

JAVA8 – 2014

Programação Funcional em Java Novidades e Impacto

JDK8



As grandes novidades

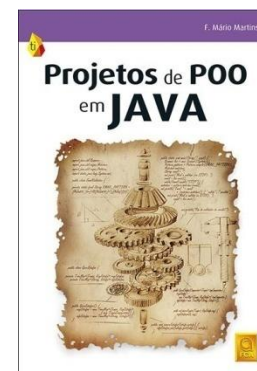
Prof. F. Mário Martins - DI/UM
CODEWEEK@DI - CESIUM
14 de Outubro de 2014



Prof. F. Mário Martins

Prof. da UM desde 1980; Prof. Associado do DI (desde 1998)

- ✓ Introduziu o paradigma da POO na UM nos anos 80 usando Smalltalk80 e Smalltalk-V;
- ✓ Introduziu Java na LESI e em LCC no final dos anos 90, substituindo Smalltalk;
- ✓ Leccionou diversas linguagens funcionais como LISP, Xmetoo e ML, anteriores a Haskell;
- ✓ Criou e Presidiu o **Centro de Competência da Java do DI/UM com Sun** (2009-2011);



- ✓ Sócio Honorário N° 1 do CESIUM.



Linguagens em 2014 (sem modas nem tiques)

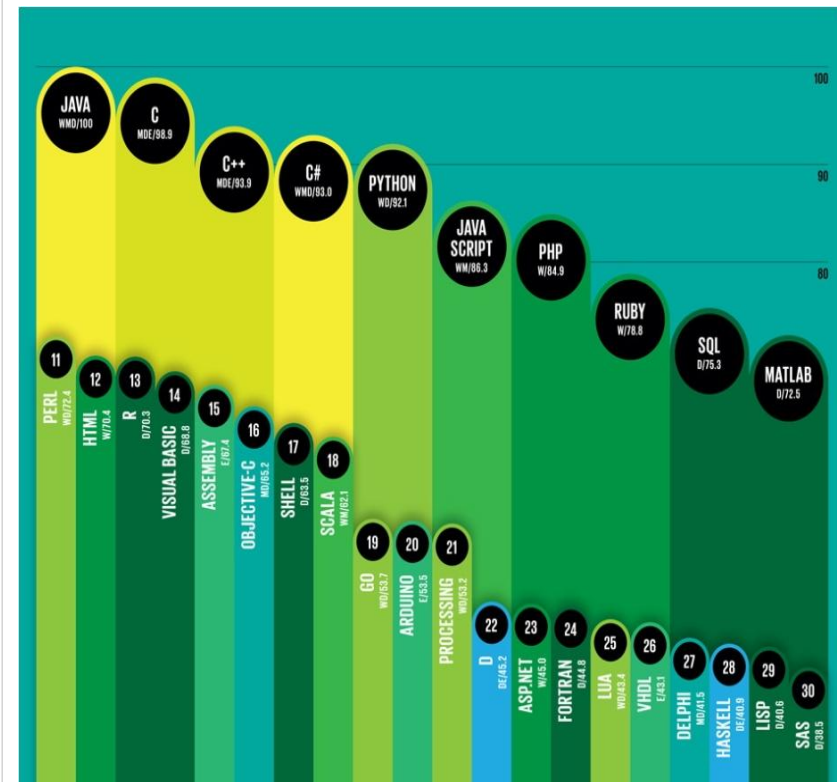
Atenção desenvolvedores! De acordo com o ranking da IEEE, a linguagem de programação Java é a que vem sendo mais utilizada no mundo. A pesquisa foi feita a partir de 12 fontes de dados, incluindo Google, GitHub, Stack Overflow e o fórum Hacker News.

Ao todo, 49 linguagens de programação entraram na lista e você pode conferir o ranking completo através do [site do IEEE](#). O Java aparece em primeiro lugar no ranking, atingindo 100 pontos, mas é seguido de perto pela linguagem C com 99,2 pontos e em terceiro lugar está o C++ com 95,5. Outras linguagens de programação web também muito utilizadas, PHP e Ruby, só aparecem na sexta e oitava posições, respectivamente. Já a linguagem de programação Swift, da Apple, não aparece no ranking, mas sua antecessora, Objective-C, está na décima sexta posição.

Language Types

☒ Web ☐ Mobile ☐ Enterprise ☐ Embedded

Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.2
3. C++		95.5
4. Python		93.4
5. C#		92.2
6. PHP		84.6
7. Javascript		84.3
8. Ruby		78.6
9. R		74.0
10. MATLAB		72.6



1. Java



IMAGE: MASHABLE COMPOSITE. IMAGE: [WIKIMEDIA COMMONS](#)

What it is: [Java](#) is a class-based, object-oriented programming language developed by Sun Microsystems in the 1990s. It's one of the most in-demand programming languages, a standard for enterprise software, web-based content, games and mobile apps, as well as the [Android](#) operating system. Java is designed to work across multiple software platforms, meaning a program written on Mac OS X, for example, could also run on Windows.

As grandes novidades



Java 8 Is Revolutionary, Java Is Back

The big talk of the conference was Java 8, exemplified by [Mark Reinhold](#) in his [Technical Keynote](#). Java 8 contains many new features, including a new Date and Time API ([JSR 310](#)), Nashorn JavaScript Engine, Type Annotations ([JSR 308](#)), Compact Profiles and Project Lambda ([JSR 335](#)).

Lambda is the single largest upgrade to the programming model. Ever. It's larger even than Generics. It's the first time since the beginning of Java that we've done a carefully coordinated co-evolution of the virtual machine, the language and the libraries, all together. Yet the result still feels like Java. — Mark Reinhold

JSR = Java Specification Request

Novidades e Impacto

- Java8 passa a ter **construções funcionais** juntamente com POO;
- Java8 passa a ter **funções** + “**closures**” (evitando outras construções pouco conhecidas e usadas, como **classes anónimas** ou **internas**);
- Java8 passa a ter “**streams de objectos e de tipos simples**”, ou seja, o equivalente funcional a listas infinitas, “lazy”, paralelizáveis e que podem ser filtradas (filter), transformadas (map), reduzidas (reduce), colectadas (collect), e muitas outras operações, funcionais ou não, como calcular o máximo de uma **stream** dada a função de cálculo, etc.;
- Em termos práticos e objectivos, como veremos, as **streams** de Java8, cf. **java.util.stream** API, em conjunto com o package **java.function**, vêm tornar a programação com **coleções** em Java completamente diferente, introduzindo a diferença entre **internal** e **external iteration**, e, até, **paralelismo explícito**.
- Java8 passa a ter um package **java.time** para questões temporais.

O que é, verdadeiramente, POO ou PPO ?

Getting The Message The Essentials of Message-Oriented Programming with Smalltalk

By Alan Lovejoy.

Smalltalk is a foundational programming language that is based on pervasive **message passing**, pervasive **dynamic strong typing**, pervasive **reflection** and pervasive **object orientation**.

Message passing: Almost all computation in Smalltalk happens via the sending of messages. The only way to invoke a method is to send a message—which necessarily involves dynamic binding (by name) of message to method at runtime (and never at compile time.) The internals of an object are not externally accessible, ever—the only way to access or modify an object's internal state is to send it a message. So function and data abstraction are both complete and universal. Pervasive **message passing** is Smalltalk's most important feature—a point that was lost on most of those who have tried to emulate Smalltalk when designing other programming languages.

Dynamic and strong typing: Although any object can be assigned to any variable, the only way to access or modify the internal state of an object is to send it a message—and the sending of any invalid message is detected and prevented at run time. So, even though Smalltalk's pervasive use of dynamic typing enables the programmer to define highly **polymorphic** abstractions with an extremely high degree of applicability and reusability, it is impossible to apply a function to a value for which there is no valid, defined behavior.

Reflection: In most programming languages, the **specifications** of types, classes, functions and subroutines exist only in the source code, and so are not accessible at runtime. But in Smalltalk, all specifications of all program constructs (classes, methods, etc.) are live objects that exist both at compile time and at runtime—and those objects are fully accessible to a running program, and can be queried or modified by sending them messages. So a Smalltalk program can not only fully introspect on itself, it has full power to change itself.

Object-orientation: In Smalltalk, all values are objects—even integers and other numbers, characters, strings, classes and blocks of code. Smalltalk is one of the first *object-oriented programming* languages. Its design was influenced by Lisp, Logo, Sketchpad, Flex and Simula. Smalltalk was developed as a research project at Xerox PARC in the 1970s by a team whose members included Dr. Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, [Diana-Merry Shipiro], Scott Wallace and others.

Warning: Terms such as "object," "class," "type," "method" and hence "object-oriented programming" itself, as used in the context of Smalltalk, do not have the same meanings as they do when used in the context of other programming languages. The term *object-oriented programming* ("OOP") was coined by Dr. Alan Kay, the inventor of Smalltalk. He intended the term to describe the essential nature of Smalltalk. Unlike Smalltalk, most of the programming languages that market themselves as "object oriented" do not satisfy Dr. Kay's definition of object oriented programming:

"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them." - Dr. Alan Kay

The full import of "object-oriented programming" as originally defined by Dr. Kay—and how and why the meaning of "OOP" as it applies to Smalltalk differs from the meaning of "OOP" as commonly understood outside of a Smalltalk context, is fully explained in the sections that follow. In addition, Dr. Kay's article "The Early History of Smalltalk" is highly recommended reading for anyone who wants to gain an even deeper insight into why and how Smalltalk came to be what it is, and why it is so different from the mainstream programming languages.

Enquadramento – Porquê mudar Java ?

- Java é uma linguagem de POO de (quase) 1ª classe; Em Java, tirando os tipos primitivos, tudo são objectos. Até os *arrays* são objectos.
- Em Java, as classes criam instâncias que são objectos, mas não existe nenhuma forma de definir um pedaço de comportamento, seja método ou função, que exista por si próprio, ou seja, fora de um objecto !!
- Em Java não existe forma de passar um método como parâmetro ou devolver um método como resultado.
- Em Java um objecto é uma cápsula de dados e de comportamento. Encapsular dados é muito importante pois garante rigor e segurança. Encapsular comportamento garante reutilização. Mas ...

Enquadramento – Porquê mudar Java ?

- As linguagens de POO **encapsulam comportamento em objectos**. As linguagens funcionais **encapsulam comportamento em funções**. Ambas as entidades são passadas como parâmetros para certos contextos e são aí executadas. Em ambas o problema de encapsular comportamento está de facto resolvido. **Porém, em Java, um objecto é mais do que apenas uma cápsula de comportamento e torna-se, em geral, “demasiado pesado”**.
- Alguma experiência de programação em Java, permite-nos verificar que quando pretendemos apenas encapsular comportamento, ou seja, implementar uma dada funcionalidade, Java resolve (**mal**) o problema e obriga-nos a criar objectos particulares sem atributos e que apenas implementam um único método, adicionalmente devendo satisfazer uma dada interface.
- Recordemos o que em Java é (**era cf. Java8**) necessário fazer para criar uma função de comparação entre dois objectos de tipo T.

Enquadramento – Porquê mudar Java ?

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

Assim, criamos uma classe que implementa o método **compare()**, cf.

```
public EmpregadoComparator implements Comparator<Empregado> {  
    public int compare(Empregado e1, Empregado e2) {  
        return e1.getNome().compareTo(e2.getNome())  
    }  
}
```

e depois passamos como parâmetro **uma instância de tal classe** para o construtor, tal como em:

```
Set<Empregado> emps = new TreeSet<Empregado>( new EmpregadoComparator() );
```

Enquadramento – Porquê mudar Java ?

- Outra técnica, menos conhecida, é a utilização de **classes anónimas** (muito comum em tecnologias como **Swing**).

```
public interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

Nestes casos, em vez de se criar uma classe que implemente a interface, dado que teríamos que ter uma destas classes para cada possível evento, é usual escrever-se código que defina tal classe "inline" e a instancie, cf. o código seguinte:

```
button.addActionListener(  
    new ActionListener() { // classe anónima com 1 único método  
        public void actionPerformed(ActionEvent e) {  
            out.println(e.toString());  
        }  
    }  
);
```

Enquadramento – Porquê mudar Java ?

■ Dada esta clara dificuldade de, em Java, porque Java sempre pretendeu ser uma linguagem de “first class objects only”, se poder “**encapsular comportamento simples**”, chegou o momento de, após muitos anos de estudo, se incorporar em Java os resultados do **Projecto Lambda (FSR 335)** de **Brian Goetze**.

■ **Objectivo:** Introduzir em Java construções funcionais clássicas (as **expressões lambda**, que são **funções anónimas**), que sejam objectos de 1ª classe como os outros, ou seja, que possuam **um tipo compatível** com o existente sistema de tipos de Java, e que permitam, de facto, **encapsular comportamento de forma simples e reutilizável em vários contextos** (ou seja, usadas como parâmetros de outras construções, etc.).

■ **Novidade? Não!** Estas construções funcionais existem desde os anos 40 do século 20 (como veremos). **Novidade? Não!** C#, Scala, Ruby e Python, etc., já têm construções funcionais dentro de POO. **Problema de Java: o impacto !!**

Expressões Lambda – História Antiga

1 Definition

The λ calculus can be called the *smallest universal programming language of the world*. The λ calculus consists of a single transformation rule (variable substitution) and a single function definition scheme. It was introduced in the 1930s by Alonzo Church as a way of formalizing the concept of effective computability. The λ calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to Turing machines. However, the λ calculus emphasizes the use of transformation rules and does not care about the actual machine implementing them. It is an approach more related to software than to hardware.

The central concept in λ calculus is the “expression”. A “name”, also called a “variable”, is an identifier which, for our purposes, can be any of the letters a, b, c, \dots . An expression is defined recursively as follows:

$$\begin{aligned}\langle \text{expression} \rangle &:= \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle \\ \langle \text{function} \rangle &:= \lambda \langle \text{name} \rangle . \langle \text{expression} \rangle \\ \langle \text{application} \rangle &:= \langle \text{expression} \rangle \langle \text{expression} \rangle\end{aligned}$$

Expressões Lambda – História Antiga

Expressões Lambda são funções sem nome

$\lambda x. x + 10$

$\lambda x, y. x * y$

Funções podem ser definidas usando expressões lambda

$F = \lambda x. x + 10$

$H = \lambda x, y. x * y$

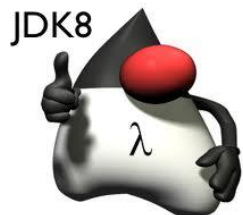
Em Lisp: `(lambda (x) (* x x))`

Em Smalltalk80: `[:x :y | x * y]` <- também designado bloco

Em Scala: `(x : Int, y : Int) => x + y`

etc.

Assim, **lambdas** são um conceito muito antigo, usado em linguagens muito antigas, mesmo de POO, e que Java8 decidiu finalmente incorporar mas, **mais uma vez procurando não alterar o bytecode e a JVM.**



Expressões Lambda em Java: Questões

- Projecto Lambda (FSR 335) de Brian Gotze.
- Passar comportamento como parâmetro eliminando a necessidade de se usarem classes anónimas, etc.;
- Não modificar o sistema de tipos de Java (não criar novos tipos);
- Sintaxe simples + Inferência automática de tipos (pelo compilador);
- Compatibilizar interfaces como **Comparator<T>**, etc. com **lambdas**;



JDK8



Expressões Lambda em Java: Sintaxe

Estruturas Básicas:

`() -> corpo`
`(arg1, arg2, ...) -> corpo`
`(arg1, arg2, ...) -> { corpo }`
`(tipo1 arg1, tipo2 arg2, ...) -> corpo`
`(tipo1 arg1, tipo2 arg2, ...) -> { corpo }`

Exemplos:

`() -> System.out.println("Bom dia !")`
`() -> random()`
`(x, y) -> x + y`
`(int x, int y) -> x * y`
`(Empregado e) -> e.getSalario()`

JDK8



Expressões Lambda em Java: Sintaxe

Invocação de métodos pré-definidos em classes, directamente ou usando uma nova expressão de referência de métodos (de instância ou static):

`x -> String.valueOf(x)`

`x -> String::valueOf`

`i -> i::toString()`

`() -> int[]::new`

Questão: Qual o tipo estático (em tempo de compilação) de uma expressão lambda ? Ou seja, lambdas são compatíveis com quê em Java8 ?

JDK8



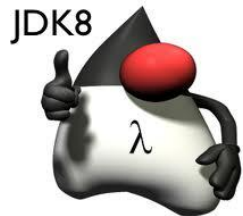
Expressões Lambda em Java: Tipos

A ideia de Gotze foi incrível. Dado que existem já em Java muitas e muitas interfaces apenas com um só método (designadas tipos SAM, ou, *Single Abstract Method* interfaces) então todas as lambdas são compatíveis com as SAM existentes, cf. `Comparator<T>`, `Runnable`, etc., e novas interfaces de 1 só método serão criadas num package novo designado `java.util.function`.

Todas estas interfaces de 1 só método, passam a designar-se por **Functional Interfaces** e são o tipo correcto das expressões lambda.

Deste modo, todas as interfaces de 1 só método antigas passam a ser **Interfaces Funcionais** e as novas passam a estar disponíveis no novo package `java.util.function`.

Lambdas passam de imediato a ser compatíveis com passado e futuro.



Expressões Lambda em Java: Compatibilidade com as interfaces existentes em Java7

Exemplos (usando lambdas com classes e interfaces antigas):

```
Comparator<String> scomp = (n1, n2) -> n1.compareTo(n2);
```

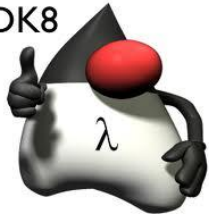
```
FileFilter javaFiles = (File f) -> f.getName().endsWith(".java");
```

// Java7

```
Runnable r1 = new Runnable() {  
    public void run() { System.out.println("Hello! I'm running.");  
    }  
};  
r1.run();
```

```
Runnable r2 = () -> System.out.println("Hello! I'm running.");  
r2.run();
```

JDK8



Expressões Lambda em Java: Functional Interfaces

`java.util.function`

Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

All Known SubInterfaces:

`UnaryOperator<T>`

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface  
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

Since:

1.8

JDK8



Expressões Lambda: `java.util.function`

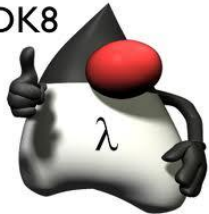
A biblioteca `java.util.function`

Todas as interfaces da biblioteca `java.util.function` são interfaces funcionais pelo que todas elas possuem a nova anotação `@FunctionalInterface`.

Sendo interfaces funcionais, para cada uma delas **há a necessidade de conhecer a assinatura do método abstracto a implementar** sempre que as pretendermos utilizar de forma explícita, ou seja, sendo nós a realizar a sua programação, e não apenas satisfazendo a sua implementação pela passagem de uma expressão lambda, tal como já vimos anteriormente.

Apresenta-se em seguida a lista completa de interfaces funcionais do package `java.util.function`, das quais apresentaremos exemplos de utilização das mais úteis e mais frequentemente requeridas ou usadas.

JDK8



Expressões Lambda: `java.util.function`

Predefined Functional Interfaces

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>

IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>



Expressões Lambda: `java.util.function`

`Function<T, R>` - função que transforma um T num R

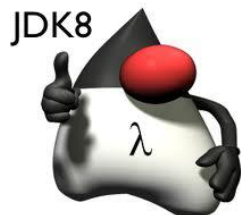
A primeira e mais natural destas interfaces é a interface `Function<T, R>` que representa uma função (ou um método) que recebe um parâmetro T e produz um resultado do tipo R.

```
Function<Integer, String> intToStrUp = i -> i.toString().toUpperCase();
```

```
Function<Ponto2D, String> pointToStr = Point2D::toString;
```

O método `R apply(T t)` é o responsável pela transformação, cf. :

```
String s = intToStrUp.apply(200);
```



Expressões Lambda: `java.util.function`

`Predicate<? Super T>` - função booleana sobre um valor `T`

A interface `Predicate<T>` representa uma função (ou um método) que recebe um parâmetro `T` e produz um resultado do tipo `Boolean`.

```
Predicate<Empregado> bemPagos = e -> e.getSalario() > 5000;
```

```
Predicate<Ponto2D> simetrico = p -> p.getX() == p.getY();
```

- Servirão para implementar filtros.

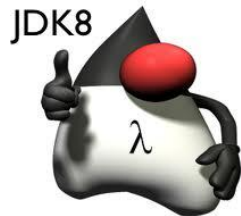


Expressões Lambda: `java.util.function`

Em Java8 **lambdas** podem não ser expressões puramente funcionais. De facto, sendo encapsulamento de comportamento destinado a ser executado nos mais variados contextos, as expressões lambda podem usar variáveis que não são seus parâmetro mas sim variáveis pertencentes ao contexto onde estas são empregues.

Por esta razão, muitos autores chamam-lhes **closures** tal como são designadas noutras linguagens.

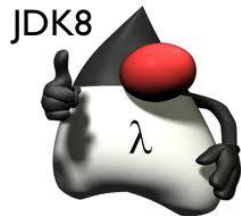
While a lambda is essentially the same as a function, taking parameters and returning a value, a closure is a slightly different beast: It "closes over" a scope, and may have access to values declared in that scope that are not explicitly passed as parameters.



Expressões Lambda: Porquê ?

The key reason for adding lambdas to the Java programming language is the need to evolve the JDK and in particular the JDK's collection framework. The traditional API design of the Java collections in package `java.util` renders certain optimizations impossible. At the same time, these optimizations are badly needed. Considering the current and future hardware architecture that we design our software for, we need support for increased parallelism and improved data locality. Let us see why.

Multi-core and multi-CPU platforms are the norm. While we deal with a couple of cores today, we will have to cope with hundreds of cores some time in the future. Naturally, we need to increase the effort for replacing serial, single-threaded execution by parallel, multi-threaded execution.



Expressões Lambda: Porquê ?

The JDK intends to support this effort by offering *parallel bulk operations for collections*. An example of a parallel bulk operation is the application of a transformation to each element in a collection. Traditionally such a transformation is done in a loop, which is executed by a single thread, accessing one sequence element after the other. The same can be accomplished in parallel with multiple threads: break down the task of transforming all elements in a sequence into many subtasks each of which operates on a subset of the elements.

A key point of the extension is to separate "what" operations are applied to sequence elements in a collection from "how" the operations are applied to these elements.

New abstractions have been added to the collection framework in package `java.util`, most prominently the *stream abstractions in package `java.util.stream`*. Collections can be converted to streams and streams, different from collections, access the sequence elements via internal iteration (as opposed to external iteration) .

Angelika Langer

JDK8



Streams em Java: `java.util.stream`

Uma **stream** é, em Java8, **uma abstracção** que representa uma **view** sobre uma colecção. Ao contrário de uma colecção, uma **stream não contém elementos**, é apenas uma visão sobre os dados/objectos, e usa os elementos da sua fonte através de um grupo vasto de operações pré-definidas.

O package `java.util.stream` possui implementadas várias **streams** para tipos primitivos, cf. `IntStream`, `DoubleStream` e `LongStream`.

O tipo **`Stream<T>`** é o directamente compatível com `Collection<T>`.

Todas as `Collection<T>` de Java8 possuem um método `stream()` que permite converter a colecção numa **`Stream<T>`** à qual se aplicam as diversas operações definidas sobre streams. **Todas as operações com streams usam lambdas !**

Streams em Java: `java.util.stream`

O programador deixa de ter que explicitar detalhadamente **COMO** cada iteração é feita (**external programming**) usando `for/each` ou `Iterator<T>`), passando apenas a declarar quais as operações que pretende realizar sobre a stream (**internal programming**), ou seja, indica **O QUÊ**.

A programação torna-se muito mais declarativa e compete à implementação decidir como cada uma delas vai ser implementada, ou seja, sequencialmente ou de forma paralela (daí a expressão **internal programming).**

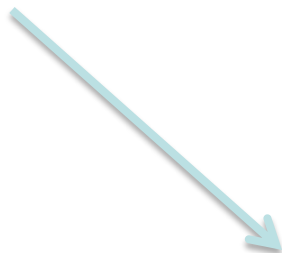
Streams em Java: `java.util.stream`

O programador deixa de ter que explicitar detalhadamente **COMO** cada iteração é feita (**external programming** usando `for/each` ou `Iterator<T>`), passando apenas a declarar quais as operações que pretende realizar sobre a stream (**internal programming**), ou seja, indica **O QUÊ**.

A programação torna-se muito mais declarativa e compete à implementação decidir como cada uma delas vai ser implementada, ou seja, sequencialmente ou de forma paralela (daí a expressão **internal programming**).

Assim, dada uma colecção, o método `stream()` converte-a na equivalente **stream** à qual se poderão aplicar as operações que se apresentam a seguir.

Collection<T>
.stream()
. ???

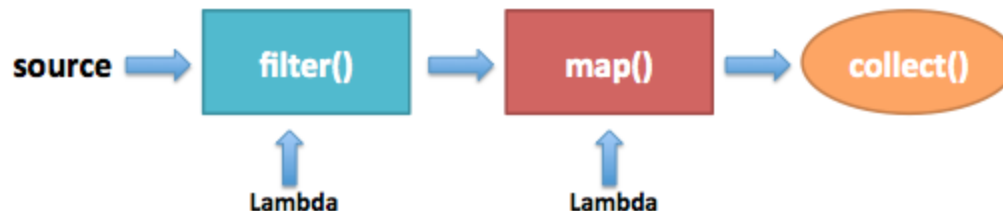


Method	Description
allMatch	true if all elements of the stream match the predicate
anyMatch	true if any element of the stream match the predicate
filter	return a stream of subset of the elements matching the predicate
findAny	return an element of the stream matching the predicate
findFirst	return the first element of the stream matching the predicate
flatMap	Return a stream where each input element is transformed into 0 or more values
	Return a stream where each input element is transformed into a stream
forEach	Perform an operation on each element of the stream (usually for side effects)
limit	Return a stream with no more than the first maxSize elements of this stream
map	Transform the stream into another containing the results after applying the function mapper on each element of the stream
max	Return the maximum element of this stream based on the supplied comparator
min	Return the minimum element of this stream based on the supplied comparator
peek	Return the same stream even as the elements are also provided to the consumer
reduce	Reduce the stream to a single value, performing the reducer operation on each element along with the accumulated value (accumulated value starting with the identity).
sorted	Sort the stream based on natural order or supplied comparator
subStream	Return a stream after discarding the first startingOffset elements (and those after optionally provided endingOffset elements)
toArray	Convert stream to array

Streams em Java: template típico - pipeline

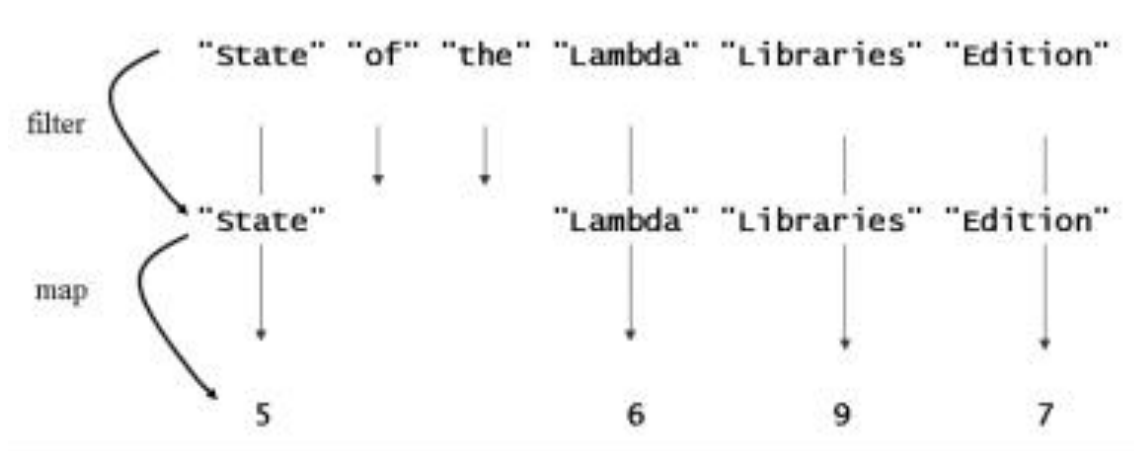
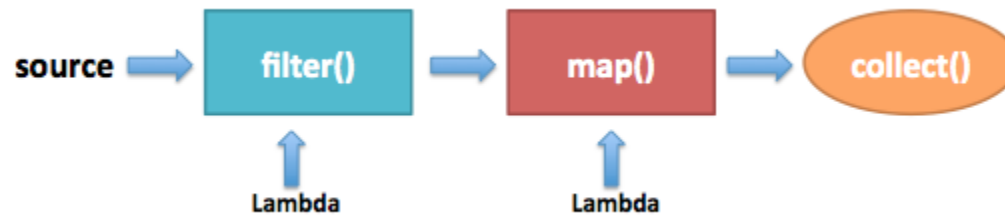
Vamos então visualizar e analisar uma estrutura de pipeline típica de Java8 com coleções, lambdas e streams.

```
List<String> nomes = Arrays.asList("Rui", "Rita", "Pedro")  
    .stream()  
    .map(n -> "Viva " + n)  
    .collect(Collectors.toList());
```

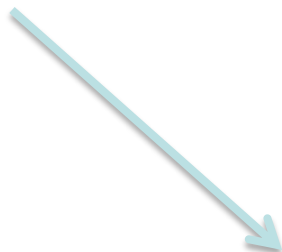


- A lista de nomes é criada usando `Arrays.asList()`;
- **`stream()` converte a lista de strings numa stream de strings;**
- **`map(n-> "Viva" + n)` transforma cada nome em "Viva ...";**
- **`collect(Collectors.toList())` cria um novo Collector que recolherá os resultados da pipeline para uma lista (que será de strings).**

Streams em Java: visualização da execução



Collection<T>
.stream()
. ???



**Relembremos as
operações**

Method	Description
allMatch	true if all elements of the stream match the predicate
anyMatch	true if any element of the stream match the predicate
filter	return a stream of subset of the elements matching the predicate
findAny	return an element of the stream matching the predicate
findFirst	return the first element of the stream matching the predicate
flatMap	Return a stream where each input element is transformed into 0 or more values
	Return a stream where each input element is transformed into a stream
forEach	Perform an operation on each element of the stream (usually for side effects)
limit	Return a stream with no more than the first maxSize elements of this stream
map	Transform the stream into another containing the results after applying the function mapper on each element of the stream
max	Return the maximum element of this stream based on the supplied comparator
min	Return the minimum element of this stream based on the supplied comparator
peek	Return the same stream even as the elements are also provided to the consumer
reduce	Reduce the stream to a single value, performing the reducer operation on each element along with the accumulated value (accumulated value starting with the identity).
sorted	Sort the stream based on natural order or supplied comparator
subStream	Return a stream after discarding the first startingOffset elements (and those after optionally provided endingOffset elements)
toArray	Convert stream to array

Streams em Java: Exemplo

```
List<Bloco> blocos = ... ;  
int somaPesos = 0;  
for (Bloco bloco : blocos) {  
    if (bloco.getCor() == Cor.AZUL)  
        somaPesos += bloco.getPeso();  
}
```

programação vertical
velho Java

```
List<Bloco> blocos = /* ... */;  
int somaPesos = blocos  
    .stream()  
    .filter(b -> b.getCor() == Cor.AZUL)  
    .map(b -> b.getPeso())  
    .sum();
```

programação pipeline
Java8

Streams em Java: Exemplo

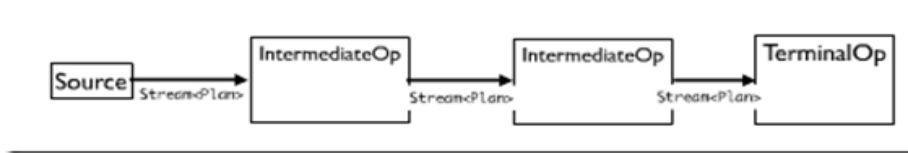
```
public boolean existeEmp(String cod) {  
    boolean existe = false;  
    Empregado emp = null;  
    Iterator<Empregado> itEmp = emps.iterator();  
    while(itEmp.hasNext() && !existe) {  
        emp = itEmp.next();  
        if(emp.getCodigo().equals(cod))  
            existe = true;  
    }  
    return existe;  
}
```

programação vertical
velho Java

```
public boolean existeEmp8(String cod) {  
    return emps.stream()  
        .anyMatch(e -> e.getCodigo().equals(cod));  
}
```

programação pipeline
Java8

Streams em Java: grupos de operações da pipeline



As operações sobre *streams* dividem-se em dois grandes grupos:

- **intermédias** (*intermediate operations*): operações que operam sobre uma *stream* e dão como resultado outra *stream* (do mesmo ou de outro tipo). São exemplos destas operações, que analisaremos em seguida, as operações: **filter**, **map**, **sort**, **distinct**, **limit**, **skip**, **concat**, **substream**, etc. **Operações intermédias são lazy!**
- **terminais** (*terminal operations*): trabalham sobre uma *stream* e dão como resultado um objecto de outro tipo qualquer ou até void, cf. **forEach**, **reduce**, **collect**, **sum**, **max**, **count**, **matchAny**, **findFirst**, **anyMatch**, **allMatch**, **noneMatch**, **findAny**, etc.

As operações intermédias, que devolvem streams, podem ser encadeadas umas nas outras, oferecendo uma forma de programar baseada em pipelines.

Streams em Java: Exemplos diversos



```
List<Integer> intList = Arrays.asList(1, 3, 5, 7, 12, 22);
```

```
// sequencial
```

```
int soma = intList.stream().reduce(0, Integer::sum);
```

```
intList.stream()  
    .forEach(i -> System.out.println(i));
```

```
// com paralelismo
```

```
int totalSoma = intList.parallelStream().reduce(0, Integer::sum);
```

Streams em Java: Exemplos

```
public double totalSalarios() {  
    double total = 0.0;  
    for(Empregado emp : emps)  
        total += emp.salario();  
    return total;  
}
```

```
public double totalSalarios8() {  
    return emps.stream()  
        .mapToDouble(e -> e.salario())  
        .sum();  
}
```

```
public double totalSalarios8A() {  
    return emps.stream()  
        .collect(Collectors.summingDouble(Empregado::salario));  
}
```


Streams em Java: Exemplos

```
public Empregado daFichaEmp(String cod) {  
    boolean existe = false; Empregado emp = null;  
    Iterator<Empregado> itEmp = emps.iterator();  
    while(itEmp.hasNext() && !existe) {  
        emp = itEmp.next();  
        if(emp.getCodigo().equals(cod))  
            existe = true;  
    }  
    return existe ? emp.clone() : null;  
}
```

```
public Optional<Empregado> daFichaEmp8(String cod) {  
    return emps.stream()  
        .filter(e -> e.getCodigo().equals(cod))  
        .findFirst();  
}
```



Streams em Java: Exemplos

Utilização de `Optional<T>` no `main()`:

```
Optional<Empregado> emp = empresa1.daFichaEmp8("E29");  
if(emp.isPresent())  
    System.out.println("Encontrado: " + emp.get().toString());  
else  
    System.out.println("Não existe empregado com esse código !");
```

O tipo `Optional<T>` visa remover de Java todas as possíveis confusões resultantes do uso do valor **null**.



Streams em Java: Exemplos

```
public int totalDeCategoria(String tipo) {  
    int total = 0;  
    for(Empregado emp : emps)  
        if(emp.getClass().getName().equals(tipo))  
            total++;  
    return total;  
}
```

```
public int totalDeCategoria8(String tipo) {  
    return (int) emps.stream()  
        .filter(e -> e.getClass().getSimpleName().equals(tipo))  
        .count();  
}
```

Streams em Java: Exemplos

```
public double totalKms() { // percorridos pelos motoristas no mês
    double totalKm = 0.0;
    for(Empregado emp : emps)
        if(emp instanceof Motorista)
            totalKm += ((Motorista) emp).getKms();
    return totalKm;
}
```

```
public double totalKms8() {
    return emps.stream()
        .filter(e -> e instanceof Motorista)
        .mapToDouble(e -> ((Motorista) e).getKms())
        .sum();
}
```

Polimorfismo mantém-se intacto e casting continua a ser necessário.

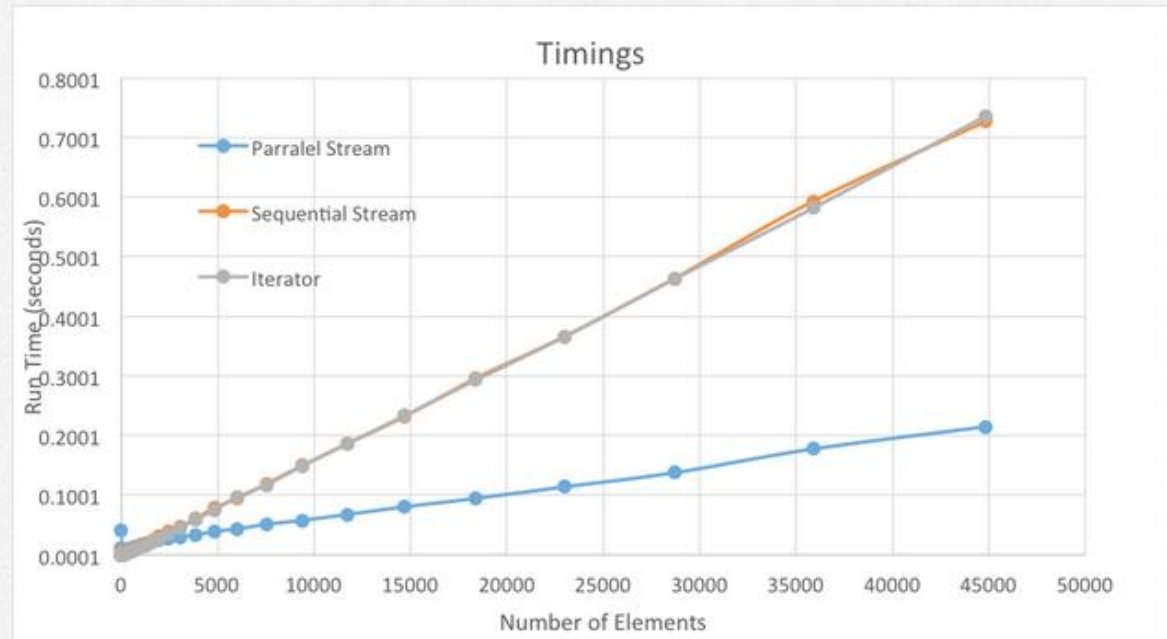


Streams em Java: Exemplos

Um exemplo final com Maps (para reflexão apenas):

```
Map<Departamento, Double> salariosPorDepart =  
    emps.stream()  
        .collect(Collectors.groupingBy(Empregado::getDept, somaSalarios));  
  
Collector<Empregado, ?, Double> soma Salarios =  
    Collectors.summingDouble(Empregado::getSalario);
```

Algumas questões simples sobre performance



In this test case, the parallel stream started to show a performance improvement above 2,000 elements and continued to perform significantly faster with even greater sizes. The sequential stream and iterator showed similar results at all data points. Parallel streams should not be used for small lists because the overhead of parallelizing the computation dwarfs the performance gains.

JSR310 >> `java.time`

- ✓ A nova API para datas, tempos, instantes e períodos (ou durações), `java.time`, que é **retrocompatível** com `GregorianCalendar` e `Date`, mas que propõe 15 classes distintas e muitos métodos, usando o formato **ISO-8601 de calendários**, que é uma **formalização proléptica do calendário Gregoriano**, ou seja, feita de tal modo que o calendário usado é correcto mesmo para anos anteriores à sua criação, ou seja do **calendário Juliano**, e trata bem os 10 dias que deixaram de existir na Terra na transição entre calendários realizada em Outubro de 1592.

JSR310 >> `java.time`

- ✓ A nova API para datas, tempos, instantes e períodos (ou durações), `java.time`, que é **retrocompatível** com `GregorianCalendar` e `Date`, mas que propõe 15 classes distintas e muitos métodos, usando o formato **ISO-8601 de calendários**, que é uma **formalização proléptica do calendário Gregoriano**, ou seja, feita de tal modo que o calendário usado é correcto mesmo para anos anteriores à sua criação, ou seja do **calendário Juliano**, e trata bem os 10 dias que deixaram de existir na Terra na transição entre calendários realizada em Outubro de 1592.
- ✓ O package é formado por 15 classes incríveis das quais destaco as seguintes:
`Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`
`Period`, `Duration`, `ChronoUnit`.

Default Methods: Exemplos

Default methods can be provided to an *Interface* without affecting implementing *Classes* as it includes an implementation. If each added *method* in an *Interface* defined with implementation then no implementing *Class* is affected. An implementing *Class* can override the *default implementation* provided by the *Interface*.

For *Java 8*, the *JDK collections* have been extended and *forEach* *method* is added to the entire *collection* (which work in conjunction with [lambdas](#)). With conventional way, the *code* looks like below,

```
1 public interface Iterable<T> {  
2     public void forEach(Consumer<? super T> consumer);  
3 }
```


Since this result each implementing *Class* with *compile errors* therefore, a *default method* added with a required implementation in order that the existing implementation should not be changed.

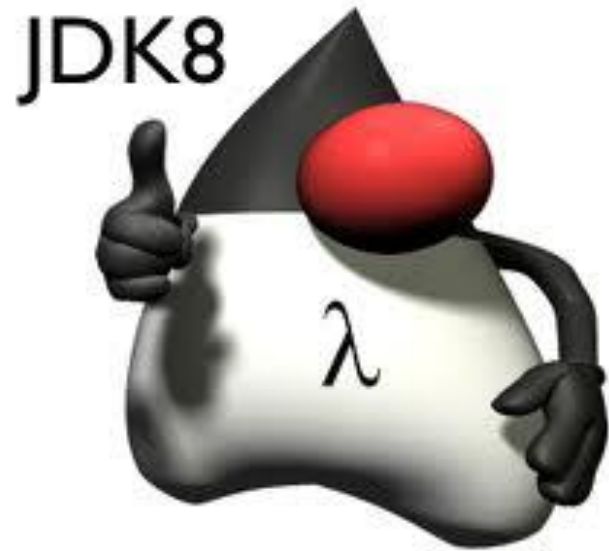
The [Iterable](#) *Interface* with the *Default method* is below,

```
1 public interface Iterable<T> {  
2     public default void forEach(Consumer  
3         <? super T> consumer) {  
4         for (T t : this) {  
5             consumer.accept(t);  
6         }  
7     }  
8 }
```

The same mechanism has been used to add [Stream](#) in *JDK* *Interface* without breaking the implementing *Classes*.

Esta foi a técnica empregue para
alterar JCF por forma a passar a ser
compatível com lambdas e streams





Temos que fazer uma workshop ☺



Contrapartidas pela Palestra



HODI KIBERA -> LIKES