

# 第3章 クラス的设计

オブジェクト指向の基本

# この章の目的

---

クラスの正しい設計方法について学ぶ.

- ◆ 良いクラスの設計方法を知ることによって、プログラム全体の品質を向上させる.
- ◆ 不正値からクラスや変数を守る術を知ろう.

# 内容

## 内容とキーワード

### ◆ 良いクラスとは

- 自己防衛責務

### ◆ クラスを成熟させよう

- ガード節
- 不変 (`final`修飾子)
- プリミティブ型

### ◆ 設計パターン (おまけ)

- 完全コンストラクタ
- 値オブジェクト

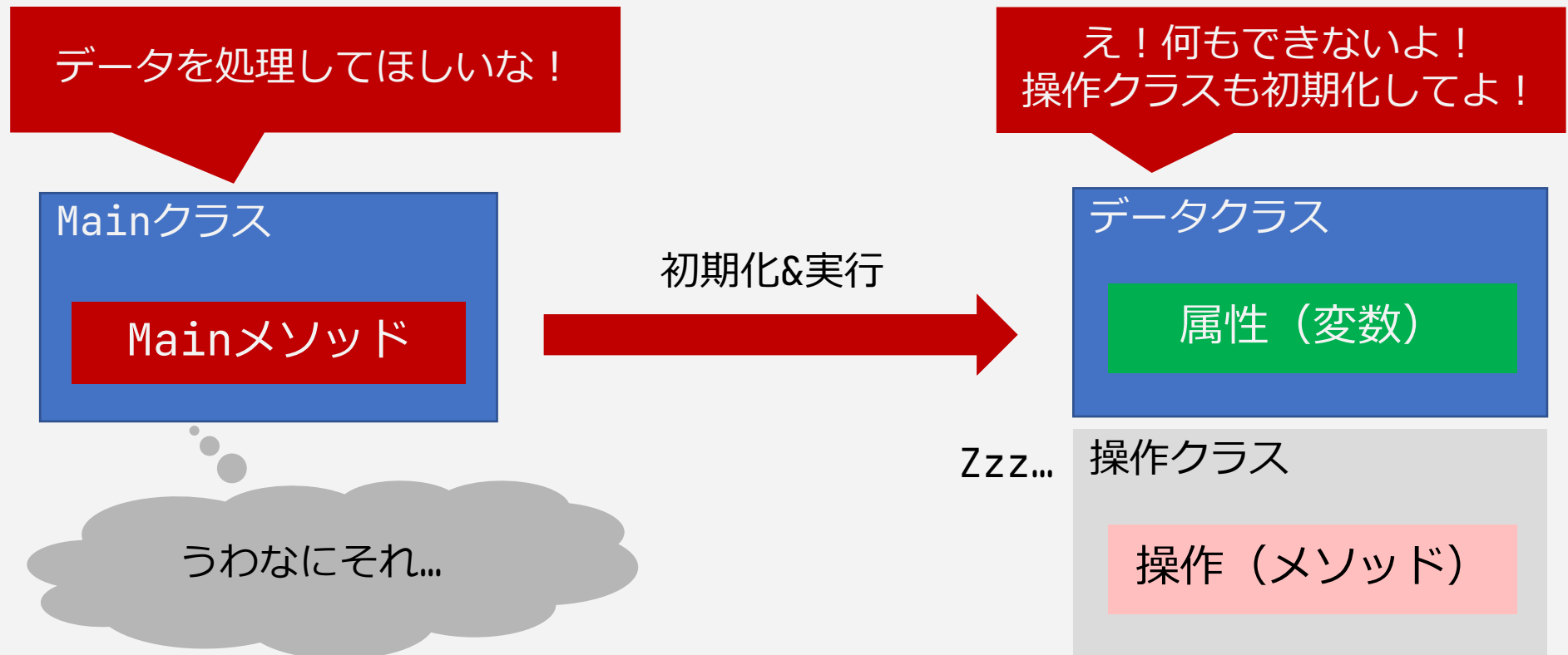
良いクラスって？

---

# 良いクラスってどんなの？

クラスが単体で正常に動作するような設計にしよう。クラスが単体で動作しないと起こる弊害は第1章で学習しましたね。

◆一番大切な考え方は、**クラスが単体で正常に動作するように設計すること**



# 単体で正常に動作するには？

インスタンス変数とそれらを不正状態から防御するメソッドから構成される。

## ◆良いクラスの構成要素は以下の通り

### ■インスタンス変数

- クラスが初期化されない限り空な変数

### ■インスタンス変数を不正状態から防御し，正常に動作するメソッド

- このようなメソッドを関連するインスタンス変数が所属しているクラスと同じクラスに所属させることで実装漏れを無くす

※目的によっては例外的に片方のみでも許される場合がある

# 自己防衛責務という考え方

自分の身は自分で守らせよう.

◆構成部品であるクラス一つ一つが品質的に完結していることで、ソフトウェア品質が向上する.

■詳細な初期化処理や下準備をしないと使い物にならないクラスはあまり使いたくない...

■自分の身は自分で守らせる、つまり**全てのクラスが自己防衛責務を備えることが必要**.

# クラスを成熟させよう

---



# クラスを成熟させよう(1/10)

典型的なデータクラスを成熟させる手順を見てみよう.

◆ゲームのキャラクターを表したクラスを例に,  
良いクラスへと成熟させていく手順を知る.

キャラクターのパラメータを持つ  
キャラクタークラス

```
class Character {  
    int hitPoint;  
    int magicPoint;  
}
```

※Pythonの場合

```
class Character:  
  
    def __init__(self):  
        self.hitPoint = 0  
        self.magicPoint = 0
```

# クラスを成熟させよう(2/10)

典型的なデータクラスを成熟させる手順を見てみよう.

## ◆手順1

■コンストラクタで確実に正常値を設定しよう.

```
class Character {  
    int hitPoint;  
    int magicPoint;  
  
    Character (int hitPoint, int magicPoint) {  
        this.name = name;  
        this.hitPoint = hitPoint;  
        this.magicPoint = magicPoint;  
    }  
}
```

しかしこのままだと...



不正値が混入！

```
Character cha = new Character(int -10, int 20);
```

# クラスを成熟させよう(3/10)

典型的なデータクラスを成熟させる手順を見てみよう.

## ◆手順1

■コンストラクタで確実に正常値を設定しよう.

```
class Character {  
    int hitPoint;  
    int magicPoint;  
  
    Character (int hitPoint, int magicPoint) {  
        if (hitPoint < 0 || magicPoint < 0) {  
            throw new IllegalArgumentException("hpまたはmpが不正");  
        }  
        this.hitPoint = hitPoint;  
        this.magicPoint = magicPoint;  
    }  
}
```

HPとMPは0以上しか受け付けなくなった  
※上限は無視

このような処理の対象外となる条件をメソッドの先頭に定義する方法を  
**ガード節**と呼ぶ

# クラスを成熟させよう(4/10)

典型的なデータクラスを成熟させる手順を見てみよう.

## ◆手順2

■計算ロジックをデータ保持側に寄せる.

```
class Character {  
    int hitPoint;  
    // 省略  
    public void hpDecreaseByAttack(int damage) {  
        int remainingHitPoint = this.hitPoint - damage;  
        if (remainingHitPoint < 0) {  
            // 戦闘に敗北したときの挙動  
        }  
        this.hitPoint = remainingHitPoint;  
    }  
}
```

HP関連の操作はもちろんHPをもつCharacterクラスで実装する！

# クラスを成熟させよう(5/10)

典型的なデータクラスを成熟させる手順を見てみよう。

## ◆手順3

可変だといつ再代入されるか分からないが...

■不変で思わぬ変更を防ぐ。

```
class Character {  
    final int hitPoint;  
    final int magicPoint;  
  
    Character (int hitPoint,int magicPoint) {  
        if (hitPoint < 0 || magicPoint < 0) {  
            throw new IllegalArgumentException("hpまたはmpが不正");  
        }  
        this.hitPoint = hitPoint;  
        this.magicPoint = magicPoint;  
    }  
}
```

final修飾子を付けることで不変にできる！  
(再代入できなくなる, 詳しくは第4章で)

え, じゃあHPとか変更できなくね？

# クラスを成熟させよう(6/10)

典型的なデータクラスを成熟させる手順を見てみよう.

## ◆手順4

■変更したい場合は新しいインスタンスを作成する.

```
class Character {  
    final int hitPoint;  
    // 省略  
    public Character hpDecreaseByAttack(int damage) {  
        int remainingHitPoint = this.hitPoint - damage;  
        if (remainingHitPoint < 0) {  
            // 戦闘に敗北したときの挙動  
        }  
        return new Character(remainingHitPoint, this.magicPoint);  
    }  
}
```

変更値を持った新しいインスタンスを返す！

# クラスを成熟させよう(7/10)

典型的なデータクラスを成熟させる手順を見てみよう。

## ◆手順5

■メソッド引数やローカル変数も不変にしよう。

```
class Character {  
    final int hitPoint;  
    // 省略  
    public Character hpDecreaseByAttack(final int damage) {  
        final int remainingHitPoint = this.hitPoint - damage;  
        if (remainingHitPoint < 0) {  
            // 戦闘に敗北したときの挙動  
        }  
        remainingHitPoint = 100;  
        return new Character(remainingHitPoint, this.magicPoint);  
    }  
}
```

コンパイルエラー！  
(予期せぬ変更から防御)

# クラスを成熟させよう(8/10)


典型的なデータクラスを成熟させる手順を見てみよう。

## ◆手順6

■値の渡し間違いを型で防止する

あれ、逆じゃね？

```
class Character {  
    // 省略  
    Character (int hitPoint,int magicPoint){  
        // 省略  
    }  
  
    public Character hpDecreaseByAttack(final int damage) {  
        final int remainingHitPoint = this.hitPoint - damage;  
        if (remainingHitPoint < 0) {  
            // 戦闘に敗北したときの挙動  
        }  
        return new Character(this.magicPoint, remainingHitPoint);  
    }  
}
```





# クラスを成熟させよう(8/10)

典型的なデータクラスを成熟させる手順を見てみよう.

## ◆手順6

■値の渡し間違いを**型**で防止する

int等の基本データ型を**プリミティブ型**と呼ぶ

これらは意図が異なる変数も同じように扱えてしまう

HPとMPは逆にしたらだめ！

うーんどっちもint型やし同じじゃん...

int:hitPoint

int:magicPoint

コンパイラ

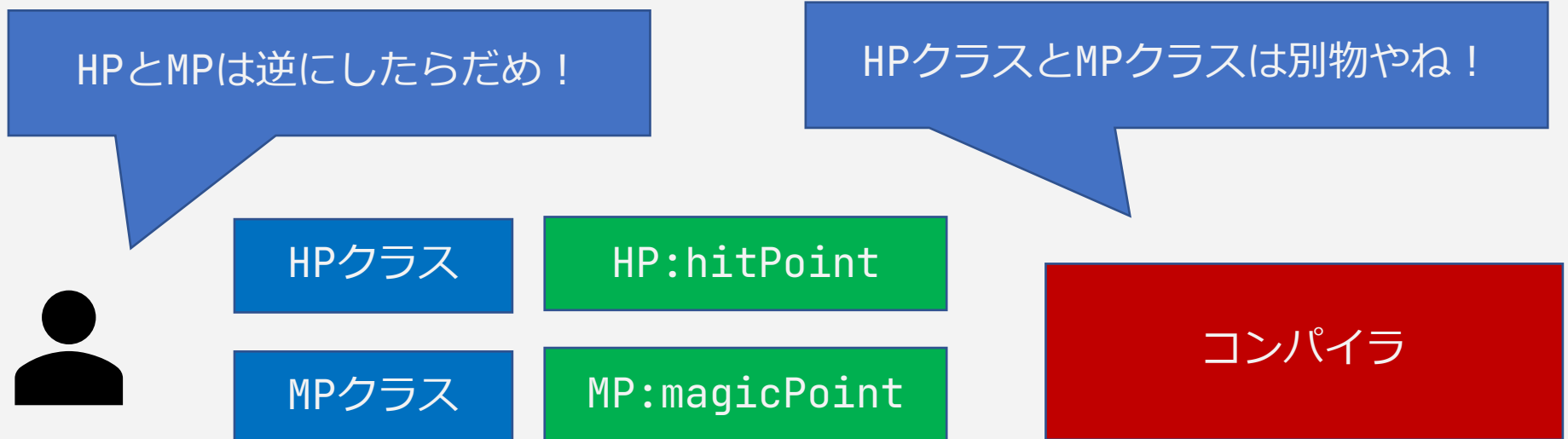


# クラスを成熟させよう(9/10)

典型的なデータクラスを成熟させる手順を見てみよう。

## ◆手順6

- 値の渡し間違いを型で防止する  
プリミティブ型に変わる独自の型（クラス）を用意しよう！



# クラスを成熟させよう(10/10)

典型的なデータクラスを成熟させる手順を見てみよう.

## ◆手順6

### ■値の渡し間違いを型で防止する

```
class Character {  
    HitPoint hitPoint;  
    MagicPoint magicPoint;
```

このような値をクラスとして表現する  
設計パターンを値オブジェクトと呼ぶ

```
    Character (HitPoint hitPoint, MagicPoint magicPoint) {  
        this.hitPoint = hitPoint.getHitPoint();  
        this.magicPoint = magicPoint.getMagicPoint();  
    }  
}
```

```
class HitPoint {  
    private final int hitPoint:  
    // コンストラクタ (ガード節あり)  
    // getメソッド  
}
```

```
class MagicPoint {  
    private final int magicPoint:  
    // コンストラクタ (ガード節あり)  
    // getメソッド  
}
```

# 結局どうなったか？

---

- ◆必要な操作がクラスに集まったので別のクラスにコードが書き散らされることはなくなり、可読性が向上した.
- ◆コンストラクタで必ず正しい値が設定されるようになり、不正値が混入しにくくなった.
- ◆不変を活用することで再代入による副作用から解放された.
- ◆引数を値オブジェクトにすることで値の渡し間違いがコンパイラで防止できるようになった.

# 設計パターン（おまけ）

プログラム構造を改善する設計手法を設計パターンと呼ぶ。

◆完全コンストラクタ

◆値オブジェクト

◆ストラテジ

◆ポリシー

◆ファーストクラスコレクション

◆スプラウトクラス

などなど（興味があれば聞いてね）

# 次の章に向けて

---

- ◆ 良いクラスは関連する属性と操作が集まっている
- ◆ 値オブジェクトなどの設計パターンを活用しよう
- ◆ 次の章では不変について学ぶ