

Java講習

後期第7回(最終回)

復習

- Charaに色々なものをまとめた
- 抽象化の本質は「メインルーチンの共通化」

```
public abstract class Chara {  
  
    Chara(String name, int hp, int mp, int atk, int def, int speed) { ...  
  
    void printStatus() { ...  
  
    /* "AはBにXXXのダメージを与えた"という表示を行う */  
    void printDamage(Chara chara, int damage) { ...  
  
    /* "AのHPは残りXXX"という表示を行う */  
    void printRestHP(Chara chara) { ...  
  
    abstract void attack(Chara chara);  
}
```

今日やること

1. 現在の実装の問題点
2. カプセル化
3. Getter, Setter



実は完了していた

実はこの講習で作ろうとしていた機能は全部作りました

RPGの戦闘画面をCLIで作ったシンプルなものでしたが、
オブジェクト指向の重要な概念は結構出てきました(まだあるのが恐ろしい)

```
[STATUS]
KNIGHT HP : 100 / 100, MP : 100 / 100, ATK : 100, DEF : 15, SPEED : 12
WIZARD HP : 85 / 85, MP : 35 / 35, ATK : 5, DEF : 15, SPEED : 20
TANK HP : 0 / 0, MP : 100 / 100, ATK : 20, DEF : 40, SPEED : 5
ENEMY1 HP : 400 / 400, MP : 30 / 30, ATK : 10, DEF : 0, SPEED : 15
ENEMY2 HP : 1200 / 1200, MP : 0 / 0, ATK : 50, DEF : 5, SPEED : 1
```

KNIGHTのターンです

[COMMAND]

- 1. Attack
 - 0. Quit
-

誰に攻撃しますか？

- 0 : KNIGHT
- 1 : WIZARD
- 2 : TANK
- 3 : ENEMY1
- 4 : ENEMY2

2

TANKに攻撃する

WIZARDはTANKに0のダメージを与えた
TANKのHPは残り0

では何するのか？

今の設計だと**致命的にまずい部分**があります

RPGゲームの動作に関わるというよりは, もっと違う角度からの問題です

それを知るために, 第1回の資料に遡りましょう



オブジェクト指向とは？(再掲)

オブジェクト指向は「将来的に苦労しないため」のアプローチの1つ

言い換えれば保守・拡張が容易になる

ゲーム作りたいたいなら絶対理解しないと無理！！(Unityでも同じ)



```
int mikata1Hp = 150;
int mikata1Mp = 15;
int mikata1Atk = 20;
int mikata1Def = 10;
int mikata1Speed = 5;

int mikata2Hp = 100;
int mikata2Mp = 20;
int mikata2Atk = 5;
int mikata2Def = 12;
int mikata2Speed = 10;

int mikata3Hp = 80;
int mikata3Mp = 80;
int mikata3Atk = 80;
int mikata3Def = 80;
int mikata3Speed = 80;

int enemy1Hp = 200;
int enemy1Mp = 200;
int enemy1Atk = 200;
int enemy1Def = 200;
int enemy1Speed = 200;

int enemy2Hp = 250;
int enemy2Mp = 250;
int enemy2Atk = 250;
int enemy2Def = 250;
int enemy2Speed = 250;
```

```
/* 味方1の攻撃処理 */
target = sc.nextInt(); // どの敵に攻撃するか
if (target == 1) {
    enemy1Hp = mikata1Atk - enemy1Def;
} else if (target == 2) {
    enemy2Hp = mikata1Atk - enemy2Def;
}

/* 味方2の攻撃処理 */
target = sc.nextInt(); // どの敵に攻撃するか
if (target == 1) {
    enemy1Hp = mikata2Atk - enemy1Def;
} else if (target == 2) {
    enemy2Hp = mikata2Atk - enemy2Def;
}

/* 味方3の攻撃処理 */
target = sc.nextInt(); // どの敵に攻撃するか
if (target == 1) {
    enemy1Hp = mikata3Atk - enemy1Def;
} else if (target == 2) {
    enemy2Hp = mikata3Atk - enemy2Def;
}

/* 敵1の攻撃処理 */
target = sc.nextInt(); // どの味方に攻撃するか
if (target == 1) {
    mikata1Hp = enemy1Atk - mikata1Def;
} else if (target == 2) {
    mikata2Hp = enemy1Atk - mikata2Def;
} else if (target == 3) {
    mikata3Hp = enemy1Atk - mikata3Def;
}

/* 敵2の攻撃処理 */
target = sc.nextInt(); // どの味方に攻撃するか
if (target == 1) {
    mikata1Hp = enemy2Atk - mikata1Def;
} else if (target == 2) {
    mikata2Hp = enemy2Atk - mikata2Def;
} else if (target == 3) {
    mikata3Hp = enemy2Atk - mikata3Def;
}
```

無理(再掲)

これだけで心が折れてきたのが分かるように, 無理です
なぜ無理なのか？



原因1：関連したデータが独立している(再掲)

下の画像のmikata1__という変数は全て味方1に関するデータ
...だけど変数がそれぞれ独立しているので, 関連したものだと分かりにくい！

それに味方1も2も同じ種類のデータを持ってる...

```
// 味方1の情報
int mikata1Hp = 150;
int mikata1Mp = 15;
int mikata1Atk = 20;
int mikata1Def = 10;
int mikata1Speed = 5;

// 味方2の情報
int mikata2Hp = 100;
int mikata2Mp = 20;
int mikata2Atk = 5;
int mikata2Def = 12;
int mikata2Speed = 10;
```

原因2：見えてる変数が多い(再掲)

ぱっと見て, 変数が非常に多い

これは**精神衛生上良くない！！！！！！！！**

しかも, これらのパラメータが**自由**にいじれてしまう！

atk, def, speedはどこでも使うわけではない

⇒これらを使うメソッドから**のみ**見えればいい！

```
int mikata1Hp = 150;  
int mikata2Hp = 100;  
int mikata3Hp = 80;  
  
int mikata1Mp = 150;  
int mikata2Mp = 100;  
int mikata3Mp = 80;  
  
int mikata1Atk = 150;  
int mikata2Atk = 100;  
int mikata3Atk = 80;  
  
int mikata1Def = 150;  
int mikata2Def = 100;  
int mikata3Def = 80;  
  
int mikata1Speed = 150;  
int mikata2Speed = 100;  
int mikata3Speed = 80;  
  
int enemy1Hp = 200;  
int enemy2Hp = 250;  
  
int enemy1Mp = 200;  
int enemy2Mp = 250;  
  
int enemy1Atk = 200;  
int enemy2Atk = 250;  
  
int enemy1Def = 200;  
int enemy2Def = 250;  
  
int enemy1Speed = 200;  
int enemy2Speed = 250;
```

原因3：似たロジックを沢山書く必要がある(再掲)

「変数名が違っただけで処理は同じ」という箇所が多い

メソッドにまとめればいいんだけど、パラメータの変数が多いので難しい。

```
/* 味方3の攻撃処理 */
target = sc.nextInt(); // どの敵に攻撃するか
if (target == 1) {
    enemy1Hp = mikata3Atk - enemy1Def;
} else if (target == 2) {
    enemy2Hp = mikata3Atk - enemy2Def;
}

/* 敵1の攻撃処理 */
target = sc.nextInt(); // どの味方に攻撃するか
if (target == 1) {
    mikata1Hp = enemy1Atk - mikata1Def;
} else if (target == 2) {
    mikata2Hp = enemy1Atk - mikata2Def;
} else if (target == 3) {
    mikata3Hp = enemy1Atk - mikata3Def;
}
```

オブジェクト指向プログラミングの発想(再掲)

結局, オブジェクト指向プログラミング(OOP)とは何なのかというと

データを「まとめて, 隠して, たくさん作る」

ことで、先ほどのような問題を解決することです

※これだけでなく, 他にも沢山の機能があります。



問題提起

コレ, 解決できてますか？

オブジェクト指向プログラミングの発想(再掲)

結局, オブジェクト指向プログラミング(OOP)とは何なのかというと

データを「まとめて, 隠して, たくさん作る」

ことで, 先ほどのような問題を解決することです

※これだけでなく, 他にも沢山の機能があります.

コレ

今までやったこと

まとめる ... クラスの定義(2), ライブラリ(4), 継承(5), 抽象化(6)

たくさん作る ... インスタンス生成(2), コンストラクタ(3)

隠す ... ???

コンストラクタの定義

インスタンスのデータの初期化を一気に行うために、**コンストラクタ**を使う
コンストラクタは**クラス内**で以下のように定義する

```
(public) <クラス名> (<引数>) {  
    処理...  
}
```

メソッドとの違いは**返り値の型が無いこと**

クラス宣言方法

```
(public) class <クラス名> {  
    ...  
}
```

- ・クラスの中には**データ(属性)**や**メソッド**を書く
- ・先頭のpublicはあってもなくてもいい(重要ではある)
- ・クラス名の先頭は**必ず大文字**

```
public class Mikata {  
    int age;  
    int sex;  
    int job;  
    int def;  
    int speed;  
  
    void printName() {  
        System.out.println("name : " + name);  
    }  
}
```

ライブラリとは？

そういった「汎用的な」処理を先人たちがまとめて1つにしたものを
ライブラリといいます(クラスやメソッドの集まり)

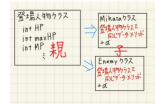
Javaはライブラリが豊富な言語として名を馳せています*

PythonやJavaScriptといったスクリプト言語や、Java互換のあるKotlinやScalaの登場によって影が薄くなりつつある

- プログラミングにおいて、抽象化を行う意義は「**メインルーチンの共通化**」にある
- 継承は「**サブルーチン(メソッドなど)の共通化**」
- 型によって様々な実装があるattack()を、メインからは**同じように**呼び出せるattack()を1つの窓口にまとめて扱えるようにした感じ
- このようなことを**ポリモーフィズム**といいます(意識しなくていいです)

継承

継承元のクラス(登場人物クラス)を**親クラス(スーパークラス)**と呼び
継承先のクラス(Mikata, Enemyクラス)を**子クラス(サブクラス)**と呼びます



子クラスは親クラスのデータとメソッドを持ちます

実際に書いてみよう

まずは、Charaクラスに

- attack()を持っているという**抽象的な情報**
- Charaクラスが「**不完全な(抽象的な)クラス**」であるという情報を書きましよう

abstractというキーワードを付与することで実現できます

```
abstract class Chara {  
    // 抽象的な、理論的な、概念的な、新しい、難解な、抽象派の、アブストラクトの
```

引用元: <https://ejje.weblio.jp/content/abstract>

「隠す」

クラスの定義や抽象化も「隠す」効果があるとは言えるのですが...
それっぽい概念はまだ出ていません

しかし, コードを見てみると
「隠蔽」が不十分であるのが分かるはずです

```
public class Game {
    static final int COMMAND_ATTACK    = 1;
    static final int COMMAND_QUIT     = 0;

    public static void main(String[] args) {
        clearTerminal();
        Scanner sc = new Scanner(System.in);

        Chara[] charas = new Chara[5];
        charas[0] = new Mikata("KNIGHT", 100, 100, 100, 15, 12);
        charas[1] = new Mikata("WIZARD", 85, 35, 5, 15, 20);
        charas[2] = new Mikata("TANK", 0, 100, 20, 40, 5);
        charas[3] = new Enemy("ENEMY1", 400, 30, 10, 0, 15);
        charas[4] = new Enemy("ENEMY2", 1200, 0, 50, 5, 1);

        int cmd = 0;
        int turn = 0;
        while (true) {
            printStatus(charas);
            printCurrentActor(charas[turn]);
            printCommand();
            cmd = sc.nextInt();

            if (cmd == COMMAND_QUIT) {
                break;
            } else if (cmd == COMMAND_ATTACK) {
                Chara target = selectTarget(charas);

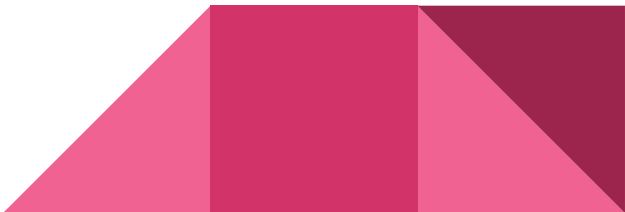
                System.out.println(target.name + "に攻撃する");
                charas[turn].attack(target);

                printStatus(charas);
            }
        }

        // 何か押されるまで待機
        sc.nextLine();
        ...
    }
}
```

Game.javaに定義されている変数

```
static final int COMMAND_ATTACK // 攻撃コマンドを表す定数(1)
static final int COMMAND_QUIT  // 終了コマンドを表す定数(0)
Scanner sc // 文字入力用
int cmd // どのコマンドを選択したかを表す
int turn // 現在だれが行動しているかを表す
Chara charas[0] // 中身にはMikataが入っている
Chara charas[1] // 中身にはMikataが入っている
Chara charas[2] // 中身にはEnemyが入っている
Chara charas[3] // 中身にはEnemyが入っている
Chara charas[4] // 中身にはEnemyが入っている
```



Game.javaに定義されている変数

よく考えてみると, Chara型の変数の中にあるMikataやEnemyには
name, hp, maxhp, mp, maxmp, atk, def, speed があります

それが5つあります

そしてそれらは自由に変更・参照ができてしまいます

「見えてる変数が少ない」とは言えないですね？



変数が見えていると良くない理由

そもそもなんで「変数の数」に厳しいのでしょうか？

機能としては実装出来ているし, 問題は無いように見えます

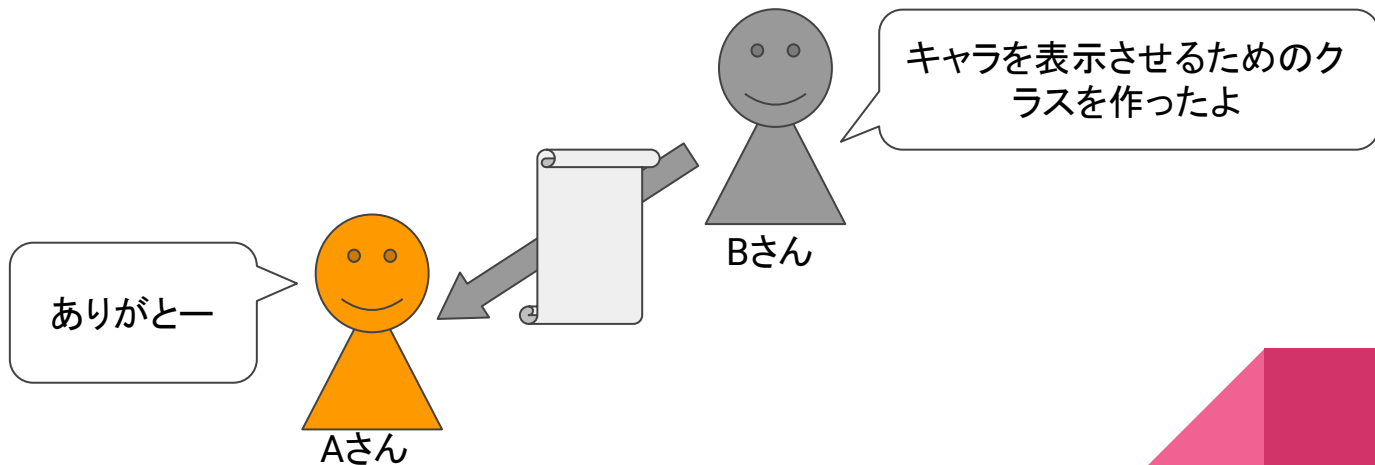
しかし, このプログラムがさらに大規模になったことを考えてみましょう



1. 他の人が作ったものを使うとき

何人かで共同開発しているのを想像してください

Bさんがクラス(ライブラリ)を作って, Aさんに提供したとします



1. 他の人が作ったものを使うとき

しかしソースコードは3000行もあり,
「そのクラスの中**だけ**で使う変数・メソッド」も混在していました

使い方を調べるためにコードを読んでいたAさんも,
さすがに嫌になってしまいました

何のメソッド使うのか
分からんし, 行数も多
くてだるい

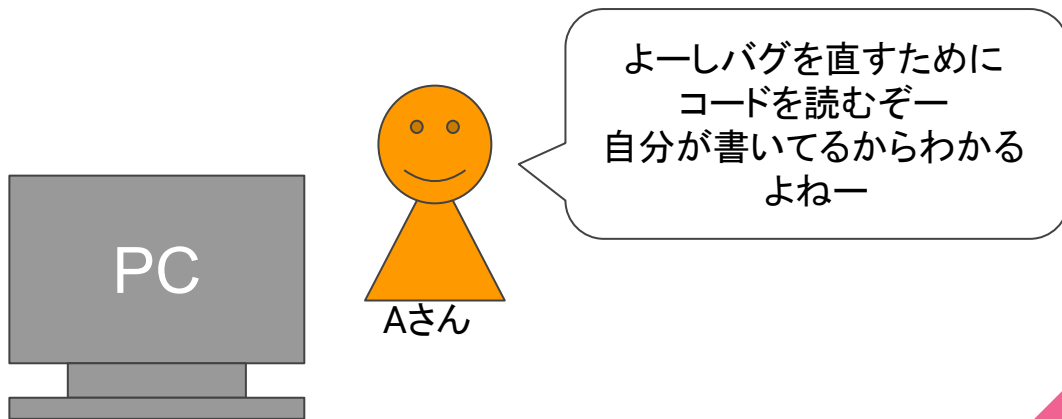


```
class CharaPrinter {  
  public void aaa() {}  
  public int bbb() {}  
  public double ccc() {}  
  public long ddd() {}  
}
```

2. 自分が昔作ったものを参照するとき

Aさんは昔(半年前くらい)にとあるアプリを作りました

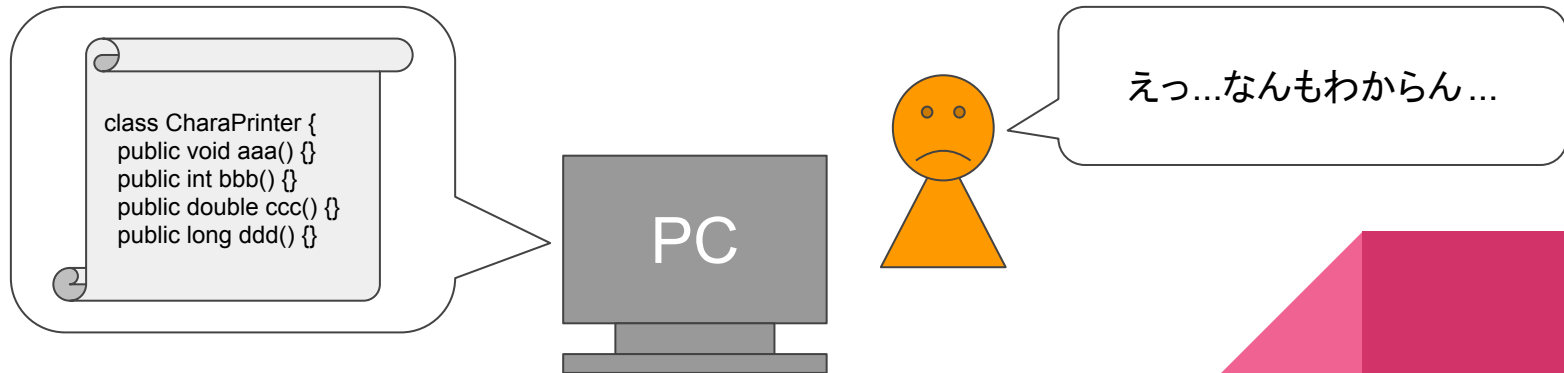
そして今, そのアプリのバグを修正するためにコードを見ようとしています



2. 自分が昔作ったものを参照するとき

しかし, どの変数・クラス・メソッドを使えばいいか,
どの箇所を直せば適切にバグを取り除けるのか

半年前のAさんなら分かっていた情報を**完全に忘れており**,
コードを把握するだけでものすごい時間がかかってしまいました

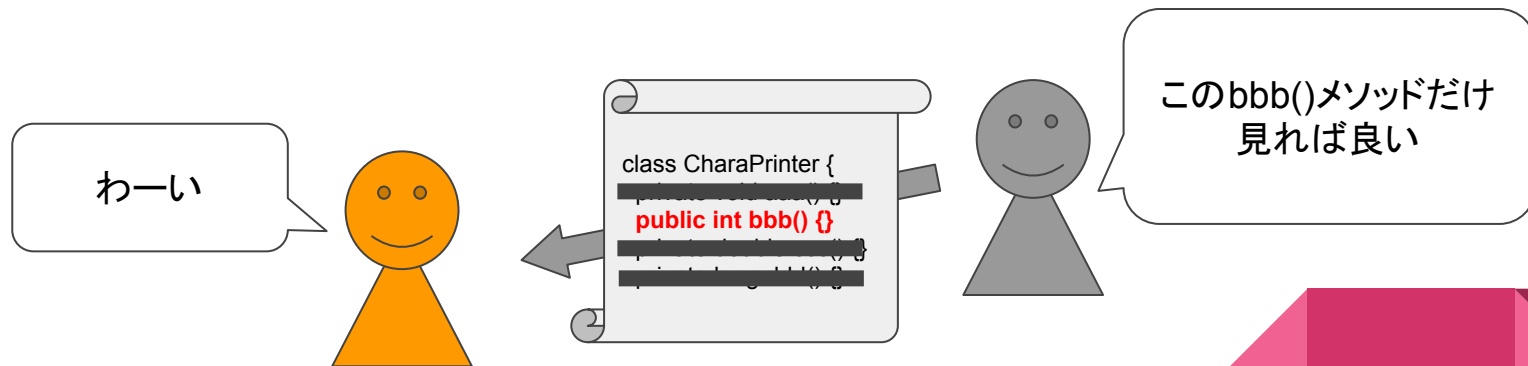


この問題への対策

1.の場合なら, Bさん(作成者)が

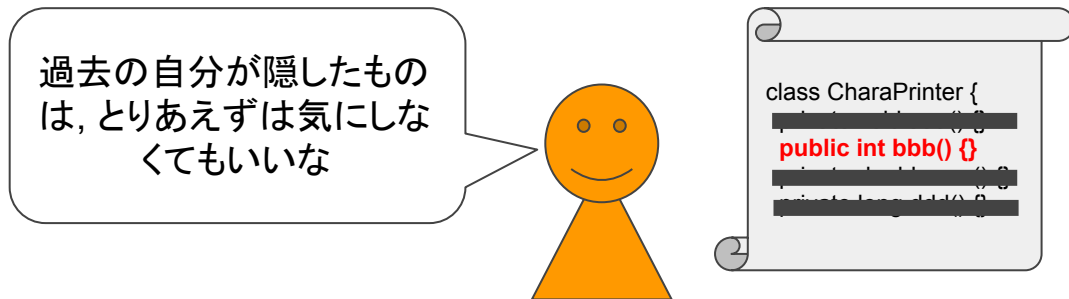
「**使い手(Aさん)にとって不要な変数・メソッドも見えている**」ことが問題

なので**隠してあげる**ことで, 使い手は**見るべきものが減って分かりやすくなる!**



この問題への対策

2.の場合なら, Aさん(作成者)が昔作った時に
「クラスの**内部**でしか使わない変数・メソッドも**公開**していた」のが問題
なので**隠してあげる**ことで, 将来の保守・改善が楽になる！



「隠す」アプローチ

このように「隠す」ことで将来の保守・アップデートが楽になります

これをオブジェクト指向では**カプセル化**といい、
継承, ポリモーフィズムと並んでOOPの三大要素といいます

(Cなどのオブジェクト指向ではない言語でも)意識すべき重要な考え方です！！



「隠し方」

このような「隠し方」はプログラミング言語によって違います

- pythonなら変数名, メソッド名の先頭に" _ "を付ける
- Goなら周りに見せるメソッド名の先頭を大文字にする
- Cならヘッダファイルとstaticを使って上手くやる

なので、「○○をしたらカプセル化できるよ！」ということより

「何故カプセル化が必要なのか」を理解するということが大事です



javaでの「隠し方」

javaでは変数・メソッドの定義に**アクセス修飾子**を付けることで隠せます

```
public int a = 1;  
private int b = 2;  
protected int c = 3;  
int d = 4;
```

上の四つの違いは, **変数・メソッドの見える範囲の違い**です



公開範囲の違い

- ・public ... **全ての**クラスからアクセスできる
- ・private ... 現在のクラスから**のみ**アクセスできる
- ・protected ... 同じパッケージのクラスとその子クラスからのみアクセスできる
- ・ ... 同じパッケージのクラスからのみアクセスできる

基本的には, protectedと"何も書かない"はpublicだと思っていいです
なのでpublicとprivateだけ意識すれば良いです



実際のコードを修正してみる

Charaクラスだと

name, hp, mp, atk, def, speed,
maxHp, maxMpは,

外部に公開しないので

privateにします

コンストラクタとクラスは

公開すべきなので**public**にします

printStatus()も**public**にします

```
public abstract class Chara {  
  
    private String name;  
  
    private int hp;  
    private int mp;  
    private int atk;  
    private int def;  
    private int speed;  
  
    private int maxHp;  
    private int maxMp;  
  
    public Chara(String name, int hp, int mp, int atk, int def, int speed, int maxHp, int maxMp) {  
        this.name = name;  
        this.hp = hp;  
        this.mp = mp;  
        this.atk = atk;  
        this.def = def;  
        this.speed = speed;  
        this.maxHp = maxHp;  
        this.maxMp = maxMp;  
    }  
  
    public void printStatus() { ...  
}
```

実際のコードを修正してみる

printDamage()やprintRestHP()は子クラスで使うのでpublic

attack()はMain.javaでも使ったのでpublic

```
public void printDamage(Chara chara, int damage) { ...  
  
public void printRestHP(Chara chara) { ...  
  
public abstract void attack(Chara chara);  
}
```

実際のコードを修正してみる

attackのダメージ計算部分をcalcDamage()にまとめてみました

この場合, attack()は親クラスでpublicと定義されているのでそれに揃えます
calcDamage()についてはEnemyの内部だけで使用されるのでprivate

```
public class Enemy extends Chara {  
    public Enemy(String name, int hp, int mp, int atk, int def, int speed) {  
        super(name, hp, mp, atk, def, speed);  
    }  
  
    private int calcDamage(Chara chara) {  
        return this.atk - chara.def;  
    }  
  
    public void attack(Chara chara) {  
        int damage = calcDamage(chara);  
    }  
}
```

デモ(都合により実行しない)

privateになっている変数・メソッドに外部からアクセスすると...

```
public class Main {  
    public static void main(String[] args) {  
        Mikata mikata = new Mikata();  
        mikata.calcDamage(mikata); // private method  
    }  
}
```

The method calcDamage(Chara) from the type Mikata is not visible Java(67108965)

問題の表示 クイック フィックス... (Ctrl+.)

The method calcDamage(Chara) from the type Mikata is **not visible**
(Mikata型のcalcDamageメソッドは**見えません**)

ここまでのまとめ

- ・データの隠蔽はチーム開発や保守のために行うべき
- ・publicが外部に公開, privateが外部に公開しない
- ・隠し方は言語によって違う, 方法よりも意義を理解しよう



休憩



隠したけど困った

データを隠したのは良いのですが, Charaに定義されたname, atkなどの変数はprivateなので子クラスやMain.javaから見えない(今のままだとエラー)

```
private int calcDamage(Chara chara) {  
    return (this.atk * 2) - chara.def;  
}
```

```
chara.hp = chara.hp - (charas[turn].attack(target));  
if (chara.hp <= 0) {  
    chara.hp = 0;  
}  
  
System.out.println(target.name + "に攻撃する");
```

privateメンバにアクセスしたい

nameやhpといったデータは外部から参照したいことがあるかもしれませんが
ただ, 公開するのもどうかと思います

そこでGetter, Setterというメソッドを間においてアクセスできるようにします



GetterとSetter

Getter(ゲッター)はインスタンスのデータを**取り出す**メソッドです

Setter(セッター)はインスタンスのデータを**設定する**メソッドです

```
public void setHp(int hp) {  
    this.hp = hp;  
}
```

```
public void setMp(int mp) {  
    this.mp = mp;  
}
```

```
public String getName() {  
    return this.name;  
}
```

```
public int getHp() {  
    return this.hp;  
}
```

作り方

名前は

`get○○○○()`

`set○○○○()`

のように決めます(どの言語でも大体そう)

また, (基本的には)外部に公開するので
publicにするのが適切です

```
public abstract class Chara {  
  
    private String name;  
  
    private int hp;  
    private int mp;  
    private int atk;  
    private int def;  
    private int speed;  
  
    private int maxHp;  
    private int maxMp;  
  
    public String getName() { ...  
    public int getHp() { ...  
    public int getMp() { ...  
    public int getAtk() { ...  
    public int getDef() { ...  
    public void setHp(int hp) { ...  
    public void setMp(int mp) { ...  
}
```

補足：命名規則

GetterとSetterの名前は、実は何でもいいのですが...

`get○○○○()`, `set○○○○()`と決めることで、一目でどんな関数かが分かります

例えば...

`get○○○()`, `set○○○()` ... ゲッターとセッター

`isAvailable` ... boolean型のメソッド

`SAMPLING_FREQ` ... 定数

のように、書き方でその名前が何を表しているのかが分かるようになります



意義

privateにしたものをGetter, Setterでアクセスできるようにする意義は
外部からの変更・参照を制御することです



1. 外部からの参照を制御する

Charaクラスには以下のGetter, Setterを作りました
これはCharaが持つデータ全てを網羅しているわけではありません
ですが, maxHpやspeedは外部から使っていないデータなので
あえてGetter, Setterを設けないことで適切にデータを隠ぺいできます

```
public String getName() { ...  
public int getHp() { ...  
public int getMp() { ...  
public int getAtk() { ...  
public int getDef() { ...  
public void setHp(int hp) { ...  
public void setMp(int mp) { ...
```

2. 外部からの読み書きを制御する

Getterだけ実装したメソッドを設ければ,
読み取り専用のデータを作ることができます(逆も然り)

以下の例では, atk, defのGetterを作らずに, 外部から読み取りだけできるようになっています

```
public int getAtk() { ...  
public int getDef() { ...
```

3. 外部からの変更を制御する

メソッドを介してデータにアクセスすることの利点は
間にエラー値をはじくプログラムなども書けるということです

```
public void setHp(int hp) {  
    if (hp < 0) {  
        hp = 0;  
    }  
    this.hp = hp;  
}
```

ただし, for, whileや他の重たい処理を呼ぶのはやめましょう
Getter, Setterは $O(1)$ であるべきです

まとめ

- ・大規模な開発になればなるほど**隠蔽の技術が重要**
- ・**最小特権の原理**を守ろう
(データの公開範囲は可能な限り**最小**にすべき)
- ・基本的にクラスの持つデータはprivateにし, Getter, Setterを用いてアクセス
- ・Getter, Setterを使うときでも,
外部から利用できる範囲や, 値の正しさを守れるように意識する



参考文献

「オブジェクト指向でなぜ作るのか」平澤 章

「アクセス修飾子 | Javaコード入門」

<https://java-code.jp/134>

「Principle of least privilege - Wikipedia」

https://en.wikipedia.org/wiki/Principle_of_least_privilege

「Controlling Access to Members of a Class - Oracle Java Documentation」

<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

