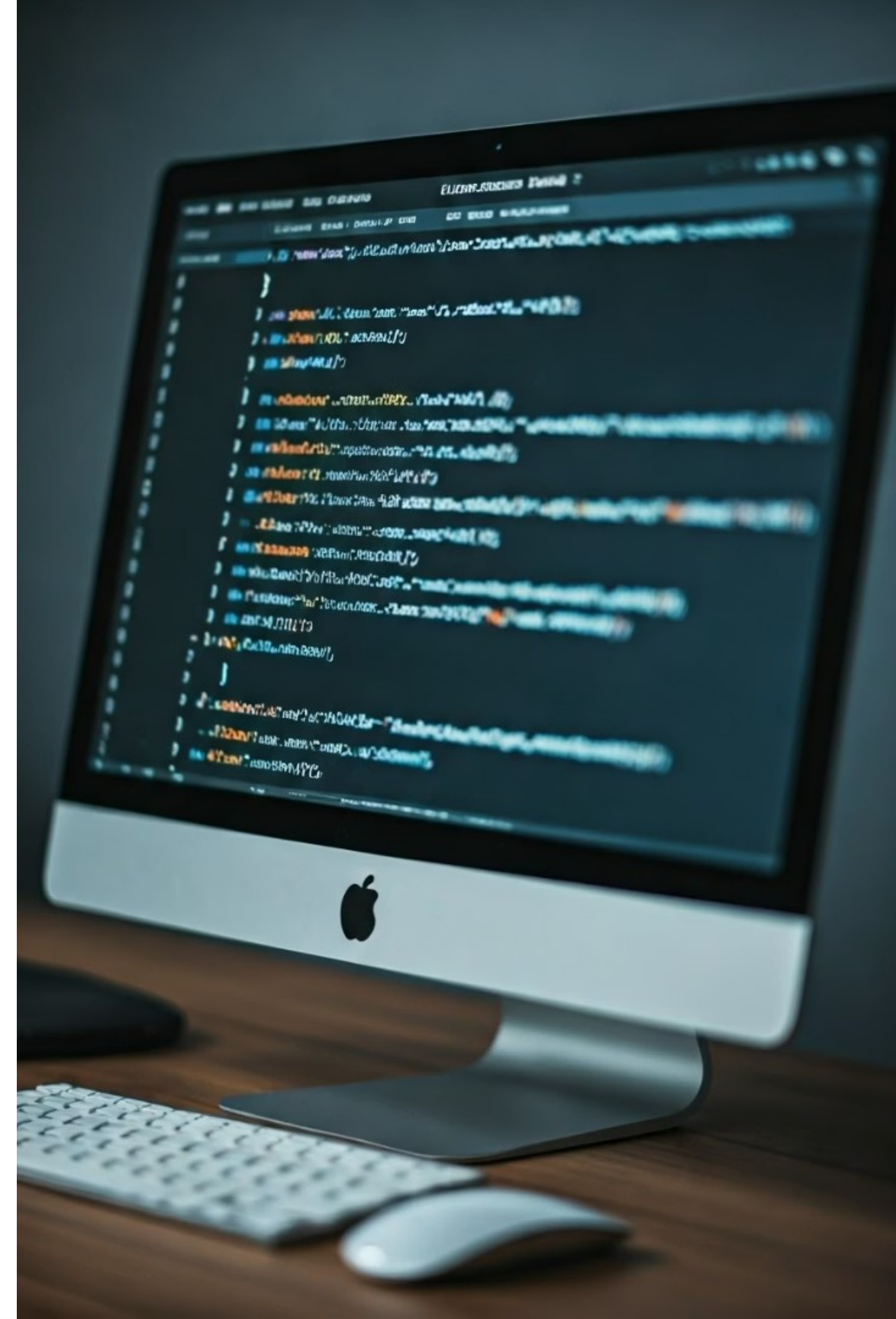


Programação Funcional em Java

- Paradigma que trata a computação como avaliação de funções
- Evita mudanças de estado e dados mutáveis.
- Recursos de programação funcional foram introduzidos no Java 8
 - adição de expressões lambda
 - pacote java.util.function
 - API Stream
 - Funções de order superior
 - imutabilidade



por Rodrigo Malara



Expressões Lambda

Definição

Expressões lambda são uma forma concisa de representar funções anônimas, fornecendo uma sintaxe clara para escrever interfaces funcionais.

Sintaxe

Substituem classes anônimas tradicionais com uma sintaxe mais simples e direta, usando o operador ->.

Benefício

São a pedra angular da programação funcional em Java, permitindo código mais conciso e legível.

Exemplo de expressão lambda comparada com classe anônima tradicional:

```
// Tradicional: Runnable runnable1 = new Runnable() { public void run()
{ System.out.println("Hello!"); } };
```

```
// Lambda: Runnable runnable2 = () -> System.out.println("Hello!");
```

Expressões Lambda

Exemplo de expressão lambda comparada com classe anônima tradicional:

```
1  ▶ public class Main {  
2  
3  ▶     public static void main(String[] args) {  
4      Runnable runnable1 = new Runnable() {  
5  ⓘ↑         public void run() {  
6              System.out.println("Hello!");  
7          }  
8      };  
9      Thread thread1 = new Thread(runnable1);  
10     thread1.start();  
11     }  
12 }  
13
```

Expressões Lambda

Exemplo de expressão lambda comparada com classe anônima tradicional:

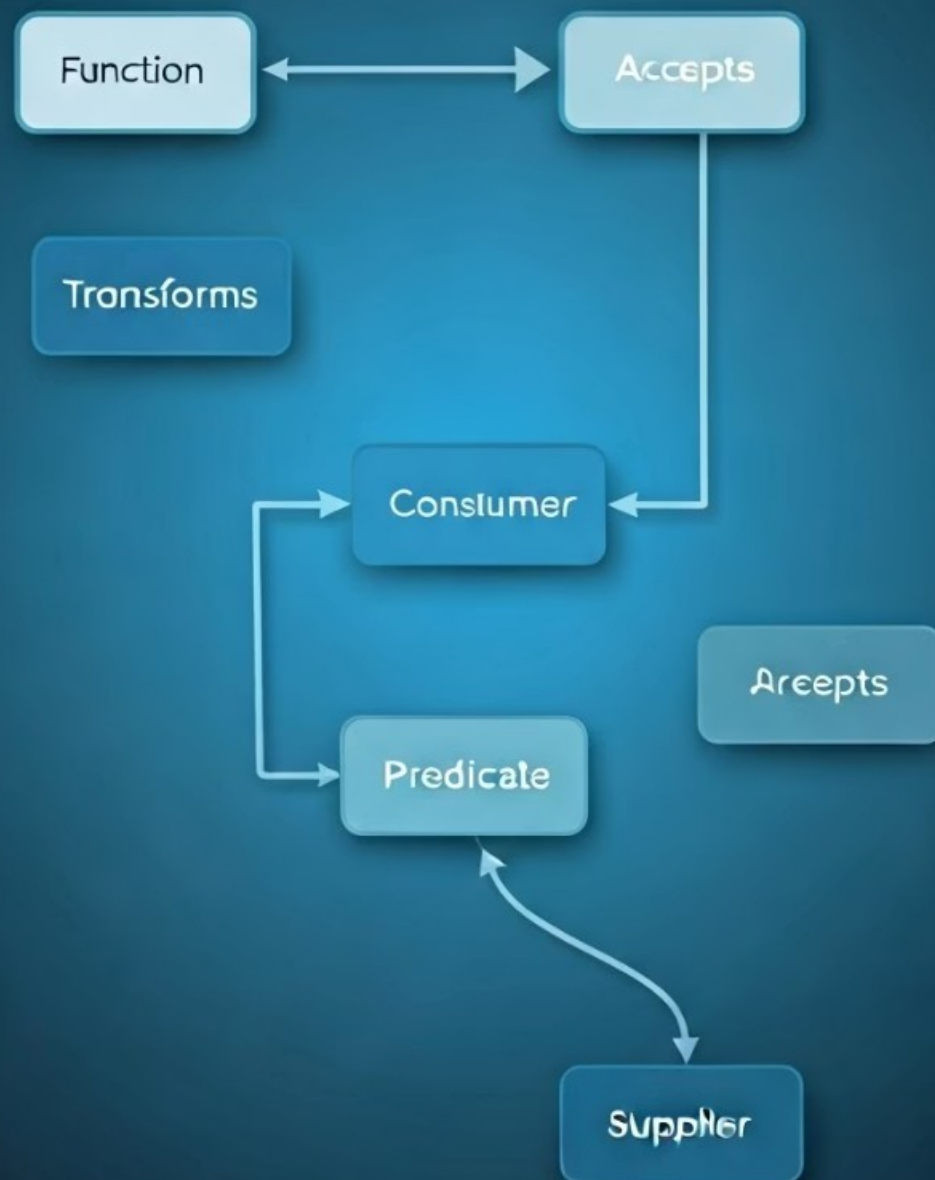
```
1  ▶ public class Main {  
2  
3  ▶     public static void vsmain(String[] args) {  
4      Runnable runnable1 = () -> System.out.println("Hello!");  
5      Thread thread1 = new Thread(runnable1);  
6      thread1.start();  
7      }  
8  }
```

Expressões Lambda

Exemplo de expressão lambda comparada com classe anônima tradicional:

```
55 @FunctionalInterface
56 public interface Runnable {
    When an object implementing interface Runnable is used to create a thread, starting the thread
    causes the object's run method to be called in that separately executing thread.
    The general contract of the method run is that it may take any action whatsoever.
    See Also: Thread.run()
68 public abstract void run();
69 }
```

Interface Funcional → Possui apenas 1 método abstrato e pode ser representada por um lambda



Interfaces Funcionais

</>

Definição

Interfaces com um único método abstrato, também chamadas de métodos funcionais.

$f(x)$

Anotação

Marcadas com `@FunctionalInterface` para garantir que contenham apenas um método abstrato.

λ

Implementação

Podem ser instanciadas usando expressões lambda:

```
Runnable myFunc = () -> System.out.println("Olá");
```

O pacote `java.util.function` fornece várias interfaces funcionais como `Function`, `Consumer`, `BiFunction`, `Predicate` e `Supplier`, que representam diferentes tipos de funções e são comumente usadas com expressões lambda.

API Stream



Filter

Filtra elementos com base em um predicado



Map

Transforma elementos aplicando uma função



Reduce

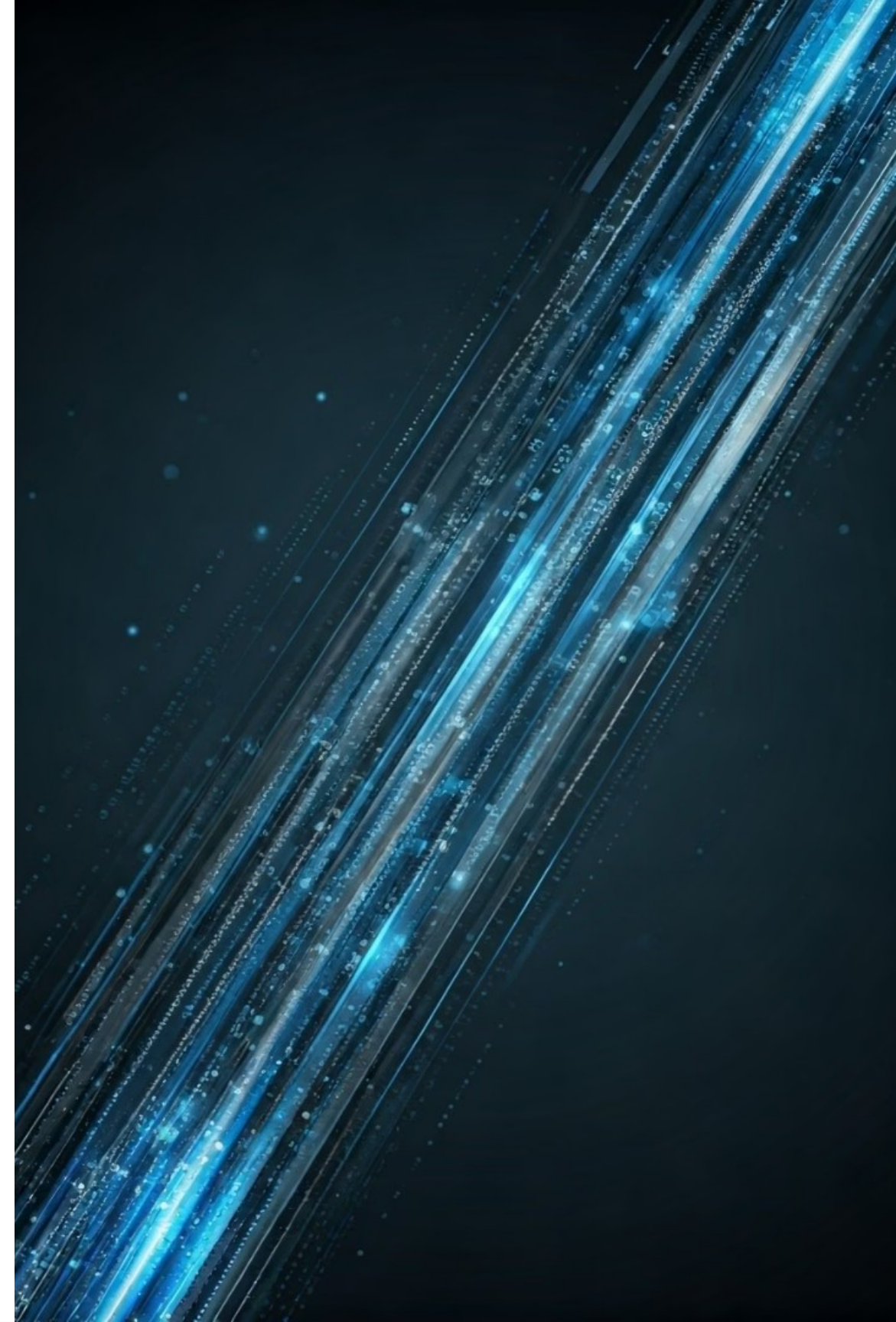
Combina elementos para produzir um resultado



Collect

Acumula elementos em uma coleção

Streams fornecem uma abordagem funcional para processar sequências de elementos. Eles permitem expressar manipulações complexas de dados usando um pipeline de operações, como map, filter e reduce.



Processamento de Coleções através de Streams

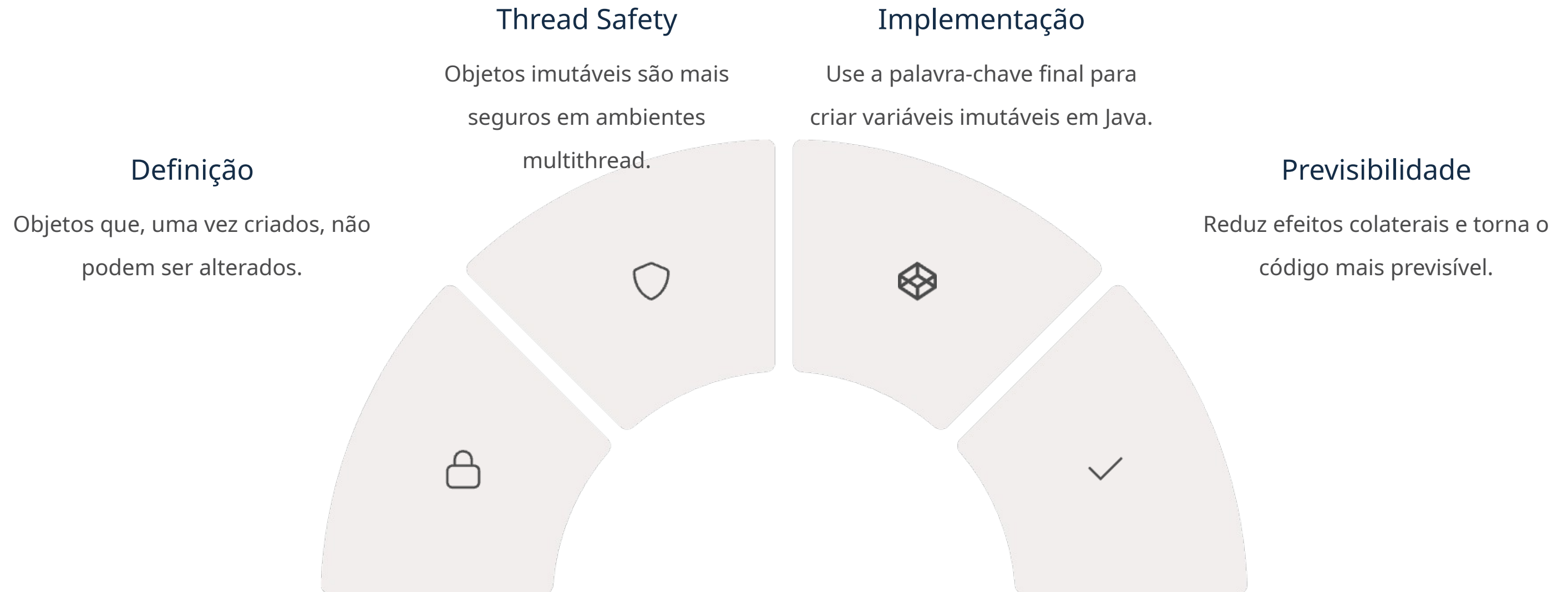
```
4 ▶ public class Main {
5
6 ▶     public static void main(String[] args) {
7         List<String> strings = List.of("algum", "bico", "carro", "dedo", "elefante");
8
9         String resultado = strings.stream()
10             .filter(String s -> s.contains("a")) // apenas strings que contem a
11             .map(String::toUpperCase) // converta para MAIÚSCULAS
12             .collect(Collectors.joining(delimiter: ", ")); // Junte tudo, separado por vírgula
13
14         System.out.println(resultado);
15     }
16 }
```

Run Main x

/home/i12/tools/jdk-11.0.23+9/bin/java ...
ALGUM, CARRO, ELEFANTE


Process finished with exit code 0

Imutabilidade



A imutabilidade é crucial em aplicações multithread. Ela permite que uma thread atue em um objeto imutável sem se preocupar com outras threads, pois sabe que ninguém está modificando o objeto. Isso torna os objetos imutáveis mais seguros que os mutáveis.




Exemplo de variável imutável em um lambda

```
4  ▶ public class Main {  
5  
6  ▶ public static void main(String[] args) {  
7      List<String> strings = List.of("algum", "bico", "carro", "dedo", "elefante");  
8      int valor = 10;  
9  
10     String resultado = strings.stream()  
11         .filter( String s -> {  
12              valor = valor - 20; // valor não pode ser alterado - imutável  
13             retu  
14         }) // ap  
15         .map(Str  
16         .map()  
17         .collect  
18  
19     System.out.println(resultado);  
20 }  
21 }
```

Variable used in lambda expression should be final or effectively final

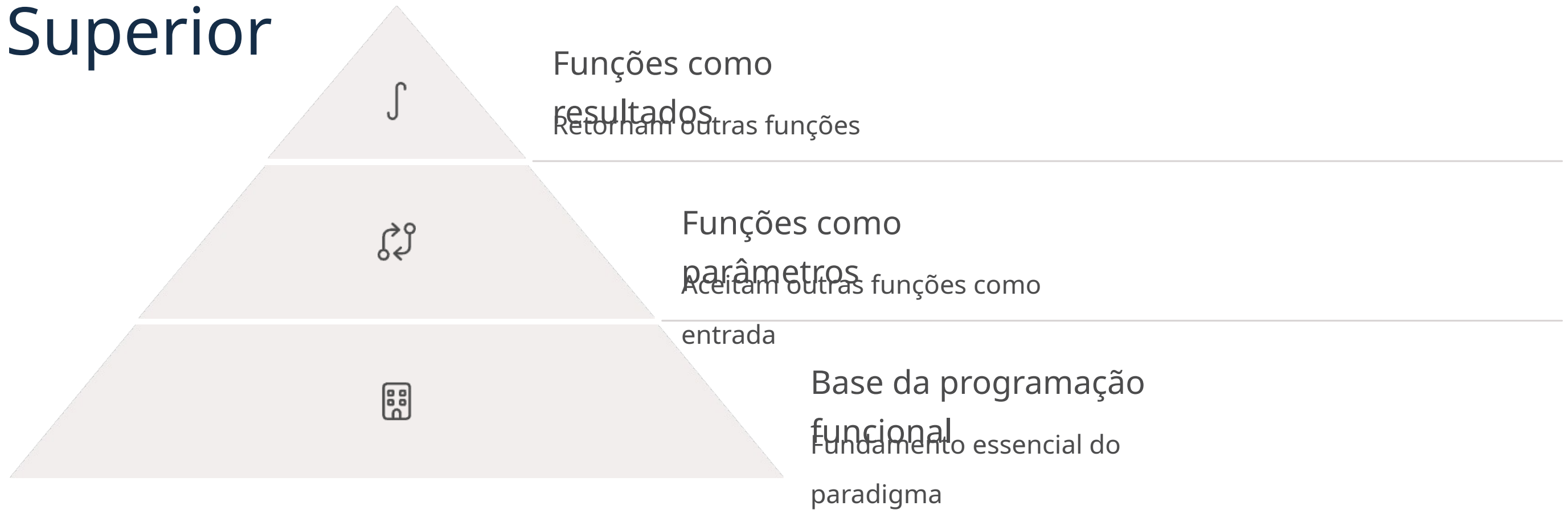
[Convert to atomic](#) Alt+Shift+Enter [More actions...](#) Ctrl+1 Alt+Enter

`int valor = 10`

 hxp-user-service  

separado por vírgula

Funções de Ordem Superior



Funções de ordem superior são funções que podem receber outras funções como parâmetros ou retornar funções como resultados. Este conceito-chave na programação funcional permite um estilo de codificação mais expressivo e modular.

Exemplo:

```
public static void processNumbers(List numbers, Function processor) { for (int i = 0; i < numbers.size(); i++) { numbers.set(i, processor.apply(numbers.get(i))); } }
```

Exemplo de funções de ordem-superior

```
1  import java.util.List;
2  import java.util.stream.Collectors;
3
4  ▶ public class Main {
5
6  ▶     public static void main(String[] args) {
7         List<String> strings = List.of("algum", "bico", "carro", "dedo", "elefante");
8
9         String resultado = strings.stream()
10            // filter e map são funções de ordem superior
11            // recebem uma função ou metodo como argumento
12            .filter(String s -> s.contains("a"))
13            .map(String::toUpperCase)
14            .collect(Collectors.joining(delimiter: ", ")); // Junte tudo, separado por vírgula
15
16         System.out.println(resultado);
17     }
18 }
```




Paralelismo

2x

Desempenho

Potencial de aumento significativo em
operações intensivas

N

Escalabilidade

Escala com o número de núcleos
disponíveis

1

Simplicidade

Uma simples mudança de `.stream()`
para `.parallelStream()`

A programação funcional incentiva a escrita de código que pode ser facilmente paralelizado. A API Stream fornece métodos para execução paralela de operações em streams, permitindo aproveitar sistemas multicore.

Exemplo:

Exemplo de paralelismo com programação funcional

```
4  ▶ public class Main {  
5  
6  ▶     public static void main(String[] args) {  
7      List<Integer> numeros = List.of(10, 20, 30, 40, 50);  
8  
9  R  ⚡ int soma = numeros.parallelStream() Stream<Integer>  
10      .mapToInt(Integer::intValue) IntStream  
11      .sum();  
12  
13      System.out.println(soma);  
14  }  
15  }
```

Run Main ×

/home/i12/tools/jdk-11.0.23+9/bin/java ...

150

Benefícios da Programação Funcional em Java



Concisão

Expressões lambda tornam o código mais conciso e legível.



Paralelismo

Facilidade para paralelizar código devido à imutabilidade e ausência de estado.



Previsibilidade

Imutabilidade reduz efeitos colaterais e torna o código mais previsível.



Testabilidade

Funções sem efeitos colaterais são mais fáceis de testar.

A programação funcional em Java complementa o paradigma de programação orientada a objetos existente e fornece aos desenvolvedores ferramentas poderosas para escrever código mais expressivo, modular e de fácil manutenção.

Ela promove o uso de funções puras, imutabilidade e funções de ordem superior, resultando em código frequentemente mais conciso e fácil de entender.

