7-9.ラムダ式

ラムダ式とは

ラムダ式は、Java8から導入された構文です。
JavaSilver試験では、直接的に関わる問題は **1〜2問程度** 出題される傾向にあります。

また、少々とっつきにくい分野に感じる人が多いです。
試験合格のための効率だけを考えると、理解が難しければ捨て問題にしてもいい分野かと思います。

関数型インターフェース

ラムダ式を簡潔に言うと、 **関数型インターフェースを実装したクラスのインスタンス生成を簡単に行う構文** となります。

まずは関数型インターフェースについて説明します。
関数型インターフェースとは、 **抽象メソッドを1つだけ持つインターフェイス** のことを指します。

インターフェースは、`interface インターフェース名`の形で定義され、`implements インターフェース名`をクラス定義につけて実装するものでした。
また、インターフェースの中には **抽象メソッド** が含まれ、他のクラスで使う時は、それを **オーバーライド** して内容を確定させた上で使用するものでした。
ラムダ式で扱える関数型インターフェースは、この抽象メソッドを1つだけ持つという条件に注意です。

関数型インターフェースは自作することも出来ますが、標準APIに初めからいくつか使用できるものが準備されています。
JavaSilver対策として押さえておくべき代表的なものには以下のものなどがあります。

```
java.lang.Runnable
// void run();  引数、戻り値無しのrunメソッド

java.util.function.Function<T,R>
// R apply(T); T型の引数を1つ受け取り、指定したR型の戻り値を返すapplyメソッド

java.util.function.Consumer<T>
// void accept(T); 指定したデータ型(T)の引数1つを受け取る、戻り値無しのacceptメソッド

java.util.function.Predicate<T>
// boolean test(T); T型の引数を1つ受け取り、boolean型の戻り値を返すtestメソッド

java.util.function.Supplier<T>
// T get(); 引数無し、T型の戻り値を返すgetメソッド
```

それぞれ別々の抽象メソッドを持っており、引数の渡し方、戻り値の返し方などが違うため場面によって使い分けます。

ラムダ式の基本構造

例えば`Runnable`インターフェースは、 **引数、戻り値無しのrunメソッド** を持っています。

これをラムダ式を用いて使用する時は、下記のように記述します。

```
public class Main {
    public static void main(String[] args) {
        Runnable test = () -> {
            System.out.println("JavaSilver");
        };
        test.run(); // "JavaSilver"を出力
    }
}

//基本構造
関数型インターフェースの型 変数名 = (引数) -> {メソッドの処理内容};
```

イコールの右側がラムダ式の部分となります。

別の関数型インターフェースも使ってみます。
例えば`Function`インターフェースは、`T型の引数を1つ受け取り、指定したR型の戻り値を返すapplyメソッド`を持っています。
TやRの部分のデータ型は、変数宣言時に`<>`をつけて自分で指定します。
また、`java.util`以下のパッケージはデフォルトで読み込まれませんので、自分でimportもしておきましょう。

?

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

public class Main {
    public static void main(String[] args) {
        Function<Integer, String> print = (Integer x) -> {
            return "入力した値は" + x;
        };
        String result = print.apply(10);
        System.out.println(result); // "入力した値は10" を出力
    }
}
```

`Function<Integer, String>`で、引数は`Integer`クラス型、戻り値は`String`クラス型の指定をします。
`(Integer x)`で引数の変数を`x`に設定します。
それを使い、処理内容は文字列と`x`を結合したものを返す内容にしています。
これらのインスタンスを、変数`print`に格納します。

`print.apply(10)`により、インスタンスの中にある`apply`メソッドに数値の`10`を渡しています。
戻り値を`result`に格納し、出力しています。

複雑な見た目ですが、少し構造が把握できてきましたでしょうか。

通常、インターフェースの内容を使いたければ

1. インターフェースを実装したクラスを定義
2. 抽象メソッドをオーバーライド
3. クラスのインスタンス化とメソッドの参照

という手順が必要になりますが、この流れをかなり簡略的に実現しています。
あまり見慣れない構造ですが、それぞれの部分の意味を知っておきましょう。

ラムダ式の省略

JavaSilver試験では、主に正しいラムダ式の記述になっているかを問う問題が出されます。
また、ラムダ式の記述はここから更に省略することもでき、その知識を問う問題が頻出です。

引数の型

引数を設定する時のデータ型は省略可能です。
条件として、複数の引数がある場合は、片方だけ省略するということはできません。

```
        Function<Integer, String> print = (Integer x) -> {
            return "入力した値は" + x;
        };

        //↓データ型を省略

        Function<Integer, String> print = (x) -> {
            return "入力した値は" + x;
        };

//以下のような形はコンパイルエラー
・・・ = (Integer x, y) -> ・・・
```

引数のカッコ

引数のカッコも省略可能です。
ただし条件が2つあり、データ型を記述した場合や、引数が複数ある場合は不可となります。
引数が1単語の時のみ省略可能とおぼえておきましょう。

```
        Function<Integer, String> print = (x) -> {
            return "入力した値は" + x;
        };

        //↓データ型を省略

        Function<Integer, String> print = x -> {
            return "入力した値は" + x;
        };

//以下はコンパイルエラー
print = String x -> ・・・・
print = x, y  -> ・・・・
```

処理を囲む{}

メソッドの処理内容を囲んだ波カッコ{}も省略可能です。
条件として、1文のみの場合しか省略できません。
また、returnがあると省略できません。
しかし、{}を省略した場合に限りreturnも省略することができ、記述可能になります。
returnを書くなら{}は必ずセット、と覚えておくといいでしょう。

```
        Function<Integer, String> print = x -> {
            return "入力した値は" + x;
        };

        ↓

        //NG {}がある時、returnのみ省略は不可
        Function<Integer, String> print = x -> {
            "入力した値は" + x;
        };

        //NG returnがあると{}は省略不可
        Function<Integer, String> print = x ->
            return "入力した値は" + x;
        ;

        //OK
        Function<Integer, String> print = x ->
            "入力した値は" + x;
        ;
```

以上のように、省略は可能だが、条件がいくつかあるというややこしい仕組みになっています。
自分の中で条件を整理しておきましょう。

課題

満点を取れるまで小テストを受験して下さい。

制限時間: 30 分

評定方法: 最高評点

合格点: 100 / 100

受験件数: 76