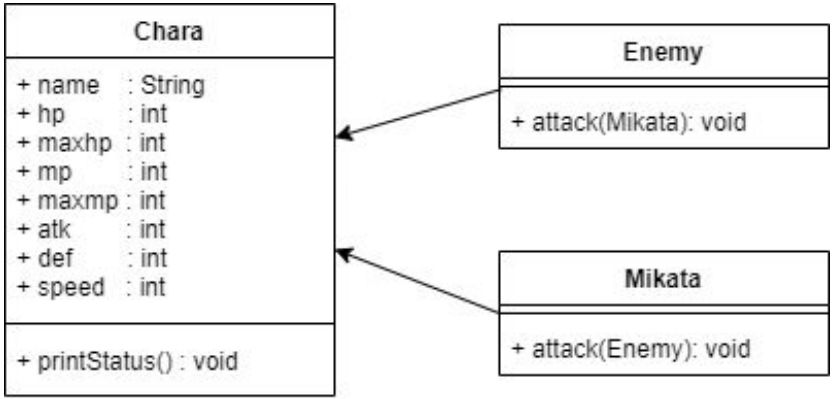
Java講習

後期第6回

前回やったこと

・MikataとEnemyに, HPとMPの最大値のパラメータを追加した

・継承を用いて共通化した



復習がてら今回の講習の準備

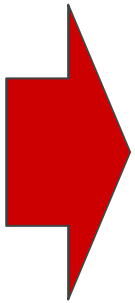
attack()メソッドがごちゃごちゃしてる...
ダメージ表示や, 残りHP表示をメソッドに分割してみましょう

```
void attack(Mikata mikata) {
    int damage = this.atk - mikata.def;

    System.out.println("--------------------");
    System.out.print(this.name + "は" + mikata.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");

    if (damage <= 0) {
        damage = 0;
    }

    mikata.hp = mikata.hp - damage;
    if (mikata.hp <= 0) {
        mikata.hp = 0;
    }
    System.out.println(mikata.name + "のHPは残り" + mikata.hp);
    System.out.println("--------------------");
}
```

復習がてら今回の講習の準備

というわけで,
printDamage()メソッドとprintRestEnemyHP()メソッドにわけてみました

```
void attack(Mikata mikata) {
    int damage = this.atk - mikata.def;

    System.out.println("--------------------");
    System.out.print(this.name + "は" + mikata.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");

    if (damage <= 0) {
        damage = 0;
    }

    mikata.hp = mikata.hp - damage;
    if (mikata.hp <= 0) {
        mikata.hp = 0;
    }
    System.out.println(mikata.name + "のHPは残り" + mikata.hp);
    System.out.println("--------------------");
}
```



```
void attack(Enemy enemy) {
    int damage = (this.atk * 2) - enemy.def - 1;

    if (damage <= 0) {
        damage = 0;
    }

    enemy.hp = enemy.hp - damage;
    if (enemy.hp <= 0) {
        enemy.hp = 0;
    }

    printDamage(enemy, damage);
    printRestEnemyHP(enemy);
}
```

復習がてら今回の講習の準備

Mikata.java側の実装はこんな感じ

```
/* "味方は敵にXXXのダメージを与えた"という表示を行う */
void printDamage(Enemy enemy, int damage) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + enemy.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "敵のHPは残りXXX"という表示を行う */
void printRestEnemyHP(Enemy enemy) {
    System.out.println(enemy.name + "のHPは残り" + enemy.hp);
    System.out.println("--------------------");
}
```

復習がてら今回の講習の準備

Enemy.java側の実装はこんな感じ

```
/* "EnemyはMikataにXXXのダメージを与えた"という表示を行う */
void printDamage(Mikata mikata, int damage) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + mikata.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "MikataのHPは残りXXX"という表示を行う */
void printRestMikataHP(Mikata mikata) {
    System.out.println(mikata.name + "のHPは残り" + mikata.hp);
    System.out.println("--------------------");
}
```

復習がてら今回の講習の準備

そして, Mikataクラスのattack()だけ処理を変えました

自分の攻撃力×2をダメージとして扱うようにしています. ちょいずるいですね

```
void attack(Enemy enemy) {
    int damage = (this.atk * 2) - enemy.def - 1;
```

↓Enemyクラスのattack()

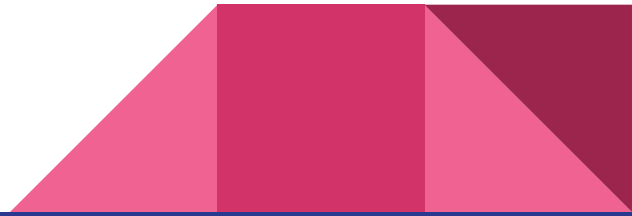
```
void attack(Mikata mikata) {
    int damage = this.atk - mikata.def;
```

今回やること

1. Charaクラスの変数
2. 抽象化とは
3. ターン制バトルを実装したい
4. attack()の抽象化を試みる

今回の内容は「ポリモーフィズム」と呼ばれる概念に関連した話です

かなり重ためなので, 頑張ってください



だから何だという話

前回作ったCharaクラスですが,

よく考えればこれも「クラス」です. Chara型の変数・インスタンスが作れます

```
Chara ch  = new Chara("Chara", 100, 10, 5, 7, 10);
```

だから何だという話

もちろん, MikataやEnemyにだけ実装したメソッドは使えません

Charaクラスに無いからね！

```
public class Chara {

    String name;

    int hp;
    int mp;
    int atk;
    int def;
    int speed;

    int maxHp;
    int maxMp;

    Chara(String name, int hp, int mp, int atk, int def, int speed) {…

    void printStatus() {…
}
```

Chara型の変数

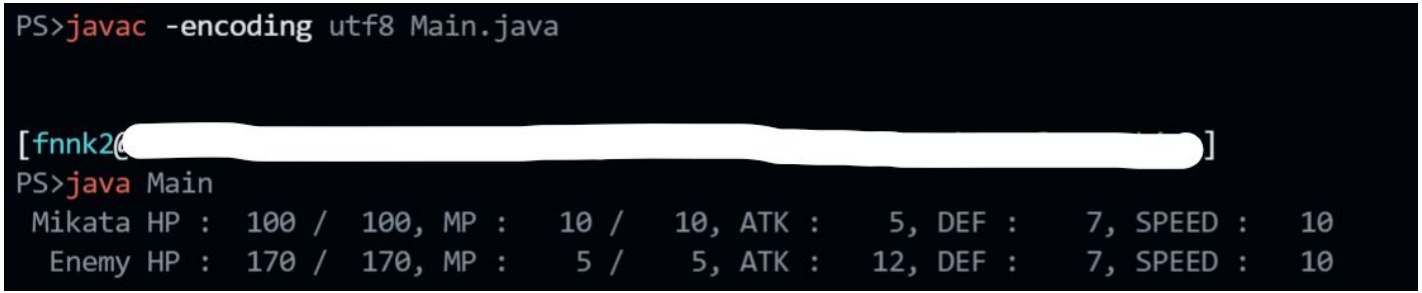
ここで, Chara型の変数にMikataとEnemyのインスタンスを入れてみましょう

エラー出そうな気がしますね

```
Chara ch1;
Chara ch2;

ch1 = new Mikata("Mikata", 100, 10,  5, 7, 10);
ch2 = new Enemy("Enemy",  170,  5, 12, 7, 10);

ch1.printStatus();
ch2.printStatus();
```

Chara型の変数

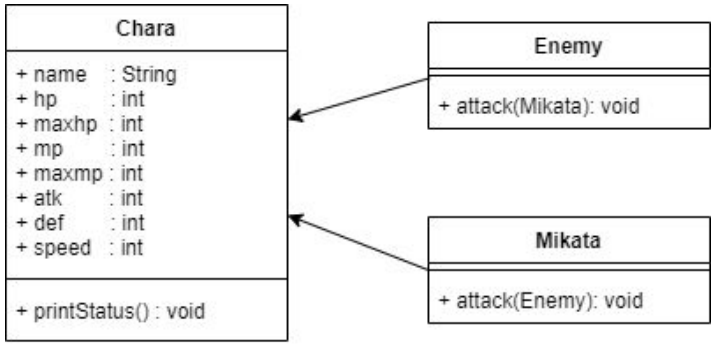
ところが, ちゃんと動きます. 何ででしょうか？



理由(厳密ではない)

雰囲気でいえば, EnemyもMikataも
Chara型の変数が持つデータとメソッドを持っているので, 似たような感じです

それゆえ, 親クラスの型(Chara)の変数に,
子クラスの型(Mikata, Enemy)のインスタンスを入れることができます



Chara型の変数

なので, このように扱うことができます

```
Chara ch1;
Chara ch2;

ch1 = new Mikata("Mikata", 100, 10,  5, 7, 10);
ch2 = new Enemy("Enemy",  170,  5, 12, 7, 10);

ch1.printStatus();
ch2.printStatus();
```

注意

中にMikataやEnemyのインスタンスが入っていても,

Chara型の変数として呼び出すことになるので, Charaに無いものは呼べません

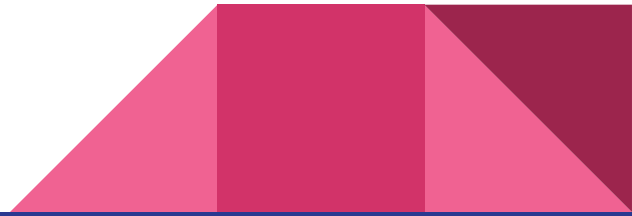
```
Chara ch = new Mikata("KNIGHT", 100, 100, 100, 15, 12);

ch.printStatus();  // Charaにあるメソッドは呼べる
ch.printDamage();  // Mikataのみに実装したメソッドは呼べない！
```

```
Main.java:19: エラー: シンボルを見つけられません
        ch.printDamage();  // Mikataのみに実装したメソッドは呼べない！
           ^
  シンボル:   メソッド printDamage()
  場所: タイプCharaの変数 ch
エラー1個
```

これができると嬉しい

これを聞いて「ふーん」と思われた方が多いと思います, 僕もです

しかし, 結構良いことあるので説明していきます



嬉しさ1 : Chara型の変数に「どっちも」入る

これは先ほどやりましたが,

Chara型の変数にMikataとEnemyどっちも入れられます

```
Chara ch1;
Chara ch2;

ch1 = new Mikata("Mikata", 100, 10,  5, 7, 10);
ch2 = new Enemy("Enemy",  170,  5, 12, 7, 10);

ch1.printStatus();
ch2.printStatus();
```

嬉しさ1 : Chara型の変数に「どっちも」入る

なので, MikataとEnemyを一括で扱えます

Charaの配列を使えばこんな感じでまとめることができます, すごい！

```
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);

for (int i = 0; i < charas.length; i++) {
    charas[i].printStatus();
}
```

```
PS>java Main
 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
```

今までは...

MikataとEnemyを別々で扱っていたので, ちょっと大変だった

```
static void printStatus(Mikata m1, Mikata m2, Mikata m3, Enemy e1, Enemy e2) {
    ColorPrint.greenPrintf("[PARTY]\n");
    m1.printStatus();
    m2.printStatus();
    m3.printStatus();
    ColorPrint.yellowPrintf("[ENEMY]\n");
    e1.printStatus();
    e2.printStatus();
    System.out.println("--------------------");
}
```

嬉しさ2 : さらに共通化できる

これは, 復習で作ったprintDamage()とprintRestXXXHP()です

MikataクラスとEnemyクラス別々に書いています, 何か無駄な気がしますよね？

```java
/* "味方は敵にXXXのダメージを与えた"という表示を行う */
void printDamage(Enemy enemy, int damage) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + enemy.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "敵のHPは残りXXX"という表示を行う */
void printRestEnemyHP(Enemy enemy) {
    System.out.println(enemy.name + "のHPは残り" + enemy.hp);
    System.out.println("--------------------");
}
```

```java
/* "EnemyはMikataにXXXのダメージを与えた"という表示を行う */
void printDamage(Mikata mikata, int damage) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + mikata.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "MikataのHPは残りXXX"という表示を行う */
void printRestMikataHP(Mikata mikata) {
    System.out.println(mikata.name + "のHPは残り" + mikata.hp);
    System.out.println("--------------------");
}
```

嬉しさ2 : さらに共通化できる

しかし, よく見てみると引数の型の違いだけです

しかもCharaで定義されているデータしか使っていません！

```
/* "味方は敵にXXXのダメージを与えた"という表示を行う */
void printDamage(Enemy enemy, int damage) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + enemy.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "敵のHPは残りXXX"という表示を行う */
void printRestEnemyHP(Enemy enemy) {
    System.out.println(enemy.name + "のHPは残り" + enemy.hp);
    System.out.println("--------------------");
}
```

```
/* "EnemyはMikataにXXXのダメージを与えた"という表示を行う */
void printDamage(Mikata mikata, int damage) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + mikata.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "MikataのHPは残りXXX"という表示を行う */
void printRestMikataHP(Mikata mikata) {
    System.out.println(mikata.name + "のHPは残り" + mikata.hp);
    System.out.println("--------------------");
}
```

嬉しさ2 : さらに共通化できる

なのでChara型の変数に変えましょう

Mikata.javaとEnemy.javaの両方とも, 全く同じ書き方で実装できます

```
/* "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) {
    System.out.println("--------------------");
    System.out.print(this.name + "は" + chara.name + "に");
    ColorPrint.redPrintf("%d", damage);
    System.out.println("のダメージを与えた");
}

/* "AのHPは残りXXX"という表示を行う */
void printRestHP(Chara chara) {
    System.out.println(chara.name + "のHPは残り" + chara.hp);
    System.out.println("--------------------");
}
```

嬉しさ2 : さらに共通化できる

なのでCharaにまとめられます！やった！！

```
public class Chara {

    String name;

    int hp;
    int mp;
    int atk;
    int def;
    int speed;

    int maxHp;
    int maxMp;

    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) {…
}
```

きれいになったMikata.java

attack()はMikataとEnemyで違うので共通化できませんが,

それ以外はきれいさっぱり無くなりました

```
public class Mikata extends Chara {
    Mikata(String name, int hp, int mp, int atk, int def, int speed) {
        super(name, hp, mp, atk, def, speed);
    }

    void attack(Enemy enemy) {
        int damage = (this.atk * 2) - enemy.def - 1;

        if (damage <= 0) {
            damage = 0;
        }

        enemy.hp = enemy.hp - damage;
        if (enemy.hp <= 0) {
            enemy.hp = 0;
        }

        printDamage(enemy, damage);
        printRestHP(enemy);
    }

}
```

ここまでのまとめ

注目すべき要素を抜き出し, 詳細を捨てることを**抽象化**といいます

今回の例だと, 引数の型をMikata, Enemyと限定するのではなく,
「Charaを継承したもの」という点に着目して共通化しようとすることです

数学なんかでもそうですが, 抽象化することで
- 問題がより一般的(普遍的)になる
- 厳密には違うが似ている概念(Mikata, Enemy)を同じものとして扱える

という良い効果があります

ただし, 抽象化が絶対ではないことに注意してください

休憩

ターン制バトルを実装したい

今は味方1人しか操作できていません

```
--------------------
KNIGHTはENEMY1に199のダメージを与えた
ENEMY1のHPは残り201
--------------------
[PARTY]
 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
[ENEMY]
 ENEMY1 HP :  201 /  400, MP :   30 /   30, ATK :   10, DEF :    0, SPEED :   15
 ENEMY2 HP : 1200 / 1200, MP :    0 /    0, ATK :   50, DEF :    5, SPEED :    1
--------------------
```

やっぱり全員操作できるようにしたいですよね

アイデア

順番にターンを回していく -> for文を使うことになる

ここで, 先ほど作ったChara型の配列を使えば, ターンを回す処理ができそう！

```
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);

for (int i = 0; i < charas.length; i++) {
    charas[i].printStatus();
}
```

というわけで実装してみた

Main.javaを修正しました, 色々変更点があるので説明していきます

printStatus()も配列渡すだけのシンプルなものになりました

```java
public static void main(String[] args) {
    clearTerminal();

    Chara[] charas = new Chara[5];

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

```java
static void printStatus(Chara[] charas) {
    System.out.println("[STATUS]");
    for (int i = 0; i < charas.length; i++) {
        charas[i].printStatus();
    }
    System.out.println("--------------------");
}
```

行動順の管理方法

変数turnを「現在誰が行動しているか」を表すのに使う

行動者を表示するメソッドとしてprintCurrentActor()を作りました

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

```
turn++;
// 行動順を先頭に戻す
if (turn >= charas.length) {
    turn = 0;
}
```

```
static void printCurrentActor(Chara chara) {
    System.out.println(chara.name + "のターンです");
}
```

攻撃処理の改善

攻撃コマンド("1")が押されたら, まず「誰を攻撃するか」を選択する

selectTarget()は「攻撃の対象」を選ぶ表示を出し,
対象となるCharaのインスタンス(へのポインタ)を返すメソッドです

```
} 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);
}
```

selectTarget()の実装

読めば分かると思います
(GitHubからダウンロードしてから見た方がよさそう)

```
static Chara selectTarget(Chara[] charas) {
    Scanner sc = new Scanner(System.in);

    for (int i = 0; i < charas.length; i++) {
        System.out.printf("%2d : %s\n", i, charas[i].name);
    }
    System.out.println();

    int target = sc.nextInt();
    while (target >= charas.length) {
        target = sc.nextInt();
    }

    return charas[target];
}
```

これで完璧！さあコンパイルしよう

さあコンパイルしましょう！

これで完璧！さあコンパイルしよう

さあコンパイルしましょう！

```
[redacted]>javac -encoding utf8 Game.java
Game.java:35: エラー: シンボルを見つけられません
                charas[turn].attack(target);
                            ^
  シンボル:   メソッド attack(Chara)
  場所: クラス Chara
エラー1個
```

あれ？

これ何でエラーなのでしょうか(ヒント：今日やった)

正解発表

正解はCharaクラスにattack()が実装されていないからでした！

先程言ったように, 中身がMikata, EnemyでもChara型の変数からは
Charaクラスに実装されたものしか呼び出せないので,
先程のコードはエラーになります

```
    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) {…
}
```

attack()は共通化できない！

attack()の実装はMikataとEnemyで違うので, Charaに共通化できない！

同じ実装にしてもいいけど, 味気ない...

```
void attack(Enemy enemy) {
    int damage = (this.atk * 2) - enemy.def - 1;
```

```
void attack(Mikata mikata) {
    int damage = this.atk - mikata.def;
```

抽象化の限界

では, Chara型の配列を使って攻撃処理を行うのは不可能なのでしょうか？

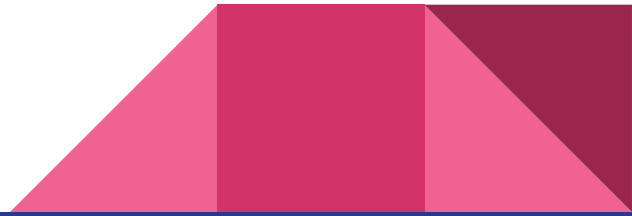
MikataとEnemyをそれぞれ作って管理するしかないのでしょうか？

```
} 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);
}
```

ここで一つ考えてみよう

諦めるのはまだ早い！

ということで, どうやったら共通化できるかを考えてみましょう



基本的な発想

Charaにattackがあればいいんです

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

何が悪いって

attack()がMikataとEnemyで違う実装であること

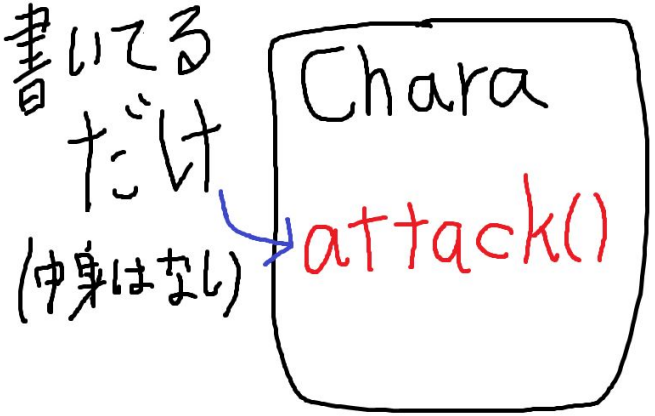
ことなんですよ

そのせいでCharaにattack()をまとめることができないです

基本的な発想

そこで,「Charaクラスはattack()メソッドを持つ！」と言ってしまいましょう

ここで重要なのは, Charaは「attack()がある」と言っているだけで
「どのように実装されているのか」という具体的な情報は無いということです

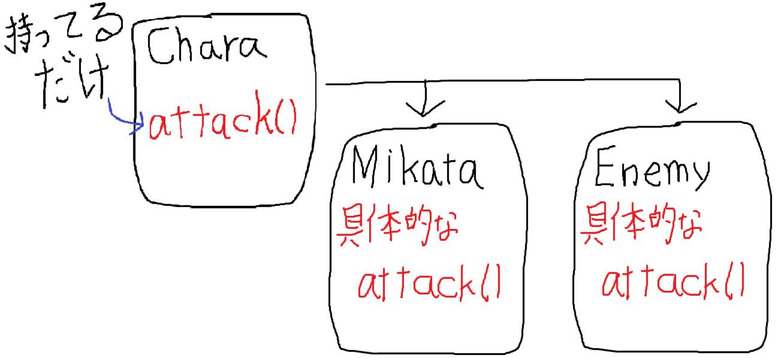


基本的な発想

では具体的なものはどこで書くのかというと, 継承先です
MikataとEnemyでそれぞれの実装を行います

- Charaにattack()が定義されていない
- MikataとEnemyで実装が違う

という問題はこれで実質的に解消されてます

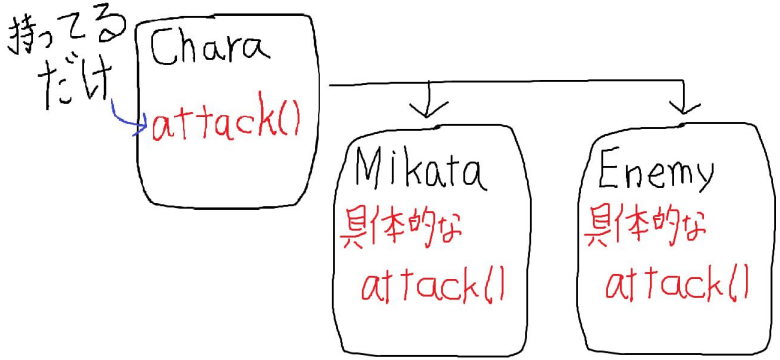


例

Chara型にはattack()があると"書いてある"ので

```
Chara aaa = new Mikata();
aaa.attack();
```

と書いても「Charaにはattack()メソッドが無い！」というエラーが無くなる



注意点

しかし, この考え方の前提として

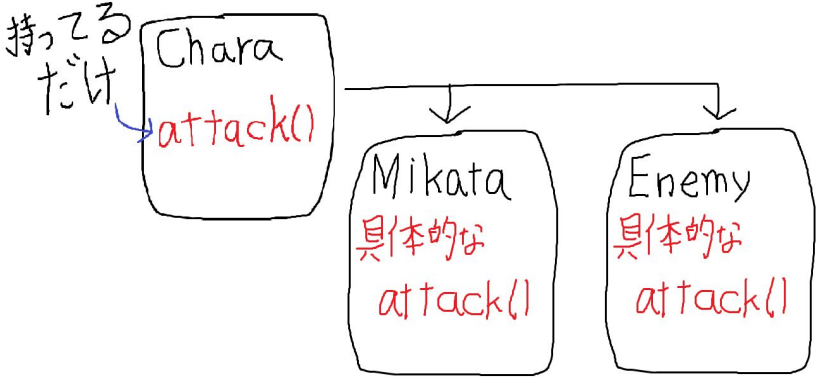
- Charaの子クラスには必ずattack()を実装しなければならない
- Charaは中身のないメソッドを持つ不完全なクラスである

というものがあります

「必ずattack()を実装しなければならない」

Charaは「attack()を持っている」と宣言してしまっているので
子クラスにattack()が具体的に書かれていなかったら実際に何するか分からない！

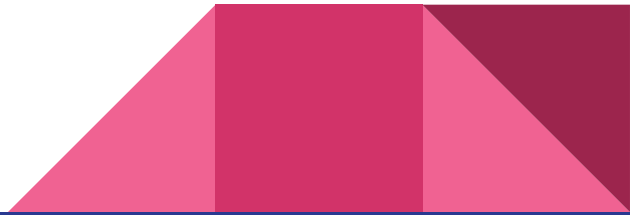
なので**絶対に**attack()を実装する必要があります



「Charaは不完全なクラスである」

Charaでのattackはいわばからっぽなメソッドなので,
Charaクラスのインスタンスを作って, そのattackを呼び出すことが出来ません

attackの具体的な実装が欠けているクラスを
他のクラスのようにインスタンス化して使うことはできません



まとめると...

- attack()を「Chara型の変数(中身はMikata, Enemy)から呼び出したい」

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

- Charaに「attack()がある」という抽象的な情報だけ記述する
- 子クラスにはattack()の実装を強制する

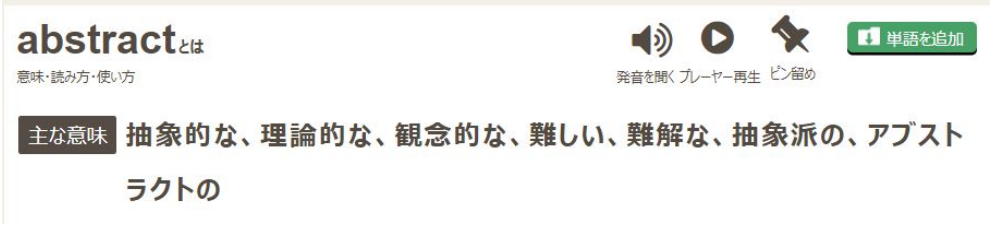
これによって, Chara型の変数としてまとめて扱ったものから
attack()を呼び出すことができる

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

を書きましょう

abstractというキーワードを付与することで実現できます



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

書き方

このように書きます

Charaは「attack()という抽象的なメソッドを持つ抽象的なクラス」
というのが分かりやすいですね

```
public abstract class Chara {

    /* "AのHPは残りXXX"という表示を行う */
    void printRestHP(Chara chara) {…

    abstract void attack(Chara chara);
```

補足

Charaクラスのインスタンスを作ろうとすると…

```
Chara chara;
chara = new Chara("KNIGHT", 100, 100, 100, 15, 12);
```

```
Main.java:5: エラー: Charaはabstractです。インスタンスを生成することはできません
        chara = new Chara("KNIGHT", 100, 100, 100, 15, 12);
                ^
エラー1個
```

子クラス側で実装しよう

あとは継承先で具体的な実装を書いてあげましょう
引数の型, 戻り値の型, 名前は「Charaにある抽象的な定義」と一致させます

下はMikataクラスの例ですが, Enemyも同様にします

```
void attack(Chara chara) {
    int damage = (this.atk * 2) - chara.def;

    if (damage <= 0) {
        damage = 0;
    }
```

以下略

補足

Charaクラスを継承しているのにattack()を実装していないクラスがあると...

```
public class Mikata extends Chara {
    Mikata(String name, int hp, int mp, int atk, int def, int speed) {
        super(name, hp, mp, atk, def, speed);
    }
}
```

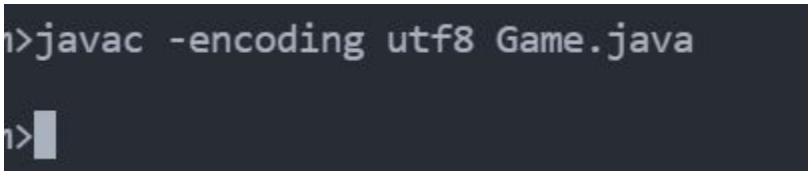
```
Chara chara;
chara = new Mikata("KNIGHT", 100, 100, 100, 15, 12);
```

```
.\Mikata.java:1: エラー: Mikataはabstractでなく、Chara内のabstractメソッドattack(Chara)をオーバーライドしません
public class Mikata extends Chara {
       ^
エラー1個
```

実際に動かしてみましょう

Game.java(メイン側)はいじらなくて大丈夫です

コンパイルも通りました！



デモ

```
--------------------
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
--------------------
```

まとめ

今までは型が違うものを扱っていたので, 処理をまとめるのが難しかった

```
Mikata m1 = new Mikata("KNIGHT", 100, 100, 100, 15, 12);
Mikata m2 = new Mikata("WIZARD",  85, 35,  5, 15, 20);
Mikata m3 = new Mikata("TANK",    0,  100, 20, 40, 5);
Enemy e1 = new Enemy("ENEMY1",  400, 30, 10, 0, 15);
Enemy e2 = new Enemy("ENEMY2", 1200,  0, 50, 5, 1);
```

```
m1.attack(e1);
printStatus(m1, m2, m3, e1, e2);

// m1がMikataに攻撃するならOOO
// m1がEnemyに攻撃するならXXX
// ...みたいな分岐処理をm1~e2まで全員分書く必要があった
```

まとめ

しかし,「機能は同じだが実装が少し違うもの」を
「実装は知らないが機能として持つ」と抽象的にしてCharaクラスにまとめたら

```
public abstract class Chara {
```

```
/* "AのHPは残りXXX"という表示を行う */
void printRestHP(Chara chara) {…

abstract void attack(Chara chara);
```

まとめ

メイン側で処理をまとめることができた！

```
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);

for (int i = 0; i < charas.length; i++) {
    charas[i].printStatus();
}
```

```
Chara target = selectTarget(charas);

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

printStatus(charas);
```

まとめ

- 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)

参考文献

「オブジェクト指向でなぜ作るのか」平澤 章
「デザインパターン紹介 - Abstract Classパターン」結城 浩
https://www.hyuki.com/dp/dpinfo.html