Interfaces Funcionais Java

# 1. Runnable

Finalidade:

Executa uma tarefa sem retorno de valor. Muito usada para threads. Possui o método run() que não recebe argumentos nem retorna nada.

Exemplo de código:

Runnable tarefa = () -> System.out.println("Executando uma tarefa!");

tarefa.run();

Explicação:

A interface Runnable foi usada para representar uma ação que será executada sem entrada e sem retorno, no exemplo, ela define uma tarefa simples que imprime uma mensagem. A interface permite encapsular essa tarefa como um objeto funcional, é especialmente útil quando queremos passar ações para execução posterior e permite usar lambdas para definir comportamentos compactamente.

# 2. Consumer<T>

Finalidade:

Recebe um argumento do tipo T e executa uma ação com ele, sem retornar nada.

Exemplo de código:

Consumer<String> imprimir = nome -> System.out.println("Olá, " + nome);

imprimir.accept("Gabriel");

Explicação:

A interface Consumer é usada para representar uma operação que consome um dado (entrada) sem gerar saída. No exemplo, ela consome uma String e realiza uma ação com ela (imprimir), ela permite aplicar uma lógica em cada elemento recebido, ideal para situações onde queremos apenas processar valores. Facilita o uso de lambdas como operações reutilizáveis.

# 3. BiConsumer<T, U>

Finalidade:

Recebe dois argumentos (T e U) e realiza uma ação com eles, sem retornar nada.

Exemplo de código:

BiConsumer<String, Integer> mostrarIdade = (nome, idade) ->

System.out.println(nome + " tem " + idade + " anos.");

mostrarIdade.accept("Ana", 25);

Explicação:

BiConsumer funciona como o Consumer, mas, com dois parâmetros de entrada, no exemplo, ela consome um nome e uma idade, e executa uma ação com ambos. A interface torna possível representar operações com pares de dados, muito útil para operações com mapas ou listas de pares, e permite definir comportamentos que envolvam dois elementos relacionados.

# 4. Function<T, R>

Finalidade:

Recebe um valor do tipo T e retorna um valor do tipo R.

Exemplo de código:

Function<Integer, String> parOuImpar = num -> num % 2 == 0 ? "Par" : "Ímpar";

System.out.println(parOuImpar.apply(7));

Explicação:

A interface Function representa uma transformação de um dado em outro, no exemplo, transforma um número inteiro em uma String que indica "Par" ou "Ímpar". Ela é usada sempre que precisamos aplicar uma lógica de conversão. Permite criar funções reutilizáveis que encapsulam lógica de transformação e facilita o uso de expressões funcionais em fluxos de dados.

# 5. BiFunction<T, U, R>

Finalidade:

Recebe dois argumentos (T, U) e retorna um resultado (R).

Exemplo de código:

BiFunction<Integer, Integer, Integer> soma = (a, b) -> a + b;

System.out.println(soma.apply(10, 5));

Explicação:

BiFunction é usada quando queremos transformar dois valores em um único resultado,  
no exemplo, soma dois inteiros e retorna o total. A interface permite combinar dois inputs em um output de forma clara, útil em cenários de cálculo, comparação ou concatenação e ajuda a isolar e reutilizar lógica que envolve múltiplas entradas.

# 6. Supplier<T>

Finalidade:

Não recebe argumento algum, mas retorna um valor do tipo T.

Exemplo de código:

Supplier<Double> gerarAleatorio = () -> Math.random();

System.out.println(gerarAleatorio.get());

Explicação:

A interface Supplier fornece um valor sob demanda, sem precisar de entrada, no exemplo, ela gera um número aleatório quando chamada. Ela encapsula a lógica de criação ou fornecimento de valores, muito útil para gerar valores dinâmicos, como IDs ou datas, permite implementar comportamentos preguiçosos (lazy loading).

# 7. Predicate<T>

Finalidade:

Recebe um valor do tipo T e retorna um boolean. Usado para testar condições.

Exemplo de código:

Predicate<String> ehVazia = s -> s.isEmpty();

System.out.println(ehVazia.test(""));

Explicação:

A interface Predicate representa uma condição que retorna true ou false, permitindo a definição de testes lógicos de forma compacta e reutilizável. No exemplo estudado, ela é usada para verificar se uma string está vazia, esse recurso é muito útil para filtragem e validação de dados em aplicações Java, além disso, facilita o uso de expressões booleanas dentro de pipelines de processamento, seu uso torna o código mais limpo, modular e fácil de manter.