

## **9 - Construções Funcionais**

### **Métodos Vs Funções**

Nas linguagens funcionais, as funções produzem resultados exclusivamente a partir dos seus parâmetros (sem efeitos colaterais).

Em java, é possível escrever métodos que recebam parâmetros e apenas trabalhem com estes, produzindo um resultado. No entanto um método pertence sempre a uma classe,

Portanto, um método não é uma função.

## **9 – Construções Funcionais**

### **Métodos Vs Funções**

- Um método só pode ser usado quando é realizada a sua invocação, isto significa que os seus argumentos são de imediato calculados para que o método seja executado. (Os argumentos são passados por valor)

Isto significa que os métodos não podem ser passados como parâmetro de outro método, nem podem ser resultado de um método.

## **9 – Construções Funcionais**

### **Interfaces Funcionais**

- Interface com um único método abstrato a implementar.

***Anotação informativa para o compilador: @FunctionalInterface***

Interfaces funcionais são uma forma de definir tipos de funções que podem ser usados como tipos de dados. Podem ser passadas como parâmetros de métodos e devolvidas como tipos de resultado.

**Instâncias de interfaces funcionais podem ser criadas com expressões lambda, referências de métodos ou construtores.**

## 9 – Construções Funcionais

### Funções anónimas (expressões lambda) em java

Sintaxe: **(argument-list) -> {body}**

Lista de argumentos – Pode ser ou não vazia;

```
() -> {  
    // ...  
}
```

```
(p1) -> {  
    // ...  
}
```

```
(p1,p2) -> {  
    // ...  
}
```

## 9 – Construções Funcionais - exemplos

```
public interface Drawable {  
    public void draw();  
}  
public class Nova {  
    public static void main(String[] args) {  
        int width=10;  
  
        //Implementação da interface com uma classe anónima  
  
        Drawable d = new Drawable(){  
            public void draw(){System.out.println("Drawing "+ width);}  
        };  
        d.draw();  
    }  
}
```

“Anonymous inner class”  
que implementa a  
interface Drawable

Output: Drawing 10

//Exemplos retirado de <https://www.javatpoint.com/java-lambda-expressions>

## 9 – Construções Funcionais - exemplos

```
public class Nova2 {  
    public static void main(String[] args) {  
        int width=10;
```

**//Implementar a interface com uma expressão lambda**

```
        Drawable d2 = ()-> {  
            System.out.println("Drawing "+width);  
        };  
        d2.draw();  
    }  
}
```

**Output: Drawing 10**

## 9 – Construções Funcionais - exemplos

```
interface Sayable{  
    public String say();  
}  
public class Nova3 {  
public static void main(String[] args) {
```

```
    Sayable s= () -> {  
        return "I have nothing to say."  
    };
```

```
    System.out.println( s.say() );
```

## 9 – Construções Funcionais - exemplos

```
interface Sayable2{
    public String say(String name);
}
public class Nova4 {

    public static void main(String[] args) {

        // expressão lambda com UM parâmetro
        Sayable2 s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println( s1.say("world") );
    } }
```

## 9 – Construções Funcionais - exemplos

// Podemos omitir os parêntesis da função

```
Sayable2 s2= name -> {  
    return "Hello, "+name;  
};  
System.out.println( s2.say("World") );  
}  
}
```

## 9 – Construções Funcionais - exemplos

```
interface Addable{
    int add(int a,int b);
}
public class Nova5 {
    public static void main(String[] args) {

        // Expressão lambda com vários parâmetros
        Addable ad1= (a,b) -> (a+b);

        System.out.println( ad1.add(10,20) );

        // Expressão lambda com vários parâmetros e tipos de dados
        Addable ad2=(int a,int b)->(a+b);

        System.out.println( ad2.add(100,200) );
    }
}
```

## 9 – Construções Funcionais - exemplos

```
// Expressão lambda sem return.
```

```
Addable ad3=(a,b)->(a+b);
```

```
System.out.println( ad3.add(10,20) );
```

```
// Expressão lambda com return
```

```
Addable ad4=(int a,int b)->{  
    return (a+b);  
};
```

```
System.out.println( ad4.add(100,200) );
```

```
}  
}
```

## 9 – Construções Funcionais

### Referências a métodos de interfaces funcionais\*

Em vez de usar uma expressão lambda podemos referir o método através de uma referência do tipo:

<nome da classe> :: < método static>

<nome da classe> :: < método de instância>

<referência do objeto> :: < método de instância>

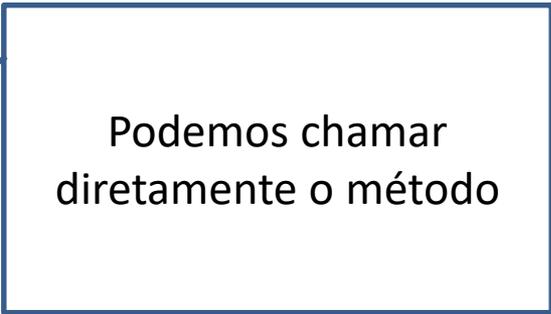
<nome da classe> :: new

\* <https://www.javatpoint.com/java-8-method-reference>

## 9 – Construções Funcionais

### Referências de métodos para referir métodos de interfaces funcionais - exemplo

```
interface Raiz {  
    double raizQuadrada(double x);  
}  
  
public class Nova7 {  
    public static void main(String[] args) {  
  
        // expressão lambda  
        Raiz r = (d) -> Math.sqrt(d);  
  
        // referência para método  
        Raiz q = Math::sqrt;  
  
        System.out.println(r.raizQuadrada(5));  
        System.out.println(q.raizQuadrada(5));  
    }  
}
```



Podemos chamar  
diretamente o método

## 9 – Construções Funcionais

### Interfaces funcionais pré-definidas em java

A partir do Java 8, todas as interfaces só com um método são interfaces funcionais. Exemplos:

```
public interface Comparable <T> {  
    public int compareTo ( T outro );  
}
```

```
public interface Comparator <T> {  
    public int compare ( T outro1, T outro2 );  
}
```

## 9 – Construções Funcionais

### Interfaces funcionais pré-definidas em java

Lembram-se de quando quisemos ordenar uma lista de Contas pelo nome do titular (ver T10)?

Definimos uma classe que implementava a interface `Comparator`,

```
import java.util.Comparator;
public class ComparaNomeConta implements Comparator<Conta>{
    @Override
    public int compare (Conta c1, Conta c2){
        return c1.getNome().compareTo (c2.getNome());
    }
}
```

E depois ordenamos uma `ArrayList<Conta>`:

```
Collections.sort(lista2, new ComparaNomeConta());
```

## 9 – Construções Funcionais

### Interfaces funcionais pré-definidas em java

Podemos agora, simplesmente associar o comparador a uma função anónima,

```
Comparator<Conta> novoCompara = (Conta c1, Conta c2)->  
    c1.getNome().compareTo (c2.getNome());
```

E depois passar o comparador para o método sort:

```
Collections.sort(lista2, novoCompara);
```

## 9 – Construções Funcionais

### O package `java.util.function`\*

Todas as interfaces de `java.util.function` são interfaces funcionais.

-O package possui seis interfaces base e depois várias especializações destas:

#### Interface

`Predicate<T>`

`Supplier <T>`

`Consumer <T>`

`Function < T, R >`

`UnaryOperator <T>`

`BinaryOperator <T>`

#### Método abstrato

`boolean test (T t)`

`T get()`

`void accept (T t)`

`R apply (T t)`

`T apply (T t)`

`T apply (T t1, T t2)`

## 9 – Construções Funcionais

### Interface

Predicate<T>

*// permite verificar uma propriedade de T*

```
import java.util.function.*;
```

```
public class Exemplo {
```

```
    public static void main(String[] args) {
```

```
        Conta cx = new Conta (1);
```

```
        cx.setNome("Mais rico");
```

```
        cx.setSaldo(2000000);
```

```
Predicate<Conta> milionario = (Conta c) -> c.getSaldo() > 1000000;
```

```
// ou apenas: Predicate<Conta> milionario = c -> c.getSaldo() > 1000000;
```

```
System.out.println(milionario.test(cx)) ;
```

### Método abstrato

boolean test (T t)

**Output: true**

## 9 – Construções Funcionais

### O package `java.util.function`\*

#### Interface

`Supplier <T>`

// permite gerar um objeto do tipo T

#### Método abstrato

`T get()`

```
Supplier<LocalDate> s = () -> LocalDate.now();
```

```
LocalDate time = s.get();
```

```
System.out.println(time);
```

## 9 – Construções Funcionais

### O package `java.util.function`\*

Interface

`Consumer <T>`

Método abstrato

`void accept (T t)`

// recebe um objeto do tipo T e opera sobre ele

`Conta cx = ....;`

`Consumer<Conta> juro = c -> c.setSaldo(c.getSaldo()*1.02);`

`juro.accept(cx);`

## 9 – Construções Funcionais

### O package `java.util.function`\*

#### Interface

`Function < T, R >`

#### Método abstrato

`R apply (T t)`

// recebe um parâmetro do tipo T e devolve um resultado do tipo R

```
Function<String, Integer> func = y -> y.length();
```

```
Integer apply = func.apply("POO");
```

```
System.out.println(apply); //3
```

## 9 – Construções Funcionais

### O package `java.util.function`\*

#### Interface

`UnaryOperator <T>`

#### Método abstrato

`T apply (T t)`

// recebe um parâmetro do tipo T e devolve um resultado do tipo T

```
UnaryOperator<Integer> func2 = x -> x * 2;
```

```
Integer result = func2.apply(2);
```

```
System.out.println(result);
```

## 9 – Construções Funcionais

### O package `java.util.function`\*

#### Interface

`BinaryOperator <T>`

#### Método abstrato

`T apply (T t1, T t2)`

// recebe dois parâmetros do tipo T e devolve um resultado do tipo T

```
BinaryOperator<Integer> func2 = (x1, x2) -> x1 + x2;
```

```
Integer result2 = func2.apply(2, 3);
```

```
System.out.println(result2); // 5
```