01

# Que venham os lambdas!

**Transcrição**

Vamos retomar o nosso `forEach`, ele precisa da classe que implementa `Consumer`:

```
class ConsumidorDeString implements Consumer<String> {
    public void accept(String s) {
        System.out.println(s);
    }
}
```

E a invocação:

```
Consumer<String> consumidor = new ConsumidorDeString();
palavras.forEach(consumidor);
```

Se você já está acostumado com Java há mais tempo, sabe que nesses casos não criamos uma classe isolada. Fazemos tudo ao mesmo tempo, criando a classe e instanciando-a:

```
Consumer<String> consumidor = new Consumer<String>() {
    public void accept(String s) {
        System.out.println(s);
    }
};
palavras.forEach(consumidor);
```

São as chamadas classes anônimas, que usamos com frequência para implementar listeners e callbacks que não terão reaproveitamento.

Poderíamos até mesmo evitar a criação da variável `consumidor`, passando a classe anônima diretamente para o `forEach`:

```
palavras.forEach(new Consumer<String>() {
    public void accept(String s) {
        System.out.println(s);
    }
});
```

Quando começamos a aprender Java, essa sintaxe pode intimidar. Ela aparece com frequência, em especial nesses casos onde a implementação é curta e simples.

## Lambda para simplificar

Tendo essas dificuldade e verbosidade da sintaxe das classes anônimas em vista, o Java 8 traz uma nova forma de implementar essas interfaces ainda mais sucinta. É a sintaxe do lambda. Em vez de escrever a classe anônima, deixamos de escrever alguns itens que podem ser inferidos.

Como essa interface só tem um método, não precisamos escrever o nome do método. Também não daremos new. Apenas declararemos os argumentos e o bloco a ser executado, separados por `->` :

```
palavras.forEach((String s) -> {
    System.out.println(s);
});
```

É uma forma bem mais sucinta de escrever! Essa sintaxe funciona para qualquer interface que tenha apenas um método abstrato, e é por esse motivo que nem precisamos falar que estamos implementando o método `accept` , já que não há outra possibilidade. Podemos ir além e remover a declaração do tipo do parâmetro, que o compilador também infere:

```
palavras.forEach((s) -> {
    System.out.println(s);
});
```

Quando há apenas um parâmetro, nem mesmo os parenteses são necessários:

```
palavras.forEach(s -> {
    System.out.println(s);
});
```

Dá pra melhorar? Sim. podemos remover as chaves de declaração do bloco, assim como o ponto e vírgula, pois só existe uma única instrução:

```
palavras.forEach(s -> System.out.println(s));
```

Pronto. Em vez de usarmos classes anônimas, utilizamos o lambda para escrever códigos simples e sucintos nesses casos. Uma interface que possui apenas um método abstrato é agora conhecida como interface funcional e pode ser utilizada dessa forma.

Outro exemplo é o próprio `Comparator` que já vimos. Se utilizarmos a forma de classe anônima, teremos essa situação:

```
palavras.sort(new Comparator<String>() {
    public int compare(String s1, String s2) {
        if (s1.length() < s2.length())
            return -1;
        if (s1.length() > s2.length())
            return 1;
        return 0;
    }
});
```

Como aplicar a mesma lógica para transformar isso em um lambda? Basta removermos quase tudo da assinatura do método, assim como o `new Comparator` e adicionar o `->` entre os parâmetros e o bloco. Além disso, podemos tirar o tipo dos parâmetros:

```
palavras.sort((s1, s2) -> {
    if (s1.length() < s2.length())
        return -1;
    if (s1.length() > s2.length())
        return 1;
    return 0;
});
```

Melhor? Parece que sim. Mas ainda não muito interessante. O lambda se encaixa melhor quando a expressão dentro do bloco é mais curta. Normalmente com apenas um statement. Conhecendo a API do Java, podemos ver que há um método que compara dois inteiros e retorna negativo/positivo/zero dependendo se o primeiro for menor/maior/igual ao segundo. É o `Integer.compare` . Com ele, reduzimos o lambda para o seguinte:

```
palavras.sort((s1, s2) -> {
    return Integer.compare(s1.length(), s2.length());
});
```

Dá para fazer melhor. Como há apenas um único statement, podemos remover as chaves. Além disso, o `return` pode ser eliminado que o compilador vai inferir que deve ser retornado o valor que o próprio `compare` devolver:

```
palavras.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Compare com a nossa primeira versão. Muito melhor! Claro que poderíamos ter utilizado o `Integer.compare` desde o capítulo anterior, mas a combinação com o lambda deixa tudo mais legível e simples.

Vale lembrar que não é porque digitamos menos linhas que o código é necessariamente mais simples. Às vezes, pouco código pode tornar difícil de entender uma ideia, um algoritmo. Não é o nosso caso.