Vamos falar sobre a interface `Predicate`?

O que é a interface `Predicate`?

`Predicate` é uma interface funcional do Java 8 que **representa uma função que recebe um argumento e retorna um valor booleano**. É usada para **avaliar** condições sobre objetos.

Definição

A interface `Predicate` é definida assim:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Como Funciona?

A interface `Predicate` tem um **único método abstrato chamado** `test`, que recebe um argumento do tipo `T` e retorna um booleano. Essa assinatura **permite que você use expressões lambda para criar instâncias** de `Predicate`.

Exemplos Práticos

Vamos ver alguns exemplos para entender melhor como funciona.

Exemplo 1: Verificando se um número é par

Aqui, vamos usar um `Predicate` para verificar se um número é par:

```
Predicate<Integer> isEven = n -> n % 2 == 0;

System.out.println(isEven.test(4)); // true
System.out.println(isEven.test(7)); // false
```

Exemplo 2: Filtrando uma lista de strings

Agora, vamos usar `Predicate` **como parâmetro de *stream.filter()*** para filtrar uma lista de strings, mantendo apenas as que começam com a letra "A":

```
List<String> names = Arrays.asList("Alice", "Bob", "Anna", "John");
Predicate<String> startsWithA = s -> s.startsWith("A");

List<String> filteredNames = names.stream()
                                  .filter(startsWithA)
                                  .collect(Collectors.toList());

System.out.println(filteredNames); // [Alice, Anna]
```

Métodos Default

A interface `Predicate` também **fornece métodos *default* que facilitam a composição de predicates**. Vamos ver os principais:

and

Combina dois predicates usando o operador lógico "e":

```
Predicate<Integer> isPositive = n -> n > 0;
Predicate<Integer> isEvenAndPositive = isEven.and(isPositive);

System.out.println(isEvenAndPositive.test(4));  // true
System.out.println(isEvenAndPositive.test(-4)); // false
```

or

Combina dois predicates usando o operador lógico "ou":

```
Predicate<Integer> isOdd = n -> n % 2 != 0;
Predicate<Integer> isEvenOrOdd = isEven.or(isOdd);

System.out.println(isEvenOrOdd.test(4)); // true
System.out.println(isEvenOrOdd.test(5)); // true
```

negate

Inverte o resultado de um predicate:

```
Predicate<Integer> isOddNegate = isOdd.negate();

System.out.println(isOddNegate.test(4)); // true
System.out.println(isOddNegate.test(5)); // false
```

Outros exemplos do uso da interface funcional Predicate:

EXEMPLO 1

O código abaixo define uma série de predicados em Java para validar strings, especificamente nomes de usuários. Ele cria três predicados: `isLengthBetween(int min, int max)` que verifica se o comprimento da string está entre os valores mínimos e máximos especificados, `containsOnlyLetters()` que valida se a string contém apenas letras (maiúsculas ou minúsculas), e `startsWithCapital()` que verifica se a string começa com uma letra maiúscula.

Esses predicados são então compostos em um único predicado chamado `validUsername`, que é usado no método `main` para testar diferentes strings e determinar se cada uma delas é um nome de usuário válido com base nas condições definidas. A saída resultante indica se cada string testada é válida ou não.

```java
public class PredicateComposition {

    // Método que retorna um Predicate<String> para verificar se o comprimento
    // de uma string está entre um valor mínimo e máximo
    public static Predicate<String> isLengthBetween(int min, int max) {
        // O Predicate retornado verifica se o comprimento da string é maior ou
        //igual ao mínimo e menor ou igual ao máximo
        return s -> s.length() >= min && s.length() <= max;
    }

    // Método que retorna um Predicate<String> para verificar se uma
    // string contém apenas letras (maiúsculas ou minúsculas)
    public static Predicate<String> containsOnlyLetters() {
        // O Predicate retornado usa uma expressão regular para verificar se a string contém apenas letras
        return Pattern.compile("^[a-zA-Z]+$").asPredicate();
    }

    // Método que retorna um Predicate<String> para verificar se uma string começa com uma letra maiúscula
    public static Predicate<String> startsWithCapital() {
        // O Predicate retornado verifica se a string não é vazia e se o primeiro caractere é uma letra maiúscula
        return s -> !s.isEmpty() && Character.isUpperCase(s.charAt(0));
    }

    public static void main(String[] args) {
        // Combina os Predicates definidos acima para criar um Predicate<String> que valida um nome de usuário
        Predicate<String> validUsername = isLengthBetween(5, 20)
                .and(containsOnlyLetters())
                .and(startsWithCapital());

        // Testa o Predicate validUsername com diferentes strings e imprime o resultado
        System.out.println(validUsername.test("ValidUsername")); // true - A string atende a todos os critérios
        System.out.println(validUsername.test("invalid_username")); // false - A string contém caracteres não permitidos
        System.out.println(validUsername.test("short")); // false - A não começa com letra maiúscula
        System.out.println(validUsername.test("good")); // false - A string é muito curta

    }
}
```

```
Console  Problems  Debug Shell
<terminated> PredicateComposition [Java Application] C:\Program Files\eclipse-java-2022-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (
true
false
false
false
```

EXEMPLO 2

O exemplo a seguir é um pouco mais complexo, pois envolve recursividade. A recursividade em Java é uma técnica de programação poderosa onde uma função chama a si mesma para resolver um problema.

O problema envolve achar em uma estrutura do tipo árvore, onde os nós estão interlig ados quais os valores contidos em cada nó que satisfazem o método `test()` da classe `P redicate`

A classe `PredicateTree é` uma estrutura de árvore personalizada que usa `Predicate` para filtrar elementos dessa árvore. Vamos passar por cada parte e entender o que está rolando, especialmente a parte da recursividade, que pode ser a mais confusa.

```
 7 public class PredicateTree<T> {
 8
 9     private T value;
10     private List<PredicateTree<T>> children;
11     private Predicate<T> predicate;
12
13     public PredicateTree(T value, Predicate<T> predicate) {
14         this.value = value;
15         this.predicate = predicate;
16         this.children = new ArrayList<>();
17     }
18
19     public void addChild(PredicateTree<T> child) {
20         children.add(child);
21     }
22
23     public List<T> findMatches() {
24         return findMatches(predicate);
25     }
26
27     private List<T> findMatches(Predicate<T> accumulatedPredicate) {
28         List<T> matches = new ArrayList<>();
29         if (accumulatedPredicate.test(value)) {
30             matches.add(value);
31         }
32         for (PredicateTree<T> child : children) {
33             matches.addAll(child.findMatches(accumulatedPredicate.or(child.predicate)));
34         }
35         return matches;
36     }
37 }
```

Estrutura Básica

1. **Classe PredicateTree<T>:**
    - `T value`: Armazena o valor que está sendo avaliado no nó da árvore.
    - `List<PredicateTree<T>>` children: Uma lista de nós filhos, permitindo que cada nó tenha múltiplos "filhos" na árvore.
    - `Predicate<T> predicate`: Um predicado associado a esse nó específico, usado para testar o valor armazenado.
2. **Construtor:**
    - Quando você cria um novo objeto `PredicateTree<T>`, você passa um valor e um predicado. Isso inicializa o nó com essas informações e uma lista vazia de filhos.

3. **Método addChild(PredicateTree<T> child)**:
    - Permite adicionar um nó filho ao nó atual.

Recursividade no Método findMatches

Aqui é onde a coisa começa a ficar mais complexa.

- **Método findMatches():**
    - Esse método é o ponto de entrada para encontrar os valores que correspondem a um conjunto de predicados. Ele chama um método privado sobrecarregado `(findMatches(Predicate<T> accumulatedPredicate))` passando o predicado do próprio nó.
- **Método privado findMatches(Predicate<T> accumulatedPredicate):**
    - **Passo 1:** Cria uma lista vazia chamada matches.
    - **Passo 2:** Testa o valor do nó atual com o predicado acumulado (`accumulatedPredicate`), que inicialmente é o predicado do nó raiz. Se o valor do nó satisfaz o predicado, ele é adicionado à lista matches.
    - **Passo 3:** Para cada nó filho, o método é chamado recursivamente, mas com um novo predicado acumulado. Esse novo predicado é uma combinação do predicado atual (`accumulatedPredicate`) **ou** do predicado do filho (`child.predicate`).

Como a Recursividade Funciona

Imagine que cada nó na árvore é como uma pessoa em uma rede social, e o predicado é uma condição, tipo "Gosta de música clássica". O que esse código faz é percorrer cada pessoa (nó) e ver se ela ou qualquer uma de suas conexões (filhos) gosta de música clássica ou se está conectada a alguém que gosta.

- A recursividade ocorre porque, para cada nó, ele chama o mesmo método em cada um de seus filhos. Isso cria uma espécie de "corrente" de chamadas até que chegue a um nó sem filhos (caso base da recursão).
- **accumulatedPredicate.or(child.predicate)**: Isso é importante porque vai acumulando as condições ao longo do caminho. Se qualquer um dos predicados (o atual ou o do filho) for verdadeiro, o valor será adicionado à lista.

Exemplo Prático

```java
package com.exemplo.predicate;

import java.util.List;

public class Exemplo2 {

    public static void main(String[] args) {
            Predicate<Integer> isEven = n -> n % 2 == 0;
            Predicate<Integer> isGreaterThan10 = n -> n > 10;

            PredicateTree<Integer> root = new PredicateTree<>(1, isEven.or(isGreaterThan10));
            PredicateTree<Integer> child1 = new PredicateTree<>(4, isEven.and(isGreaterThan10));
            PredicateTree<Integer> child2 = new PredicateTree<>(7, isEven.or(isGreaterThan10));
            PredicateTree<Integer> child3 = new PredicateTree<>(12, isEven.and(isGreaterThan10));

            root.addChild(child1);
            root.addChild(child2);
            child2.addChild(child3);

            List<Integer> matches = root.findMatches();
            System.out.println(matches); // [4, 12]
    }

}
```

```
Console  Problems  Debug Shell
<terminated> Exemplo2 [Java Application] C:\Program Files\eclipse-java-2022-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe  (30 de ago. de 2024 12:25:12 – 12:25:14) [pid: 17128]
[4, 12]
```

Inicializando os Nós da Árvore

Uma árvore de predicados foi criada com a seguinte estrutura:

- **Raiz (root)**: Valor = 1, Predicado = n -> n % 2 == 0 || n > 10 (Ou seja, "é par **ou** maior que 10")
- **Filho 1 (child1)**: Valor = 4, Predicado = n -> n % 2 == 0 && n > 10 (Ou seja, "é par **e** maior que 10")
- **Filho 2 (child2)**: Valor = 7, Predicado = n -> n % 2 == 0 || n > 10
- **Filho 3 (child3)**: Valor = 12, Predicado = n -> n % 2 == 0 && n > 10

E a árvore foi construída assim:

- root tem dois filhos: child1 e child2.
- child2 tem um filho: child3.

Chamando findMatches()

- Agora, você chamou `findMatches()` na raiz (root). Vamos ver como isso se desenrola.

1. **No Raiz (root)**
    - **Valor**: 1
    - **Predicado**: `isEven.or(isGreaterThan10)` (Ou seja, "é par **ou** maior que 10")
    - O predicado **não** é satisfeito para o valor 1 (não é par e não é maior que 10), então 1 **não** é adicionado à lista matches.

- A função agora vai descer para os filhos.

2. **No Filho 1 (child1)**
   - **Valor**: 4
   - **Predicado acumulado**: `isEven.or(isGreaterThan10).or(isEven.and(isGreaterThan10))` (Ou seja, "é par **ou** maior que 10 **ou** é par **e** maior que 10")
   - **Explicando o predicado acumulado**:
     - O predicado acumulado inclui a condição da raiz "é par **ou** maior que 10" combinada com a condição do child1 "é par **e** maior que 10".
   - Para o valor 4, o predicado acumulado é satisfeito porque 4 é par. Então, 4 é adicionado à lista matches.
3. **No Filho 2 (child2)**
   - **Valor**: 7
   - **Predicado acumulado**: `isEven.or(isGreaterThan10).or(isEven.or(isGreaterThan10))` (Basicamente, "é par **ou** maior que 10")
   - Para o valor 7, o predicado acumulado **não** é satisfeito (não é par e não é maior que 10), então 7 **não** é adicionado à lista matches.
   - Agora, a função vai descer para o filho do child2, que é child3.
4. **No Filho 3 (child3)**
   - **Valor**: 12
   - **Predicado acumulado**: `isEven.or(isGreaterThan10).or(isEven.and(isGreaterThan10))` (Ou seja, "é par **ou** maior que 10 **ou** é par **e** maior que 10")
   - Para o valor 12, o predicado acumulado é satisfeito (12 é par **e** maior que 10), então 12 é adicionado à lista matches.

Resultado Final

Depois que o método recursivo percorre toda a árvore, a lista matches contém [4, 12].

- **4**: Foi adicionado porque é par, satisfazendo o predicado acumulado quando foi verificado em child1.
- **12**: Foi adicionado porque é par **e** maior que 10, satisfazendo o predicado acumulado quando foi verificado em child3.

Como a Recursividade Está Funcionando

Vamos resumir como a recursividade interage com a árvore:

- O método findMatches começa na raiz e tenta encontrar correspondências no nó atual.
- Se o nó atual não satisfaz o predicado, ele passa para os filhos.
- Quando desce para os filhos, ele acumula o predicado do nó pai com o predicado do filho.
- Esse processo continua até que todos os nós tenham sido verificados.

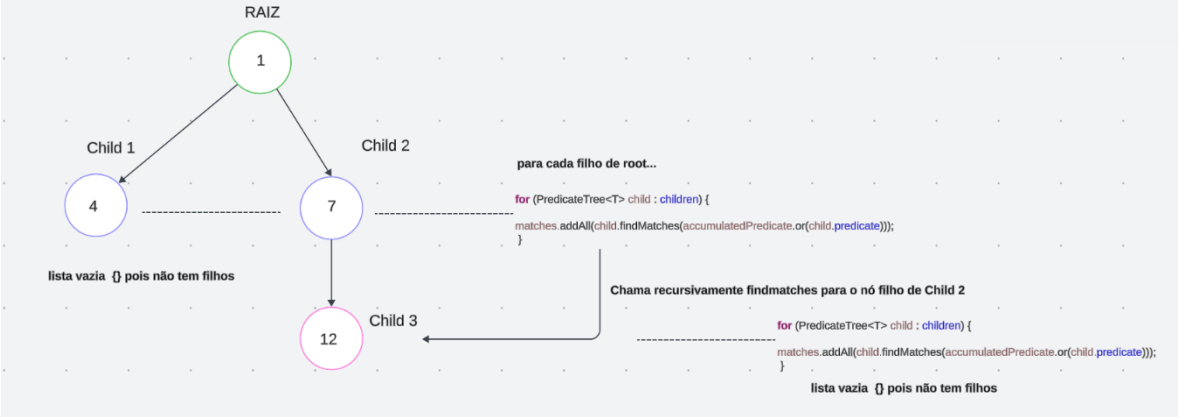
Cada chamada recursiva ao método findMatches vai acumulando predicados, combinando as condições de cada nó, e essa combinação é testada para cada valor na árvore.

Visualizando a Recursividade

Podemos pensar nisso como uma árvore de decisões:

- **No Raiz (root)**: Decide se deve adicionar 1 à lista.
    - **No Filho 1 (child1)**: Desce e verifica 4 (adiciona 4).
    - **No Filho 2 (child2)**: Desce e verifica 7 (não adiciona 7).
        - **No Filho 3 (child3)**: Desce e verifica 12 (adiciona 12).

Visualizando isso num gráfico temos o seguinte:



Perceba que nNo método `findMatches(Predicate<T> accumulatedPredicate`), o laço for que percorre os filhos é essencial para a recursividade. Esse laço for é chamado em cada nível da árvore, ou seja, para cada nó que tem filhos.

Mas porque usar `addAll()` e não `add()`?

- **addAll()**: É utilizado para adicionar todos os elementos de uma lista à lista matches atual. Então, quando você chama child.findMatches(...), ele retorna uma lista de todos os valores que satisfazem os predicados nos sub-nós daquele filho. addAll() pega todos esses valores e os adiciona à lista matches do nó atual.
- **add()**: Esse método só adicionaria um único elemento à lista. Mas como cada chamada recursiva a findMatches retorna uma **lista inteira** (que pode ter nenhum, um ou vários elementos), precisamos de addAll() para juntar todas essas listas em uma só.