

# 09 OOP - その3



Created by GT F

Last updated 2019-08-15

- 抽象メソッド
- 抽象クラス
- 抽象クラスを継承する
- インターフェース
- インターフェース UML 表現
- 実装方法
- インプリメント
- 抽象クラス&インターフェースの利用方法
  - 一般的の実装
  - 内部クラスで実装
  - 匿名クラス
  - ラムダ式
- 列挙型
  - 列挙型とSwitch
  - 列挙型をループする
  - 列挙型とString
  - フィールド&メソッド
  - 現場で列挙型の利用パターン
- 質問集

## 抽象メソッド

修飾子 `abstract` を宣言したメソッドは抽象メソッドになります。

1. 抽象的メソッドはキーワード `abstract` を付けること。
2. 抽象的メソッドは実体なし。
3. 抽象的メソッドは必ず抽象クラス又インターフェースを所属する。

抽象的メソッド定義

```
1 public abstract void hello();
```

## 抽象クラス

修飾子 `abstract` を宣言したクラスは抽象クラスになります。同様に、`abstract` が付けられたメソッドは抽象的メソッドを呼びます。抽象的なメソッドは自体ない、抽象的なクラスはインスタンス化


できません。抽象メソッド所属（しよぞく）するクラスは必ず抽象クラスであること。

```
1 public abstract class BasePet {
2     public abstract void hello(); // 抽象メソッド
3     public void eat() {}; // 普通のメソッド
4 }
```

※現場に抽象クラスは一般的に頭文字を**Base**を付けること。

抽象クラスは、それ自体を利用することはなく、単独では意味ありません。継承しなければ、利用できません。

 抽象クラスを利用する場合、継承する必要があります。

 抽象メソッドは自体がないです。

## 抽象クラスを継承する

抽象クラスを継承する場合、下位クラスは必ず抽象メソッドを実現（メソッドの自体を実装する）しなければいけません。

```
1 // Cat.java
2 public class Cat extends BasePet {
3     // 抽象メソッドをオーバーライドしなければいけません。
4     @Override
5     public void hello() {
6         System.out.println("ニャー");
7     }
8 }
```

※抽象的クラスを抽象的クラス継承する場合、メソッドを実現しなくてもいいです。

## インターフェース

インスタンスは特殊のクラスの1つで、属性やメソッドを**宣言**は持ちますが、**実装は持ちません**。インスタンスの宣言はキーワード**interface**を使用します。interfaceにてメソッドはすべて `public` であること。アクセス制御を省略可能です。インターフェースは**多重継承**ができます。インターフェースの別名は**完全抽象クラス**です。

## インターフェース UML 表現

UMLの標準的な**クラスの表記法**によ `<<interface>>` というキーワードを付けることです。以下はInterface記述の例を示しています。

<<Interface>> Interface
+ field1: Type + field2: Type
+ method1(Type): Type + method2(Type, Type): Type


## 実装方法

キーワードは「interface」を使用して、インターフェースを定義する。

```
1 public interface IPet {  
2     public abstract void hello();  
3 }
```


※多くの現場では、インターフェースクラスの見頭文字は大文字のIを付けます。

インターフェースはクラスの定義を似ていますが、ただ、異なる所は、すべてのメソッドは `public` と `abstract` (抽象的)に設定必要があります。

 インターフェースに定義したメソッドは `public` と `abstract` の為、省略可能です。

例： `public` & `abstract` を省略する場合

```
1 public interface IPet {  
2     // public abstract 省略可能  
3     void hello();  
4 }
```

 普通のクラスに、`public` を省略した場合、`private` をデフォルトに付けます。

## インプリメント

キーワード `implements` を使用して、インターフェースをインプリメントします。インターフェースはインプリメントする場合、具象的のクラスは必ずすべてのメソッドの**本体**を実装しなければならない。

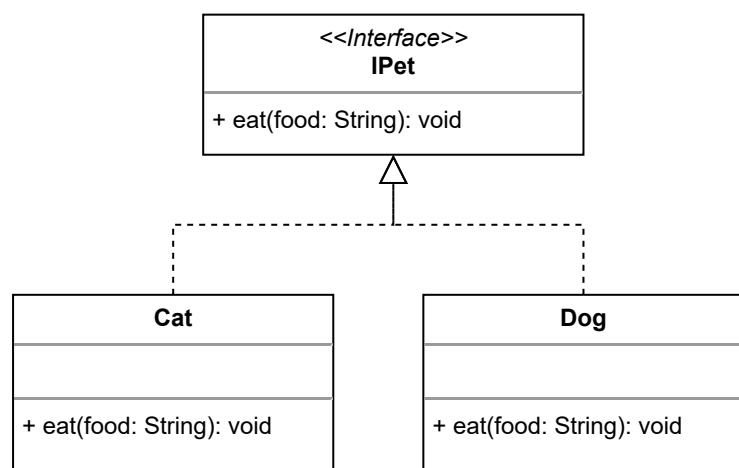
```
1 // Cat.java  
2 public class Cat implements IPet {  
3     @Override  
4     public void hello() {  
5         System.out.println("ニャー");  
6     }  
}
```

```

7  }
8  // Dog.java
9  public class Dog implements IPet {
10     @Override
11     public void hello() {
12         System.out.println("ワン");
13     }
14 }

```

インターフェースをインプリメントするUML表現は



⚠️ 一つのクラスは複数のインスタンスをインプリメントできます。

## 抽象クラス&インターフェースの利用方法

抽象的クラスとインターフェースの利用する場合、継承又はインプリメントが必要です。以下4つの利用方法があります。

1. 一般的の実装
2. 内部クラスで実装
3. 匿名クラス
4. ラムダ式

### 一般的の実装

全文に記述した通り、クラスを新規にして、メソッドを実装します。以下通り3つのクラスを作成しました。

```

1  // IPet.java
2  public interface IPet {
3      // public abstract 省略可能

```

```

4     void hello();
5 }
6 // Cat.java
7 public class Cat implements IPet {
8     @Override
9     public void hello() {
10         System.out.println("ニャー");
11     }
12 }
13 // Kicker.java
14 public class Kicker {
15     public static void main(String... args) {
16         IPet cat = new Cat();
17         cat.hello();
18     }
19 }

```

## 内部クラスで実装

いちいちクラスを新規作成は面倒の為、内部クラスで実装可能です。以下場合2つクラウを作成しました。

```

1 // IPet.java
2 public interface IPet {
3     void hello();
4 }
5 // Kicker.java
6 public class Kicker {
7     // 内部クラス
8     public static class Cat implements IPet {
9         @Override
10        public void hello() {
11            System.out.println("ニャー");
12        }
13    }
14    public static void main(String... args) {
15        IPet cat = new Cat();
16        cat.hello();
17    }
18 }

```

## 匿名クラス

抽象的クラスとインターフェースは匿名で可能。

```

1 // IPet.java
2 public interface IPet {
3     void hello();
4 }
5 // Kicker.java
6 public class Kicker {

```

```

7      public static void main(String...args) {
8          // 匿名クラス
9          IPet cat = new IPet() {
10             @Override
11             public void hello() {
12                 System.out.println("ニャー");
13             }
14         };
15         cat.hello();
16     }
17 }

```

## ラムダ式

更に、匿名クラスを簡化する場合、ラムダ（ $\lambda$ ）式で記述可能です。ラムダ式は Java1.8 (Java8) の新し機能である。構文は

```

1  (実装するメソッドの引数) -> {
2      // 処理内容
3  };

```

以下は実装の一例：

```

1  // IPet.java
2  public interface IPet {
3      void hello();
4  }
5  // Kicker.java
6  public class Kicker {
7      public static void main(String...args) {
8          // ラムダ式
9          IPet cat = () -> {
10             System.out.println("ニャー");
11         };
12         cat.hello();
13     }
14 }

```

ラムダ式は強制的に使用しなくてもいいです。なお、IDEの自動コデイン機能を利用して、内部クラスを自動的にラムダ式変更可能。

## 列挙型

列挙型は特定の値のみを持つ型で、プログラマが任意に定義できます。キーワード `enum` を利用して定義します。

```

1  // 性別
2  public enum SEX {
3      MAIL, // 男性
4      FEMAIL // 女性
5  }

```

列挙型の追加方法は「列挙型名.列挙した値」とします。

```
1 SEX sex = SEX.MALE; // 男性
```

## 列挙型とSwitch

列挙型はSwitch文で利用可能です。

```
1 public class Kicker {
2
3     public static void main(String... args) {
4         SEX sex = SEX.FEMALE;
5         switch (sex) {
6             case MALE:
7                 System.out.println("男性");
8                 break;
9             case FEMALE:
10                System.out.println("女性");
11                break;
12            default:
13                throw new AssertionError(sex.name());
14        }
15    }
16 }
```

## 列挙型をループする

列挙型のクラスに静的なメソッド `values()` を持っています。該当メソッドはEnumに定義された選択肢を返します。

```
1 public class Kicker {
2
3     public static void main(String... args) {
4         for (SEX s : SEX.values()) {
5             System.out.println(s);
6         }
7     }
8 }
```

## 列挙型とString

列挙型の名称を取得するメソッド: `name()`

```
1 public static void main(String... args) {
2     for (SEX s : SEX.values()) {
3         System.out.println(s.name()); //Enumの名称を取得する
4     }
5 }
```


Stringから列挙型を変更するメソッド：`valueOf(String value)`

```
1 public static void main(String... args) {
2     SEX s = SEX.valueOf("MAIL"); // 名称からEnum値を変換する
3     System.out.println(s.name());
4 }
```

## フィールド&メソッド

列挙型にもフィールド、メソッド、コンストラクタを定義可能です。

```
1 // SEX.java
2 public enum SEX {
3     MALE("1", "男性"),
4     FEMALE("0", "女性")
5     ; // 注意：ここにセミコロン「;」
6
7     // 属性
8     final String code;
9     final String description;
10
11     // コンストラクタ（列挙型のコンストラクタは必ずprivate）
12     SEX(String code, String description) {
13         this.code = code;
14         this.description = description;
15     }
16     // メソッド
17     public String getDescription() {
18         return description;
19     }
20 }
```

 列挙型は `final` である、これ以下の継承はできないが、他のクラスを継承し、インターフェースをインプリメントする可能です。

## 現場で列挙型の利用パターン

現場で列挙型はコード値として使用する場面が多いです。

```
1 // SEX.java
2 public enum SEX {
3     MALE("1", "男性"),
4     FEMALE("0", "女性");
5     final String code;
6     final String description;
7
8     SEX(String code, String description) {
9         this.code = code;
10        this.description = description;
11    }
```



```

12 // コード値からEnum値を変換する
13 public static SEX getEnum(String code) {
14     for (SEX s : SEX.values()) {
15         if (s.code.equals(code)) {
16             return s;
17         }
18     }
19     return null;
20 }
21 }
22 // Kicker.java
23 public class Kicker {
24     public static void main(String... args) {
25         String code = "1";
26         System.out.println(SEX.getEnum(code).description); // 男性
27     }
28 }

```

## 質問集

質問 1 : 以下ソースにラムダ式の使う場所を解釈してください。

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5
6 public class GUISample {
7
8     public static void main(String[] args) {
9         JFrame window = new JFrame("DCNet Java 教育");
10        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        window.setSize(800, 600);
12        JButton btn = new JButton("hello world");
13        window.getContentPane().add(btn);
14
15        btn.addActionListener((ActionEvent e) -> {
16            System.out.println("ボタンをクリックしました。");
17        });
18
19        window.setVisible(true);
20    }
21 }

```

質問 2 : 以下ソースに匿名クラスの利用する行は？

```

1 public class Kicker {
2
3     public static void main(String... args) {
4         new Thread() {
5             @Override
6             public void run() {
7                 System.out.println("thread running...");

```

```
8         }  
9     }.start();  
10 }  
11 }
```

 [Like](#) Be the first to like this

No labels 