Expressões Lambda e Referências de Métodos

1. Expressão Lambda:

Uma expressão lambda é uma maneira concisa de implementar interfaces funcionais, como `Supplier<T>`, diretamente no código. Ela pode ser usada para definir o comportamento do método `get()`.

Exemplos com Lambda:

```
// Usando uma expressão lambda para retornar um valor fixo.
Supplier<String> stringSupplier = () -> "Hello, World!";
System.out.println(stringSupplier.get());
```

Aqui, a expressão lambda `() -> "Hello, World!"` está implementando o método `get()` da interface `Supplier<String>`. Ela diz que, quando `get()` for chamado, ele deve retornar a string `"Hello, World!"`.

```
Supplier<Double> randomSupplier = () -> Math.random();
// Saída: (um número aleatório)
System.out.println(randomSupplier.get());
```

Neste exemplo, a expressão lambda `() -> Math.random()` está criando um `Supplier<Double>` que retorna um número aleatório quando `get()` é chamado.

2. Referências de Métodos:

Referência de métodos é outra forma de fornecer uma implementação para métodos funcionais, mas, em vez de usar uma lambda, você se refere diretamente a um método existente.

Exemplos com Referência de Método:

```
// Método que retorna uma string.
public static String provideString() {
    return "Hello, Method Reference!";
}

// Usando referência de método para o Supplier.

Supplier<String> stringSupplier = MyClass::provideString;
System.out.println(stringSupplier.get());
```

Neste exemplo, `MyClass::provideString` é uma referência de método. Ele refere-se ao método `provideString()` existente, que será invocado quando `get()` for chamado no `Supplier<String>`.

```
Supplier<Double> randomSupplier = Math::random;
// Saída: (um número aleatório)
System.out.println(randomSupplier.get());
```

Aqui, `Math::random` é uma referência ao método `random()` da classe `Math`, e faz exatamente a mesma coisa que o exemplo de lambda, mas de forma mais direta.

3. Comparando Expressão Lambda e Referência de Métodos:

- **Expressão Lambda:**
  - É uma forma concisa de definir o comportamento diretamente no ponto de uso.
  - É útil quando o comportamento é simples ou único e não justifica a criação de um método separado.
  - Permite escrever código mais compacto e legível.
  - Útil em contextos que exigem implementações de interfaces funcionais, como Runnable, Comparator e outras.

- **Referência de Métodos:**
  - É mais apropriada quando você já tem um método existente que faz exatamente o que você precisa.
  - Promove a reutilização de código, evitando a duplicação e tornando o código mais legível, especialmente se o método já está bem nomeado.
  - Deixa o código mais expressivo e facilita a compreensão, pois é mais explícito sobre qual método está sendo utilizado.
  - Geralmente resulta em código mais curto e conciso do que usar uma expressão lambda.

As referências por método em Java podem aceitar parâmetros, mas há algumas nuances a serem consideradas, dependendo do tipo de método que você está referenciando. Vamos explorar algumas dessas nuances:

- **Métodos de Instância vs. Métodos Estáticos**:
  - Ao referenciar um método de instância, o primeiro parâmetro da interface funcional deve ser uma instância do tipo que contém o método.
  - Ao referenciar um método estático, não há necessidade de uma instância, e o primeiro parâmetro da interface funcional não precisa ser do tipo que contém o método estático.

- **Correspondência de Parâmetros**:
  - Os parâmetros da referência do método devem corresponder exatamente aos parâmetros esperados pela interface funcional.
  - Se a interface funcional tiver mais parâmetros do que o método referenciado, você não poderá usá-la.
  - Se o método referenciado tiver mais parâmetros do que a interface funcional, você precisará usar uma expressão lambda em vez de uma referência de método.

- **Construtores como Referências de Método**:
  - Você também pode referenciar construtores como se fossem métodos, usando a sintaxe ClassName::new.
  - Nesse caso, os parâmetros da interface funcional devem corresponder exatamente aos parâmetros do construtor.

Referências por métodos em Java podem aceitar parâmetros, mas há algumas nuances a serem consideradas dependendo do tipo de método que você está referenciando. A seguir estão alguns exemplos que vão tornar mais claras essas nuances.

ALGUMAS REGRAS COM EXEMPLOS

I. Quando você referencia um método estático que aceita parâmetros, o tipo da interface funcional que você está implementando deve ser compatível com a assinatura do método. Observe também que não preciso de uma instância da classe para usar a interface funcional, pois o método estático pertence à classe, e não a um objeto específico.

```java
import java.util.function.Function;
public class MyClass {
    // Método estático que aceita um parâmetro
    public static Integer parse(String s) {
        return Integer.parseInt(s);
    }
    public static void main(String[] args) {
        // Referência ao método estático 'parse' que aceita um
        // parâmetro String
        Function<String, Integer> stringToInteger = MyClass::parse;
        // Usando a função
        Integer result = stringToInteger.apply("123");
        System.out.println(result); // Saída: 123
    }
}
```

→ `Function<String, Integer>` é uma interface funcional que aceita um parâmetro (`String`) e retorna um valor (`Integer`).

→ `MyClass::parse` é uma referência ao método estático `parse(String)` que aceita uma `String` como parâmetro e retorna um `Integer`.

II. Se você tem um método de instância que aceita parâmetros, você pode referenciar esse método para um objeto específico.

```
import java.util.function.BiFunction;
public class MinhaClasse2 {
    // Método de instância que aceita dois parâmetros
    public Integer add(Integer a, Integer b) {
        return a + b;
    }

    public static void main(String[] args) {
        MinhaClasse2 obj = new MinhaClasse2();

        // Referência ao método de instância 'add' do objeto myClass2
        BiFunction<Integer, Integer, Integer> addFunction = obj::add;

        // Usando a função
        Integer result = addFunction.apply(10, 20);
        System.out.println(result); // Saída: 30
    }
}
```

→ `BiFunction<Integer, Integer, Integer>` é uma interface funcional que aceita dois parâmetros (`Integer, Integer`) e retorna um valor (`Integer`).

→ `obj::add` refere-se ao método de instância `add(Integer, Integer)` no objeto `obj`.

III. Quando você referencia um método de instância para objetos de um tipo específico, o primeiro parâmetro na interface funcional se torna o objeto no qual o método é chamado.

```
import java.util.function.BiFunction;
public class MinhaClasse3 {
    public Integer subtract(Integer a) {
        return  -1*a;
    }

    public static void main(String[] args) {
        // Referência ao método de instância 'subtract' para qualquer
        // objeto MyClass
        BiFunction<MinhaClasse3, Integer, Integer> subtractFunction =
        MinhaClasse3::subtract;

        // Usando a função
        MinhaClasse3 obj = new MinhaClasse3();
        Integer result = subtractFunction.apply(obj, 5);
        System.out.println(result);
    }
}
```

→ `BiFunction<MyClass3, Integer, Integer>` aceita dois parâmetros: o primeiro é um `MyClass3` e o segundo é o `Integer` passado ao método `subtract`.

Perceba que quando você refere um **método de instância de um tipo arbitrário** (ou seja, um método de instância de qualquer objeto desse tipo), **a interface funcional precisa incluir um parâmetro para o objeto da instância**,como no exemplo acima. Isso acontece porque o método precisa de uma instância do objeto para ser chamado.

Isso acontece porque os métodos de instância **precisam de um objeto de instância para operar**. Em Java, quando você chama um método de instância, você implicitamente passa o objeto no qual o método será chamado.

Com as referências de métodos, quando você está referenciando um método de instância de um tipo arbitrário, você precisa explicitamente fornecer o objeto ao qual o método pertence, daí a necessidade do primeiro parâmetro na interface funcional. Resumindo:

Método de Instância de Objeto Específico (Sem Parâmetro Extra):

```
Supplier<String> supplier = myObject::instanceMethod;
```

Aqui, `myObject` é conhecido, então não há necessidade de passar um objeto.

Método de Instância de Tipo Arbitrário (Com Parâmetro Extra):

```
Function<MyClass, String> function = MyClass::instanceMethod;
```

Aqui, `function.apply(myObject)` passa explicitamente o objeto `myObject` para o método.

IV. Referências de métodos podem ser usadas para referenciar construtores, que aceitam parâmetros como métodos normais.

```
import java.util.function.Function;

public class MinhaClasse4 {
    private final String name;

    // Construtor que aceita um parâmetro
    public MinhaClasse4(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "MyClass{" + "name='" + name + '\'' + '}';
    }

    public static void main(String[] args) {
        // Referência ao construtor de MyClass
        Function<String, MinhaClasse4> myClassConstructor =
        MinhaClasse4::new;

        // Usando a função
        MinhaClasse4 myObject = myClassConstructor.apply("Hello");
        System.out.println(myObject); // Saída: MyClass{name='Hello'}
    }
}
```

→ `Function<String, MyClass>` refere-se a um construtor de `MyClass` que aceita um `String` e retorna uma nova instância de `MyClass`, que é exatamente a assinatura da interface funcional `Function<T, R>`.