目次

オブジェクト指向とは
オブジェクト指向とは、複数のオブジェクトを組み合わせてプログラムを構築する考え方の一つです。オブジェクトとはデータと処理の集まりです。この集まりは漠然とした集まりではなく、あるテーマを持った集まりです。オブジェクト指向と反する考え方は、手続き型です。世界は手続き型であり、オブジェクト指向でもあります。まずは、この考え方の違いをそれぞれ見て、オブジェクト指向の理解を深めます。

手続き型とは
手続き型とは、時間の流れに沿って物事が順番に起こっていく流れを表したものです。
例えば、朝起きて、歯を磨き、着替えて、会社に行く。こうした流れを手続き型のプログラムで表現できます。
しかし、現実世界では予期せぬ出来事があります。朝起きたら具合が悪く、病院に立ち寄りたいときもあります。こういう予期せぬ出来事が起きる可能性がある場合は、手続き型の考え方は向いていません。

オブジェクト指向とは
前述の通り、オブジェクトとはあるテーマをもった集まりです。
日常には、さまざまなオブジェクトに関わりがあります。
朝起きて、歯を磨き、着替えて、会社に行く。こうした活動の中にも様々なオブジェクトがあり、まずはこうした活動をする「人間」がいて、その「人間」が所属する「会社」があります。
「人間」というオブジェクトには、様々なデータがあります。名前・年齢・血液型などです。また、オブジェクトは振る舞い（処理）を持ちます。時間になったら起きる、「名前を教えて」と聞かれたら「名前を答える」、体調が悪かったら病院に行くなどです。
こうした振る舞いを事前にプログラムしておくと、予期せぬ出来事をプログラムする必要はありません。

なぜオブジェクト指向が必要なのか
オブジェクト指向はプログラムを構築（設計）する考え方の一つです。アプリケーションを作るときに、完全かつ正確に仕様を満たしていて、このアプリケーションは永久に変化しないとすると、設計を気にする必要はありません。
しかし、現実問題として永久に変化しないということはなく、必ず変化が訪れます。このときに重要になってくるのが設計です。設計は将来訪れる変化に対応して変更するために行うのです。オブジェクト設計は、オブジェクトが変更を許容できるようなかたちで依存関係を構築するための考え方です。つまり、オブジェクト指向は未来のための変更コストを削減するためにあるのです。次に、具体的な例を用いて説明します。

具体例
生徒管理アプリケーションを開発していると想定してみます。
以下のような機能があります：
生徒の名前と緊急連絡先を登録できる
部活の名前を登録できる
生徒は部活と紐づく(所属する)

Ruby on Rails では以下のようになります

```
class Student < ActiveRecord::Base
  belongs_to :club

  # DBにname, phone_numberカラムを持つ
end

class Club < ActiveRecord::Base
  has_many :student

  # DBにnameカラムを持つ
end
```

この学校は生徒が部活に絶対所属する校則がある学校でした。
そしてアプリケーションを作って納品しましたが、校則が変わり生徒が部活に所属しないことが可能になりました。
しかしながら、ビューに生徒の部活を表示する部分がたくさんあります。

ビューはこのようになっています。

```
<%= @student.name %> : <%= @student.club.name %>
```

以下のように変更して対応しましたが、表示する箇所がたくさんあり変更は容易ではありません。

```
<% if @student.club %>
  <%= @student.name %> : <%= @student.club.name %>
<% else %>
  <%= @student.name %> : <%= '帰宅部' %>
<% end %>
```

オブジェクト指向の考え方で、設計をしておけばこのような自体を避けられました。具体的にいうと、生徒の所属を表示する処理を実装しておくべきです。

```
class Student < ActiveRecord::Base
  belongs_to :club

  def display_info
    if club
      "#{name} : #{club.name}"
    else
      "#{name} : 帰宅部"
    end
```

```
  end
end

# view
<%= @Student.display_info %>
```

オブジェクト指向の三大要素

カプセル化

他のオブジェクトから、干渉されないようにして、外部に対して必要な情報や手続きのみを提供することです。
オブジェクトは、外部から自身のデータをカプセル化（隠匿）しています。オブジェクトはどれだけ多く、あるいは少なくデータを晒すか自身で決めます。

具体例
以下のコードだと、studentクラス（オブジェクト）からインスタンス変数を生成して、nameの呼び出しをする際にエラーがでます。

```
class Student
  def initialize(name)
    @name = name
  end
end

riku = Student.new('Riku')
puts riku.name #=>undefined method `name' for #<Student:0x007fa3f4057d88
@name="Riku">
```

なぜカプセル化が必要なのか？
なぜカプセル化が必要なのでしょうか。これは他のオブジェクトに対して「余計な処理を見せて惑わせない」「処理の意図しない使われ方」をさせてないという配慮です。具体例をあげて説明します。

具体例
もし下記の例のようにStudentクラスのインスタンスのデータが勝手に書き換えることが可能になってしまうと、他のオブジェクトに干渉されてしまいますし、今後何か変更が起きたときに、Studentクラスは他のオブジェクトまで責任をとらなければいけません。
しかし、カプセル化とは絶対にデータを提供しないという原則ではありませんので、必要に応じて外部に晒す必要性があるデータに関しては、提供しても大丈夫です。

```ruby
class Student
  def initialize(name)
    @name = name
  end
end

riku = Student.new('Riku')
riku.name = 'taro'
```

継承

特定のオブジェクトの振る舞い（メソッド）を引き継いで使うことを継承といいます。
継承されるクラスを「スーパークラス（親クラス）」、継承したクラスを「サブクラス（小クラス）」といい、継承には親子の関係が存在します。
しかしながら継承するのに適切かどうかは性質や概念が共通している必要があります。判断する方法があり、「サブクラスはスーパークラスの一種である」と読んだときに違和感がないか確かめることです。これは「is-aの関係」を呼ばれます。

> **具体例**
> 車オブジェクトがあったとします。
> それぞれ、加速・減速・クラクションを鳴らす振る舞い（メソッド）が定義されてます。新しくパトカーのオブジェクトを作成したいとします。すると以下のようになります。また、「パトカーは車の一種である」と読んでも違和感がありません。「is-aの関係」が成立していることもわかります。

```ruby
class Car
  def speed_up
    # 加速する処理
  end

  def speed_down
    # 減速する処理
  end

  def horn
    # クラクションを鳴らす処理
  end
end
```

```ruby
class PatrolCar < Car #クラスの継承

  def siren
    # サイレンを鳴らす処理
  end
end
```

なぜ継承が必要なのか？
コードの再利用ができる点にあります。つまり、継承をしないと上記の例だと、PatrolCarクラスに加速の処理、減速の処理・・・とCarクラスに書いてあるものを再度定義しなければなりません。また、加速の処理に重大な欠陥が見つかったときに、継承をしていればCarクラスの修正のみで済みます。

ポリモーフィズム

ポリモーフィズムとは「多様性・多態性」と訳されます。
オブジェクト指向においては、同じ名前のメソッドを複数のクラスで利用できるようにし、そのメソッドを通じて動作を切り替えることができるようにすることです。

具体例
作業員クラス（Workerクラス）があり、プログラマーとデザイナークラスに継承されています。それぞれjobメソッドが定義されています。同じjobメソッドを呼び出しても違う処理が実行されるようになります。これがポリモーフィズムです。

```
class Worker
  def job
    raise NotImplementedError
  end
end

class Programmer < Worker
  def job
    # プログラムを書く処理
  end
end

class Designer < Worker
  def job
    # デザインをする処理
  end
end

programmer = Programmer.new
designer = Designer.new
programmer.job
designer.job
```

なぜポリモーフィズムが必要か？
ポリモーフィズムは、多岐に渡るオブジェクトが、同じメッセージに応答できることを指します。つまり、メッセージの送り手が、受け手を気にする必要がなく、受け手はそれぞれ独自化した振る舞いを提供します。

参考文献

Sandi Metz、髙山泰基『オブジェクト指向設計ガイド〜Rubyでわかる進化しつづける柔軟なアプリケーションの育て方』株式会社技術評論社、2016年

tutinoco『オブジェクト指向と10年戦ってわかったこと』
https://qiita.com/tutinoco/items/6952b01e5fc38914ec4e