OPEN STREAM

214-08-1.Java8::関数型プログラミング

関数型インタフェース,ラムダ式

Make IT your Success

目次

関数型プログラミングとは
関数型インタフェース
    関数型インタフェースとは
    関数型インタフェースの作成
    汎用の関数型インタフェースの基本の4つ
ラムダ式
    ラムダ式とは
    ラムダ式の省略
    ラムダ式の利用例

この講義について

目的：

- オブジェクト指向プログラミングと関数型プログラミングの違いを学び、関数型インタフェースの基本の4つについてそれぞれの役割を学ぶ。
- ラムダ式の利用方法、処理について学ぶ。

ゴール：

- 関数型インターフェースを継承し、関数型プログラミングを記述することが出来る。
- ラムダ式の利点について理解し、記述することが出来る。

関数型プログラミングとは

関数

入力に対して、ただ一つの出力が定まる式のことをいいます。

```
y=f(x)
```

プログラミング言語の関数は、状態(メンバ変数など)によって同じ引数でも戻り値が異なったり、関数の実行によってシステムの状態が変化したり、戻り値のない関数があったり、と純粋な関数とは異なる事が多いです。

参照透過性

関数そのものと関数の結果(値)を置き換えても、プログラムの振る舞いが変わらない事を、関数は参照透過であるといいます。

つまり、同じ x の場合、必ず同じ y が返ります。 → これを**純粋関数**といいます

※ クラスメンバのメソッドの場合、メンバ変数の値によって y が異なる場合がある

副作用

機能(引数や戻り値)以外に、作用がある(処理が行われる)事を**副作用**があるといいます。

例えば、キュー構造のインスタンスがあるとします。 dequeueすると値を一つ取り出しますが、メモリ内部の状態が変化します。

この副作用によって、 dequeueは参照透過性ではありません。

関数型プログラミングとは

関数型プログラミング

オブジェクト指向プログラミングによって、規模の大きい開発が容易になりました。

状態(メンバ変数)と機能(メソッド)を合わせてオブジェクトとして扱う事が良い反面、カプセルの内部の副作用が複雑性を上げてしまってもいます。

関数型プログラミングとは、純粋な関数を書いて、隠れた機能(副作用)をできるだけ取り除き、コードの入出力の関係だけでプログラムを書きましょうという考え方です。

※「カプセル化」と「副作用が無い」は無関係ですので注意しましょう

関数を第一級オブジェクト(引数や戻り値として使えるオブジェクト)として扱う事で、関数を組み合わせてプログラミングを行います。

メリット

宣言的な書き方の為、コードが簡潔に書ける

副作用が無い場合、証明がしやすい　→　テストのコストを下げられる

オブジェクト指向プログラミング、関数型プログラミング、どちらが優れているというものではありません。

必要に応じて使い分けていきましょう。

関数型インタフェース

関数型インタフェースとは

実装すべきメソッドは1つであるインタフェース。

→ 抽象メソッドが1つのみ

1つの抽象メソッドをラムダ式で実装します。

※ラムダ式はステートメント(文＝手続き　→処理)を書くための記法

実装されたオブジェクトは1つのメソッドを持ちます。

このことによって関数(メソッド)をオブジェクトとして扱う事ができます。

→引数や戻り値に使う事ができる

関数型インタフェースの注意点

- デフォルトメソッドやstaticメソッドは定義されてても良い
- Objectクラスのpublicメソッド(toStringやequalsなど)は無視される

→ 実装するメソッドが1つである事

※匿名クラスもラムダ式もメソッド参照も、関数をオブジェクトとして扱える仕組み

（匿名クラスはそれだけではない）

関数型インタフェースの作成

実装する抽象メソッドが1つである事が条件。

@FunctionalInterfaceを付与すると関数型インタフェースかチェックします。

使用例

```
public class LambdaTest {
    public static void main(String[] args) {
        funcTest fu = (String name) -> {    //抽象メソッドの実装を式として扱う
            return "hi! " + name + “!”;
        };
        System.out.println( fu.method( "duke" ) );
    }
}
@FunctionalInterface
interface funcTest{
    public String method(String name);    //抽象メソッド
}
```

汎用の関数型インタフェースの基本の4つ

Java8では、よく使われる汎用的な関数型インタフェースが提供されています。
以下に基本的な4つをピックアップします。

インタフェース名	実装するメソッド	内容
Function<T,R>	R apply(T t)	引数と返り値のあるメソッドを定義するインタフェース
Consumer<T>	void accept(T t)	引数はあるが返り値がvoidのメソッドを定義するインタフェース
Predicate<T>	Boolean test(T t)	引数と、booleanの返り値のメソッドを定義するインタフェース
Supplier<T>	T get()	引数がなく、返り値のみのメソッドを定義するインタフェース

汎用の関数型インタフェースの基本の4つ

実装の例（ Function型）

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

public class LambdaTest {
	public static void main(String[] args) {
		// Function型　引数と返り値のあるメソッドを定義
		Function<String, Integer> function = str -> str.length();
		// 実行結果
		 function. apply( "Function型 applyの実行結果 : " + str );
	}
}
```

汎用の関数型インタフェースの基本の4つ

実装の例（ Consumer型）

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

public class LambdaTest {
	public static void main(String[] args) {
		// Consumer型　引数はあるが返り値がvoidのメソッドを定義
		Consumer<String> consumer = str -> System.out.println(str);
		// 実行結果
		consumer.accept( " Consumer型 acceptの実行結果：" + str );
	}
}
```

汎用の関数型インタフェースの基本の4つ

実装の例（ Predicate型）

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

public class LambdaTest {
	public static void main(String[] args) {
		// Predicate型　引数と、booleanの返り値のメソッドを定義
		Predicate<String> predicate = str -> str.isEmpty();
		// 実行結果
		System.out.println( "Predicate型 testの実行結果：" + predicate.test(str));
	}
}
```

汎用の関数型インタフェースの基本の4つ

実装の例（ Supplier型）

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

public class LambdaTest {
	public static void main(String[] args) {
		// Supplier型　引数がなく、返り値のみのメソッドを定義
		Supplier<String> supplier = () -> "aaa";
		// 実行結果
		System.out.println( "supplier型 getの実行結果：" + supplier.get());
	}
}
```

ラムダ式とは

ステートメント(文)を記述する為の記法です。
記述を省略する事ができるので、簡潔にステートメントを書くことができます。
関数型インタフェースの抽象メソッドを実装する為に用いられます。

ラムダ式の使える場所

- 型を関数インタフェースとする変数の初期化や割り当てでの右辺。
- メソッド呼び出しの引数。
- キャスト演算子の対象。

構文

```
(引数リスト) -> {  処理  }
```

メソッドの引数　メソッドの処理ブロック

※ ラムダ式など、名前を付けない関数を匿名関数(無名関数)といいます。

ラムダ式の省略

例)

```
(String t) -> {
        return "** " + t + " ##";
}
```

実行する処理が一文だけなら、中括弧「{～}」は省略可能。
このときの行がreturn文だけであれば、returnも省略する必要があります。

```
(String t) -> "** " + t + " **";
```

引数の型は推論されるので省略することができます。

```
(t) -> "** " + t + "##";
```

さらに、引数がひとつのときにはカッコも省略できます。

```
t -> "** " + t + "##";
```

ラムダ式の利用例

```
public class Test {
	public static void main(String... args) {
		List<String> strs = Arrays.asList("hoge", "fuga", "yeah");

		strs.replaceAll(t -> "** " + t + "##");
		System.out.println(strs.toString());
	}
}
```

※collectionのサブクラスListクラスのreplaceAllメソッドの引数は関数型インタフェース
UnaryOperator