Javaプログラミング実習

34. 匿名クラスとラムダ式

株式会社ジードライブ

今回学ぶこと

- インナークラス
- 匿名クラスの使い方
- ラムダ式の記述方法

インナークラスとは

- クラス内部に定義されるクラス
  - メソッド内部に定義することもできる
- GUIを持ったアプリケーションのイベント処理など、一度しか利用しない(再利用しない)クラスが必要な場面で主に使われる
  ⇒ アプリ開発時に多く使用される

インナークラスの利用例

例：MyApp.java（Memberはインナークラス）

```
public class MyApp {
    public static void main(String[] args) {
        Member member = new Member("山田太郎");
        member.showInfo();
    }

    static class Member {
        private String name;
        public Member(String name) {
            this.name = name;
        }
        public void showInfo() {
            System.out.println("氏名：" + name);
        }
    }
}
```

staticは必須ではない
⇒ staticメソッド内でMemberクラスを利用しているため記述している

練習

- 練習34-1

匿名クラスとは

- インナークラスの一種
- クラスに名前を付けないので、**無名クラス**とも呼ばれる
- インターフェースを実装(または抽象クラスを継承)したクラスを定義しつつ、同時にインスタンス化して使用する

書式（implementsやextendsといったキーワードは不要）

```
変数 = new インターフェース名 or 抽象クラス名() {
    // 抽象メソッドをオーバーライドして定義する
};
```

※ 変数に代入せず、メソッドの引数として渡す使い方なども可能

匿名クラスの記述例

- 以下のようなインターフェースがあるとする

```
public interface Greet {
        void hello(String name);
        void goodbye(String name);
}
```

- 通常、このインターフェース型のオブジェクトを利用する前に、あらかじめ実装クラスを用意しておく必要がある

```
public class JapaneseGreet implements Greet {
        @Override
        public void hello(String name) { ... }
        @Override
        void goodbye(String name) { ... };
}
```

匿名クラスの記述例

- 匿名クラスの記法を使用することで、実装クラスをあらかじめ定義する必要がなくなり、インスタンス生成時に定義することができる

```
Greet japanese = new Greet() {
        @Override
        public void hello(String name) {
                System.out.println(name + "さん、こんにちは");
        }

        @Override
        public void goodbye(String name) {
                System.out.println(name + "さん、さようなら");
        }
};

japanese.hello("山田");   // 山田さん、こんにちは
japanese.goodbye("山田"); // 山田さん、さようなら
```

匿名クラスの利用例：forEach

Listのもつ**forEach**メソッドでは、引数として**Consumer**型のオブジェクトをとる

MyApp.java

```
public class MyApp {
   public static void main(String[] args) {
       List<String> fruits = Arrays.asList("りんご", "バナナ", "ぶどう");
       fruits.forEach(     );
   }
}
```

引数はConsumer型

Consumerはインターフェースなので
new Consumer() とすることはできない
⇒ Consumerをimplementsするクラスが必要

匿名クラスの利用例：forEach

対応方法１：
forEachメソッドで利用するための ConsumerImplのようなクラスを準備する

MyApp.java

```
public class MyApp {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("りんご", "バナナ", "ぶどう");
        fruits.forEach(new ConsumerImpl());
    }

    class ConsumerImpl implements Consumer<String> {
        @Override
        public void accept(String item) {
            System.out.println(item);
        }
    }
}
```

匿名クラスの利用例：forEach

対応方法２：
引数として匿名クラスを利用

MyApp.java

```
public class MyApp {
   public static void main(String[] args) {
      List<String> fruits = Arrays.asList("りんご", "バナナ", "ぶどう");
      fruits.forEach(new Consumer<String>() {
          @Override
          public void accept(String item) {
              System.out.println(item);
          }
      });
   }
}
```

練習

- 練習34-2
- 練習34-3

関数型インターフェース

- オーバーライドすべきメソッドが1つだけ定義されているインターフェースを関数型インターフェースと呼ぶ
- java.util.functionパッケージには、あらかじめ様々な関数型インターフェースが用意されている
  - https://docs.oracle.com/javase/jp/21/docs/api/java.base/java/util/function/package-summary.html

代表的な関数型インターフェース

インターフェース	実装すべきメソッド	説明
Function	R apply(T arg)	引数を処理して、何かしらのデータを返す
Consumer	void accept(T arg)	引数を消費(利用)して、何かしらの処理をする ※ forEachメソッドの引数は、このConsumer型
Supplier	T get()	何かしらのデータを供給する
Predicate	boolean test(T arg)	引数を判定して、true または false を返す

ラムダ式とは

- 匿名クラスを簡易的に記述するための表記方法
  - **関数型インターフェース**を実装する場合に利用できる

通常の書式

```
new インターフェース名() {
    @Override
    アクセス修飾子 戻り値型 メソッド( 引数 ) {
        処理
    }
}
```

ラムダ式

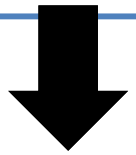
```
( 引数 ) -> { 処理 }
```

ラムダ式の記述例

通常の書式

```
fruits.forEach(new Consumer<String>() {
    @Override
    public void accept(String item) {
        System.out.println(item);
    }
});
```

削除可能な部分



ラムダ式

```
List<String> fruits = Arrays.asList("りんご", "バナナ", "ぶどう");
fruits.forEach((String item) -> {
    System.out.println(item);
});
```

ラムダ式の記述方法

- 匿名クラスからの変更内容
  - **new インターフェース名 (){ }** を削除する
  - メソッドの **アクセス修飾子 戻り値の型 メソッド名**を削除する
  - メソッドの **(引数)** と **{処理内容}** との間に **->** を記述する

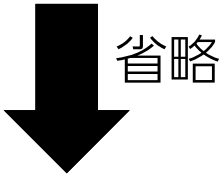
- 引数の型は省略可
  - 引数が1個の場合、**()**も省略可能
  - 引数が0個の場合は**()**が必要
- ステートメントが1個の場合、**セミコロン**と**{}**を省略可
  - ステートメントがreturn文だけの場合、returnキーワードは記述しない

ラムダ式の記述例

省略した場合の記述例

```
List<String> fruits = Arrays.asList("りんご", "バナナ", "ぶどう");
fruits.forEach((String item) -> {
    System.out.println(item);
});
```

省略可能な部分




```
List<String> fruits = Arrays.asList("りんご", "バナナ", "ぶどう");
fruits.forEach(item -> System.out.println(item));
```

メソッド参照

- ラムダ式の引数をそのままメソッドの引数に渡す処理の場合には、メソッド参照という構文が利用できる

書式

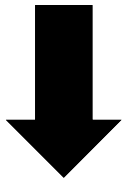
```
インスタンス名::インスタンスメソッド名
```

```
クラス名::クラスメソッド名
```

メソッド参照

ラムダ式

```
List<Integer> scores = Arrays.asList(60, 20, 80);
scores.stream()
    .forEach(e -> System.out.println(e));
```



メソッド参照

```
scores.stream()
    .forEach(System.out::println);
```

練習

- 練習34-4