Java Developer

Java Core

Sumário

Objetivos deste Módulo

Ao final deste módulo, objetiva-se que o aluno adquira:

Conhecimentos:

- Generics
- Streams, lambdas e optional

1. Generics

Os **generics** (genéricos) são um recurso que permite criar classes, interfaces e métodos que podem ser parametrizados por um ou mais tipos, permitindo que você escreva código mais flexível, seguro e reutilizável. Os generics fornecem a capacidade de criar componentes que funcionam com tipos específicos sem se vincular a um tipo particular.

A principal vantagem dos generics em Java é a segurança de tipo em tempo de compilação, o que significa que erros relacionados a tipos são detectados pelo compilador antes de o código ser executado. Isso ajuda a evitar erros de tempo de execução e facilita a manutenção do código.

Ao criar uma classe, interface ou método genérico, você pode usar parâmetros de tipo para representar tipos que serão determinados em tempo de uso. Esses parâmetros de tipo são colocados entre ângulos ("<" e ">") após o nome da classe, interface ou método. Por exemplo:

```
public class Caixa<T> {
    private T conteudo;

    public Caixa(T conteudo) {
        this.conteudo = conteudo;
    }

    public T getConteudo() {
        return conteudo;
    }
}
```

Neste exemplo, T é um parâmetro de tipo que representa o tipo do conteúdo armazenado na Caixa. Ele será determinado quando você criar uma instância da classe Caixa.

Ao usar uma classe ou método genérico, você fornece o tipo concreto que deseja usar entre os ângulos ("<" e ">"). Isso é chamado de "parametrização". Por exemplo:

```
Caixa<String> caixaDeString = new Caixa<>("Texto");

String conteudo = caixaDeString.getConteudo();
```

Neste caso, **Caixa**<String> indica que estamos usando uma Caixa que armazena strings.

Você pode definir classes genéricas com múltiplos parâmetros de tipo. Por exemplo:

```
public class Par<K, V> {
    private K chave;
    private V valor;

    public Par(K chave, V valor) {
        this.chave = chave;
        this.valor = valor;
    }

    @Override
    public String toString() {
        return chave + " - " + valor;
    }

    // Getters e setters
}
```

Aqui, Par<K, V> é uma classe genérica que armazena um par de chaves e valores, com tipos K e V.

```
Par<String, String> b = new Par<>("10", "Marcia");
System.out.println(b);
```

É possível aplicar restrições aos tipos genéricos usando a palavra-chave extends para especificar que o tipo genérico deve ser uma subclasse de um tipo específico, ou super para especificar que o tipo genérico deve ser uma superclasse de um tipo específico. Além disso, você pode usar wildcards (?) para tornar o código mais flexível. Por exemplo:

```
public static <T extends Number> double soma(List<T> lista) {
    double total = 0.0;
    for (T elemento : lista) {
        total += elemento.doubleValue();
    }
    return total;
}
```

Neste exemplo, a função soma aceita uma lista de números (subclasses de Number) como parâmetro.

```
        // public final class Integer extends Number
        // public final class Double extends Number
        Double soma = soma(List.of(1, 2, 3));
        System.out.println(soma);

        // Não compila
        Double soma2 = soma(List.of("1", "2", "3"));
```

Já usamos generics algumas vezes:

```
List<String> lista = new ArrayList<>();
lista.add("Marcia");
lista.add("João");
```

2. Lambda

Você pode usar expressões lambda em muitos contextos em Java, como em operações de coleção, threads, ou sempre que precisar de uma função que pode ser passada como argumento. Um exemplo comum é o uso de expressões lambda com métodos como forEach de coleções ou Runnable em threads.

Aqui está um exemplo de uso de expressões lambda com forEach:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
numeros.forEach(numero -> System.out.println(numero));
```

Neste exemplo, a expressão lambda numero -> System.out.println(numero) é usada para imprimir cada número na lista.

As expressões lambda são uma maneira poderosa de tornar o código mais conciso e legível, especialmente quando você precisa passar funções como argumentos ou implementar interfaces funcionais de maneira rápida. Elas foram introduzidas no Java 8 para melhorar o suporte à programação funcional na linguagem.

Em Java, uma expressão lambda é uma **função anônima** que pode ser usada para implementar interfaces funcionais de maneira concisa e elegante. Uma interface funcional é uma interface que contém apenas um método abstrato. As expressões lambda permitem que você forneça uma implementação desse método de forma concisa, sem a necessidade de criar uma classe anônima ou implementar toda a interface manualmente.

A sintaxe básica de uma expressão lambda em Java é a seguinte:

(**parametros**) -> **expressao**

Expressões lambda são **frequentemente usadas para implementar interfaces funcionais**. Uma interface funcional é uma interface que tem exatamente **um método abstrato**. O Java fornece anotações especiais como @FunctionalInterface para indicar que uma interface é funcional e, portanto, pode ser usada com expressões lambda.

Exemplo de uma interface funcional:

```
@FunctionalInterface
interface Operacao {
    int calcular(int a, int b);
}
```

Você pode então criar uma expressão lambda para implementar esse único método:

```
Operacao soma = (a, b) -> a + b;
System.out.println(soma.calcular(1,2));
```

As expressões lambda são uma adição significativa ao Java, tornando a programação funcional mais acessível e permitindo escrever código mais limpo e expressivo. Elas são amplamente utilizadas em Java 8 e versões posteriores e são uma parte importante da modernização da linguagem Java.

3. Stream

Em Java, Stream é uma abstração que representa uma **sequência de elementos** que podem ser processados de uma maneira funcional e declarativa. Streams são uma parte fundamental da programação funcional introduzida no Java 8, e eles são frequentemente usados para operações de processamento de coleções de dados de forma mais concisa e eficiente.

Aqui estão algumas características e conceitos importantes relacionados a Streams em Java:

Origens de Streams: Você pode criar um Stream a partir de várias fontes de dados, como coleções (por exemplo, List, Set, Map) ou arrays. Aqui está um exemplo de criação de um Stream a partir de uma lista:

```
List<String> nomes = Arrays.asList("Alice", "Bob", "Carol", "David");
Stream<String> stream = nomes.stream();
```

Operações Intermediárias: Operações intermediárias são aquelas que podem ser encadeadas para modificar o Stream original, mas não executam nenhuma operação de terminal até que uma operação de terminal seja chamada. Alguns exemplos de operações intermediárias incluem filter, map, distinct, sorted e limit. Essas operações retornam outro Stream.

Exemplo de filtro de nomes que começam com a letra "A":

```
Stream<String> nomesComA = nomes.stream()
   .filter(nome -> nome.startsWith("A"));
```

Operações de Terminal: As operações de terminal são aquelas que produzem um resultado final ou acionam uma ação no Stream. Exemplos de operações de terminal incluem forEach, collect, reduce, count e anyMatch. Após a execução de uma operação de terminal, o Stream não pode ser reutilizado.

Exemplo de uso do forEach para imprimir cada elemento do Stream:

```
nomesComA.forEach(System.out::println);
```

Encadeamento de Operações: Você pode encadear várias operações intermediárias em um único Stream para criar um pipeline de processamento de dados. Isso permite que você escreva código mais legível e conciso.

Exemplo de encadeamento de operações para filtrar e mapear:

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

     List<Integer> quadradosPares = numeros.stream()
          .filter(n -> n % 2 == 0)
          .map(n -> n * n)
          .collect(Collectors.toList());

System.out.println(quadradosPares); // [4, 16]
```

Imutabilidade: Streams são imutáveis, o que significa que as operações em um Stream não alteram o Stream original, mas criam um novo Stream com as transformações aplicadas.

Collector: A interface Collector fornece uma variedade de métodos estáticos para realizar operações de terminal e coletar os elementos do Stream em várias formas, como Listas, Conjuntos, Mapas ou outros tipos de coleções personalizadas. É comumente usado em conjunto com a operação de terminal collect para coletar os elementos do Stream em uma coleção.

Exemplo de uso do collect para criar uma lista de nomes que começam com "A":

```
List<String> nomes = Arrays.asList("Alice", "Bob", "Carol", "David");

    List<String> nomesComA = nomes.stream()
        .filter(nome -> nome.startsWith("A"))
        .collect(Collectors.toList());

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

FlatMap: A operação flatMap é usada para lidar com Streams aninhados ou Streams de Streams, desenrolando-os em um único Stream. É útil quando você precisa aplicar uma transformação em cada elemento e, em seguida, achatá-lo em um único Stream.

Exemplo de uso de flatMap para lidar com Streams aninhados:

```
List<List<Integer>> numeros = Arrays.asList(
        Arrays.asList(1, 2, 3),
        Arrays.asList(4, 5, 6),
        Arrays.asList(7, 8, 9)
    );

    List<Integer> todosNumeros = numeros.stream()
        .flatMap(List::stream)
        .collect(Collectors.toList());


    // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
    System.out.println(numeros);

    // [1, 2, 3, 4, 5, 6, 7, 8, 9]
    System.out.println(todosNumeros);
```

Redução (Reduce): A operação de redução é usada para combinar os elementos de um Stream em um único valor. Isso é frequentemente usado para calcular somas, produtos, valores máximos, mínimos e outras operações de agregação.

Exemplo de uso do reduce para calcular a soma de números inteiros:

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

    int soma = numeros.stream()
        .reduce(0, (a, b) -> a + b);

    System.out.println(soma); // 15
```

Neste caso, o valor inicial é 0, e a função lambda (a, b) -> a + b é usada para combinar os elementos.

Os Streams em Java são uma maneira poderosa e expressiva de lidar com coleções de dados e realizar operações de processamento de maneira eficiente. Eles promovem um código mais legível, funcional e modular, facilitando a manipulação e transformação de dados em seus programas Java. É importante entender os conceitos básicos dos Streams para aproveitar ao máximo essa funcionalidade.

Mais exemplos com streams:

https://gist.github.com/marciafc/ec986b7285580933171c91ed69c16760

Usando Streams no Java

O básico de Java Streams com exemplos

Java 8 Streams: Pare de usar 'for' e simplifique seu código!

4. Optional

Optional é uma classe introduzida no Java 8 para **lidar com valores que podem ser nulos**, proporcionando uma forma mais segura e expressiva de trabalhar com esses valores. Ela é parte da API de Streams e foi projetada para **evitar exceções do tipo NullPointerException** ao acessar valores potencialmente nulos.

Aqui estão os principais conceitos e características do Optional em Java:

Evita NPEs (NullPointerExceptions): O Optional ajuda a evitar o erro comum de tentar acessar um valor nulo e, assim, lançar uma exceção NullPointerException. Ele força o programador a lidar explicitamente com a possibilidade de o valor estar ausente.

Contêiner para Valor Possivelmente Nulo: O Optional é essencialmente um contêiner que pode conter um valor não nulo ou indicar que o valor está ausente (nulo). Isso torna explícita a intenção de que um valor pode estar ausente e requer que você tome providências para lidar com ambos os casos.

Criação de Optionals: Você pode criar um Optional de várias maneiras, incluindo o uso de métodos estáticos da classe Optional, como of, ofNullable e empty. Por exemplo:

```
Optional<String> optionalNome = Optional.of("Alice"); // Contém um valor não nulo

Optional<String> optionalVazio = Optional.empty();    // Indica ausência de valor

String possivelNulo = null;
Optional<String> optionalPossivelNulo = Optional.ofNullable(possivelNulo); // Contém um valor que pode ser nulo

System.out.println(optionalNome);
System.out.println(optionalVazio);
System.out.println(optionalPossivelNulo);
```

Métodos para Acessar Valores: Para acessar o valor contido em um Optional, você pode usar métodos como get() (tenha cuidado, pois isso pode lançar uma exceção se o valor estiver ausente), orElse(), orElseGet() e orElseThrow(). Exemplos:

```
Optional<String> optionalNome = Optional.empty();
//Optional<String> optionalNome = Optional.of("A");

// Lança uma exceção se o valor estiver ausente
//System.out.println(optionalNome.get());
```

```
// Obtém o valor ou um padrão se estiver ausente
String nome = optionalNome.orElse("Nome Padrão");
System.out.println("nome " + nome);

// Obtém o valor ou chama uma função para fornecer um padrão
String nome2 = optionalNome.orElseGet(() -> obterNome());
System.out.println("nome2 " + nome2);

// Lança uma exceção personalizada se o valor estiver ausente
String nome3 = optionalNome.orElseThrow(() -> new RuntimeException("Valor ausente"));
System.out.println("nome3 " + nome3);

static String obterNome(){
        return null;
    }
```

Métodos para Lidar com Valores Presentes e Ausentes: O Optional fornece métodos como ifPresent(), ifPresentOrElse(), filter(), map(), flatMap(), entre outros, que permitem que você execute ações com base na presença ou ausência de um valor.

```
Optional<String> optionalNome = Optional.of("Alice");
optionalNome = Optional.empty();

optionalNome.ifPresent(valor -> System.out.println("Valor presente: " + valor));

optionalNome.ifPresentOrElse(
            valor -> System.out.println("Valor presente: " + valor),
            () -> System.out.println("Valor ausente")
);

if(optionalNome.isPresent()) {
    System.out.println(optionalNome.get());
} else {
    System.out.println("Não tem valor");
 }
```

É Imutável: Um Optional é imutável, o que significa que, uma vez criado, seu valor não pode ser alterado. Qualquer operação que modifica um Optional retorna um novo Optional com o valor modificado ou semelhante.

Usos Comuns: Optional é frequentemente usado em APIs para indicar que um método pode retornar um valor que pode estar ausente. Também é usado como um meio de expressar a intenção de que um valor pode ser nulo, sem causar exceções indesejadas.

Considerações: Embora Optional seja útil para evitar NPEs e expressar intenções, deve ser usado com parcimônia. Não é uma solução universal para todos os casos de valores possivelmente nulos. Deve ser usado principalmente em situações onde a ausência de um valor é um estado legítimo que deve ser tratado de maneira diferente de um valor nulo.

O Optional é uma ferramenta poderosa para lidar com valores possivelmente nulos em Java, tornando o código mais seguro e expressivo. No entanto, é importante usá-lo com sabedoria e em situações apropriadas para evitar o excesso de complexidade desnecessária em seu código.

5. Exercícios

1) Dada uma lista de números inteiros, multiplique cada número por 2 e colete os resultados em uma nova lista com uso de Stream.

2) Dada uma lista de números, encontre o maior número usando Streams.

3) Dada uma lista de objetos Aluno, filtre os alunos com notas maiores ou iguais a 7 e liste seus nomes.

```
List<Aluno> alunos = Arrays.asList(
        new Aluno("Alice", 8),
        new Aluno("Bob", 6),
        new Aluno("Carol", 9),
        new Aluno("David", 7),
        new Aluno("Eva", 8)
    );
```

4) Dada uma lista de produtos, agrupe os produtos por categoria com uso de Streams.