title

Javaでオブジェクト指向プログラミングを完全に理解する

tags

Java, OOP, オブジェクト指向, 初心者

はじめに

こんにちは!

今回はJavaでオブジェクト指向プログラミングを説明したいと思います. ロールプレイングゲームを題材にして, 説明していきます.

前回はPythonでオブジェクト指向についての記事を書きましたが, わかりにくいなと感じたので, 改めて記事を書くことにしました.

わかりやすさを重視するので, 本来の説明とは違う表現があるかもしれません. ご了承ください.

わかりにくいところや明らかに間違えているところがあれば(優しく)指摘してください!

前提知識

- 特になし
- 環境構築の説明はいたしません.

対象読者

- プログラミング初心者
- Javaが初めてのプログラム言語の人
- 授業などでJavaの課題が出ているが、まったくわからない人
- オブジェクト指向プログラミングについて理解したい人

さっそく説明するよ

本題

そもそもオブジェクト指向プログラミングって何?

説明すると長くなってしまうので, Wikipediaを読んでください!

オブジェクト指向プログラミング-Wikipedia

ざっくり説明すると, オブジェクト(モノや動作)をプログラムで表現しよう!ということです.

Javaについて教えて!

Javaはオブジェクト指向型言語です.

拡張子は`ファイル名.java`です.

基本的なクラスを以下に示します.

```
// 世界に挨拶をするクラス
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

色々でてきましたね.

ですが, 大丈夫です. しっかり説明します!

最初の行は何?

コメントです.

一行のコメントは `// hogehoge~`

複数行のコメントは `/* hogehoge~~ */`

のように書きます.

コメント化することをコメントアウトすると言ったりします.

コメントは, クラスの意図や処理の意図の説明, デバッグなどに利用できます.

`public class Main { ... }` って何?

`Main`クラスを定義するよ!っていうことです.

波かっこの内側に変数やメソッド(後述)を定義します.

さて, 今回は`Main`クラスでしたが, クラス(後述)の名前は自由に付けてもいいのでしょうか?

▶ 答え

Javaでは, `ファイル名.java`には`ファイル名`のクラスを定義しなくてはなりません.

他の言語...例えば, CやPythonではこういった制限はありません.

では, `Hoge`クラスを定義する場合のファイル名はどうなるでしょうか?

▶ 答え

ファイル名とクラス名は同じにする必要があるでしょうか?

▶ 答え

`public static void main(String[] args) { ... }` って何?

`main`メソッドとよびます.

プログラムを実行するときにこの`main`メソッドが呼ばれます.

今はまだおまじないだと思っていてください.

`public static void`や`String[] args`の部分は後述します.

クラスの説明聞いてないんだけど!?

クラスは設計書のようなものだと思ってください.

変数って?

変数はデータの入れ物のことです.

あなたは神様で `Human`クラスを作るとしましょう.

人間は, 名前があり, 年齢があり, 体重や身長などのデータがあります.

それらを格納するための入れ物が変数です.

以下に, `Human`クラスの例を示します.

```
// 人間クラス
public class Human {
    String name;  // 変数
    int age;  // 変数
    double weight;  // 変数
    double height;  // 変数
}
```

`String`や`int`,`double`って何ですか?

データの型(タイプ)です.

`String`のついた変数は`String`のみ,

`int`のついた変数は`int`のみ,

`double`のついた変数は`double`のみ

を表現します.

とりあえず, ここでは上記3種類の型に焦点を当てます.

型名	何が入る?	例
int	整数	24
double	実数	3.1415
String	文字列	"Hello"

このような感じになっています.

例えば, CDとラベル付けされた収納BOXとマンガとラベル付けされた収納BOXがあるとします.

CDとラベル付けされた収納BOXにはCDを入れ, マンガとラベル付けされた収納BOXにはマンガを入れますよね.

そういうことに似ています.

メソッドって?

メソッドは動作です.

神様であるあなたは人間に名前や年齢などのデータを持たせました.

しかし, データがあるだけでは何もできませんよね.

そこであなたは人間に, (食べ物を)食べる, 寝る, 遊ぶことを覚えさせます.

```
// 人間クラス
public class Human {
    String name;  // 変数
    int age;  // 変数
    double weight;  // 変数
    double height;  // 変数

    // eatメソッド
    public void eat() {
        System.out.println("eating now!");
    }

    // sleepメソッド
    public void sleep() {
        System.out.println("sleeping now!");
    }

    // playメソッド
    public void play() {
        System.out.println("playing now!");
    }
}
```

これでひとまず人間の設計図が完成しました.

人間を作る

さて, 神様であるあなたは人間の設計図を作りました.

ですが, 設計図を作り終えて満足してはいけません.

あなたはまだ設計図を書き上げただけであり, 人間を作っていないのです.

では, どうやって作るのか?

ここからは少し用語が多くなったり, 複雑になります.

コードから先に見ていきましょう.

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human();
    }
}
```

これで人間を作ることができました. 変数名は何でもいいのですが, ここでは`human`としています.

`Human human = new Human();`ってなにしてるの?

これは, `Human`型の変数`human`を定義しています.

また, `new Human();`では`Human`クラスを実体化しています.

まって, まってちょうだい!! `=`ってなによ!?

いいところに気付きましたね.

これは変数に値(実体)を代入しています.

先ほど変数はデータを入れるための箱のようなモノと説明しました.

その箱にデータ(値)を入れているのです.

実体化って?

実体化は**クラス(設計図)**から実体を作ることを言います.

実体のことを**インスタンス**といいます. 実体化のことを**インスタンス化**といいます.

クラスとは何が違うの?

クラスはあくまでも設計図です.

インスタンスは設計図から出来上がった実物です.

混同しないように気を付けてくださいね.

変数を使う

さっきのMainクラスではhumanという人間を作っただけです.

ここでは, humanに名前や年齢などのデータを追加してあげましょう.

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human();
        human.name = "アダム";
        human.age = 23;
        human.weight = 57.7;
        human.height = 173.5;
    }
}
```

あなたはアダム(年齢23歳, 身長173.5kg, 体重57.7kg)を作ることに成功しました.

しかし, あなたは人間の実体を作ったあとに名前や年齢のデータを追加することはできるのでしょうか?

神様であるあなたでもあとから人間を作り変えることはできないですよね.

これは, 設計図に誤りがありました.

早速修正してきましょう.

修正前

```
// 人間クラス
public class Human {
    String name;  // 変数
    int age;  // 変数
    double weight;  // 変数
    double height;  // 変数

    public void eat() {
        System.out.println("eating now!");
    }

    public void sleep() {
        System.out.println("sleeping now!");
    }

    public void play() {
        System.out.println("playing now!");
    }
}
```

修正後

```
// 人間クラス
public class Human {
    private String name;  // 変数
    private int age;  // 変数
    private double weight;  // 変数
    private double height;  // 変数

    public void eat() {
        System.out.println("eating now!");
    }

    public void sleep() {
        System.out.println("sleeping now!");
    }

    public void play() {
        System.out.println("playing now!");
    }
}
```

変数の型名の前に`private`というのを追加しました.

ちょっとあんた!! `private`って何よ!?

アクセス修飾子と言います.

アクセス修飾子には3つありますが, 今回は2つだけ紹介します.

アクセス修飾子	概要
private	外部に公開しないためのもの(内部だけで参照するもの)
public	外部に公開するためのもの

アクセス修飾子は変数やメソッド, クラスの最初に書きます.

なので, 今までの`public class Main { ... }`の`public`や`public void eat() { ... }`の`public`は外部に公開するよ!!っていうことだったんです.

クラスで宣言した変数には基本的に`private`をつけます.

誰もが他人の名前や見た目を簡単に変えられることはできないですよね.

これを**カプセル化**といいます.

宣言? なにそれ?

`int a;`のようなものを**宣言**といいます.

このクラスではint型のaという変数名の変数を使うよ! ということです.

神様であるあなたは怒られる

人間クラスを修正したあなたはもう一度人間を作ります.

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human();
        human.name = "アダム";
        human.age = 23;
        human.weight = 57.7;
        human.height = 173.5;
    }
}
```

しかし, ここでエラーが起こるはずです.

あなたはコンパイラに怒られてしまいました.

もう一度`Human`クラスを修正してみましょう.

修正前

```
// 人間クラス
public class Human {
    private String name;  // 変数
    private int age;  // 変数
    private double weight;  // 変数
    private double height;  // 変数

    public void eat() {
        System.out.println("eating now!");
    }

    public void sleep() {
        System.out.println("sleeping now!");
    }

    public void play() {
        System.out.println("playing now!");
    }
}
```

修正後

```
// 人間クラス
public class Human {
    private String name;  // 変数
    private int age;  // 変数
    private double weight;  // 変数
    private double height;  // 変数
```

```
    // 追加
    public Human(String name, int age, double weight, double height) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }

    public void eat() {
        System.out.println("eating now!");
    }

    public void sleep() {
        System.out.println("sleeping now!");
    }

    public void play() {
        System.out.println("playing now!");
    }
}
```

追加した部分は数行ですが, 何やらいつもと違いますね.

クラス名と同じ名前のメソッドって何!?

これまたいいところに気付きました.

これは**コンストラクタ(初期化子)**と言って, インスタンスを生成するときにデータを一緒に渡すための特殊なものです.

人間を作るとしたら, 最初に名前や年齢など決めておいてから作るべきですよね.

待ちなさいよ!! `(String name, ... double height)`ってなによ!

これは**仮引数**もしくは**パラメータ**とよびます. メソッド内部で必要なデータを外部から渡すときに使います.

じゃあ `this.name = name;`みたいなのはなんなのよ!!

`this`は自分自身のインスタンスを表します.

つまり, `this`の部分には`Human human = new Human();`の`human`が入るわけです.

ちなみに, 仮引数と変数の名前が違うのであれば`this`を省略することが可能です.

```
    ...
    public Human(String myName, int myAge, double myWeight, double myHeight) {
        name = myName;
        age = myAge;
        weight = myWeight;
        height = myHeight;
```

```
    }
    ...
```

`name`や`age`などを**インスタンス変数**と呼びます.

改めて人間を作る

あなたは人間クラスを修正しました。

`Main`クラスも修正して, 改めて人間を作りましょう.

修正前

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human();
        human.name = "アダム";
        human.age = 23;
        human.weight = 57.7;
        human.height = 173.5;
    }
}
```

修正後

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human("アダム", 23, 57.7, 173.5);
    }
}
```

これであなたはアダムを作ることができました.

`new Human("アダム", 23, 57.7, 173.5);`ってなによ!!?

これは, コンストラクタを呼び出しています.

`Human`クラスのコンストラクタは`public Human(String name, int age, double weight, double height)`でしたね.

これを呼び出しています.

なんで仮引数に`"アダム"`や`23`を渡すの??

`"アダム"`や`23`のことを**実引数**とよぶ.

`Human human = new Human("アダム", 23, 57.7, 173.5);`は

```
    public Human(String name, int age, double weight, double height) {
        // this.name = name;
        human.name = "アダム"
        // this.age = age;
        human.age = 23;
        // this.weight = weight;
        human.weight = 57.7;
        // this.height = height;
        human.height = 173.5;
    }
```

といった処理が行なわれる.

これ自体は先ほどの失敗例と同じだが, 生成するときに値が代入される点が違う.

人間で遊んでみる

神様であるあなたは人間を作った.

あなたはきっと人間で遊びたくてうずうずしているはず.

さっそく遊んでみよう.

遊ばせてみたり, 食べ物を食べさせてみたり, 寝させてみたり.

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human("アダム", 23, 57.7, 173.5);
        human.play();  // playing now!と表示される
        human.eat();  // eating now!と表示される
        human.sleep();  // sleeping now!と表示される
    }
}
```

あなたはきっとこうおもう.

「せっかく名前とか考えたのに, 表示されないのはつまんな〜い」

人間クラスを改良してみよう.

食べたら太る, 遊んだら痩せる, 寝たら身長が伸びるようにしてみよう.

改良前

```
// 人間クラス
public class Human {
```

```
    private String name;  // 変数
    private int age;  // 変数
    private double weight;  // 変数
    private double height;  // 変数

    public Human(String name, int age, double weight, double height) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }

    public void eat() {
        System.out.println("eating now!");
    }

    public void sleep() {
        System.out.println("sleeping now!");
    }

    public void play() {
        System.out.println("playing now!");
    }
}
```

改良後

```
// 人間クラス
public class Human {
    private String name;  // 変数
    private int age;  // 変数
    private double weight;  // 変数
    private double height;  // 変数

    public Human(String name, int age, double weight, double height) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }

    public void eat() {
        System.out.println(name+" is eating now!");
        weight = weight + 0.5;
        System.out.println("weight: "+weight);
    }

    public void sleep() {
        System.out.println(name+" is sleeping now!");
        height = height + 0.1;
        System.out.println("height: "+height);
    }
```

```
    public void play() {
        System.out.println(name+" is playing now!");
        weight = weight - 0.2;
        System.out.println("weight: "+weight);
    }
}
```

これできっとあなたが満足する人間の設計図ができたはずだ.

四則演算

まだ四則演算について説明していなかったね. 忘れてたわけじゃないよ.

これは言葉で説明するよりも, 表を見てもらったほうが早いと思う.

記号	意味	使い方	省略形	使い方	説明
+	足し算	A+B	+=	A+=B	A=A+B
-	引き算	A-B	-=	A-=B	A=A-B
*	掛け算	A*B	*=	A*=B	A=A*B
/	割り算	A/B	/=	A/=B	A=A/B
%	mod	A%B	%=	A%=B	A=A%B

優先順位は数学と同じで, `()`を使うこともできる.

人間で遊んでみる part2

もう一度さっきの`Main`クラスを実行してみよう.

```
public class Main {
    public static void main(String[] args) {
        Human human = new Human("アダム", 23, 57.7, 173.5);
        human.play();
        // アダム is playing now!
        // weight: 57.5 と表示される
        human.eat();
        // アダム is eating now!
        // weight: 58.0 と表示される
        human.sleep();
        // アダム is sleeping now!
        // height: 173.6 と表示される
    }
}
```

これでどうだろう.

そういえばまだメソッドの書き方聞いてないよ?

そうですね. 説明し忘れていました.

メソッドは`(アクセス修飾子) (static) (戻り値の型) メソッド名(仮引数) { 処理 };`といった構成でできています.

`static`や**戻り値**など新しい単語が出てきました.

実は`static`は一度は目にしているはずです. そうです.

`main`メソッドには`static`がついていました.

`static`とは?

`static`がついているメソッドや変数はインスタンス化をしなくても使うことができます.

`static`がついている変数を**クラス変数**, `static`がついているメソッドを**クラスメソッド**といいます.

ここでは, これだけ覚えていれば十分です.

戻り値とは?

戻り値は関数で処理した内容を渡せるようにするためのものです.

例えば, 足し算をするメソッド`add(int a, int b){ a+b; }`を定義しても, これでは足し算した結果をどうすることもできませんよね.

そこで**戻り値**を用意してあげることでメソッド内で処理したデータを外部に受け渡すことができます.

`void`は**戻り値**がないということを意味する特別なものです.

具体的なメソッドの書き方を教えてよ!

そうですね.

ここでは, 足し算をする`Adder`クラスについて考えてみましょう.

```
public class Adder {
    public int add(int a, int b) {
        return a + b;
    }
}
```

こんな感じになると思います.

次に足し算を行なう`AddMain`クラスを用意します.

```
public class AddMain {
    public static void main(String[] args) {
        int result;
```

```
        Adder adder = new Adder();
        result = adder.add(10, 5);
        System.out.println("result = "+result);
    }
}
```

おそらく, `result = 15`と表示されたはずです.

これで関数の定義のしかたについてわかっていただけたかと思います.

また, `add`メソッドに`static`をつけると以下のように書くことができます.

```
public class Adder {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

```
public class AddMain {
    public static void main(String[] args) {
        int result;
        result = Adder.add(10, 5);
        System.out.println("result = "+result);
    }
}
```

`(アクセス修飾子) (static) (戻り値の型) メソッド名(仮引数) { 処理 };`を心の中で3回唱えましょう!

そうすることで, きっと関数を定義できるようになるはずです!

ちなみに`private`変数を外部で使いたいときはどうすればいいの?

`private`の変数を外部で使いたい場合は, `public`なメソッドを使いましょう.

例えば, `private String name;` があるとしたら, `public String getName() { return name; }`のようなメソッドを作りましょう.

こういうメソッドを**ゲッター**と呼びます.

`private`変数の値を外部から変えたい場合はどうすればいいの?

上記と同じように`public`なメソッドを使います.

例えば, `private String name;` があるとしたら, `public void setName(String newName) { name = newName; }`のようなメソッドを作りましょう.

こういうメソッドを**セッター**と呼びます.

Javaのまとめ

用語	意味
mainメソッド	プログラムの本体
クラス	設計図
インスタンス	実体
インスタンス化	実体化
変数	データの入れ物
型	データが何であるか
メソッド	データの加工や動作
戻り値	データを加工した結果を渡すためのもの
仮引数	実引数を受け取るための宣言
実引数	メソッドに実際に渡すデータ
コンストラクタ	インスタンス化するときに変数を初期化するもの
インスタンス変数	クラス内で宣言した変数
クラス変数	インスタンス化しなくても呼び出せる変数
クラスメソッド	インスタンス化しなくても呼び出せるメソッド
アクセス修飾子	外部に公開するか否かを決める
private	外部に公開しない
public	外部に公開する
カプセル化	情報を隠蔽する
セッター	カプセル化した変数の値を変えるためのメソッド
ゲッター	カプセル化した変数の値を取得するためのメソッド
void	特別な戻り値
宣言	変数の型と変数名のみを記したもの
定義	宣言 + 実際のデータを記したもの

そろそろオブジェクト指向プログラミング教えてよ!!

お待たせしました.

RPGといえば, 戦闘ですよね.

戦闘シーンを目標にプログラムを作っていきます.

プレイヤークラスの作成

最初に, プレイヤーのクラスを作っていきます.

```
public class Player {
    private String name;
    private int level;
    private int hitPoint;
    private int strength;
    private int defence;

    public Player(String name, int level) {
        this.name = name;
        this.level = level;
        this.hitPoint = 24 + 8 * level;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    public String getName() { return name; }
    public int getLevel() { return level; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void showStatus() {
        System.out.println("Name: "+getName());
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

こんな感じでしょうか.

敵クラスの作成

次に, 敵のクラスを作っていきます.

```
public class Enemy {
    private String name;
    private int level;
    private int hitPoint;
    private int strength;
    private int defence;

    public Enemy(String name, int level) {
        this.name = name;
        this.level = level;
        this.hitPoint = 24 + 4 * level;
        this.strength = 9 + 2 * level;
        this.defence = 6 + 1 * level;
    }
```

```
    public String getName() { return name; }
    public int getLevel() { return level; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void showStatus() {
        System.out.println("Name: "+getName());
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

`Player`クラスを少しいじったら完成ですね.

メインクラスの作成

RPGのメインクラスを作ります.

```
public class RPGMain {
    public static void main(String[] args) {
        Player player = new Player("アリス", 1);
        Enemy enemy = new Enemy("スライム", 1);
        player.show();
        enemy.show();
    }
}
```

良い感じですね.

共通項をまとめる

`Player`クラスと`Enemy`クラスはステータスの初期値が違うだけで他は一緒です.

どうにかできないでしょうか??

一度`Player`クラスと`Enemy`クラスの共通部分を抜き出して新しいクラスを作ってみます.

```
public class Role {
    private String name;
    private int level;
    private int hitPoint;
    private int strength;
    private int defence;

    public Role(String name, int level) {
        this.name = name;
```

```
        this.level = level;
        this.hitPoint = 32;
        this.strength = 12;
        this.defence = 8;
    }

    public String getName() { return name; }
    public int getLevel() { return level; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void showStatus() {
        System.out.println("Name: "+getName());
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

ステータスは固定にしました.

メインクラスも書き換えてみます.

```
public class RPGMain {
    public static void main(String[] args) {
        Role player = new Role("アリス", 1);
        Role enemy = new Role("スライム", 1);
        player.show();
        enemy.show();
    }
}
```

こちらはあまり変わりませんね.

しかし, これではステータスが違うキャラクターを生成することができません.

そこで, **継承**を利用します.

継承って?

継承は, あるクラスからそのクラスの特徴を持った別なクラスを作ることです. 文字通りですね.

is-a関係ともいいます.

例えば, 人間は動物(is-a)であるとか, そういう感じです. 犬や猫も動物なので, (is-a)が当てはまります.

動物クラスを**スーパークラス(親クラス)**,

人間や犬クラスを**サブクラス(子クラス)**

と言います.

プログラムでは`extends`を使います.

継承を利用する

プレイヤーも敵も役職(role)です.

- `Player` is a `Role`
- `Enemy` is a `Role`

ということになりそうです.

それでは, `Role`クラスと`Player`クラスと`Enemy`クラスを修正していきましょう.

```
public class Role {
    protected String name;
    protected int level;
    protected int hitPoint;
    protected int strength;
    protected int defence;

    public Role(String name, int level) {
        this.name = name;
        this.level = level;
    }

    public String getName() { return name; }
    public int getLevel() { return level; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void showStatus() {
        System.out.println("Name: "+getName());
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

```
public class Player extends Role {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }
}
```

```
public class Enemy extends Role {

    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 4 * level;
        this.strength = 9 + 2 * level;
        this.defence = 6 + 1 * level;
    }
}
```

え? `protected`ってなに?

3つめの**アクセス修飾子**です.

`private`は外部に公開しないためのもので, `public`は外部に公開するためのものでした.

`private`はサブクラスにも公開されません.

`protected`はサブクラスまでOK!, というアクセス修飾子です.

実行してみる

メインクラスを少し修正します.

```
public class RPGMain {
    public static void main(String[] args) {
        Role player = new Player("アリス", 1);
        Role enemy = new Enemy("スライム", 1);
        player.show();
        enemy.show();
    }
}
```

変数`player`の型が`Role`だけど大丈夫なの!?って思われたかもしれませんが, 大丈夫です.

継承を利用しているので, `Player`は`Role`でもあるのです.

実行結果はというと...

```
Name: アリス
HP : 32
STR: 15
DEF: 10
Name: スライム
HP : 28
STR: 11
DEF: 7
```

良い感じですね. コードもすっきりして見やすくなりました.

攻撃のメソッドを用意しよう

次は攻撃のメソッドを作ります.

`attack`メソッドは`Role`クラスではなく, `Player`クラスと`Enemy`クラスそれぞれに実装します.

理由は, `Role`クラスは必ず戦闘に参加するわけではないからです.

```
public class Player extends Role {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    public attack(Role target) {
        System.out.println(target.getName()+"を攻撃した!");
    }
}
```

```
public class Enemy extends Role {

    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 4 * level;
        this.strength = 9 + 2 * level;
        this.defence = 6 + 1 * level;
    }

    public attack(Role target) {
        System.out.println(target.getName()+"を攻撃した!");
    }
}
```

ちゃんと二つのクラスに`attack`メソッドを実装することができました.

しかし, これでは`attack`メソッドを実装し忘れたり, 他のメソッドの実装を忘れるかもしれません.

そこで, **インターフェース**を利用します

インターフェースって何!?

インターフェースは宣言のみを記したクラスのことです.

これを利用することで, 確実にメソッドが実装されていることが保証されます.

使い方は継承と似ていますが, クラスの定義に`interface`を使用し, 利用する側では`extends`ではなく`implements`を使用します.

インターフェースで`attack`メソッドを宣言しよう.

`Role`クラスのキャラクターは必ずしも戦闘に参加するとは限りません.

ここでは, 戦闘に参加するキャラ用のインターフェースを作ります.

```
public interface Battler {

    public void attack(Battler target);

}
```

これだけです.

`Player`クラスと`Enemy`クラスを修正しよう

こちらが修正した2つのクラスです.

```
public class Player extends Role implements Battler {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    @Override
    public void attack(Battler target) {
        System.out.println(target.getName()+"を攻撃した!");
    }
}
```

```
public class Enemy extends Role implements Battler{

    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 4 * level;
        this.strength = 9 + 2 * level;
        this.defence = 6 + 1 * level;
    }

    @Override
```

```
    public void attack(Battler target) {
        System.out.println(target.getName()+"を攻撃した!");
    }
}
```

`@Override`ってなんですか?!

`@Override`はメソッドの上書きを意味してます. インターフェースでは実装を持たないので, サブクラスで実装をします.

そういった場合に`@Override`が使われます.

実行してみよう

メインメソッドを少し修正して実行してみます.

```
public class RPGMain {
    public static void main(String[] args) {
        Battler player = new Player("アリス", 1);
        Battler enemy = new Enemy("スライム", 1);
        player.showStatus();  // ここでエラー
        enemy.showStatus();
        player.attack(enemy);
    }
}
```

たぶん, エラーが出たと思います.

`Battler`型だから`showStatus`メソッドなんて知らないぞ!?と言っています.

`Role`型でも`attack`メソッドなんて知らないぞ!?と言われると思います.

しかし, 解決法がないわけではありません!!

修正してみる

`Battler`インターフェースに`getRole`メソッドを追加

```
public interface Battler {

    public void attack(Battler target);
    public Role getRole();

}
```

`Player`クラスと`Enemy`クラスでそれぞれ実装

```
public class Player extends Role implements Battler {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    @Override
    public void attack(Battler target) {
        System.out.println(target.getName()+"を攻撃した!");
    }

    @Override
    public Role getRole() {
        return new Player(name, level);
    }
}
```

```
public class Enemy extends Role implements Battler{

    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 4 * level;
        this.strength = 9 + 2 * level;
        this.defence = 6 + 1 * level;
    }

    @Override
    public void attack(Battler target) {
        System.out.println(target.getName()+"を攻撃した!");
    }

    @Override
    public Role getRole() {
        return new Enemy(name, level);
    }
}
```

メインクラスの修正

```
public class RPGMain {
    public static void main(String[] args) {
        Battler player = new Player("アリス", 1);
        Battler enemy = new Enemy("スライム", 1);
        player.getRole().showStatus();
        enemy.getRole().showStatus();
```

```
            player.attack(enemy);
        }
    }
```

これでちゃんと実行できたと思います.

receiveメソッドを作ろう!

まずはBattlerインターフェースにreceiveメソッドを追加します.

```
public interface Battler {

    public void attack(Battler target);
    public void receive(int damage);
    public Role getRole();

}
```

PlayerクラスとEnemyクラスで実装します.

```
public class Player extends Role implements Battler {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    @Override
    public void attack(Battler target) {
        System.out.println(target.getName()+"を攻撃した!");
    }

    @Override
    public void receive(int damage) {
        System.out.println(damage+"ダメージ受けた!");
    }

    @Override
    public Role getRole() {
        return new Player(name, level);
    }
}
```

```
public class Enemy extends Role implements Battler{
```

```
    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 4 * level;
        this.strength = 9 + 2 * level;
        this.defence = 6 + 1 * level;
    }

    @Override
    public void attack(Battler target) {
        System.out.println(target.getName()+"を攻撃した!");
    }

    @Override
    public void receive(int damage) {
        System.out.println(damage+"ダメージ受けた!");
    }

    @Override
    public Role getRole() {
        return new Enemy(name, level);
    }
}
```

ひとまずこれで完成です.

ダメージ計算の処理を書こう!

`attack`メソッドと`receive`メソッドを実装しましたが、文章を表示するだけでした.

ダメージ計算の処理を書いていきましょう.

`attack`メソッドでは, `自身の攻撃力 * 2 + random([0, 自身のレベル*4])`をダメージとします.

`receive`メソッドでは, `attack`メソッドで計算したダメージから `自身の守備力 / 4` を引いたダメージを自分が受けるダメージとします.

`Role`クラスで`Random`を使えるように`import`します.

また, インスタンス変数`random`を宣言します.

コンストラクタで初期化します.

戦闘のことを考えて, ここで最大HPの変数を追加します. それに伴って, ゲッターも追加します.

さらに, HPを変化させるための`setHp`メソッドも追加します.

```
import java.util.Random;

public class Role {
    protected Random random;
    protected String name;
    protected int level;
```

```
    protected int maxHitPoint;
    protected int hitPoint;
    protected int strength;
    protected int defence;

    public Role(String name, int level) {
        random = new Random();
        this.name = name;
        this.level = level;
    }

    public String getName() { return name; }
    public int getLevel() { return level; }
    public int getMaxHp() { return maxHitPoint; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void setHp(int newHp) { hitPoint = newHp; }
    public void showStatus() {
        System.out.println("Name: "+getName());
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

```
public class Player extends Role implements Battler {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.maxHitPoint = this.hitPoint;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    @Override
    public void attack(Battler target) {
        int damage = strength * 2 + random.nextInt()%(level*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - defence / 4;
        setHp(getHp() - damage);
        System.out.println(damage+"ダメージ受けた!");
    }
    @Override
    public Role getRole() {
        return new Player(name, level);
```

```
    }
}
```

`Enemy`のステータスも調整しちゃいます.

```
public class Enemy extends Role implements Battler{

    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 36 + 4 * level;
        this.maxHitPoint = this.hitPoint;
        this.strength = 9 + 2 * level;
        this.defence = 8 + 1 * level;
    }

    @Override
    public void attack(Battler target) {
        int damage = strength * 2 + random.nextInt()%(level*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - defence / 4;
        setHp(getHp() - damage);
        System.out.println(damage+"ダメージ与えた!");
    }

    @Override
    public Role getRole() {
        return new Enemy(name, level);
    }
}
```

`attack`メソッドと`reveive`メソッドを調整してひとまず, 戦闘できるまで準備が整いました.

あとは戦闘用のプログラムを書くだけですね.

戦闘のルール

- HPが0以下になったら戦闘終了
- 必ずプレイヤーが先攻
- 攻撃か逃げるのみ

`import`ってなに??

クラスライブラリを使えるようにするためのおまじないです.

死亡判定を書いていこう!

`Battler`インターフェースに`isDead`メソッドを追加します.

```
public interface Battler {

    public void attack(Battler target);
    public void receive(int damage);
    public Role getRole();
    public boolean isDead();

}
```

実装していきます.

HPが0以下のとき`true`になる**論理式**を`return`します.

```
public class Player extends Role implements Battler {

    public Player(String name, int level) {
        super(name, level);
        this.hitPoint = 24 + 8 * level;
        this.maxHitPoint = this.hitPoint;
        this.strength = 12 + 3 * level;
        this.defence = 8 + 2 * level;
    }

    @Override
    public void attack(Battler target) {
        int damage = strength * 2 + random.nextInt()%(level*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - defence / 4;
        setHp(getHp() - damage);
        System.out.println(damage+"ダメージ受けた!");
    }
    @Override
    public Role getRole() {
        return new Player(name, level);
    }

    @Override
    public boolean isDead() {
        return hitPoint <= 0;
    }
}
```

```
public class Enemy extends Role implements Battler{

    public Enemy(String name, int level) {
        super(name, level);
        this.hitPoint = 36 + 4 * level;
        this.maxHitPoint = this.hitPoint;
        this.strength = 9 + 2 * level;
        this.defence = 8 + 1 * level;
    }

    @Override
    public void attack(Battler target) {
        int damage = strength * 2 + random.nextInt()%(level*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - defence / 4;
        setHp(getHp() - damage);
        System.out.println(damage+"ダメージ与えた!");
    }

    @Override
    public Role getRole() {
        return new Enemy(name, level);
    }

    @Override
    public boolean isDead() {
        return hitPoint <= 0;
    }
}
```

`boolean`? はじめてみたぞ!!

`boolean`は`false`か`true`の2値のどちらかしかありません.

戦闘プログラムを書いていこう!

先にプログラムを載せておきます. `Scanner`クラスはユーザーから入力を得るためのクラスです.

```
import java.util.Scanner;

public class Battle {

    public int battle(Battler player, Battler enemy) {
        String playerName = player.getRole().getName();
        String enemyName = enemy.getRole().getName();
        int action;
```

```
        Scanner scanner = new Scanner(System.in);

        System.out.println(enemyName+"があらわれた!");

        while (true) {
            System.out.println(playerName+"はどうする?");
            System.out.print("[1]こうげき [0]にげる > ");
            action = scanner.nextInt();
            if (action == 0) {
                System.out.println(playerName+"は逃げた!");
                return -2;
            }
            player.attack(enemy);
            if (enemy.isDead()) {
                System.out.println(enemyName+"をたおした!");
                return 1;
            }
            enemy.attack(player);
            if (player.isDead()) {
                System.out.println(playerName+"はたおされてしまった...");
                return -1;
            }
        }
        return 0;
    }
}
```

色々新しいものが出てきましたね.

`while (true) { ... }`ってなんなのさ!?

`while`文は`()`内の**条件式**が`true`の間ループ(繰り返し)処理を行ないます. この場合は`true`が指定されているので, 無限ループになります.

条件式の表を示しておきます.

条件式	使用例	意味
==	A==B	AとBが等しいかどうか?
!=	A!=B	AとBが等しくないかどうか?
&&	A&&B	AとBがともに真か?(論理積)
||	A||B	AとBのいずれかが真か?(論理和)
!	!A	Aがtrueであればfalse, falseであればtrue(否定)
>=	A>=B	AがB以上かどうか?
>	A>B	AがBよりも大きいか?
<=	A<=B	AがB以下かどうか?

条件式	使用例	意味
<	A< B	AがBより小さいか?

じゃあ`if(~){ ... }`はなんなのさ??

`if`文は条件分岐です.

`()`の式が成立するなら `{}`内の処理を行ないます.

メインクラスを修正

`Battle`クラスをメインクラスで呼びます.

```
public class RPGMain {
    public static void main(String[] args) {
        Battler player = new Player("アリス", 1);
        Battler enemy = new Enemy("スライム", 1);
        player.getRole().showStatus();
        enemy.getRole().showStatus();
        Battle battle = new Battle();
        battle.battle(player, enemy);
    }
}
```

これでRPGが完成しました!!

ステータスをいじれるようにしたい!!

なるほど, たしかに今のままではステータスが固定ですね.

ならばステータスを分離してしまいましょう.

`Role`クラスからステータスを取り出して, 汎用的に使えるように変数やメソッドを追加しました.

```
public class Status {

    private int level;
    private int maxHitPoint;
    private int hitPoint;
    private int diffHp;
    private int strength;
    private int diffStr;
    private int defence;
    private int diffDef;

    public Status(int level, int baseHp, int baseStr, int baseDef, int diffHp, int
diffStr, int diffDef) {
        this.diffHp = diffHp;
        this.diffStr = diffStr;
```

```
        this.diffDef = diffDef;
        this.level = level;
        hitPoint = baseHp + addHp(level);
        maxHitPoint = hitPoint;
        strength = baseStr + addStr(level);
        defence = baseDef + addDef(level);
    }

    public int addHp(int level) { return diffHp * level; }
    public int addStr(int level) { return diffStr * level; }
    public int addDef(int level) { return diffDef * level; }

    public int getLevel() { return level; }
    public int getMaxHp() { return maxHitPoint; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void setHp(int newHp) { hitPoint = newHp; }
    public void showStatus() {
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

```
import java.util.Random;

public class Role {
    protected Random random = new Random();
    protected String name;
    protected Status status;


    public Role(String name, Status status) {
        this.name = name;
        this.status = status;
    }

    public Status getStatus() { return status; }
    public String getName() { return name; }
}
```

だいぶすっきりしましたね.

ステータスが変わったことで, `Player`クラスや`Enemy`クラスも少し修正が必要になります.

```
public class Player extends Role implements Battler {

    public Player(String name, Status status) {
```

```
        super(name, status);
    }

    @Override
    public void attack(Battler target) {
        int damage = status.getStr() * 2 + random.nextInt()%
(status.getLevel()*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - status.getDef() / 4;
        status.setHp(status.getHp() - damage);
        System.out.println(damage+"ダメージ受けた!");
    }

    @Override
    public Role getRole() {
        return new Player(name, status);
    }
    @Override
    public boolean isDead() {
        return status.getHp() <= 0;
    }
}
```

```
public class Enemy extends Role implements Battler{

    public Enemy(String name, Status status) {
        super(name,  status);
    }

    @Override
    public void attack(Battler target) {
        int damage = status.getStr() * 2 + random.nextInt()%
(status.getLevel()*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - status.getDef() / 4;
        status.setHp(status.getHp() - damage);
        System.out.println(damage+"ダメージ与えた!");
    }

    @Override
    public Role getRole() {
        return new Enemy(name, status);
    }
```

```
    @Override
    public boolean isDead() {
        return status.getHp() <= 0;
    }
}
```

`Battle`クラスや`Battler`インターフェースは変更なしです.

メインクラスは少し修正が必要です.

`Player`クラスのコンストラクタの引数に`Status`のインスタンスを渡すようになりました.

また, `Enemy`クラスのコンストラクタも同様です.

```
public class RPGMain {
    public static void main(String[] args) {
        // Your code here!
        Battler player = new Player("アリス", new Status(1, 32, 8, 6, 4, 3, 2));
        Battler enemy = new Enemy("スライム", new Status(1, 36, 8, 6, 2, 2, 2));
        Battle battle = new Battle();
        battle.battle(player, enemy);
    }
}
```

このように, クラスから分離してインスタンスを引数にすることを**コンポジション**といいます.

ところで, `player`とか`enemy`の型を`Player`とか`Enemy`じゃなくて`Battler`にしているのはなんでですか?

これには理由があります.

インターフェースの役割は覚えていますか?

そうです, 変数やメソッドの実装を強制させることです.

`Battler`インターフェースは戦闘に必要なメソッドの実装を強制しています. そのため, `Player`のインスタンスであろうと, `Enemy`のインスタンスであろうと, `Battler`であれば戦闘に参加することができます.

例えば, 味方同士で戦闘したいとき, `battle`メソッドの引数が`battle(Player player, Enemy enemy)`だったらどうでしょうか?

新しく `battle(Player player1, Player player2)`メソッドを作らないといけませんよね.

これを**オーバーロード**といいます.

ですが, これは無駄ですよね.

わざわざメソッドを複数作らなくても, 引数を`Battler`にすればいいんですから.

この型であれば実体は何でもいいぜ! というのが **ポリモーフィズム** です.

オーバーロードってなに?

オーバーロードはメソッドの名前や戻り値の型が同じで, 引数だけが異なるメソッドを複数作ることです.

ポリモーフィズムって??!

多様性を意味してます.

例えば, 車は見た目は違っても動作は同じですよね?

もっと言えば, 乗り物は全部乗れて, 操縦できますよね??

アクセル, ブレーキとかは共通しているわけです.

プログラミングの世界では, 飛行機だろうと船だろうと車だろうとアクセルがあってブレーキがあれば何でもいいんやで!っていった感じです.

プログラムのまとめ

```
public class Status {

    private int level;
    private int maxHitPoint;
    private int hitPoint;
    private int diffHp;
    private int strength;
    private int diffStr;
    private int defence;
    private int diffDef;

    public Status(int level, int baseHp, int baseStr, int baseDef, int diffHp, int
diffStr, int diffDef) {
        this.diffHp = diffHp;
        this.diffStr = diffStr;
        this.diffDef = diffDef;
        this.level = level;
        hitPoint = baseHp + addHp(level);
        maxHitPoint = hitPoint;
        strength = baseStr + addStr(level);
        defence = baseDef + addDef(level);
    }

    public int addHp(int level) { return diffHp * level; }
    public int addStr(int level) { return diffStr * level; }
    public int addDef(int level) { return diffDef * level; }

    public int getLevel() { return level; }
    public int getMaxHp() { return maxHitPoint; }
    public int getHp() { return hitPoint; }
    public int getStr() { return strength; }
    public int getDef() { return defence; }
    public void setHp(int newHp) { hitPoint = newHp; }
```

```
    public void showStatus() {
        System.out.println("HP : "+getHp());
        System.out.println("STR: "+getStr());
        System.out.println("DEF: "+getDef());
    }
}
```

```
import java.util.Random;

public class Role {
    protected Random random = new Random();
    protected String name;
    protected Status status;


    public Role(String name, Status status) {
        this.name = name;
        this.status = status;
    }

    public Status getStatus() { return status; }
    public String getName() { return name; }
}
```

```
public interface Battler {

    public void attack(Battler target);
    public void receive(int damage);
    public Role getRole();
    public boolean isDead();
}
```

```
public class Player extends Role implements Battler {

    public Player(String name, Status status) {
        super(name, status);
    }

    @Override
    public void attack(Battler target) {
        int damage = status.getStr() * 2 + random.nextInt()%
(status.getLevel()*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
```

```
        damage = damage - status.getDef() / 4;
        status.setHp(status.getHp() - damage);
        System.out.println(damage+"ダメージ受けた!");
    }

    @Override
    public Role getRole() {
        return new Player(name, status);
    }
    @Override
    public boolean isDead() {
        return status.getHp() <= 0;
    }
}
```

```
public class Enemy extends Role implements Battler{

    public Enemy(String name, Status status) {
        super(name,  status);
    }

    @Override
    public void attack(Battler target) {
        int damage = status.getStr() * 2 + random.nextInt()%
(status.getLevel()*4)+1;
        target.receive(damage);
    }

    @Override
    public void receive(int damage) {
        damage = damage - status.getDef() / 4;
        status.setHp(status.getHp() - damage);
        System.out.println(damage+"ダメージ与えた!");
    }

    @Override
    public Role getRole() {
        return new Enemy(name, status);
    }
    @Override
    public boolean isDead() {
        return status.getHp() <= 0;
    }
}
```

```
import java.util.Scanner;

public class Battle {
```

```
    public int battle(Battler player, Battler enemy) {
        String playerName = player.getRole().getName();
        String enemyName = enemy.getRole().getName();
        int action;
        Scanner scanner = new Scanner(System.in);
        System.out.println(enemyName+"があらわれた!");

        while (true) {
            System.out.println(playerName+"はどうする?");
            System.out.print("[1]こうげき [0]にげる > ");
            action = scanner.nextInt();
            if (action == 0) { return -2; }
            player.attack(enemy);
            if (enemy.isDead()) { return 1; }
            enemy.attack(player);
            if (player.isDead()) { return -1; }
        }
    }
}
```

```
public class RPGMain {
    public static void main(String[] args) {
        Battler player = new Player("アリス", new Status(1, 32, 8, 6, 4, 3, 2));
        Battler enemy = new Enemy("スライム", new Status(1, 36, 8, 6, 2, 2, 2));
        Battle battle = new Battle();
        battle.battle(player, enemy);
    }
}
```

おわりに

いかがでしたか?

この記事が参考になったらLGTMお願いします!

p.s.

深夜テンションで書いたのとソースコードが多くなりすぎてすごいわかりにくくなってしまったかもしれません。