Java の関数型インターフェース

java.util.function パッケージは、Java 8 で導入された関数型インターフェースのコレクションを提供し、関数型プログラミングをサポートするために設計されています。これらのインターフェースはそれぞれ 1 つの抽象メソッドを持ち、ラムダ式やメソッド参照と互換性があります。このレスポンスでは、このパッケージの最も一般的な関数型インターフェース—Function<T, R>、Predicate<T>、Consumer<T>、Supplier<T>—の使用方法について説明し、実用的な例を示します。

関数型インターフェースとは？

関数型インターフェースは、正確に 1 つの抽象メソッドを持つインターフェースです。java.util.function パッケージは、一般的なタスク用の事前定義された関数型インターフェースを提供するため、独自のインターフェースを作成する必要はありません。これらのインターフェースは、ラムダ式、メソッド参照、およびストリーム API とともに使用され、簡潔で表現力のあるコードを書くために広く使用されています。

以下に、主要なインターフェースの使用方法を示します。

---

1. Function<T, R>: 入力を出力に変換

Function<T, R> インターフェースは、入力の型 T を受け取り、出力の型 R を生成する関数を表します。その抽象メソッドは apply です。

例: 文字列の長さを取得

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

public class Main {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();
        System.out.println(stringLength.apply("Hello")); // 出力: 5
    }
}
```

- **説明**: ラムダ式 s -> s.length() は、String (T) を受け取り、Integer (R) を返す Function を定義します。apply メソッドはこのロジックを実行します。

---

2. `Predicate<T>`: 条件をテスト

`Predicate<T>` インターフェースは、入力の型 `T` を受け取り、ブール値を返す関数を表します。その抽象メソッドは `test` です。

例: 数字が偶数かどうかを確認

```
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // 出力: true
        System.out.println(isEven.test(5)); // 出力: false
    }
}
```

- **説明**: ラムダ `n -> n % 2 == 0` は、入力が偶数の場合に `true` を返す `Predicate` を定義します。`test` メソッドはこの条件を評価します。

---

3. `Consumer<T>`: アクションを実行

`Consumer<T>` インターフェースは、入力の型 `T` を受け取り、結果を返さない操作を表します。その抽象メソッドは `accept` です。

例: 文字列を印刷

```
import java.util.function.Consumer;

public class Main {
    public static void main(String[] args) {
        Consumer<String> printer = s -> System.out.println(s);
        printer.accept("Hello, World!"); // 出力: Hello, World!
    }
}
```

- **説明**: ラムダ `s -> System.out.println(s)` は、入力を印刷する `Consumer` を定義します。`accept` メソッドはこのアクションを実行します。

---

4. `Supplier<T>`: 結果を生成

`Supplier<T>` インターフェースは、入力を受け取らず、型 `T` の値を返す結果の供給元を表します。その抽象メソッドは `get` です。

例: 乱数を生成

```
import java.util.function.Supplier;
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        Supplier<Integer> randomInt = () -> new Random().nextInt(100);
        System.out.println(randomInt.get()); // 0 から 99 の間の乱数を出力
    }
}
```

- **説明**: ラムダ `() -> new Random().nextInt(100)` は、乱数を生成する `Supplier` を定義します。`get` メソッドはその値を取得します。

ストリームと関数型インターフェースの使用

これらのインターフェースは、Java Stream API で特に輝きます。以下は、文字列のリストをフィルタリング、変換、印刷する例です。

例: 文字列のリストを処理

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("a", "bb", "ccc", "dddd");

        Predicate<String> longerThanTwo = s -> s.length() > 2;       // 長さが 2 より長い文字列をフィルタリン
        Function<String, String> toUpperCase = s -> s.toUpperCase(); // 大文字に変換
```

```
        Consumer<String> printer = s -> System.out.println(s);       // 各結果を印刷

        strings.stream()
               .filter(longerThanTwo)    // "ccc" と "dddd" を保持
               .map(toUpperCase)         // "CCC" と "DDDD" に変換
               .forEach(printer);        // 出力: CCC, DDDD (別々の行に)
    }
}
```

- **説明**:
  - `filter` は、長さが 2 より長い文字列を保持するために `Predicate` を使用します。
  - `map` は、文字列を大文字に変換するために `Function` を使用します。
  - `forEach` は、各結果を印刷するために `Consumer` を使用します。

メソッド参照の使用　メソッド参照を使えば、さらに簡潔にできます:

```
strings.stream()
       .filter(s -> s.length() > 2)
       .map(String::toUpperCase)      // Function 用のメソッド参照
       .forEach(System.out::println); // Consumer 用のメソッド参照
```

関数型インターフェースの組み合わせ

いくつかのインターフェースは、より複雑な操作のために組み合わせを許可します: - **関数の組み合わせ**: `andThen` または `compose` を使用 `java Function<String, Integer> toLength = s -> s.length(); Function<Integer, String> toString = i -> "Length is " + i; Function<String, String> combined = toLength.andThen(toString); System.out.println(combined.apply("Hello")); // 出力: Length is 5` - **述語の組み合わせ**: `and`、`or`、`negate` を使用 `java Predicate<String> isLong = s -> s.length() > 5; Predicate<String> startsWithA = s -> s.startsWith("A"); Predicate<String> isLongAndStartsWithA = isLong.and(startsWithA); System.out.println(isLongAndStartsWithA.test("Avocado")); // 出力: true`

要約

これらのインターフェースの使用方法とタイミング: - `Function<T, R>`: 入力を出力に変換します (例: `apply`)。 - `Predicate<T>`: 条件をテストします (例: `test`)。 - `Consumer<T>`: 入力に対してアクションを実行します (例:

accept)。- Supplier<T>: 入力なしで値を生成します (例: get)。- **ストリームとの使用**: 強力なデータ処理のために組み合わせます。- **組み合わせ**: 組み合わせまたは複雑なロジックのために連鎖します。

これらのインターフェースは、ラムダ式 (例: s -> s.length()) またはメソッド参照 (例: String::toUpperCase) を使用して実装できます。これらは、特にストリーム API とともに使用することで、Java のコードをより簡潔で読みやすく、再利用可能にするための関数型プログラミングスタイルを可能にします。