# ポケモンの世界をオブジェクトで考えよう！

2018/1/29

**1. オブジェクトを考える**

オブジェクト指向とは、世の中にある ”モノ" をプログラムを開発しやすいように整理しやすいようにする考え方です。
Java言語の世界では、クラスとして、データ(状態)と処理(ふるまい)を持つ形で実装されます。

データ（状態）　：　クラスが所持している状態を表すもの。フィールドとして表現される
機能（ふるまい）　：　クラスが所持している処理や動き、機能などを表すもの。メソッドとして表現される

オブジェクト指向の例として…

| | 人間 | 木 | 車 |
|---|---|---|---|
| クラス名 | Human | Tree | Car |
| フィールド<br>→ 名詞で表現 | name：名前 | height：高さ | gas：ガソリン残量 |
| | age：年齢 | growthRate:成長速度 | speed：走行速度 |
| | sex：性別 | | direction：進行方向 |
| メソッド<br>→ 動詞で表現 | eat()：食べる | grow()：成長する | charge()：ガソリンを補給する |
| | sleep()：寝る | getHeiht()：現在の高さを知る | ccelerateaccelerate()：加速する |
| | work()：仕事する | | drive()：走る |

ポケモンをJavaの世界で、表現することを考えてみましょう。
ポケモンが持っているデータ（状態）と機能（ふるまい）をそれぞれ、グループで話し合ってみましょう。
＜個人で考える：3分　・　チームで考える：10分　・　発表内容を整理する：2分＞

| 演習：15分 | ポケモン(ポケットモンスター) |
|---|---|
| クラス名 | Pokemon |
| フィールド<br>→ クラスの持つ<br>　データを表す<br>→ 名詞で表現される | ①<br>②<br>③<br>④ |
| メソッド<br>→ クラスの持つ<br>　機能を表す<br>→ 動詞で表現される | ①<br>②<br>③<br>④<br>⑤ |

サンプルプログラム　階層

```
pokemon
  ├ PokemonType.java                ポケモンのタイプを表現する列挙型クラス
  │
  ├────work                         演習問題用フォルダ：演習には「//TODO」とコメント記載アリ
  │   ├────pokemon01
  │   │   └────createclass          クラス生成演習
  │   │       Pokemon.java          ※ 以降の演習にも含まれるため、本日は省略
  │   │       PokemonMain.java
  │   │
  │   ├────pokemon02                コンストラクタ演習
  │   │   └────constructor          ※ 以降の演習にも含まれるため、本日は省略
  │   │       Pokemon.java
  │   │       PokemonMain.java
  │   │
  │   ├────pokemon03                継承関係演習
  │   │   └────inheritance
  │   │       Eve.java
  │   │       Pikachu.java
  │   │       Pokemon.java
  │   │       PokemonMain.java
  │   │
  │   └────pokemon04                ポリモーフィズム演習
  │       └────polymorphism
  │           Eve.java
  │           Pikachu.java
  │           Pokemon.java
  │           PokemonIF.java
  │           PokemonMain.java
  │
  └────answer                       workフォルダと同階層で、回答を用意
      ├────pokemon01                ※ 解説や動作確認時に利用
      ├────pokemon02
      ├────pokemon03
      └────pokemon04
```

## 2. 継承関係を考える

1つ1つすべての"モノ"をプログラムするのは、大変です。
似ている"モノ" や 共通する"モノ" もあり、あるクラスの機能を引き継ぐために、継承という仕組みがあります。
以下の関係性を持つ、プログラムを作成しましょう。
＜演習①②③　：　20分　・余裕があれば　補足演習④⑤＞

**演習①**　サンプルプログラムにて、実装されている　Pokemon　クラスに目を通しましょう。
サンプルプログラム内に解説コメントが記載されています。
pokemon>work>pokemon03>inheritance>Pokemon.java

| スーパークラス | **ポケモンクラス** |
|---|---|
| クラス名 | Pokemon |
| フィールド<br>→ 名詞で表現 | name：名前 |
| | type：タイプ |
| | hp：HP |
| | mp：MP |
| | item：持っているアイテム |
| コンストラクタ<br>→ 初期値設定 | Pokemon()：引数なしのコンストラクタ |
| | Pokemon(名前,タイプ,アイテム)：引数3つのコンストラクタ |
| | Pokemon(名前,タイプ)：2つのコンストラクタ |
| メソッド<br>→ 動詞で表現 | attack()：攻撃する |
| | useItems()：アイテムを利用する |
| | escape()：逃げる |
| | showStatus()：ステータスを表示する |

**演習②**　サンプルプログラムにて、実装されている　Pikachu　クラスに目を通しましょう。
pokemon>work>pokemon03>inheritance>Pikachu.java

| サブクラス | **ピカチュウクラス** | |
|---|---|---|
| クラス名 | Pikachu | ← スーパークラスを継承して、サブクラスとして定義 |
| フィールド<br>→ 名詞で表現 | name：名前 | ← スーパークラスからフィールドを引き継いで利用可 |
| | type：タイプ | Pikachuクラス内では、定義しない |
| | hp：HP | ← サブクラス特有のフィールドがあれば |
| | mp：MP | 定義をする |
| | item：持っているアイテム | |
| コンストラクタ<br>→ 初期値設定 | Pikachu()：引数なしのコンストラクタ | ← コンストラクタは、 |
| | Pikachu(名前,タイプ,アイテム)：引数3つのコンストラクタ | 継承されないので注意 |
| | Pikachu(名前,タイプ)：2つのコンストラクタ | |
| メソッド<br>→ 動詞で表現 | attack()：攻撃する | ← スーパークラスから処理を引き継ぐが、 |
| | useItems()：アイテムを利用する | Pikachuクラス特有の処理を再定義する |
| | escape()：逃げる | |
| | showStatus()：ステータスを表示する | ← サブクラス特有の処理を定義しない場合は、 |
| | | スーパークラスの処理を引き継ぐ |

**演習③**　サンプルプログラムにて、実装されている　Pikachu　クラスを参考にして、 Eve　クラスを実装しましょう
サンプルプログラム内に「//TODO」と記載されている部分を記載しましょう
pokemon>work>pokemon03>inheritance>Eve.java
pokemon>work>pokemon03>inheritance>PokemonMain.java

| サブクラス | **イーブイクラス** | |
|---|---|---|
| クラス名 | Eve | ← スーパークラスを継承して、サブクラスとして定義 |
| フィールド<br>→ 名詞で表現 | name：名前 | ← スーパークラスからフィールドを引き継いで利用可<br>Eve クラス内では、定義しない |
| | type：タイプ | |
| | hp：HP | ← サブクラス特有のフィールドがあれば<br>定義をする |
| | mp：MP | |
| | mp：MP | |
| コンストラクタ<br>→ 初期値設定 | Eve()：引数なしのコンストラクタ | ←　コンストラクタは、<br>継承されないので注意 |
| | Eve(名前,タイプ,アイテム)：引数3つのコンストラクタ | |
| | Eve(名前,タイプ)：2つのコンストラクタ | |
| メソッド<br>→ 動詞で表現 | attack()：攻撃する | ← スーパークラスから処理を引き継ぐが、<br>Eve クラス特有の処理を再定義する |
| | useItems()：アイテムを利用する | |
| | escape()：逃げる | ← サブクラス特有の処理を定義しない場合は、<br>スーパークラスの処理を引き継ぐ |
| | showStatus()：ステータスを表示する | |

**▼実行結果**

pokemon>work>pokemon03>inheritance>PokemonMain.java

```
C:¥¥work>javac  *.java
C:¥¥work>java  PokemonMain
----------------------------------
 名　前 :　ピカチュウ
タ イ プ :　TYPE_ELECTRIC
  H　P　:　 100
  M　P　:　 100
アイテム :　ピカチュウのアイテム
----------------------------------
ピカチュウは逃げた
----------------------------------
 名　前 :　イーブイ
タ イ プ :　TYPE_NORMAL
  H　P　:　 100
  M　P　:　 100
アイテム :　 null
----------------------------------
Pokemon.escape()

C:¥¥work>
```

…①　① フォルダ内のJavaファイルをすべてコンパイル

…②　② 実行ファイル　PokemonMain　を実行

…③　③ pikachu.showStatus()　メソッドの実行結果
ピカチュウオブジェクトに格納されたデータが
表示される

…④　④ pikachu.escape()　メソッドの実行結果
Pikachu クラスでは、スーパークラスのescape()メソッド
をサブクラスで再定義しているので、
Pikachuクラスの escape()メソッドが実行される

…⑤　⑤ eve.showStatus()　メソッドの実行結果
イーブイオブジェクトに格納されたデータが
表示される

…⑥　⑥ eve.escape()　メソッドの実行結果
Eveクラスでは、escape()が実装されていないため、
スーパークラスの escape() メソッドが実行される

**補足.　オーバーライド　と　オーバーロード　の違い**

オーバーライド：スーパークラスのメソッドを再定義（上書きのイメージ）
オーバーロード：同名のメソッドにおいて、引数が異なるメソッド
　　引数の型や個数が異なっても、同じ処理を実施できるように準備をする

| |
|---|
| オーバーロード例）<br>足し算をする　add()　メソッドの場合<br>引数に、整数が受け渡されても、小数が受け渡されても、足し算処理ができるように準備をする<br><br>add ( int　a , int　b)　　← 整数型(int型)の足し算処理<br>add ( double　a , double　b)　　← 小数型(double型)の足し算処理 |

**補足演習④**　サンプルプログラム　Eve クラスにて、実装されている userItems() メソッドを参考にしましょう。
コメントアウトされているので、余裕があれば、取り組みましょう。
pokemon>work>pokemon03>inheritance>Eve.java

useItems()　← スーパークラスで定義されている　userItems()メソッドを
サブクラス特有の処理で上書き
useItems(String item)　← Eve クラス内で、2つ userItems() メソッドが定義されているが、
引数が異なるため、宣言可能

**3.　ポリモフィズムを考える**

ポリモーフィズム（多態性）とは、同じ動作で、オブジェクトごとに異なる動作をさせることです。
ポリモーフィズを実現するために、Java言語では、インターフェースを利用します。

インターフェースが同じであれば、たとえ多少異なっていても利用することができます。

| |
|---|
| インターフェースの例）<br>テレビのリモコンの使い方を知っていれば、以下のどのリモコンであっても、利用することができます。<br>・アナログテレビ<br>・レグザ　} どんなテレビであってもチャンネルを変えるためにリモコンを利用する<br>・携帯のワンセグ |

ポリモーフィズム（多態性）とは、同じ動作で、オブジェクトごとに異なる動作をさせることです。
Java言語のインターフェースは、実装予定のクラスで必ず処理内容を記述してほしい、メソッド一覧を定義しています

インターフェースは、メソッドのシグネチャ（メソッド名と引数）のみを定義したものです。
実際の処理内容は、インターフェースを実装したクラスにて、定義します。
（このインターフェースを利用すると、この処理は必須だから必ず実装してね）

人間の役割を例にとると、以下のように整理することができます。
例えば、「食べる」「寝る」「仕事する」という3つの処理を実施するとインターフェースで定義すると…

| | 人間インターフェース |
|---|---|
| インターフェース名 | HumanIF |
| メソッド<br>→ 定義のみ | eat()：食べる |
| | sleep()：寝る |
| | work()：仕事する |

← HumanIFを実装する場合、
この3つのメソッド（処理）は必ず実装します

人間インターフェースを実装する、「先生クラス」「生徒クラス」「会社員クラス」は、それぞれ、
「食べる」「寝る」「仕事する」という3つの処理を実装する制約がかかります。

| 先生クラス | 生徒クラス | 会社員クラス | |
|---|---|---|---|
| **Teacher** | **Student** | **Employee** | ← 同じ人でも役割が異なる |
| eat()：食べる | eat()：食べる | eat()：食べる | ← クラス名が異なる |
| sleep()：寝る | sleep()：寝る | sleep()：寝る | ← ｲﾝﾀｰﾌｪｰｽを利用すると<br>メソッドの実装が必須 |
| work()：仕事する | work()：仕事する | work()：仕事する | ← クラス特有の処理 |

しかし、同じ「仕事をする」というwork()メソッドを実装しても、
　先生クラスのwork()メソッドを実行　→　勉強を教える
　生徒クラスのwork()メソッドを実行　→　勉強をする
　会社員クラスのwork()メソッドを実行　→　資料を作る
など、「役割」に応じて、「仕事をする」というクラス特有の処理内容が異なります。

繰り返しになりますが、ポリモーフィズム（多態性）とは、同じ動作（＝同じメソッド呼び出し）で、
オブジェクトごと（＝役割ごと）に異なる動作をさせることで、
現実世界を整理して、プログラムを開発しやすいように整理しやすいようにしているのです。

**補足演習⑤**　以下のクラス体系になっています。ピカチュウクラス、イーブイクラスを参考に
ニャースクラス(Meowth)を作成しましょう。