban()););
```

O lambda recebe dois objectos da lista e retorna:

- Um número negativo se o primeiro for menor que o segundo
- Zero se forem iguais
- Um número positivo se o primeiro for maior que o segundo

Tipos de funções

Isto funciona?

```
Collections.sort(contas, (c1, c2) -> "coisas");
```

Tipos de funções

Isto funciona?

Não!

```
Collections.sort(contas, (c1, c2) -> "coisas");
```



Este lambda recebe dois parâmetros do tipo Conta e retorna uma String

Para comparar dois objectos, o lambda tem que retornar um inteiro, lembrem-se?
(positivo se maior, zero se iguais, negativo se menor)

Tipos de funções

Os lambdas dão imensa flexibilidade mas temos que respeitar o tipo de lambda (função) que é necessário em cada caso

Parametrização

Parametrização por herança

Declara-se um método no pai (tipicamente abstracto) e implementam-se os diferentes comportamentos em cada sub-classe, fazendo override a esse método.

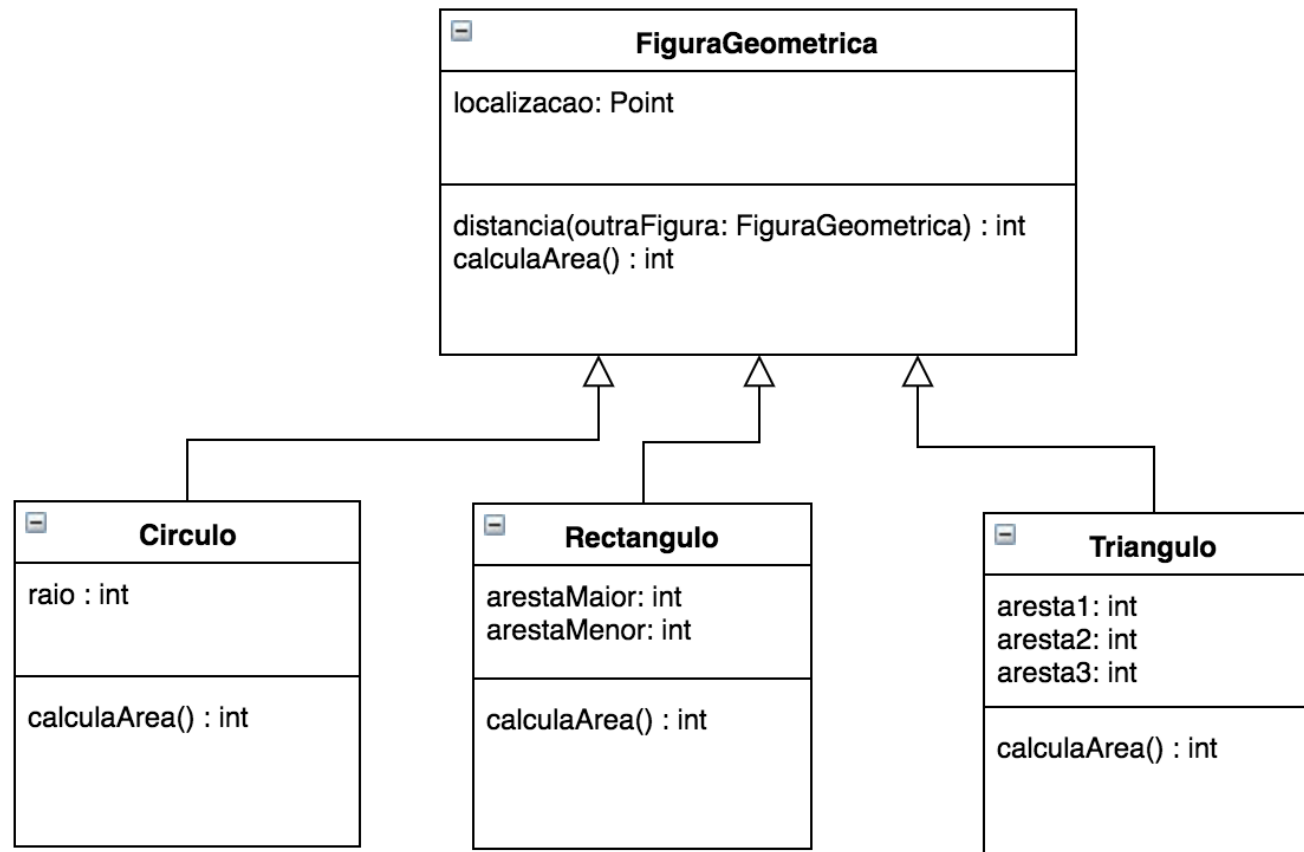
Por exemplo: para poder ter várias formas de calcular a área (consoante a figura geométrica), crio várias sub-classes (Triangulo, Rectangulo, etc.)

Parametrização por comportamento

O comportamento que pretendemos parametrizar recebe uma função genérica. Consoante o comportamento pretendido, passamos-lhe a função específica pretendida.

Por exemplo: para poder efectuar vários cálculos de juro de uma conta bancária, crio várias funções com os diferentes cálculos.

Parametrização por herança



```
class Circulo extends FiguraGeometrica {  
  
    int raio;  
  
    int calculaArea() {  
        return (int) (raio * raio * Math.PI);  
    }  
}
```

```
class Rectangulo extends FiguraGeometrica {  
  
    int arestaMaior, arestaMenor;  
  
    int calculaArea() {  
        return arestaMaior * arestaMenor;  
    }  
}
```

Parametrização por comportamento

```
public class Conta {  
  
    private int saldo;  
  
    public int calculaJurosARceber(Function<Integer,Integer> funcaoCalculo) {  
        return funcaoCalculo.apply(saldo);  
    }  
  
    // ...  
}
```

```
public static void main(String[] args) {  
  
    Function<Integer,Integer> jurosBonificados = (saldo) -> (int) (saldo * 0.2);  
    Function<Integer,Integer> semJuros = (saldo) -> saldo;  
  
    Conta2 conta1 = new Conta2();  
    int juros;  
    if (...){  
        juros = conta1.calculaJurosARceber(semJuros);  
    } else {  
        juros = conta1.calculaJurosARceber(jurosBonificados);  
    }  
}
```

Exemplo motivador

Imaginemos uma aplicação onde podemos ir guardando receitas culinárias



Lista de Receitas

```
public class Receita {  
    private String nome;  
    private String tipo;  
    private int numCalorias;  
  
    // construtores, getters e setters...  
}
```

```
ArrayList<Receita> lista = new ArrayList<>();
```

```
lista.add(new Receita("Feijoada", "portuguesa", 500));
```

```
lista.add(new Receita("Pizza Margherita", "italiana", 400));
```

```
...
```

Lista de Receitas

Como imprimir apenas as receitas pouco calóricas (< 200 calorias)?

Lista de Receitas

Como imprimir apenas as receitas pouco calóricas (< 200 calorias)?

```
for (Receita receita: lista) {  
    if (receita.getNumCalorias() < 200) {  
        System.out.println(receita.getNome());  
    }  
}
```

Fácil!

Lista de Receitas

Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

Como saber quantas receitas portuguesas têm mais do que 200 calorias?

Como saber quantos tipos de receita existem?

Estas funções são complicadas de implementar no paradigma orientado a objectos mas serão muito fáceis de implementar no paradigma funcional

Streams

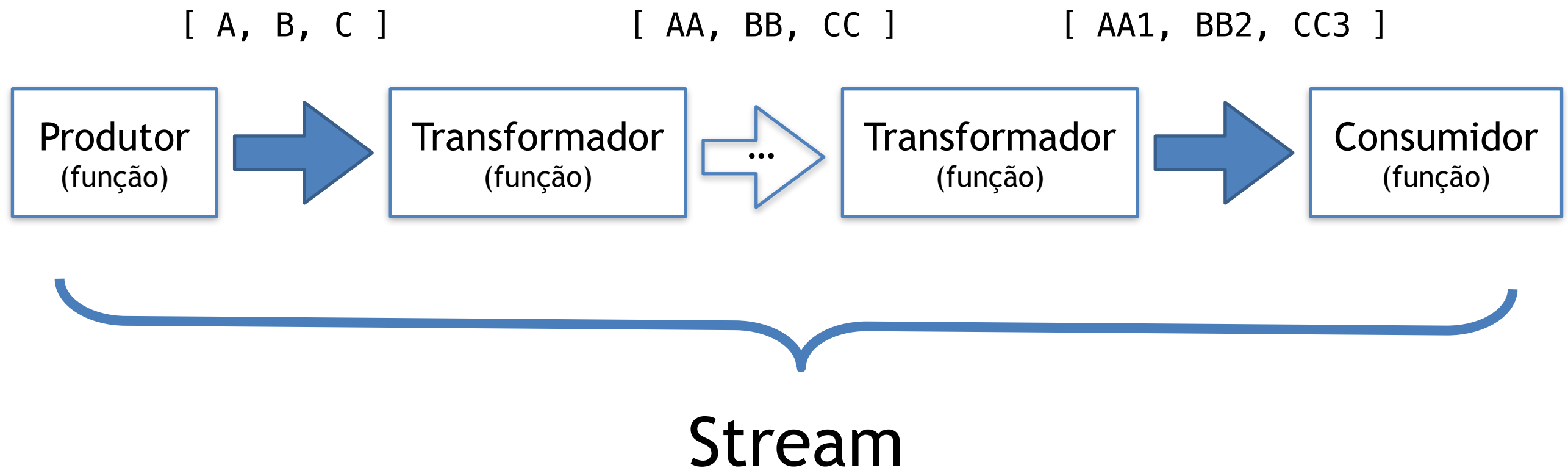


Podemos ver as Streams como uma espécie de linha de montagem através da qual passa a informação (ex: Strings, objectos, ...)

Em cada fase da linha de montagem é aplicada uma operação que vai transformando ou processando a informação que por ali passa

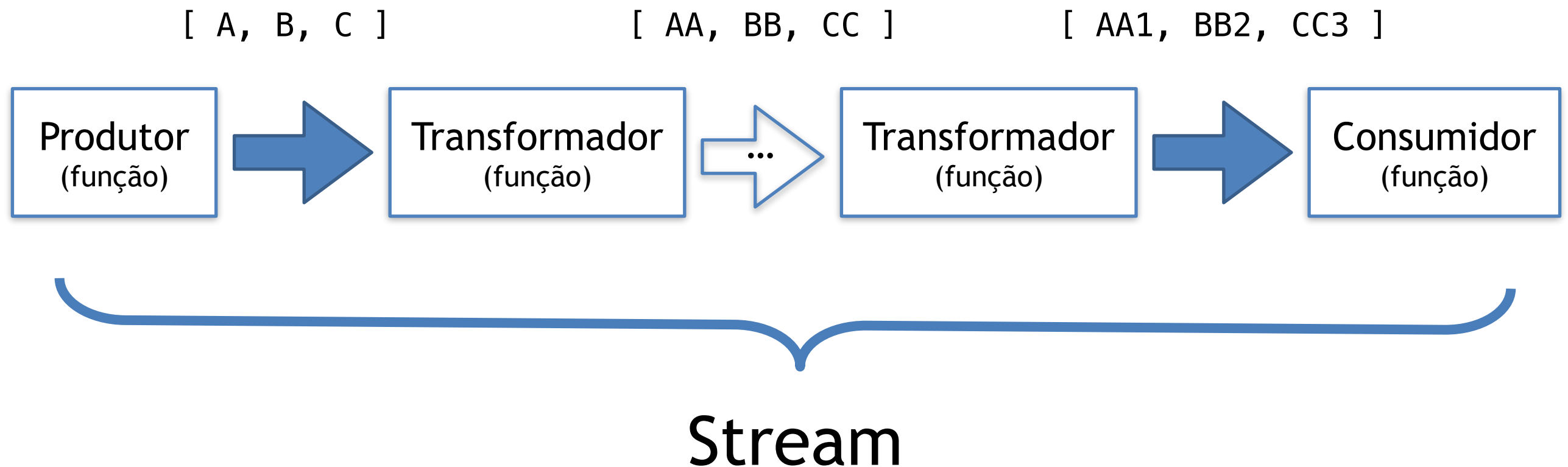
Streams

No Java 8 foi introduzido o conceito de Stream, como uma forma de representar uma composição de funções que manipulam uma sequência de dados, desde o Produtor até ao Consumidor, passando por diversas funções de transformação

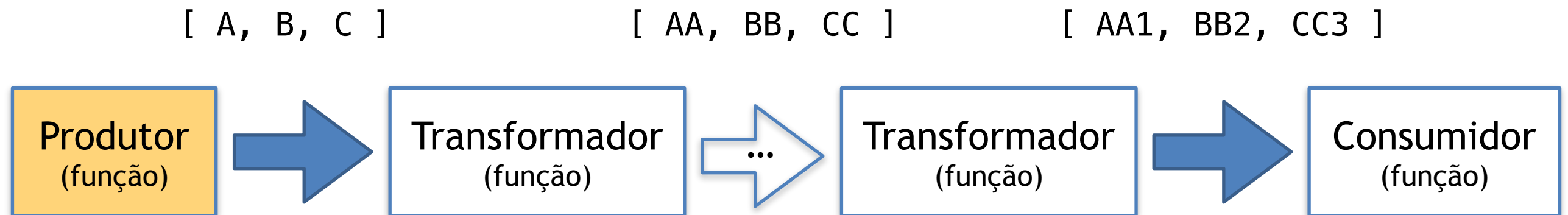


Streams

Cada etapa da “linha de montagem” tem um input e um output.
O output de uma etapa é o input da etapa seguinte.

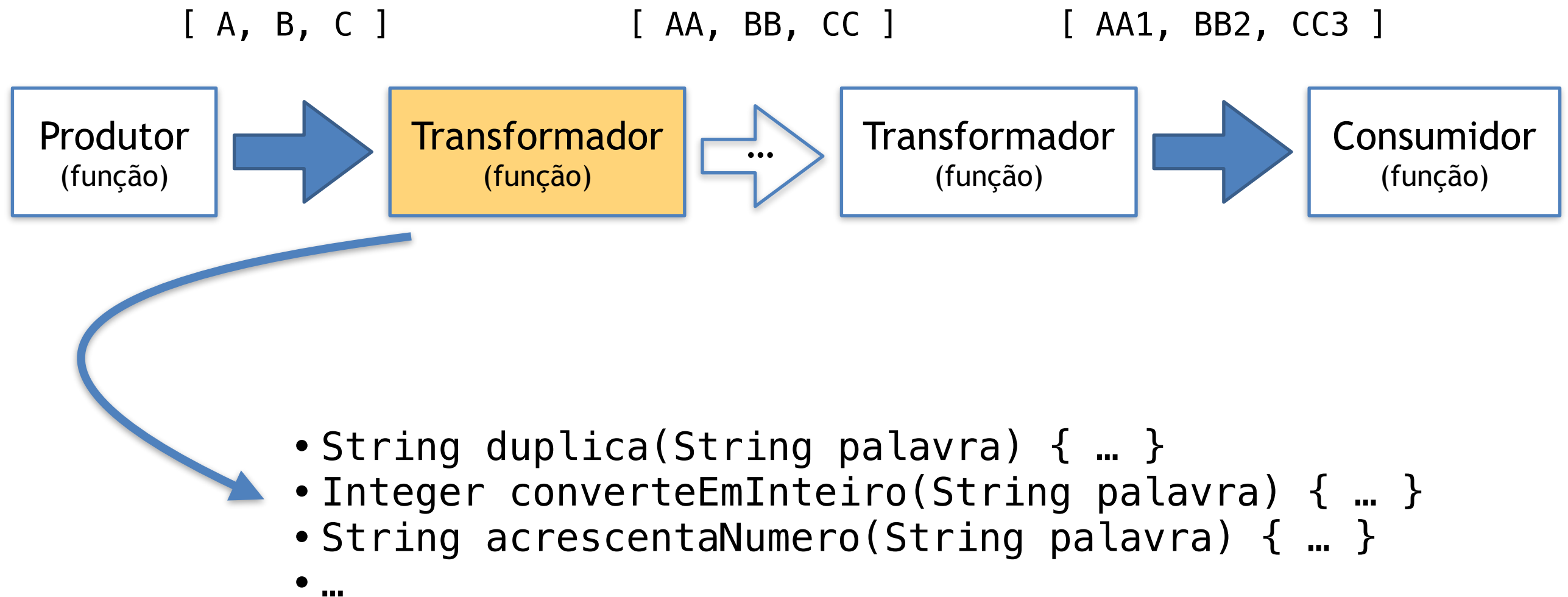


Streams



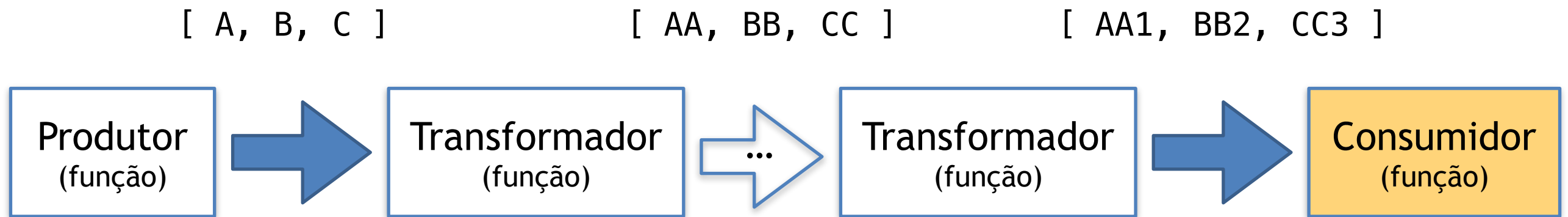
- `String[] criaArray() { ... }`
- `List<Carro> criaListaCarros() { ... }`
- `String[] leLinhasDoFicheiro() { ... }`
- `List<Post> obtemPublicacoesDoFacebook() { ... }`
- `...`

Streams



Nota: A função transformadora recebe um único parâmetro do tipo dos elementos da Stream (ex: se é uma Stream de Strings, recebe uma String). Retorna um único objecto que pode ou não ser do tipo da Stream

Streams



- `void escreveNoEcran(String palavra) { ... }`
- `void escreveNumFicheiro(String palavra) { ... }`
- `void enviaPorEmail(String palavra) { ... }`
- ...

Java 8 - Streams

O Java traz já de raiz um conjunto alargado de funções que podem actuar como produtores, transformadores e consumidores. E nós podemos criar novas funções, claro.

Exercício

Para as seguintes funções pré-existentes no Java, identifique aquelas que são produtores, transformadores ou consumidores:

1. `System.out::println`
2. `BufferedReader::lines`
3. `Math::max`
4. `Math::round`
5. `List::stream`
6. `Random::ints`
7. `Math::abs`
8. `Arrays::stream`

Resolução

Para as seguintes funções pré-existentes no Java, identifique aquelas que são Produtores, Transformadores e Consumidores:

1. `System.out::println` – Consumidor
2. `BufferedReader::lines` – Produtor
3. `Math::max` – Transformador
4. `Math::round` – Transformador
5. `List::stream` – Produtor
6. `Random::ints` – Produtor
7. `Math::abs` – Transformador
8. `Arrays::stream` – Produtor

Tipos de operações com Streams

A partir de um produtor, como é que posso compor funções de forma a processar os elementos que são produzidos?

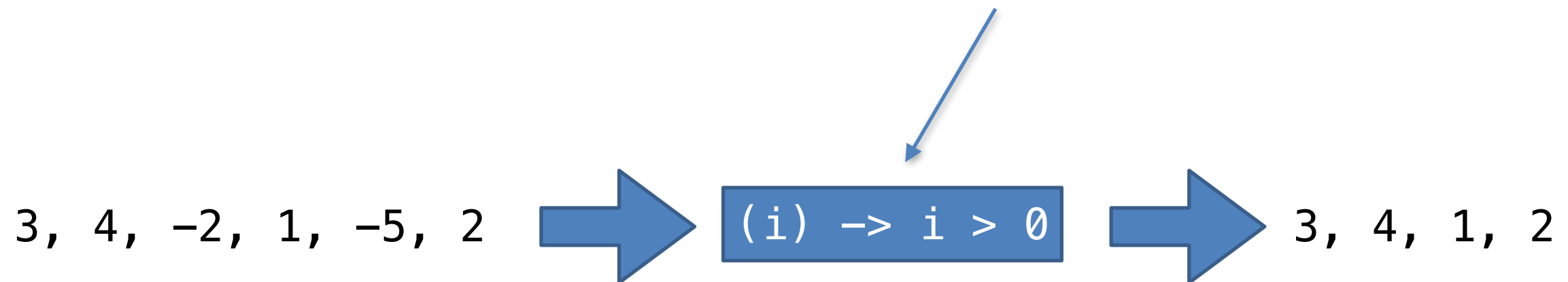
Existem 4 tipos de processamento:

- **Filtrar**
 - `stream.filter(predicado)`
- **Ordenar**
 - `stream.sorted(funcao)`
- **Transformar**
 - `stream.map(funcao)`
- **Reduzir**
 - `stream.collect(funcao)`

Filtrar streams

Quando queremos manter apenas alguns elementos da stream com base nalgum critério

```
stream.filter(predicado)
```

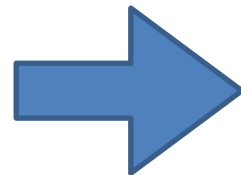


O predicado tem que ser um função que recebe um elemento da stream e retorna true se esse elemento é para manter

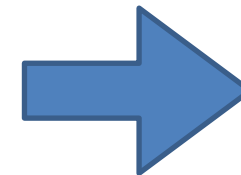
Exercício

Qual o filtro (lambda) a aplicar em cada um destes casos?

3, 4, 1, 2, 3, 6

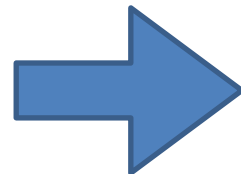


???

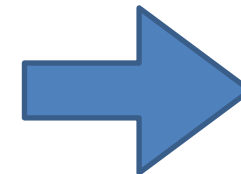


4, 2, 6

"aa", "abc", "cab", "abc"

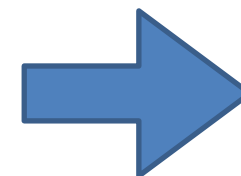


???

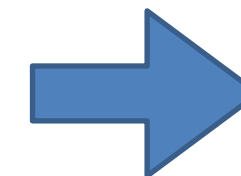


"abc", "abc"

"a", "abd", "ab", "abc"



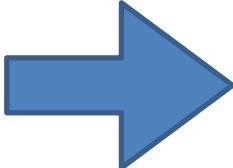
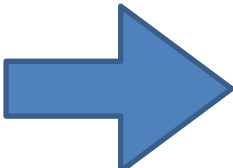
???

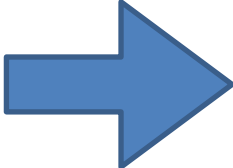
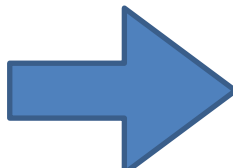


"a", "ab"

Resolução

Qual o filtro a aplicar em cada um destes casos?

3, 4, 1, 2, 3, 6  `(i) -> i%2 == 0`  4, 2, 6

"aa", "abc", "ab", "abc"  `(s) -> s.equals("abc")`  "abc", "abc"

"a", "abd", "ab", "abc"  `(s) -> s.length() < 3`  "a", "ab"

Lista de Receitas

Como imprimir apenas as receitas pouco calóricas (< 200 calorias)?

```
for (Receita receita: lista) {  
    if (receita.getNumCalorias() < 200) {  
        System.out.println(receita.getNome());  
    }  
}
```

Versão orientada a objectos

Lista de Receitas

Como imprimir apenas as receitas pouco calóricas (< 200 calorias)?

```
lista.stream()  
    .filter((r) -> r.getNumCalorias() < 200)  
    .forEach((r) -> System.out.println(r.getNome()));
```

Versão funcional

Iterar listas

Até agora, todos estes problemas implicavam:

- iterar a lista (usando ciclos)
- para cada elemento executar um conjunto de operações

```
for (Receita receita: lista) {  
    if (receita.getNumCalorias() < 200) {  
        System.out.println(receita.getNome());  
    }  
}
```

operações



Chama-se a isto iteração externa

Iterar listas

Com Streams, aplicam-se operações mas sem precisar de iterar de forma explícita

```
lista.stream()  
    .filter((r) -> r.getNumCalorias() < 200)  
    .forEach((r) -> System.out.println(r.getNome()));
```

Iterar listas

Com Streams, aplicam-se operações mas sem precisar de iterar

```
lista.stream()  
    .filter((r) -> r.getNumCalorias() < 200)  
    .forEach((r) -> System.out.println(r.getNome()));
```

operações



Chama-se a isto iteração interna

Iterar listas

Iteração externa

```
for (Receita receita: lista) {  
    if (receita.getNumCalorias() < 200) {  
        System.out.println(receita.getNome());  
    }  
}
```

Iteração interna

```
lista.stream()  
    .filter((r) -> r.getNumCalorias() < 200)  
    .forEach((r) -> System.out.println(r.getNome()));
```

Filtrar streams (distinct)

Existe um filtro especial, que permite descartar duplicados



```
lista.stream()  
    .distinct()  
    .forEach((r) -> System.out.println(r.getNome()));
```

Filtrar streams (limit)

Existe outro filtro especial, que permite preservar apenas os primeiros N elementos da stream

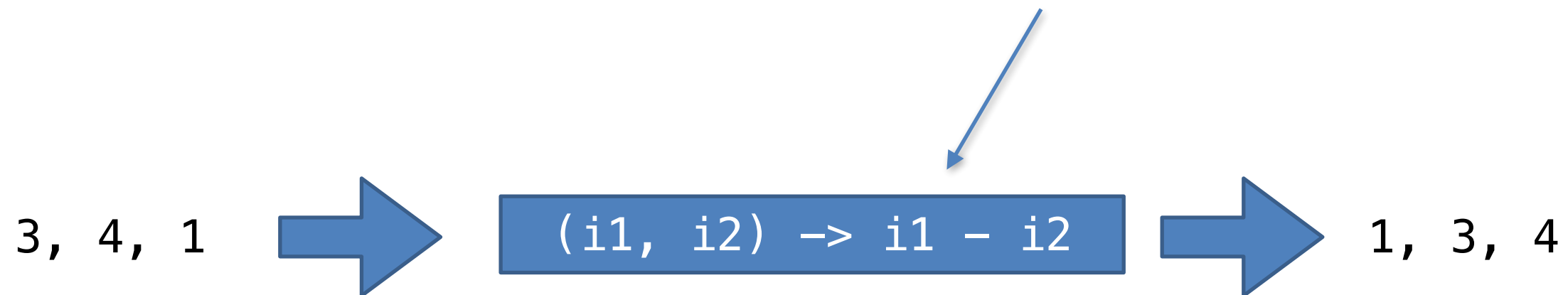


```
lista.stream()  
    .limit(3)  
    .forEach((r) -> System.out.println(r.getNome()));
```

Ordenar streams

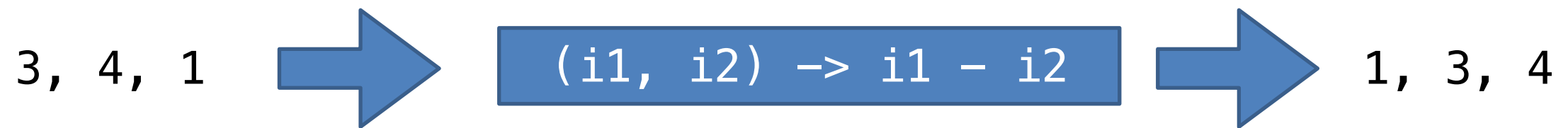
Quando queremos ordenar os elementos da stream com base numa função comparadora

`stream.sorted(comparador)`



O comparador tem que ser um função que recebe dois elementos e retorna positivo se o primeiro é maior, negativo se o primeiro é menor e zero se são iguais
(semelhante ao lambda que é passado ao `sort()`)

Ordenar streams



```
List<Integer> numeros = Arrays.asList(3, 4, 1);  
numeros.stream()  
    .sorted((i1, i2) -> i1 - i2)  
    .forEach(System.out::println);
```

Lista de Receitas

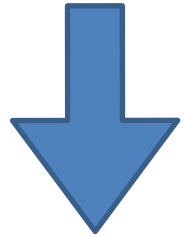
Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

Como saber quantas receitas portuguesas têm mais do que 200 calorias?

Como saber quantos tipos de receita existem?

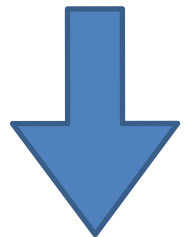
Exercício

Produtor
(lista de receitas)

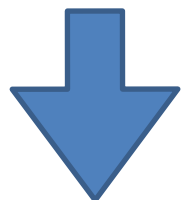


[receita1, receita2, receita3, receita4]

??



??

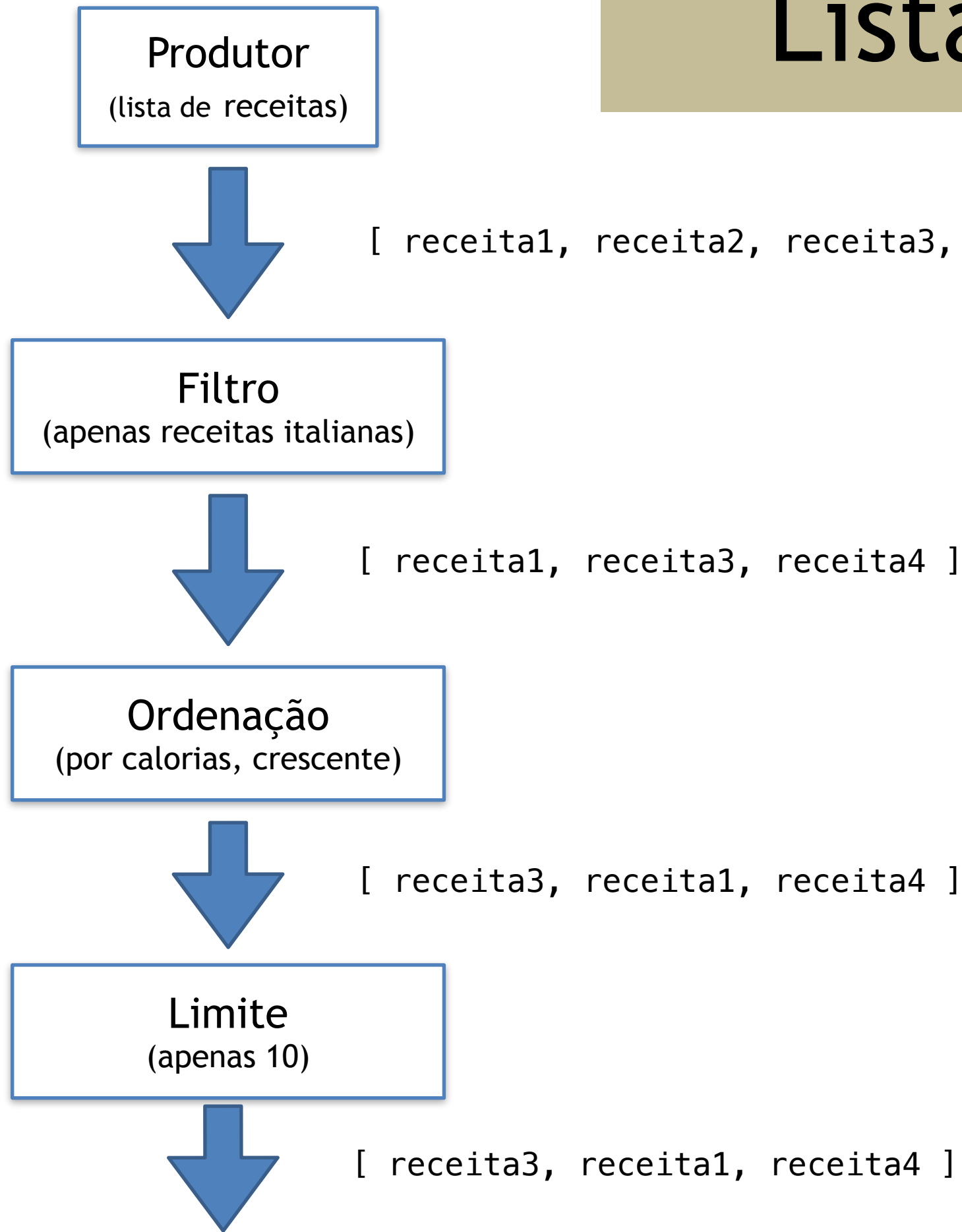


??



Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

Lista de Receitas



Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

Lista de Receitas

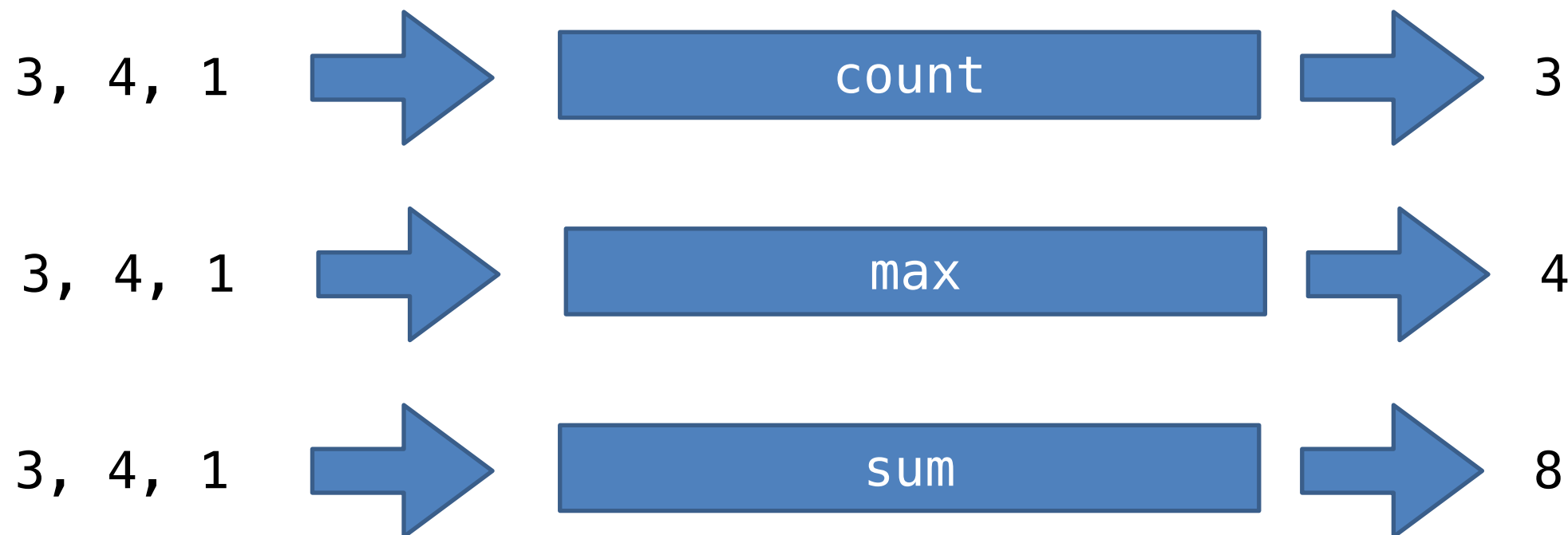
Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

```
lista.stream()  
    .filter((r) -> r.getTipo().equals("italiana"))  
    .sorted((r1, r2) -> r1.getNumCalorias() - r2.getNumCalorias())  
    .limit(10)  
    .forEach((r) -> System.out.println(r.getNome()));
```

Reduzir streams

Quando uma função transforma uma stream (uma sequência de elementos) num único elemento, diz-se que estamos a reduzir a stream

Exemplos:



Reduzir streams

Para reduzir streams, usamos o collect:

- O collect retorna um elemento (normalmente um inteiro)
- O collect tem que ser o último passo numa composição de funções, pois destrói a stream

```
List<Integer> numeros = Arrays.asList(3, 4, 1);  
  
int soma = numeros.stream()  
                  .collect(summingInt(Integer::intValue));
```

Retorno do collect

função de redução (podia ser summingInt, averagingInt, counting, maxBy, etc.)

Reduzir streams

Tipo de redução	O que faz?	Exemplo
counting()	Conta os elementos da stream	<code>stream.collect(counting())</code>
summingInt()	Soma os elementos da Stream (de inteiros)	<code>stream.collect(summingInt(Conta::getSaldo))</code>
averagingInt()	Média dos elementos da Stream (de inteiros)	<code>stream.collect(averagingInt(Conta::getSaldo))</code>
maxBy()	Máximo dos elementos da Stream	<code>stream.collect(maxBy(comparator))</code>
...

Nota: comparator é um lambda igual ao que é usado no sorted

Lista de Receitas

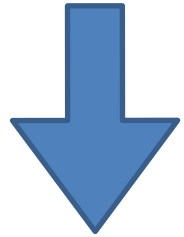
Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

Como saber quantas receitas portuguesas têm mais do que 200 calorias?

Como saber quantos tipos de receita existem?

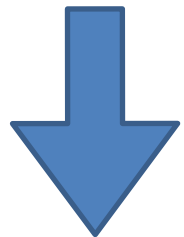
Exercício

Produtor
(lista de receitas)

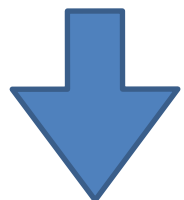


[receita1, receita2, receita3, receita4]

??



??



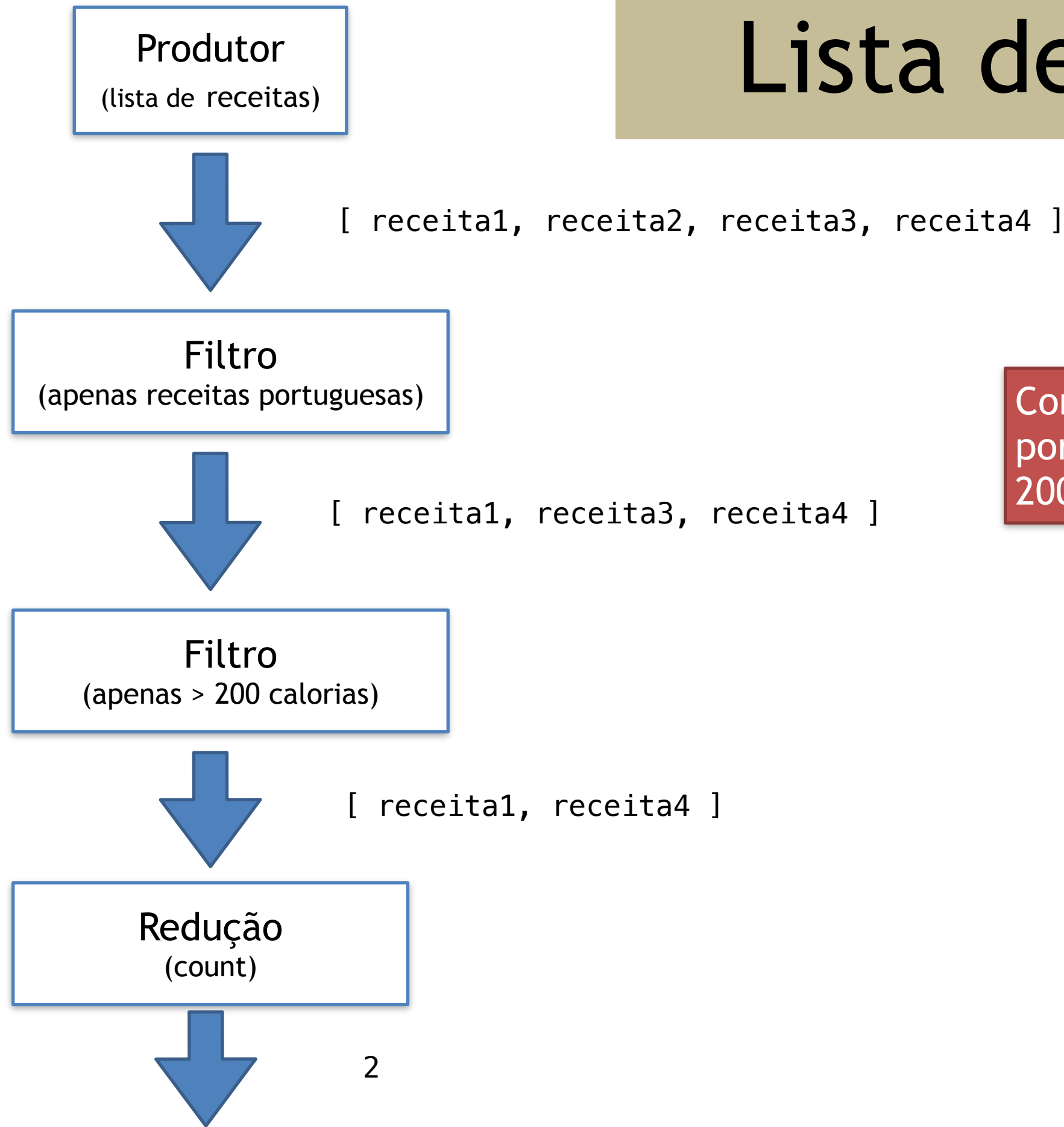
??



2

Como saber quantas receitas
portuguesas têm mais do que
200 calorias?

Lista de Receitas



Como saber quantas receitas portuguesas têm mais do que 200 calorias?

Lista de Receitas

Como saber quantas receitas portuguesas têm mais do que 200 calorias?

```
long numeroReceitas =  
    lista.stream()  
        .filter((r) -> r.getTipo().equals("portuguesa"))  
        .filter((r) -> r.getNumCalorias() > 200)  
        .collect(counting());
```

Reduzir streams

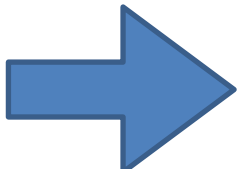

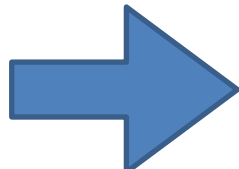
Também é possível reduzir para uma lista através da função `toList()`

```
List<Receita> receitas =  
    lista.stream()  
        .filter((r) -> r.getTipo().equals("portuguesa"))  
        .filter((r) -> r.getNumCalorias() > 200)  
        .collect(toList());
```

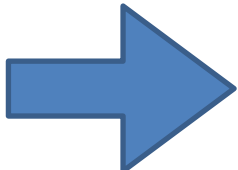
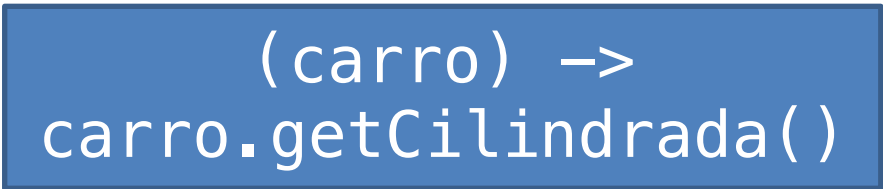
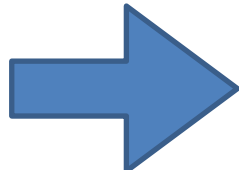
Transformar elementos das streams

Quando queremos transformar cada elemento da stream (possivelmente num tipo diferente), devemos usar a funções de mapeamento (map)

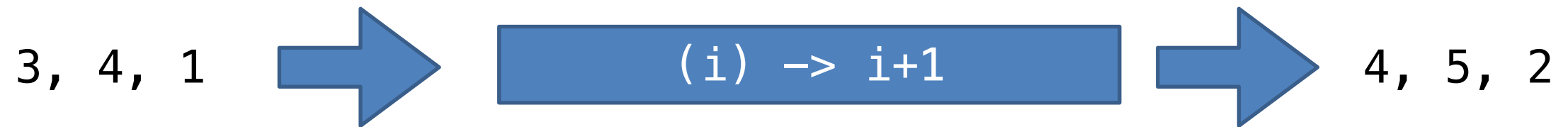
Exemplos:

3, 4, 1    4, 5, 2

3, 4, 1    "3", "4", "1"

carro1, carro2    1300, 1200

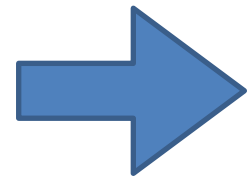
Transformar elementos das streams



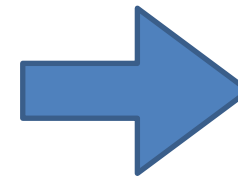
```
List<Integer> numeros = Arrays.asList(3, 4, 1);  
numeros.stream()  
    .map((i) -> i+ 1)  
    .forEach(System.out::println);
```

Transformar elementos das streams

```
new Receita("arroz de  
pato", "portuguesa",  
300),  
new Receita("empadão",  
"portuguesa", 350)
```



```
(r) -> r.getNome()
```



```
"arroz de pato",  
"empadão"
```

```
List<Receita> receitas = ...;
```

```
receitas.stream()  
    .map((r) -> r.getNome())  
    .forEach(System.out::println);
```

Lista de Receitas

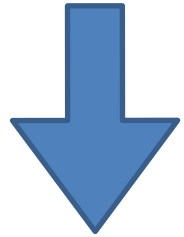
Como obter as 10 receitas italianas menos calóricas, ordenadas da menos calórica para a mais calórica?

Como saber quantas receitas portuguesas têm mais do que 200 calorias?

Como saber quantos tipos de receita existem?

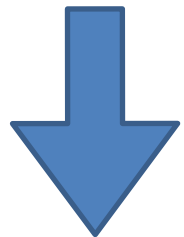
Exercício

Produtor
(lista de receitas)

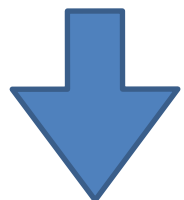


[receita1, receita2, receita3, receita4]

??



??



??

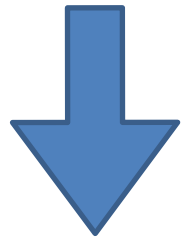


3

Como saber quantos tipos de receita existem?

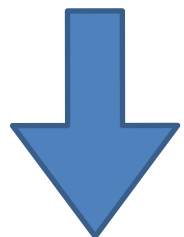
Lista de Receitas

Produtor
(lista de receitas)



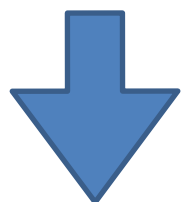
[receita1, receita2, receita3, receita4]

Map
(receita -> receita.getTipo())



["vegan", "italiano", "vegan", "japonesa"]

Filtro
(retira duplicados)



["vegan", "italiano", "japonesa"]

Redução
(count)



3

Como saber quantos tipos de
receita existem?

Lista de Receitas

Como saber quantos tipos de receita existem?

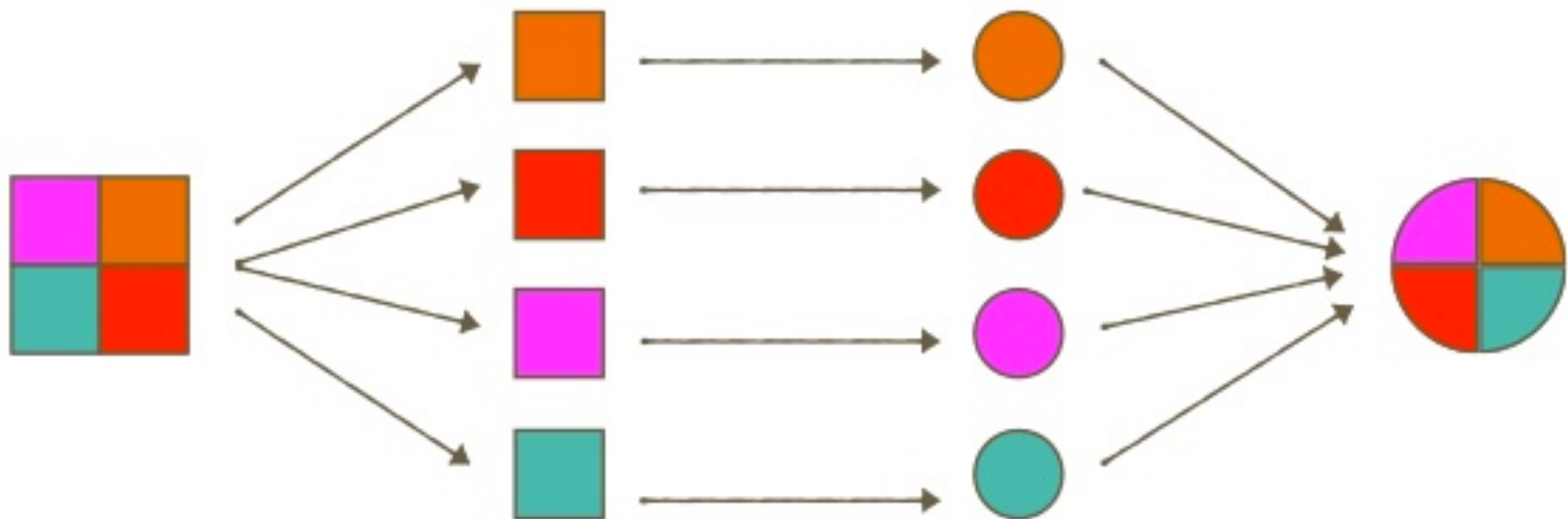
```
long numeroTiposReceita =  
    lista.stream()  
        .map((receita) -> receita.getTipo())  
        .distinct()  
        .collect(counting());
```

Resumo

Tipo de operação	O que faz?	Exemplo
filter()	Mantém apenas alguns elementos da stream	$[-3, 2, 1, -4] \longrightarrow [2, 1]$
sorted()	Ordena os elementos da stream	$[-3, 2, 1, -4] \longrightarrow [-4, -3, 1, 2]$
collect()	Reduz os elementos da stream a um único elemento	$[-3, 2, 1, -4] \longrightarrow 4$ (nº elementos)
map()	Transforma cada elemento da stream	$[-3, 2, 1, -4] \longrightarrow [-2, 1, 0, -3]$

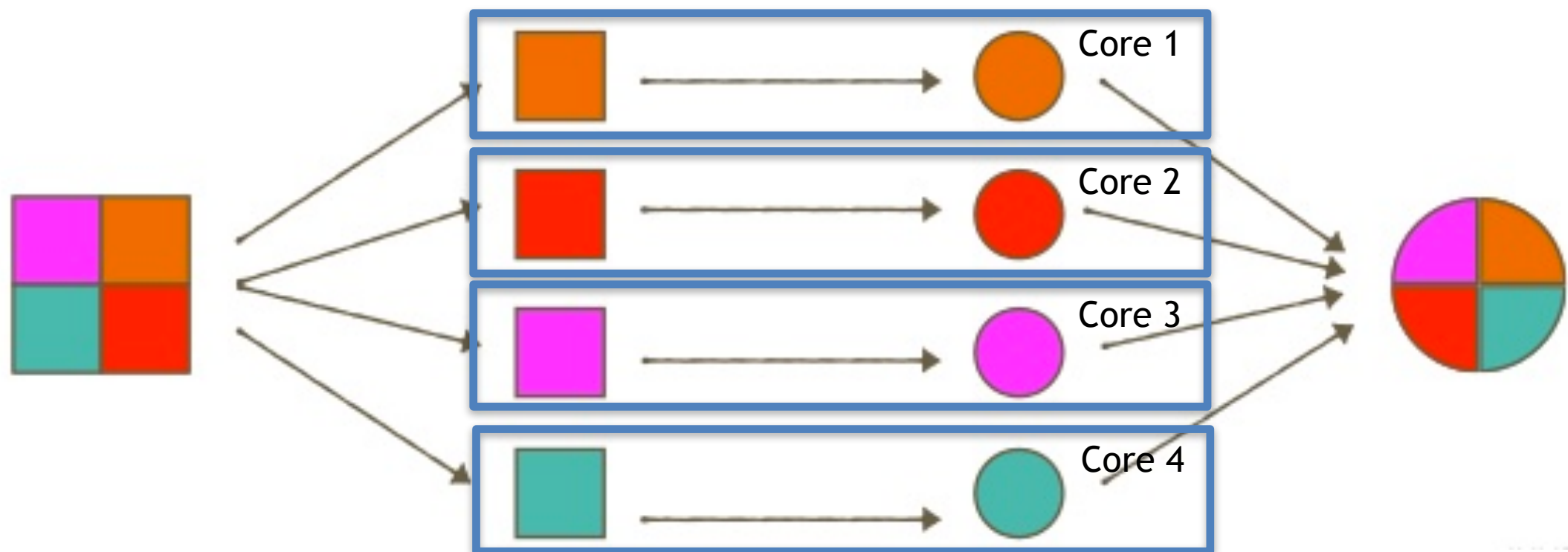
Streams - Paralelismo

A principal vantagem de uma linha de montagem é que cada etapa pode funcionar ao mesmo tempo que as outras (ex: enquanto o carroA está a pintar, o carroB pode estar a secar). Mas em computação, pode-se inclusivamente pôr a mesma etapa a processar diferentes objectos ao mesmo tempo (ex: o carroA e o carro B estão a pintar ao mesmo tempo)



Streams - Paralelismo

Isto é possível pois as Streams tiram partido das arquitecturas multi-core. Cada core fica responsável por uma das etapas da linha de montagem.



Streams - Paralelismo

Para usar paralelismo, basta usar a função `parallelStream()` em vez de `stream()`

Versão não-paralelizável

```
long numeroTiposReceita =  
    lista.stream()  
        .map((receita) -> receita.getTipo())  
        .distinct()  
        .collect(counting());
```

Versão paralelizável

```
long numeroTiposReceita =  
    lista.parallelStream()  
        .map((receita) -> receita.getTipo())  
        .distinct()  
        .collect(counting());
```

Streams

Vantagens das Streams

1. Permitem aplicar um conjunto de operações a sequências de elementos, de forma fácil e compacta
2. Permitem paralelizar o processamento dessas operações, tirando partido dos vários cores do computador

Frequência final

12 de Janeiro às 18h30 (para todos os turnos)

Salas a anunciar via Moodle

**Importante: Quem chegar atrasado
não poderá fazer a frequência**

Frequência final

A frequência cobre a matéria toda!

Na folha de teste, terão que indicar se querem a:

- Opção “avaliação contínua” - vale 50% da nota teórica fazendo média com os TPCs, os quizzes e a frequência intermédia. Tem nota mínima 8
- Não querem a opção “avaliação contínua” - vale 100% da nota e ignora os TPCs e frequência intermédia. Tem nota mínima 9,5

Nota: Só quem teve mais do 8 valores na frequência intermédia é que pode escolher a opção “avaliação contínua”.

Frequência final

A frequência começa assim:

Número: _____

Nome: _____

Pretende fazer esta frequência em regime de avaliação contínua? _____ (sim/não)

Notas prévias

- Apenas pode responder sim na pergunta acima quem teve 8 ou mais valores na frequência intermédia.
- Se respondeu sim, esta frequência terá um peso de 50% na nota final **teórica**, fazendo média com os TPCs, os quizzes e a frequência intermédia. Neste caso, a nota mínima desta frequência será 8 valores.
- Se respondeu não, esta frequência terá um peso de 100% para a nota final **teórica**, ignorando-se os TPCs, os quizzes e o teste intermédio. Neste caso, a nota mínima desta frequência será 9,5 valores.
- Utilize a folha de rascunho fornecida para preparar a resposta. Nestas folhas deverá escrever apenas a resposta definitiva de forma legível e sem rasuras.
- Preencha o seu número de aluno em todas as páginas.
- Junto a cada pergunta está a respetiva cotação entre parêntesis rectos.

Estrutura da frequência

- **Conceitos teóricos**
- **Estrutura de memória** (Map's e List's, herança, statics, ...)
- **Análise de código Java** (qual o output?, descobrir erros, completar espaços em falta na implementação)
- **Testes unitários**
- **Programação funcional** (identificar lambdas, implementar alguma funcionalidade usando streams, definir comparadores para a ordenação, ...)
- **Desenho de classes** (UML, implementar um dos métodos)

Objetivos

(apresentados na primeira aula)

Make it work

Make it right ← Foco de FP e LP1

Make it fast ← Foco de AED

-- Kent Beck

Make it easy to change ← Foco de LP2

O que conseguimos

