

**Curso**

# Pacotes, Lambdas, Streams, Interfaces Gráficas

---

Atualizado até o Java 21 &  
Eclipse 2023-09



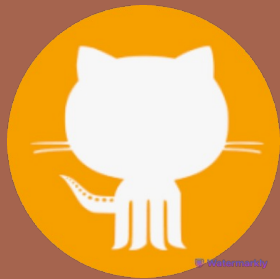
**Prof. Msc. Antonio B. C. Sampaio Jr**  
engenheiro de software & professor

@abctreinamentos  
@amazoncodebr

[www.abctreinamentos.com.br](http://www.abctreinamentos.com.br)  
[www.amazoncode.com.br](http://www.amazoncode.com.br)



# REPOSITÓRIO GITHUB



antonio-sampaio-jr / **pacotes-lambdas-streams**



# CONTEÚDO PROGRAMÁTICO




- UNIDADE 1 – PACOTES, ERROS E EXCEÇÕES
- UNIDADE 2 – ANOTAÇÕES E ENTRADA/SAÍDA
- UNIDADE 3 – GENÉRICOS
- UNIDADE 4 – FRAMEWORK COLLECTIONS
- UNIDADE 5 – NOVIDADES JAVA 8
  - Introdução às Expressões Lambdas
  - Sintaxe Lambda
  - Interfaces Funcionais
  - Var nos Parâmetros Lambda [NOVO]

# CONTEÚDO PROGRAMÁTICO



- UNIDADE 5 – NOVIDADES JAVA 8  
(Continuação)
  - Manutenção da Compatibilidade
  - Principais Utilizações das Expressões Lambdas
  - Referência de Métodos
  - Métodos Default
  - Streams
  - Parallel Streams
  - Melhorias na API Stream [NOVO]
  - Conclusão [NOVO]



# **Introdução às Expressões Lambda**

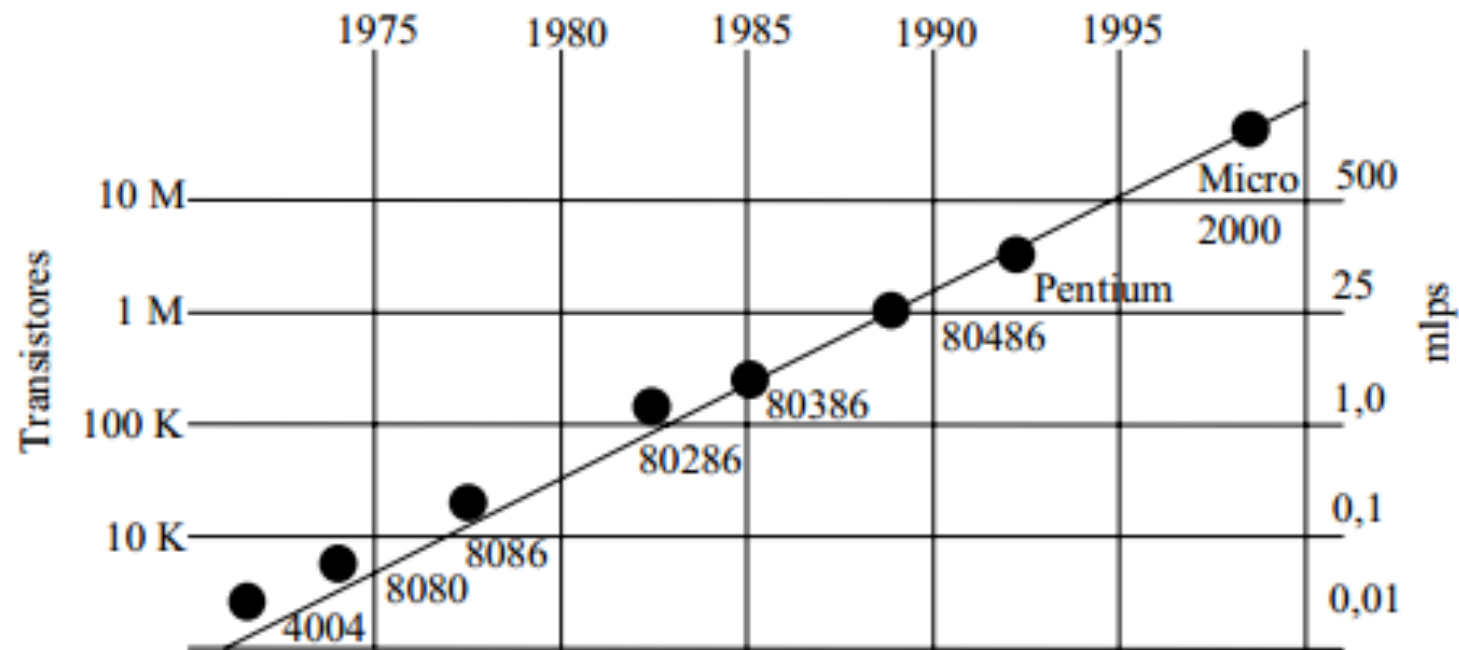
# O Fim da Era da Lei de Moore

- Em 1964, Gordon Moore, co-fundador e chefe aposentado da Intel, fez a seguinte observação:

“Após terem se passado quatro décadas depois do advento dos circuitos integrados, o número de transistores incorporados em chips de silício tem crescido amplamente. Realmente, ela tem dobrado mais ou menos a cada dezoito meses, explicando o porquê que computadores têm se tornado obsoletos em uma velocidade fantástica”.

- Ele fez esta observação - conhecida atualmente como **Lei de Moore** - em poucos anos de experiência de produção dos primeiros circuitos integrados, antes mesmo de terem sido inventados os microprocessadores.

# O Fim da Era da Lei de Moore



**Figura** – a lei de Moore informa que o número de transistores em chips de silicone dobram a cada 18 meses.

# O Fim da Era da Lei de Moore

- Contudo, a **Lei de Moore** está prestes a acabar. O principal motivo é que o processo de litografia (método de construção dos chips) está próximo do seu esgotamento. Especula-se que seja possível estender o processo de litografia por mais uma geração depois dos 10 nm, mas a partir daí esse processo atingirá o seu limite físico.

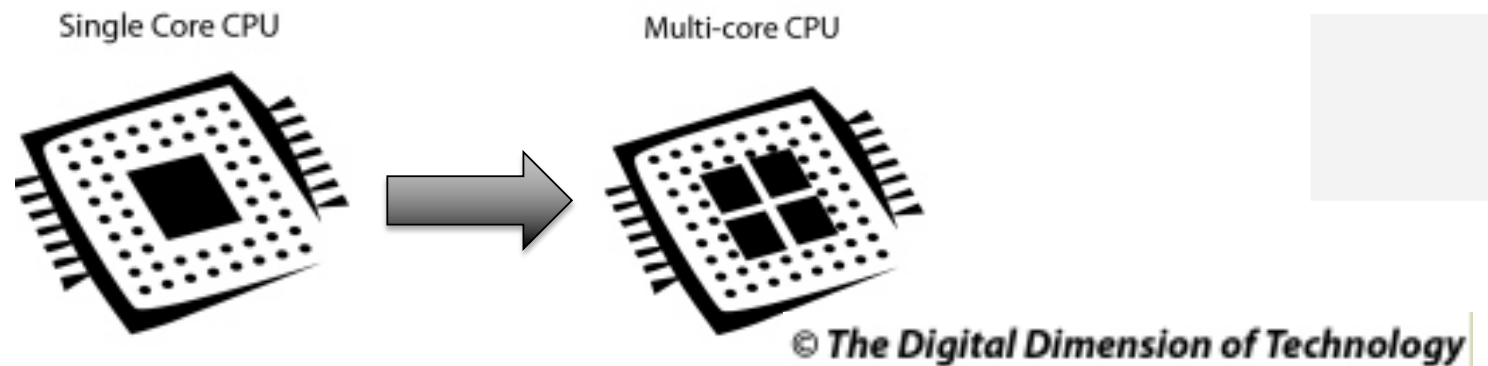
## Intel Technology Roadmap

Process Name	<u>P1266</u>	<u>P1268</u>	<u>P1270</u>	<u>P1272</u>	<u>P1274</u>
Lithography	45 nm	32 nm	22 nm	14 nm	10 nm
1 <sup>st</sup> Production	2007	2009	2011	2013	2015



# Consequências

- A melhoria na capacidade dos processadores vai se concentrar na utilização de vários núcleos. Um processador de 2 núcleos consome muito menos energia e oferece o dobro de velocidade do que um de 1 só núcleo.



- **O investimento do hardware será concentrado no paralelismo** dos seus chips, o que trará mudanças significativas no mundo do software, que se baseou durante muitos anos no processamento sequencial!
- As linguagens definidas como imperativas são aquelas baseadas na construção de programas através de uma sequência de instruções. O fundamento desse tipo de linguagem é a Máquina de Turing e é caracterizada por utilizar conceitos como variáveis, atribuição, repetição, etc.

# Ascensão do Paradigma Funcional

- O paradigma funcional fornece muitos benefícios na construção de aplicações baseadas em um hardware composto por chip de vários núcleos, visto que favorece enormemente a execução das instruções de máquina de forma paralela.
- O Paradigma Funcional é bastante antigo. A 1ª linguagem baseada neste paradigma foi a IPL em 1955. A mais popular foi a LISP em 1958.
- O paradigma funcional é baseado no conceito de função, isto é, pode-se definir a programação funcional como a simples avaliação de expressões matemáticas.
- O programador define uma função para resolver um problema e a passa para o computador avaliá-la como uma calculadora. Nessa avaliação, as expressões escritas pelo programador são simplificadas até chegar a uma forma normal.
- **Exemplo:**  $f(x) = x^3 + 3$  é definida em termos de exponenciação e adição.

# Cálculo Lambda

- É um modelo que oferece uma maneira muito formal de descrever o cálculo de uma função. Foi projetado por Alonzo Church em 1930.
- Uma **abstração lambda** é um tipo de expressão que denota uma função:  $(\lambda x. +x1)$ , onde “ $\lambda$ ” determina que existe uma função.

(       $\lambda$       x      .      +      x      1      )

uma função de x que incrementa x de 1

[© Matheus Henrik Nogueira de Santana](#)

- Atualmente, as linguagens C++, C#, JavaScript, Python, Ruby, Jruby, Scala, Clojure, já oferecem suporte ao uso de expressões lambdas!
- O Java tem evoluído para aumentar seu poder de expressão, incorporando novos recursos: Collections API , Generics, Annotations, Lambdas, Stream API, entre outros.

# Java e o Paradigma Funcional

- Até a versão 7 Java pode ser definida como uma linguagem imperativa Orientada a Objetos. A partir da versão 8, Java também passa a incorporar funcionalidades do paradigma funcional, quando adota em sua sintaxe o cálculo Lambda!

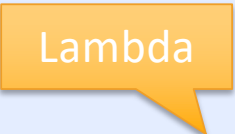
## Método até o Java 7

```
class Conta {  
    ...  
    public boolean transfere (Conta conta, float valor) {
```



## Método a partir do Java 8

```
class Conta {  
    ...  
    public boolean transfere (Conta conta, float valor,  $\lambda$ ) {
```



# Expressões Lambda

- A implementação da **JSR 335: *Lambda Expressions for the Java™ Programming Language*** provocou uma das maiores atualizações em termos de código fonte, tanto na JVM como no JDK (compilador e bibliotecas).
- Para se ter uma idéia, ao se enumerar as novas classes dos pacotes **java.util** e **java.util.concurrent** que dão suporte à JSR 335 no JDK 8, chega-se ao expressivo número de 283 classes. Se também for somado as classes dos novos pacotes **java.util.function** e **java.util.stream**, chega-se a um total de 896 classes a mais no JDK 8 em relação ao JDK 7, como pode ser visto na Tabela abaixo.

Pacote	Classes (JDK8/JDK7)	EL (JDK 8)	Total (JDK8/JDK 7)
java.util	428/299	43	471/299
java.util.concurrent	310/202	3	313/202
java.util.function	43/0	38	81/0
java.util.stream	332/0	200	532/0

# Expressões Lambda

- A principal novidade do Java 8 é o suporte às **Expressões Lambda**.
- A sua adoção pela tecnologia Java era muito aguardada, principalmente pelo suporte a elas no *Framework Collections*.

## O que são?

- **Expressões Lambda** são um novo recurso da linguagem Java para implementar de maneira simples o cálculo lambda.
- No Java 8 o lambda representa uma função anônima. Esta é definida em uma interface (***Functional Interface***) que possui apenas um único método abstrato!
- Com o uso das Expressões Lambda é possível a construção de um código mais conciso, pois evita a criação de classes anônimas de apenas um método.

# Expressões Lambda

## CLASSES ANÔNIMAS

- São classes que não possuem nome, não tem construtor e só são utilizadas dentro de um bloco de código para oferecer alguma funcionalidade.

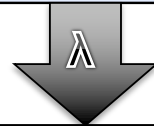
```
btnNewButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        ...  
    } }) ;
```

- Isso implica em não ser possível termos uma variável do tipo dessas classes, uma vez que não sabemos qual o seu tipo, ou melhor, qual o nome de seu tipo.
- Elas são utilizadas geralmente quando se quer uma classe para implementar determinada interface, porém, só é utilizada em um contexto muito restrito.

# Expressões Lambda

## CLASSES ANÔNIMAS X LAMBDA

```
btnNewButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        ...  
    } }) ;
```



```
btnNewButton.addActionListener(event -> {...}) ;
```

- É importante ressaltar que o lambda não é a mesma coisa que uma classe interna anônima, visto que esta pode possuir muitos métodos e o **lambda** apenas um!



# Primeiro Lambda JAVA

```
public class LambdaApp {  
    public static void main(String[] args) {  
        List<Integer>integers = Arrays.asList(1, 2, 3, 4, 5);  
        //Expressão Lambda  
        integers.forEach(x->System.out.println(x));  
    }  
}
```

- O método **forEach(...)** aceita a função anônima como entrada e chama a referida função para cada elemento da lista.
- A função anônima é representada pela expressão abaixo, onde **x** é o seu parâmetro do tipo inteiro.

***x -> System.out.println(x)***

# Ciclo de Vida Lambda JAVA

- **São dois os estágios no Ciclo de Vida Lambda Java:**
- 1) Converter a expressão lambda para uma função

`x -> System.out.print(x);`



```
public static void generatedNameOfLambdaFunction(Integer x){  
    System.out.println(x);  
}
```

© Adib Saikali

- 2) Executar a função chamada

# Segundo Lambda JAVA

```
public class LambdaApp2 {  
    public static void main(String[] args) {  
        List<Integer>integers = Arrays.asList(1, 2, 3, 4, 5);  
        integers.forEach(x ->{  
            x = x + 10;  
            System.out.println(x);  
        });  
    }  
}
```

- Neste código o valor de **x** é incrementado em **10!**

# Terceiro Lambda JAVA

- As variáveis podem ser lidas 'fora' do escopo das Expressões Lambdas.

```
public class LambdaApp3 {  
    public static void main(String[] args) {  
        int numero = 10;  
        List<Integer>integers = Arrays.asList(1, 2, 3, 4, 5);  
        integers.forEach(x ->{  
            x = x + numero;  
            System.out.println(x);  
        });  
    }  
}
```

- Contudo, essas variáveis não podem ser modificadas.

```
public static void main(String[] args) {  
    ...  
    numero = 20; //ERRO DE COMPILAÇÃO  
    x = x + numero;  
    ...  
}
```

# Quarto Lambda JAVA

- Diferentemente das variáveis locais, o acesso aos atributos de objeto e de classe podem ser lidos e modificados.Lambda

```
public class LambdaApp4 {  
    static int numero;  
    int somatorio;  
    public static void main(String[] args) {  
        List<Integer>integers = Arrays.asList(1, 2, 3, 4, 5);  
        LambdaApp2 app = new LambdaApp2();  
        integers.forEach(x ->{  
            numero = 10;  
            x = x + numero;  
            app.somatorio = app.somatorio + x;  
            System.out.println(x);  
        });  
        System.out.println(app.somatorio);  
    }  
}
```

# Quinto Lambda JAVA

```
public class LambdaApp5 {  
    public static void main(String[] args) {  
        List<Integer>integers = Arrays.asList(1, 2, 3, 4, 5);  
        integers.forEach(x ->{  
            int y = x/2;  
            System.out.println(y);  
        });  
    }  
}
```

- Neste código é criado uma variável local **y**!

# Sexto Lambda JAVA

```
public class LambdaApp6 {  
    public static void main(String[] args) {  
        List<Integer>integers = Arrays.asList(1, 2, 3, 4, 5);  
        integers.forEach((Integer x) ->{  
            x = x + 10;  
            System.out.println(x);  
        });  
    }  
}
```

- Neste código é passado um parâmetro do tipo **Integer**!

# Exercícios

- 1) Escreva as classes **LambdaApp**, **LambdaApp2**, **LambdaApp3**, **LambdaApp4**, **LambdaApp5** e **LambdaApp6**.
- 2) Altere o código **LambdaApp** e inclua a variável **var**, conforme imagem abaixo. Pergunta-se: por que o erro abaixo? Como corrigi-lo?

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
```

```
    int var = 1;  
    integers.forEach(x -> {  
        var++;
```

Error: Local variable var defined in an enclosing scope must be final or effectively final

```
        System.out.println(x);  
    });
```





# Sintaxe Lambda

# Sintaxe

## REGRAS DE SINTAXE DAS EXPRESSÕES LAMBDA

- A declaração dos tipos de parâmetros é opcional;

```
() -> System.out.println(this)
```

- A utilização de parêntesis () em volta do parâmetro é opcional se houver apenas um parâmetro;

```
(String str) -> System.out.println(str)  
str -> System.out.println(str)
```

- A utilização de colchetes {} é opcional, a menos que sejam definidos múltiplas instruções;

```
(String s1, String s2) -> {return s2.length()  
                           - s1.length();}  
(s1, s2) -> s2.length() - s1.length()
```

- O uso da palavra **return** é opcional se houver uma simples expressão que retorne um valor.

# Sintaxe

## CLASSES ANÔNIMAS → SEM EXPRESSÃO LAMBDA

```
interface IMatematica{  
    public int somar(int a, int b);  
    public int subtrair(int a, int b);  
    public int multiplicar(int a, int b);  
    public int dividir(int a, int b);  
}
```

# Sintaxe

## CLASSES ANÔNIMAS → SEM EXPRESSÃO LAMBDA

```
public class Calculadora{
    public static void main(String[] args) {
        IMatematica anonimo = new IMatematica() {
            //objeto tipo anônimo que implementa IMatematica
            public int somar(int a, int b){return a+b;}
            public int subtrair(int a, int b){return a-b;}
            public int multiplicar(int a, int b){return a*b;}
            public int dividir(int a, int b){return a/b;}
        };
        System.out.println("anônimo: " + anonimo.getClass());
        System.out.println("=> " + anonimo.somar(5,5));
        System.out.println("=> " + anonimo.subtrair(6,4));
        System.out.println("=> " + anonimo.multiplicar(3,4));
        System.out.println("=> " + anonimo.dividir(10,2));
    }
}
```

# Sintaxe

## CLASSES ANÔNIMAS → SEM EXPRESSÃO LAMBDA

- Após a execução do código, têm-se as seguintes saídas:

```
anônimo: class Calculadora$1  
=> 10 => 2  
=> 12 => 5
```

- O Java nomeia as classes anônimas como se fossem classes internas da classe onde foram declaradas. O nome individual de cada classe anônima é, na verdade, apenas um inteiro.

# Sintaxe

## CLASSES ANÔNIMAS → EXPRESSÃO LAMBDA

- Criar uma Interface Funcional com o método **int operacao(...)**

```
@FunctionalInterface
interface IMath
{
    int operacao(int a, int b);
}
```

- Criar uma Expressão Lambda que instancie a interface A.
- Sintaxe Padrão:

```
parâmetros -> corpo
```

# Sintaxe

## CLASSES ANÔNIMAS → EXPRESSÃO LAMBDA

```
public class Calculadorav2 {  
    public static void main(String[] args) {  
        Calculadorav2 ex = new Calculadorav2();  
        IMath somar = (a,b)-> a+b; //Expressão Lambda  
        IMath subtrair = (a,b)-> a-b; //Expressão Lambda  
        IMath multiplicar = (a,b)-> a*b; //Expressão Lambda  
        IMath dividir = (a,b)-> a/b; //Expressão Lambda  
  
        System.out.println(ex.execOperacao(5, 5, somar));  
        System.out.println(ex.execOperacao(6, 4, subtrair));  
        System.out.println(ex.execOperacao(3, 4, multiplicar));  
        System.out.println(ex.execOperacao(10, 2, dividir));  
    }  
    public int execOperacao(int a, int b, IMath op) {  
        return op.operacao(a, b);  
    }  
}
```

# Exercícios

- 1) Modifique as classes **Calculadora** e **CalculadoraV2**, bem como as interfaces **IMatematica** e **IMath** para incluir os métodos **exponenciação** e **radiciação**.



- 2) Transformar o código abaixo para deixar de utilizar classe anônima e passar a utilizar EL.

```
Validador<String> validadorCEP = new Validador<String>() {  
    public boolean valida(String valor) {  
        return valor.matches("[0-9]{5}-[0-9]{3}");  
    }  
};
```

```
interface Validador<T> {  
    boolean valida(T t);  
}
```





# Interfaces Funcionais

# Interfaces Funcionais

- Interfaces funcionais são o “coração” do recurso de Lambda. O Lambda por si só não existe, mas sim as **Expressões Lambda** quando associadas a uma interface funcional.
- Pode-se marcar “explicitamente” uma interface como funcional, bastando utilizar a anotação **@FunctionalInterface**.

```
@FunctionalInterface  
interface IMath  
{  
    int operacao(int a, int b);  
}
```

- No exemplo acima, sem o uso dessa anotação, o Java entenderia que a interface **IMath** é “implicitamente” Funcional.

# Interfaces Funcionais

- Como já foi citado anteriormente, foi criado no Java 8 o pacote **java.util.function** para a definição de um conjunto de **Interfaces Funcionais (43)** a serem utilizadas pelo desenvolvedor Java.

Interface Summary	
Interface	Description
<b>BiConsumer&lt;T,U&gt;</b>	Represents an operation that accepts two input
<b>BiFunction&lt;T,U,R&gt;</b>	Represents a function that accepts two argumer
<b>BinaryOperator&lt;T&gt;</b>	Represents an operation upon two operands of t
<b>BiPredicate&lt;T,U&gt;</b>	Represents a predicate (boolean-valued function
<b>BooleanSupplier</b>	Represents a supplier of boolean-valued results.
<b>Consumer&lt;T&gt;</b>	Represents an operation that accepts a single in
<b>DoubleBinaryOperator</b>	Represents an operation upon two double-valued
<b>DoubleConsumer</b>	Represents an operation that accepts a single do
<b>DoubleFunction&lt;R&gt;</b>	Represents a function that accepts a double-val
<b>DoublePredicate</b>	Represents a predicate (boolean-valued function
<b>DoubleSupplier</b>	Represents a supplier of double-valued results.

# Interfaces Funcionais

## PACOTE JAVA.UTIL.FUNCTION

- As principais Interfaces Funcionais estão listadas abaixo:
  - **Predicate <T>**
  - **Supplier <T>**
  - **Function <T,R>**
  - **Consumer <T>**
  - **Comparator <T>**
  - BiFunction
  - BiConsumer
  - IntConsumer
  - IntFunction <R>
  - IntPredicate
  - IntSupplier
- A seguir, as principais Interfaces Funcionais são apresentadas.

# Interfaces Funcionais

## PREDICATE<T>

- É uma Interface Funcional que recebe um valor T e faz um teste qualquer no objeto recebido como parâmetro. Retorna *'true'* ou *'false'*.

```
Predicate<String> predicate = (s) -> s.length() > 0;  
predicate.test("foo"); // true  
predicate.negate().test("foo"); // false
```

## SUPLIER<T>

- É uma Interface Funcional que não recebe nenhum valor e retorna um resultado T.

```
Supplier<Perssoa> pessoaSupplier = new Pessoa();  
pessoaSupplier.get(); // novo objeto Pessoa
```

# Interfaces Funcionais

## FUNCTION<T,R>

- É uma Interface Funcional que recebe um valor T e retorna um resultado R.

```
Function<String, String> atr = (name) -> {return "@" + name;};  
Function<String, Integer> leng = (name) -> name.length();  
void m() {  
    int y = 3;  
    Function<Integer, Integer> f = x -> x + y;  
    f.apply(2);  
}
```

## CONSUMER<T>

- É uma Interface Funcional que recebe um valor T e imprime um valor.

```
Consumer<Pessoa> greeter = (p) -> System.out.println  
                                ("Ola," + p.nome);  
greeter.accept(new Pessoa("Antonio", "Sampaio"));
```

# Interfaces Funcionais

## COMPARATOR<T>

- É uma Interface Funcional que recebe um valor T e retorna um resultado R.

```
Comparator<Pessoa> comparator = (p1, p2) ->
    p1.nome.compareTo(p2.nome);
Pessoa p1 = new Pessoa("João", "Pedro");
Pessoa p2 = new Pessoa("Alice", "Wonder");
comparator.compare(p1, p2); // > 0
comparator.reversed().compare(p1, p2); // < 0
```

# Exercício

- 1) Compile o código abaixo e identifique qual foi o erro gerado.

```
@FunctionalInterface
interface Validador<T> {
    boolean valida(T t);
    boolean outroMetodo(T t);
}
```





# **Var nos Parâmetros Lambda**

# VAR NOS PARÂMETROS LAMBDA

- **Definição**

- O uso do operador **var** em parâmetros lambda é uma característica introduzida a partir do Java 11, e isso permite declarar parâmetros lambda sem especificar explicitamente o tipo, deixando o compilador inferir o tipo com base no contexto. Isso torna o código mais conciso e legível em certos casos.

```
(@Nullable var x, @Nonnull Integer y) -> x + y;
```

- É possível usar anotações para marcar se um dado pode ser nulo ou não, mas essa regra só é possível se usar parâmetros explícitos ou com var.

# VAR NOS PARÂMETROS LAMBDA

- Exemplo

```
//Var nos parâmetros Lambda
public static void exemplo05()
{
    BiFunction<@NonNull String, @Nullable String, String> concatenar = (s1, s2) -> {
        if (s2 == null) {
            return s1;
        } else {
            return s1 + s2;
        }
    };

    String resultado1 = concatenar.apply(t:"Olá, ", u:null);
    String resultado2 = concatenar.apply(t:"Java ", u:"11");

    System.out.println(resultado1); // "Olá, "
    System.out.println(resultado2); // "Java 11"

    Predicate<Integer> maiorQueDez = (var x) -> x > 10;

    System.out.println(maiorQueDez.test(t:5)); // Saída: false
    System.out.println(maiorQueDez.test(t:15)); // Saída: true
}
```

# VAR NOS PARÂMETROS LAMBDA

- Exemplo

```
//Criando anotações personalizadas  
@interface Nonnull {}  
@interface Nullable {}
```



# Manutenção da Compatibilidade

# Manutenção da Compatibilidade

- Qualquer interface com um método é considerada pelo Java 8 como **Interface Funcional!**
- No Java 8, os lambdas irão trabalhar com as bibliotecas mais antigas que usam as interfaces funcionais, sem a necessidade de recompilá-las ou modificá-las.

# Manutenção da Compatibilidade

## INTERFACE RUNNABLE

- Pode-se afirmar que toda interface Java que possui apenas um método abstrato pode ser instanciada como um código lambda! Isso vale até mesmo para as interfaces antigas, pré-Java 8, como por exemplo a interface *Runnable*:

```
public interface Runnable {  
    public abstract void run();  
}
```

- Nova interface Runnable no Java 8:

```
@FunctionalInterface  
public interface Runnable  
{  
    void run();  
}
```

# Manutenção da Compatibilidade

## INTERFACE RUNNABLE

- Implementação de Runnable sem Lambda

```
public class ThreadApp {  
    public static void main(String[] args) {  
        //objeto tipo anônimo que implementa Runnable  
        Runnable r = new Runnable() {  
            public void run() {  
                for (int i = 0; i <= 1000; i++) {  
                    System.out.println(i);  
                }  
            }  
        };  
        new Thread(r).start();  
    }  
}
```



# Manutenção da Compatibilidade

## INTERFACE RUNNABLE

- Implementação de Runnable com Lambda

```
public class ThreadAppLambda {  
    public static void main(String[] args) {  
        //criação de um Lambda  
        Runnable r = () ->  
        {  
            for (int i = 0; i <= 1000; i++)  
                System.out.println(i);  
        };  
        //criando uma Thread passando como argumento um Lambda  
        new Thread(r).start();    }  
}
```

# Exercícios

- 1) Implementar as classes **ThreadApp** e **ThreadAppLambda**.
- 2) Quais as diferenças observadas nessas duas classes?



# **Principais Utilizações das Expressões Lambdas**

# Expressões Lambda

## PRINCIPAIS UTILIZAÇÕES

- 1) Substituição das Classes Anônimas utilizadas com muita frequência para ativação de eventos em Interfaces Gráficas com Java.

**//Sem Lambda**

```
btn.setAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        System.out.println("Hello!");  
    }  
});
```

**//Com Lambda**

```
btn.setAction(event -> System.out.println("Hello!"));
```

# Expressões Lambda

## PRINCIPAIS UTILIZAÇÕES

- 2) Realização de operações em objetos do *Framework Collections* de forma mais objetiva e concisa.

```
//Sem Lambda
for (String s : list) {
    System.out.println(s);
}

//Com Lambda
list.forEach(System.out::println);
```

# Lambda – Considerações Finais

## VANTAGENS

- Facilita a programação paralela;
- Possibilita a escrita de código mais compacto;
- Oferece um suporte extensivo às coleções e estruturas de dados;
- Permite o desenvolvimento de uma coleção de APIs mais enxutas.

## DESVANTAGENS

- Não são reutilizáveis;
- Limitados no seu escopo;
- Não podem ser testados de forma isolada;
- Dificuldade de manutenção;
- Diminuição da legibilidade do código;
- Dificuldade de depuração dos erros.

# Lambda – Considerações Finais

- Interface Funcional é aquela que possui apenas um método abstrato;
- O método gerado pela expressão lambda deve possuir a mesma assinatura do método abstrato definido na Interface Funcional.
- O tipo da expressão lambda deve ser o mesmo do método abstrato definido na Interface Funcional.

# Lambda – Considerações Finais

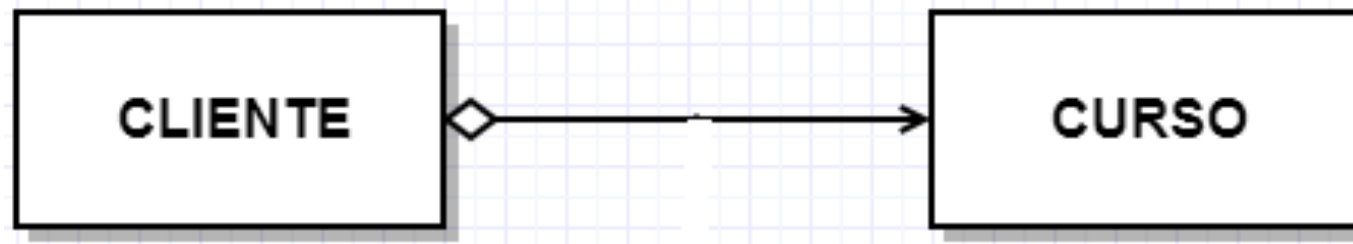
**“Qualquer tolo consegue escrever código para um computador entender. Bons programadores escrevem código que humanos consigam entender.”**

*Martin Fowler*



# Exercício

- 1) No exercício feito na Unidade 4 (classes **Cliente** e **Curso**), criar o método **listarCursos(...)** fazendo uso do recurso Lambda.





# Referência de Métodos

# Referência de Métodos

- Uma Expressão Lambda é a forma utilizada para a criação de uma função anônima! Contudo, o que acontece se a função desejada já tiver sido escrita?
- A **Referência de Métodos** pode ser utilizada para fazer uso de uma função já existente, no lugar de uma Expressão Lambda.
- No Java 8 é possível a passagem por referência de métodos via operador `::` Sendo assim, uma Expressão Lambda pode ser escrita da seguinte forma:

```
Function<String,Integer> f = s -> s.length();  
Function<String,Integer> f = String::length;
```

- Fica implícito que deverá ser chamado o método **length** do objeto String passado como parâmetro

# Referência de Métodos

- Esta abordagem é utilizada tanto para métodos de classe quanto para métodos de objeto, bem como para construtores também.
- **Exemplo - Método de Objeto**

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}
```

```
Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted); // "J"
```

# Referência de Métodos

```
Something something = new Something();  
Converter<String, String> converter = something::startsWith;  
String converted = converter.convert("Java");  
System.out.println(converted); // "J"
```

- A sintaxe é bem simples: define-se inicialmente o tipo (nesse caso a classe **Something**), seguido do delimitador `::` e o nome do método, sem parênteses.

- **Exemplo - Método de Classe**

```
Converter<String, Integer> converter =  
    (from) -> Integer.valueOf(from);  
//Substituição  
Converter<String, String> converter = Integer::valueOf;
```

# Referência de Métodos

- Exemplo - Construtor

```
Supplier<Usuario> criadorDeUsuarios = Usuario::new;  
Usuario novo = criadorDeUsuarios.get();
```

- A interface `Supplier<T>` representa uma *factory*. Pode-se guardar a expressão `Usuario::new` em um `Supplier<Usuario>` e chamar o seu método `get` sempre que for desejada a instanciação de um novo objeto do tipo `Usuario`.

# Referência de Métodos

- RESUMO

Method Reference Type	Syntax	Example
Static	ClassName::StaticMethodName	String::valueOf
Constructor	ClassName::new	String::new
Specific object instance	objectReference::MethodName	System.out::println
Arbitrary object of a particular type	ClassName::InstanceMethodName	String::toUpperCase

© [Adib Saikali](#)

# Exercícios

- 1) Alterar o método **listarCursos(...)** do exercício anterior para fazer uso de **referência de método**.
- 2) No exercício anterior, alterar a chamada ao método **listarCursos(...)** para fazer uso de **Referência de Métodos**. Utilizar a interface funcional **Consumer<T>**.





# Métodos Default

# Métodos Default

- A partir do Java 8, as Interfaces já podem possuir implementação de métodos! A esses métodos é dado o nome de *default* e são identificados pelo uso da palavra reservada **default**.

```
interface Formula {  
    double calcular(int a);  
    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}
```

- As classes que implementarem a interface Formula só terão a obrigação de implementar o método abstrato **calcular(...)**.
- Um classe pode implementar várias interfaces e herdar vários métodos **default**. Se houver dois métodos **default** iguais, um deles deverá ser anulado com a anotação **@Override**.

# Métodos Default

```
interface Foo {
    default void talk() {
        out.println("Foo!");
    }
}

interface Bar {
    default void talk() {
        out.println("Bar!");
    }
}

class FooBar implements Foo, Bar {
    @Override
    void talk() {
        Foo.super.talk();
    }
}
```

- No código acima, o método **talk()** da interface **Bar** foi anulado pela classe **FooBar**. O método será chamado da interface **Foo**.

# Métodos Default

- Os métodos **default** não podem ser acessados por Expressões Lambda.

```
Formula formula = (a) -> sqrt( a * 100);  
//Erro de compilação
```

- Os métodos defaults foram adicionados para permitir que as interfaces evoluíssem sem “quebrar” o código existente!
- Métodos default permitem evoluir uma API antiga de maneira não disruptiva, sem romper com o passado.
- A partir do Java 8, as Interfaces já podem possuir métodos **default** de classe (estáticos)!

# Métodos Default

- Uma interface pode possuir vários métodos **default** e permanecer funcional.

```
@FunctionalInterface
public interface Iterable {
    Iterator iterator();
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

- A interface **java.lang.Iterable** é mãe de **Collection**, que por sua vez é mãe de **List**.
- Como **ArrayList** implementa **List**, ele implementa este método.

# Exercício

- 1) Alterar a classe do exercício anterior para implementar a **interface Formula**.
- 2) Ao incluir a anotação **@FunctionalInterface** na **interface Formula**, é gerado algum erro de código? E se houver mais **métodos default**? E se houver mais métodos abstratos?



# Streams

# Streams

- **AVANÇOS NO FRAMEWORK COLLECTIONS**

- O *Framework Collections* já possui mais de 15 anos. A cada nova versão do Java ele sofre modificações. Contudo, nenhuma se compara com as mudanças realizadas pelo Java 8!
- O Java 8 introduziu o **Stream**. O **Stream** traz para o Java uma forma mais funcional de se trabalhar com coleções, usando uma interface específica para isso.
- Como já foi visto anteriormente, um Stream representa uma abstração de um “fluxo de dados”. No caso do Java 8, foi criado um pacote específico (**java.util.stream**) para permitir manipulações e transformações em coleções.



# Streams

- A interface **Stream<T>** representa uma sequencia de elementos nos quais uma ou mais operações (*filter, sorted, map, match, count, reduce*) poderão ser executadas.
- Essas operações poderão ser **intermediárias ou terminais**. A primeira retorna um Stream para ser utilizado em outra operação. A segunda já retorna o resultado esperado.
- Um novo método **default stream()** foi adicionado à interface **Collection**. Como esta interface é “pai” de todas as outras coleções, todas elas herdam esse novo método.

```
default Stream<E> parallelStream()  
default Stream<E> stream()
```

- A execução dessas operações se dá de forma sequencial (**Collection.stream()**) ou de forma paralela (**Collection.parallelStream()**).

# Streams

```
Class StreamApp { ...  
    //listagem de funcionários  
    List<String> funcionarios = new ArrayList<>();  
    funcionarios.add("Antonio");  
    funcionarios.add("José");  
    funcionarios.add("Pedro");  
    funcionarios.add("João");  
    funcionarios.add("Andreia");  
    //criação de um stream de funcionários para  
    //manipulação dos seus objetos  
    Stream<String> stream = funcionarios.stream();  
    ...}
```

- É importante ressaltar que o Stream gerado acima não é uma outra coleção. Ele não possui uma estrutura de dados interna para armazenar cada um dos elementos! Na verdade, ele usa uma coleção para executar uma serie de operações de forma sequencial ou paralela.

# Streams



© Adib Saikali

- Um Stream possui:
  - **Source** – representa a fonte dos objetos a serem manipulados;
  - **Pipeline** – representa o conjunto de operações intermediárias que irão atuar nos elementos do Stream;
  - **Terminal** – representa a operação final que irá extrair os elementos desejados.
- É importante ressaltar que as operações intermediárias (**pipeline**) não executam, apenas preparam as estruturas para execução.
- Nas operações finais (**terminal**) as operações intermediárias (**pipeline**) são executadas e é produzido um resultado.

# Operações com Streams

- **FILTER**

- É o método definido na interface `Stream<T>` que realiza um 'filtro' nos elementos de uma coleção. **É uma operação intermediária.**

```
funcionarios.stream().filter((s) -> s.startsWith("a")).  
    forEach(System.out::println);  
// Filtra todos os funcionários cujos nomes iniciam  
// com a letra "a" => Resultado "Antonio", "Andreia"
```

- **SORTED**

- É o método definido na interface `Stream<T>` que realiza ordena os elementos de uma coleção. **É uma operação intermediária.**

```
funcionarios.stream().sorted()  
    .filter((s) -> s.startsWith("a"))  
    .forEach(System.out::println);  
// "Andreia", "Antonio"
```

# Streams

- **MAP**
- É o método definido na interface **Stream<T>** que converte cada elemento recebido em um outro objeto, de acordo com a função passada. **É uma operação intermediária.**

```
funcionarios.stream().map(String::toUpperCase)
                .sorted((a, b) -> b.compareTo(a))
                .forEach(System.out::println);
//
```

- É importante observar que a interface **Map** não prove suporte ao método **stream()**.

# Streams

- **MATCH**

- É o método definido na interface **Stream<T>** que verifica se determinada condição é atendida. **É uma operação final.**

```
boolean anyStartsWithA =  
funcionarios.stream().anyMatch((s) -> s.startsWith("a"));  
System.out.println(anyStartsWithA); // true  
//Verifica se algum funcionário inicia com a letra "a"  
boolean allStartsWithA =  
funcionarios.stream().allMatch((s) -> s.startsWith("a"));  
System.out.println(allStartsWithA); // false  
//Verifica se todos os funcionários iniciam com a letra "a"  
boolean noneStartsWithZ =  
funcionarios.stream().noneMatch((s) -> s.startsWith("z"));  
System.out.println(noneStartsWithZ); // true  
//Verifica se nenhum funcionário inicia com a letra "z"
```

# Streams

- **COUNT**

- É o método definido na interface **Stream<T>** que retorna o número de elementos existentes na stream. **É uma operação final.**

```
long startsWithA =  
usuarios.stream().filter((s) -> s.startsWith("a")).count();  
System.out.println(startsWithA); // 2
```

- **REDUCE**

- É o método definido na interface **Stream<T>** que realiza a redução nos elementos existentes na stream, de acordo com a função passada. **É uma operação final.**

```
Optional<String> reduced = funcionarios.stream().sorted()  
.reduce((s1, s2) -> s1 + "#" + s2);  
reduced.ifPresent(System.out::println);  
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

# Exercícios

- 1) Escreva a classe Java StreamApp.
- 2) Escreva os métodos abaixo na classe StreamApp.

```
public static void filtrar(String letra){...}  
public static void ordenar(String letra){...}  
public static void contar(String letra) {...}
```





# Parallel Streams

# Parallel Streams

- Como foi citado anteriormente, um Stream pode ser sequencial ou paralelo. A principal diferença é que no primeiro utiliza-se apenas uma thread, enquanto que no segundo são utilizadas várias threads. Assim sendo, o segundo tipo de Stream é muito mais 'performático' que o primeiro.
- Criação de uma coleção de String com valores (ids) únicos.

```
public class ParallelStreamApp { ...  
    int max = 1000000;  
    List<String> values = new ArrayList<>(max);  
    for (int i = 0; i < max; i++) {  
        UUID uuid = UUID.randomUUID();  
        values.add(uuid.toString()); }  
    ...}
```

© Benjamin Winterberg

# Parallel Streams

- Ordenação com Stream 'sequencial'

```
long t0 = System.nanoTime();  
long count = values.stream().sorted().count();  
System.out.println(count);  
long t1 = System.nanoTime();  
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);  
System.out.println(String.format("Tempo: %d ms", millis));
```

- Ordenação com Stream 'paralela'

```
long t0 = System.nanoTime();  
long count = values.parallelStream().sorted().count();  
System.out.println(count);  
long t1 = System.nanoTime();  
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);  
System.out.println(String.format("Tempo: %d ms", millis));
```

© Benjamin Winterberg

# Exercícios

- 1) Implemente a classe **ParallelStreamApp** que possua dois métodos: **colecacaoStream()** e **colecacaoParallelStream()**.
  - a) Cada método irá cadastrar uma coleção de 1.000.000 de elementos do tipo String únicos;
  - b) Após o cadastro, essa coleção será ordenada;
  - c) Por fim, os elementos dessa coleção serão impressos.
- 2) Quais as diferenças de tempo observadas na chamada dos métodos **colecacaoStream()** e **colecacaoParallelStream()**?



# Melhorias na API Stream

# OBTENÇÃO DE STREAM A PARTIR DE VALORES OU VETORES

- **Definição**

- Para obter uma stream a partir de valores ou vetores basta chamar os métodos estáticos `Stream.of()` ou `Arrays.stream()`, como mostra o código a seguir:

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4, 5);  
  
IntStream numbersFromArray = Arrays.stream(new int[] {1, 2, 3, 4, 5});
```

- **Método `takeWhile()`**

- O método `takeWhile` mantém os elementos enquanto a condição especificada for verdadeira e para assim que encontra o primeiro elemento que não satisfaça a condição.

```
//Novo método takewhile  
public static void exemplo01()  
{  
    List<Integer> list  
    = Stream.of(1,2,3,4,5,6,7,8,9,10)  
        .takeWhile(i -> (i % 2 == 0)).collect(Collectors.toList());  
    System.out.println(list);  
}
```

# OBTENÇÃO DE STREAM A PARTIR DE VALORES OU VETORES

- Método **dropWhile()**
  - O método **dropWhile** descarta os elementos iniciais até que uma determinada condição seja verdadeira e retorna os elementos restantes.

```
//Novo método dropWhile
public static void exemplo02()
{
    List<Integer> list
    = Stream.of(2,2,3,4,5,6,7,8,9,10)
        .dropWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
    System.out.println(list);
}
```

# OBTENÇÃO DE STREAM A PARTIR DE VALORES OU VETORES

- Método **ofNullable()**
  - O método **ofNullable** é útil para evitar verificações de nulidade antes de criar um stream com um único elemento.

```
//Novo método ofNullable
public static void exemplo03()
{
    String nome = "Alice";
    Stream<String> stream = Stream.ofNullable(nome);
    stream.forEach(System.out::println);

    nome = null;
    stream = Stream.ofNullable(nome);
    stream.forEach(System.out::println);
}
```



# OBTENÇÃO DE STREAM A PARTIR DE VALORES OU VETORES

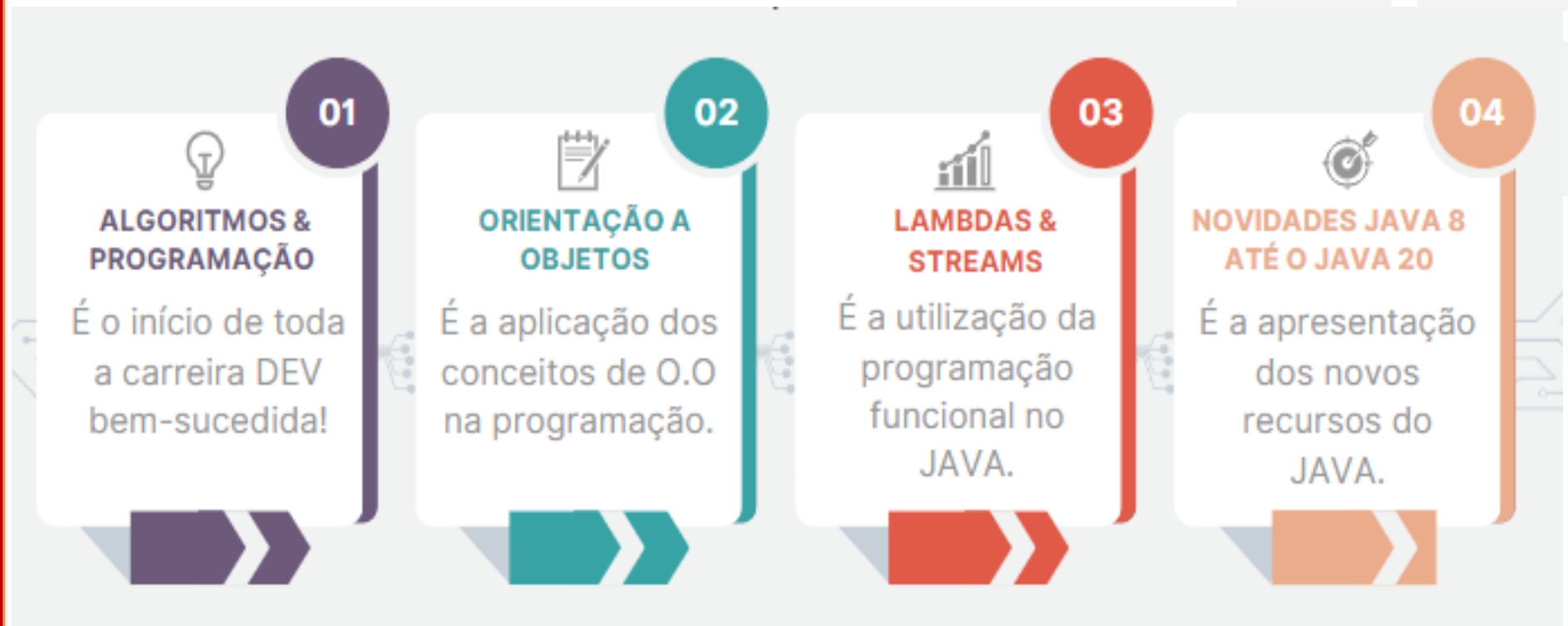
- Método **iterate()**
  - O método **iterate** é um dos métodos estáticos fornecidos pela classe Stream no Java que permite a geração de uma sequência de elementos com base em um valor inicial e uma função de iteração.

```
//Novo método iterate|
public static void exemplo04()
{
    Stream.iterate(1, i -> i <= 10, i -> i*3)
        .forEach(System.out::println);
}
```



# Conclusão

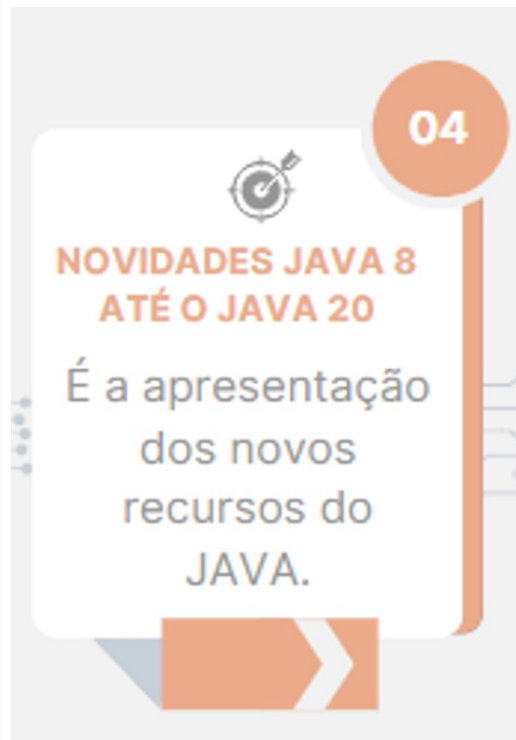
# A Nossa Trilha **DEV JAVA FULL STACK**



# A Nossa Trilha **DEV JAVA FULL STACK**



# A Nossa Trilha DEV JAVA FULL STACK



## Curso

### Domine as Inovações do Java com Spring Boot & MongoDB

As Mudanças mais Importantes do Java 8 até o Java 20



**Prof. Msc. Antonio B. C. Sampaio Jr**  
engenheiro de software & professor

@abctreinamentos  
@amazoncodebr

www.abctreinamentos.com.br  
www.amazoncode.com.br

