Princípios de Programação

Programação Funcional em Java para Programadores Haskell

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

Vasco Thudichum Vasconcelos
dezembro 2017*

Introdução

Java é uma linguagem orientada por objectos. Não quer isto dizer que não consigamos escrever código num "estilo funcional". Mas para escrever funções de ordem superior (isto é para passar uma "função" para um método) é necessário escrever bastante código.

A mais importante novidade na linguagem Java 8 permite desenvolver código "funcional" mais rapidamente, escrevendo menos código, obtendo deste modo texto mais fácil de ler. Claro que continuamos com uma linguagem orientada por objetos e, por isso, a nova e concisa sintaxe é suportada pelos conceitos básicos da linguagem, como sejam objectos, classes e interfaces.

Funções simples em Java

Comecemos com uma função Haskell simples.

```
dobro :: Int -> Int
dobro x = 2 * x
```

Como simular esta função em Java? O conceito mais próximo de uma função é um método. Mas métodos só podem aparecer dentro de classes. Temos então de criar uma classe na qual escrevemos o nosso método. Qualquer coisa como:

*Pequenos ajustes em dezembro de 2018 e em dezembro de 2019.

```
class Dobro {
  int dobro (int x) {
    return 2 * x;
  }
}
```

Para correr uma função em Haskell escrevemos

```
ghci> dobro 3
6
```

Para obter um efeito semelhante em Java, criamos um objecto da classe `Dobro` e chamamos o método `dobro`.

```
System.out.println(new Dobro().dobro(3));
```

Também poderiamos ter resolvido o problema com um método de classe (`static`). A escolha que fizémos está alinhada com o desenvolvimento que se segue.

Funções de ordem superior em Java à moda antiga

Aplicar duas vezes a mesma função Uma das primeiras funções de ordem superior que estudámos foi a função "aplicar duas vezes". Esta função recebe uma função e um elemento como parâmetros, aplica a função ao parâmetro e aplica novamente a função ao resultado. Em Haskell é simples:

```
aplicar2Vezes :: (a -> a) -> a -> a
aplicar2Vezes f x = f (f x)
```

Como exercitamos a função `aplicar2Vezes`? Em Haskell é simples:

```
ghci> aplicar2Vezes dobro 3
12
```

E como escrevemos esta função em Java? Seguindo a estratégia da função `dobro`, escrevemos uma classe com um único método `aplicar2Vezes`. O método `aplicar2Vezes` recebe uma função `f` e um parâmetro `x`. De que tipo é o parâmetro? De um tipo genérico `A`. E a função? Trata-se de uma função que recebe um `A` e devolve um `A`. Mas como escrevemos em Java uma função que recebe um elemento de um dado tipo e devolve um elemento de um outro tipo? Isto é, como escrevemos em Java o tipo Haskell `a -> b`?

Vimos que funções são métodos, e métodos aparecem em classes ou interfaces. Neste caso, e porque estamos apenas interessados no tipo (e não na sua implementação) escolhemos uma interface. Precisamos de um nome para a interface e um nome para a sua única função. Que tal chamarmos `Funcao` à classe e `aplicar` ao método? Eis o resultado:

```
public interface Funcao<A, B> {
  B aplicar (A x);
}
```

Agora já podemos escrever o método `aplicar2Vezes`:

```
<A> A aplicar2Vezes (Funcao<A,A> f, A x) {
  return f.aplicar(f.aplicar(x));
}
```

O tipo do método pode ser difícil de ler, mas conceptualmente, não é mais do que uma função que recebe dois parâmetros, um do tipo `a->a` e o outro do tipo `a`, e que devolve um objecto do tipo `a`, o que pode ser escrito `(a -> a, a)-> a`. A diferença é que o Java obriga à introdução explícita da variável de tipo (`<A>`[1]), mistura o tipo com o nome dos argumentos (`A x`) e escreve à esquerda da assinatura o tipo de retorno (enquanto que o Haskell o escreve à direita).

E como aplicamos o método `aplicar2Vezes` à função `dobro` e ao número `3`? A classe `Dobro` definida acima tem se ser adaptada à sua nova interface `Funcao`. Qualquer coisa como:

```
class Dobro implements Funcao<Integer, Integer> {
  @Override
  Integer aplicar(Integer x) {
    return 2 * x;
  }
}
```

Agora já podemos calcular o valor resultante de aplicar duas vezes a função dobro ao número 3:

```
System.out.println(new Aplicar2Vezes().aplicar2Vezes(new
    Dobro(), 3));
```

Em resumo, para fazer algo tão simples como escrever uma função de ordem superior tivémos de escrever uma interface (`Funcao`), um método (`aplicar2vezes`), e de adaptar a classe `Dobro` de modo a implementar a interface. Nada simples!

Filtrar elementos de uma lista Imaginem que pretendemos reter apenas os elementos impares de uma dada lista. Em Haskell escrevemos a função `impar` (ou usamos a função **`odd`** do Prelude):

```
impar :: Int -> Bool
impar x = x `mod` 2 == 1
```

e para filtrar os elementos de uma lista podemos escrever:

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> filter impar [1, 5, 2, 6, 8, 3, 4, 6]
[1,5,3]
```

[1] A *convenção* Java para as variáveis de tipo aponta para letras maiúsculas. Em Haskell as variáveis de tipo *têm* de ser escritas com letras minúsculas.

Em Java... Em Java começamos por escrever o predicado `impar`. Terá de ser um método numa dada classe. Mas como o predicado vai ter de ser passado como parâmetro, começamos por escrever uma interface que descreve um predicado genérico, tal como fizémos no caso das funções. Neste caso queremos um predicado `a -> Bool`. Para o nome da interface escolhemos `Predicado`; para o nome do método escolhemos `testar`.

```
public interface Predicado<A> {
  boolean testar (A x);
}
```

Agora já podemos escrever o predicado, perdão a classe, `Impar`. Notem que o método chama-se `testar` para estar de acordo com a interface.

```
class Impar implements Predicado<Integer> {
  @Override
  boolean testar(Integer x) {
    return x % 2 == 1;
  }
}
```

E como escrevemos o método `filter :: (a -> Bool)-> [a] -> [a]`? Terá de ser um método que recebe dois parâmetros: um predicado do tipo `A` e uma lista de `A`s. Uma lista de `A`s poderia ser um objeto do tipo `List<A>`. No entanto, usamos uma `Collection` de modo a tornar o nosso código mais genérico. Eis o nosso método:

```
<A> Collection<A> filtrar (Predicado<A> pred, Collection<
    A> colecao) {
  Collection<A> resultado = new ArrayList<A>();
  for (A elemento : colecao)
    if (pred.testar(elemento))
      resultado.add(elemento);
  return resultado;
}
```

Finalmente, para reter os elementos impares de uma lista, escrevemos

```
Collection<Integer> lista = Arrays.asList
    (1,5,2,6,8,3,4,6);
System.out.println(new Filtrar().filtrar(new Impar(),
    lista));
```

Funções de ordem superior à moda de Java 8

Vimos que, sem apoio especial, a escrita de funções de ordem superior requer a escrita de muito código. A versão 8 da linguagem Java trouxe três novidades que ajudam a reduzir o código que é necessário escrever. A saber:

- Um novo pacote de *interfaces funcionais*, `java.util.function`,

- Sintaxe para uma escrita compacta de funções anónimas (ou lambda), e
- Referências para métodos.

No pacote java.util.function encontramos uma série de interfaces, incluindo `Function` e `Predicate`, com a mesma funcionalidade das interfaces introduzidas acima, mas com nomes em inglês.

Na interface `Function<T,R>`, o parâmetro de tipo `T` descreve o tipo do *input* da função, e o parâmetro de tipo `R` descreve o tipo do resultado da função. O método `apply` aplica a função a um dado argumento.

```
@FunctionalInterface
public interface Function<T,R> {
  R apply(T t);
  ...
}
```

O parâmetro de tipo `T` na interface `Predicate<T>` indica o *input* do predicado. O método `test` avalia o predicado num dado argumento.

```
@FunctionalInterface
public interface Predicate<T> {
  boolean  test(T t);
  ...
}
```

A anotação FunctionalInterface é usada para indicar que a interface deve ser usada como uma interface funcional, isto é, deve conter apenas um método abstrato (sem implementação). A sua utilização é opcional, mas traz duas vantagens: o compilador verifica se a entidade anotada é efetivamente uma interface com um único método abstrato, e o javadoc inclui a informação de que a interface é funcional.

Para escrever funções anónimas contamos agora com um sintaxe simplificada da forma:

```
parâmetros -> corpo-da-função
```

Eis alguns exemplos:

```
Function<Integer, Integer> dobro = x -> 2 * x;
Function<Integer, Integer> dobroBis = x -> {return 2 * x
   ;};
Function<Integer, Integer> dobroTris = x -> {
  int dois = 2;
  int resultado = dois * x;
  return resultado;
};
```

De notar que à direita da seta encontramos uma *expressão* Java sem parêntesis ou chavetas. Alternativamente, encontramos um *bloco de código*, entre chavetas.

Para funções em que o único parâmetro e o resultado têm o mesmo tipo podemos usar a interface `UnaryOperator`, subinterface de `Function`:

```
UnaryOperator<Integer> dobroUn = x -> 2 * x;
```

Se necessário (ou por questões de legibilidade) podemos anotar o parâmetro da função com o seu tipo:

```
Predicate<Character> sim = (Character c) -> c == 'y';
```

Para funções com dois parâmetros usamos a interface `BiFunction<T,U,V>` que descreve uma função de tipo `(t, u)-> r`. Os parâmetros da função anónima são colocado entre parêntesis e separados por vírgulas.

```
BiFunction<Integer, Integer, Integer> soma = (x, y) -> x
    + y;
```

Quando os parâmetros e o resultado são todos do mesmo tipo podemos usar a subinterface `BinaryOperator<T>`:

```
BinaryOperator<Integer> somaBinOp = (x, y) -> x + y;
```

Fazendo uso das interfaces funcionais e da sintaxe das funções lambda, podemos trocar o código que vimos acima. Em vez de:

```
System.out.println(new Filtar().filtrar(new Impar(),
    lista));
```

escrevemos:

```
System.out.println(new Filtar().filtrar(x -> x % 2 == 1,
    lista));
```

com a vantagem adicional que não precisamos de escrever a classe `Impar` e a sua interface `Predicado`.

Referências para métodos

Por vezes existem métodos que implementam a funcionalidade que gostariamos de utilizar na forma de uma função (para passar para um outro método, por exemplo).

Imaginemos que precisamos de uma função para converter os carateres de uma string para maiúsculas, isto é, de uma função com assinatura `Function<String, String>`. Poderiamos escrever `s -> s.toUpperCase()`. Mas o método `toUpperCase` é em certo sentido a função que precisamos. Compararemos com a situação em Haskell. Se precisarmos de uma função para converter para maiúscula, escrevemos **`toUpper`** e não `\s -> ` **`toUpper`** `s`.

O método `toUpperCase()` da classe `String` faz exactamente o que queremos. Para passar o método diretamente para a funcão, sem ter de escrever a função lambda usamos uma *referência para um método* da forma `String::toUpperCase`. O operador `::` separa o nome de um objeto ou de uma classe do nome do método. Há três casos a considerar.

Referência para método	Exemplo
`Classe::metodoDeObjecto`	`String::toUpperCase`
`Classe::metodoDeClasse`	`Math::pow`
`objecto::metodoDeObjeto`	`System.out::println`

Consideremos um método para transformar todos os elementos de um coleção: o método `map` recebe uma função e uma coleção e devolve uma nova coleção cujos elementos são obtidos por aplicação da função dada:

```
<A, B> Collection<B> map (Function<A, B> f, Collection<A>
    c) {
  Collection<B> result = new ArrayList<B> ();
  for(A x : c)
    result.add(f.apply(x));
  return result;
}
```

Um exemplo do padrão `Classe::metodoDeObjecto` é obtido usando o método `String toUpperCase()` da classe `String`. Em vez de `x -> x.toUpperCase())` escrevemos apenas `String::toUpperCase`:

```
Collection<String> strs = Arrays.asList("ola", "como", "
   vais?");
System.out.println(map(String::toUpperCase, strs));
```

Um exemplo para o segundo caso usa o método `static int abs(int x)` da classe `Math`. Em vez de `x -> Math.abs(x)` escrevemos apenas `Math::abs`:

```
Collection<Integer> ints = Arrays.asList(3,2,3,-3,-7,3);
System.out.println(new ReferenciasParaMetodos().map(Math
   ::abs, ints));
```

Para o terceiro caso temos, por exemplo, o método `boolean equals()` da classe `Integer`. Neste caso, se `tres` for um `Integer`, em vez de escrevermos `x -> tres.equals(x)` escrevemos apenas `tres::equals`:

```
Integer tres = Integer.valueOf(3);
System.out.println(new ReferenciasParaMetodos().map(tres
   ::equals, ints));
```

Outro exemplo. Para ordenar um *array* de strings ignorando a distinção entre letras minúsculas e maiúsculas podemos usar o método

```
public static <T> void sort(T[] a, Comparator<T> c)
```

da classe `Arrays`[2] e o método

```
public int compareToIgnoreCase (String s1, String s2)
```

[2]Mais precisamente, o comparador `c` compara elementos de um qualquer tipo que seja um supertipo de `T`.

da classe `String` e escrever apenas:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

Neste caso, `String` é o nome de uma classe e `compareToIgnoreCase` é o nome de um método de classe. Trata-se de uma variante do primeiro caso: o método `compareToIgnoreCase`, sendo um método de objecto, precisa de um objecto como alvo de chamada. Além disso, tem apenas um parâmetro. Ora nós procuramos uma função com dois parâmetros. Assim, o primeiro parâmetro da função torna-se o alvo da chamada e o segundo torna-se o (único) parâmetro do método. Isto é, a expressão `String::compareToIgnoreCase` tem o mesmo significado do que

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

Streams

Eis o problema: dada uma lista de números inteiros, obter a soma dos quadrados dos número impares. Temos de

- filtrar os números impares,
- elevar ao quadrado os que sobram (os impares),
- somá-los, e
- obter o resultado, um inteiro.

Utilizando *streams* podemos resolver este tipo de problemas facilmente.

```
List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9);
Integer soma = lista.stream()
                    .filter(x -> x % 2 == 0)
                    .map(x -> x * x)
                    .reduce(0, (x, y) -> x + y);
System.out.println(soma);
```

As soluções com *streams* tomam geralmente a seguinte forma:

1. Criar um *stream* a partir de dados pré-existentes,
2. Manipular os dados através de operações *stream*, e
3. Ler os dados do *stream*.

Para *criar streams* temos várias hipóteses:

- Vimos acima como usar o método `List::stream()` para transformar uma lista num *stream*.
- A classe `Arrays` também tem um método `stream(array)`, desta vez um método de classe.

```
Stream<String> arrayStream = Arrays.stream(new String
    []{"As","Armas","e","os","barões"});
```

- A classe `Stream` tem um método `of` com um número variável de parâmetros.

  ```
  Stream<String> stringStream = Stream.of("As","Armas",
      "e","os","barões");
  ```

- O método `iterate` da classe `Stream` devolve um *stream* infinito

  ```
  static<T> Stream<T> iterate(final T seed, final
      UnaryOperator<T> f)
  ```

  onde o primeiro elemento do *stream* resultante é `seed`, o segundo é `f.apply(seed)`, o terceiro é `f.apply(f.apply(seed))`, e por aí fora. Eis um *stream* de números naturais:

  ```
  Stream.iterate(0, n -> n + 1)
  ```

Para *manipular streams* a classe `Stream` dispõe de alguns métodos importantes:

- `filter` recebe um predicado e devolve um novo *stream* com todos os elementos para os quais o predicado é verdadeiro,
- `map` recebe uma função e devolve um novo *stream* aplicando a função a todos os elementos,
- `distinct`, devolve um novo *stream* sem elementos repetidos,
- `sorted` devolve um novo *stream* ordenado,
- `limit` recebe um número inteiro n e devolve um novo *stream* com n elementos no máximo (comparar com a função **`take`** Haskell),
- `skip` recebe um número inteiro n e devolve um novo *stream* depois de deixar cair os primeiros n elementos no máximo (comparar com a função **`drop`** Haskell),
- `reduce` recebe um elemento e um operador binário:

  ```
  T reduce(T identity, BinaryOperator<T> accumulator)
  ```

  Comparar com **`foldl`** em Haskell.

Finalmente, para *ler os dados* num *stream* temos várias hipóteses:

- O método `reduce` devolve um valor, tal como ilustrado no exemplo acima.

- Os métodos de quantificação existencial (`anyMatch`) e de quantificação universal (`allMatch`) devolvem um valor lógico. Mesma coisa para o método `empty`.
- O método `toArray` devolve um array de `Object` com os elementos constantes no *stream*.
- O método `collect` coleciona os elementos de um *stream*. É geralmente utilizado com a classe `Collectors`, por exemplo, `Collectors.toList()`.

Terminamos esta secção com três exemplos. Imaginemos que pretendemos a soma dos primeiros 1000 quadrados que são números ímpares. Começamos com o *stream* infinito de números naturais discutido acima. Depois lemos os primeiros 1000 números com o método `limit`, seguido de `map` para obter quadrados, `filter` para ficarmos com os impares e `reduce` para somar. Tudo junto fica:

```
Integer soma = Stream.iterate(0, n -> n + 1)
                     .map(x -> x * x)
                     .filter(x -> x % 2 == 1)
                     .limit(1000)
                     .reduce(0, (x,y) -> x + y);
System.out.println(soma);
```

Para transformar uma lista de strings numa lista de inteiros, podemos usar o método `lista.stream` para criar o *stream*, o método `map` para converter as strings e o método método `Collectors.toList` para transformar o *stream* de volta numa lista:

```
List<String> listaStrings = Arrays.asList("1", "2", "3",
   "4");
List<Integer> listaInteiros =
  listaStrings.stream().
               map(Integer::valueOf).
               collect(Collectors.toList());
System.out.println(listaInteiros);
```

Finalmente, consideremos uma classe descrevendo um utilizador.

```
class User {
  private final int age;
  private final String name;
  private final Sex sex;
  public User(int age, String name, Sex sex) {...}
  public int getAge() { return age; }
  public Sex getSex() { return sex; }
  public String getName() { return name; }
}
enum Sex {MALE, FEMALE}
```

Suponhamos que pretendemos todos as pessoas maiores do sexo feminino.

```
List<User> maioresFeminino =
  utilizadores.stream()
              .filter(user -> user.getAge() >= 18)
              .filter(user -> user.getSex() == Sex.FEMALE
                  )
              .collect(Collectors.toList());
```

Suponhamos agora que estamos interessados em todos as pessoas maiores, mas agrupadas por sexo. Podemos guardar esta informação num mapa de `Sex` para lista de pessoas, isto é: `Map<Sex,List<User>>`. Para este efeito a classe `Collectors` conta com um método que agrupa os elementos de um *stream* por categorias:

```
Map<Sex, List<User>> agrupadas =
  utilizadores.stream()
              .filter(user -> user.getAge() >= 18)
              .collect(Collectors.groupingBy(User::getSex
                  ));
```

Produtores e consumidores

Imaginemos que queremos aplicar a mesma "função" a todos os elementos de uma coleção, mas que não estamos interessados no resultado. Em vez disso estamos interessados no efeito colateral da "função". O caso típico acontece quando queremos imprimir todos os elementos de uma coleção. Claro que isto só faz sentido numa linguagem *imperativa*, onde as instruções alteram o *estado* da computação (o estado da consola no caso do exemplo da impressão).

A interface `Consumer<T>` descreve uma operação que recebe um único argumento e não devolve resultado algum:

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
  ...
}
```

Deste modo podemos facilmente aplicar uma mesma operação a todos os elementos de uma dada coleção através do método `paraCada`:

```
class Consumir {
  static <T> void paraCada(Consumer<T> consumidor,
     Collection<T> fonte) {
    for (T elemento : fonte)
      consumidor.accept(elemento);
  }
  public static void main(String[] args) {
```

```
    List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
    Consumir.paraCada(System.out::println, numeros);
  }
}
```

De notar que neste caso a referência para método `System.out::println` tem tipo `Consumer<Integer>`, o que está de acordo com o parâmetro do método `paraCada`. Inversamente podemos utilizar a interface `Supplier<T>` para produzir elementos do tipo `T`. Cada chamada ao método `get` devolve um valor do tipo `T`.

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

Eis, por exemplo, um método para produzir uma lista com um dado número de elementos, onde cada elemento é produzido mediante uma chamada ao método `get` (de um objecto do tipo `Supplier<T>`):

```
static <T> List<T> produzir (Supplier<T> produtor, int
    quantos) {
  Collection<T> resultado = new ArrayList<T>();
  for (int i = 0; i < quantos; i++)
    resultado.add(produtor.get());
  return resultado;
}
```

Se quisermos uma lista de 10 números todos iguais a 5 escrevemos:

```
List<Integer> numeros = produzir(() -> 5, 10);
```

onde a expressão lambda `() -> 5` tém tipo `Supplier<Integer>`. Para imprimir o resultado podemos chamar o método `paraCada` desenvolvido acima.

```
Consumir.paraCada(System.out::println, numeros);
```

Se quisermos uma lista de 20 números aleatórios em vírgula flutuante, escrevemos:

```
Random random = new Random();
List<Double> doubles = produzir(() -> random.nextDouble()
    , 20);
Consumir.paraCada(System.out::println, doubles);
```

Expressões lambda sem interfaces java.util.function

A utilização de expressões lambda não está confinada às interfaces no pacote `java.util.function`. Com efeito, qualquer interface Java com apenas um método abstrato (um método sem implementação) é um bom candidato. E isto inclui tanto os novos interfaces como aqueles que predatam a versão 8 da

linguagem. É o caso da interface `java.util.Comparator<T>` que compara a ordem de dois argumentos:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  ...
}
```

Imaginemos que pretendemos ordenar uma lista de strings recorrendo ao método `sort` da classe `Arrays`. Este método recebe dois parâmetros: um array de um tipo arbitrário `T` e um comparador de elementos do tipo `T`.[3] Temos então de preparar um objecto de uma classe `Comparator<String>`. Mas o interface `Comparator` é funcional. E interfaces funcionais pedem expressões lambda. Então, para ordenar uma lista, podemos usar uma simples expressão lambda que recorre ao método `compareTo` da classe `String`.

```
String[] diasSemana = {"seg", "ter", "qua", "qui", "sex",
    "sab", "dom"};
Arrays.sort(diasSemana, String::compareTo);
System.out.println(Arrays.toString(diasSemana));
```

Neste caso, a compilador de Java descobre que o tipo da referência para método `String::compareTo` é `Comparator<String>`, o que permite utilizar como parâmetro na chamada ao método `Array.sort`. Podemos tornar o tipo da expressão explícito guardando o objeto numa variável:

```
Comparator<String> comparador = String::compareTo;
Arrays.sort(diasSemana, comparador);
```

Compare-se com a versão sem expressões lambda, onde é necessário criar uma nova classe:

```
class ComparadorStrings implements Comparator<String> {
  @Override
  public int compare(String s1, String s2) {
    return s1.compareTo(s2);
  }
}
```

para depois escrevermos:

```
Arrays.sort(diasSemana, new ComparadorStrings());
```

O novo método da velha interface Iterable

Até aqui, para iterar sobre uma lista, usávamos um iterador. Por exemplo para imprimir os elementos de uma lista que são pares usávamos um código como o abaixo.

[3] Mais precisamente, um comparador de elementos de qualquer tipo que seja um supertipo de `T`.

```
List<Integer> lista = Arrays.asList(0, 1, 2, 3, 4, 5, 6,
    7, 8, 9);
for (Integer x : lista)
  if (x % 2 == 0)
    System.out.println(x);
```

Com o Java 8, a interface `Iterable<T>` ganhou um novo método. O método `forEach` tem um só parâmetro que deve ser do tipo `Consumer<T>`.[4] Aplica a mesma função a todos os elementos da lista. Um método que pede um objecto de uma interface funcional (`Consumer` é uma interface funcional) pode ser utilizado com uma expressão lambda. Assim, o ciclo **`for`** acima pode ser escrito deste modo:

```
lista.forEach (x -> {
  if (x % 2 == 0)
    System.out.println(x);
});
```

Neste caso o código não saiu mais compacto, mas se pretendermos apenas imprimir os elementos, podemos simplesmente escrever:

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

[4]Mais precisamente deverá ser um consumidor de qualquer tipo que seja supertipo de T.