第４章
不変の活用

安定動作を構築しよう

この章の目的

可変と不変を適切にあつかえるようになる.

◆可変（ミュータブル）

■変数等の値や状態をいつでも変更できる状態

◆不変（イミュータブル）

■値や状態を変更できないこと

プログラムは変更を最小限にする設計が重要

⇒近年のプログラミングスタイルで不変は標準的！

内容

内容とキーワード

◆再代入
◆可変がもたらす意図せぬ影響
◆副作用とは
◆デフォルトは不変
◆可変にしてもいいとき

可変がもたらす意図せぬ影響

再代入

第２章でも学習したが，同じ変数を何度も繰り返し使用すると変数がいつ変更されたかがわかりにくくなり，可読性が下がる.

◆注意していても間違って再代入してしまうかも…

⇒ final修飾子を使って強制的に再代入不可に！

なんとPythonにはこの概念がない！

```
final int hitPoint = 100;

htiPoint = 200; // コンパイルエラー！
```

```
※Pythonでは一応全て大文字で書いて定数とする決まりはある
HIT_POINT = 100

HIT_POINT = 200 // 普通に動作する

※一応定数を定義することはできるらしいが…
```

可変がもたらす意図せぬ影響

ゲームを例に，不変の重要性について考えてみる.

◆ケース１：可変インスタンスの使い回し

可変がもたらす意図せぬ影響

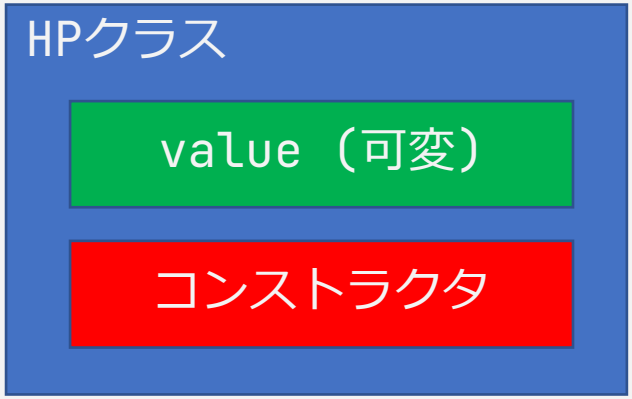
◆ケース１：可変インスタンスの使い回し

■HPを表現するクラス

```
class HitPoint {
 static final int MIN = 0;
 int value; // 可変

 HitPoint(int value) {
  if (value < MIN) {
    throw new IlligalArgumentException()
   }
    this.value = value;
 }
}
```

Hpの値オブジェクトです



可変がもたらす意図せぬ影響

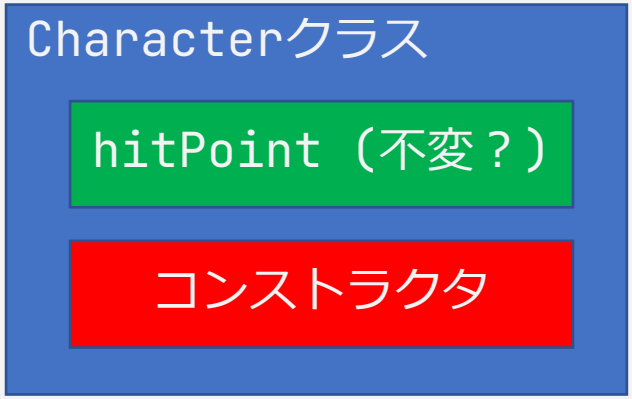
◆ケース１：可変インスタンスの使い回し

■キャラクターを表現するクラス

```
class Character {
 final HitPoint hitPoint;

 Character(HitPoint hitPoint) {
  this.hitPoint = hitPoint;
 }
}
```

キャラクタークラスです



可変がもたらす意図せぬ影響

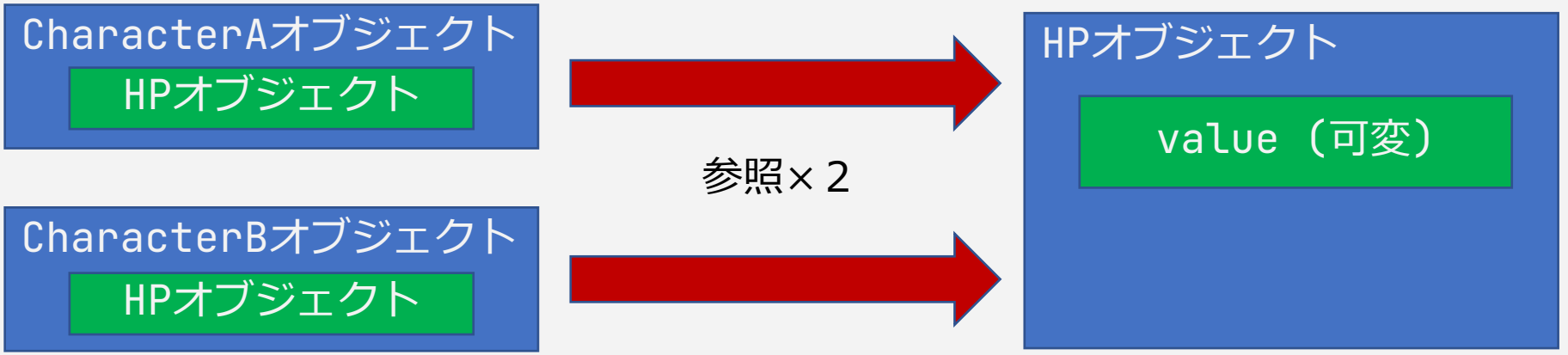
◆ケース１：可変インスタンスの使い回し

■さきほどのクラスを用いてインスタンスを生成すると…

```
HitPoint hitPoint = new HitPont(100);

Character characterA = new Caracter(hitPoint);
Character characterB = new Caracter(hitPoint);
```

HP同じだし使い回しちゃお！



可変がもたらす意図せぬ影響

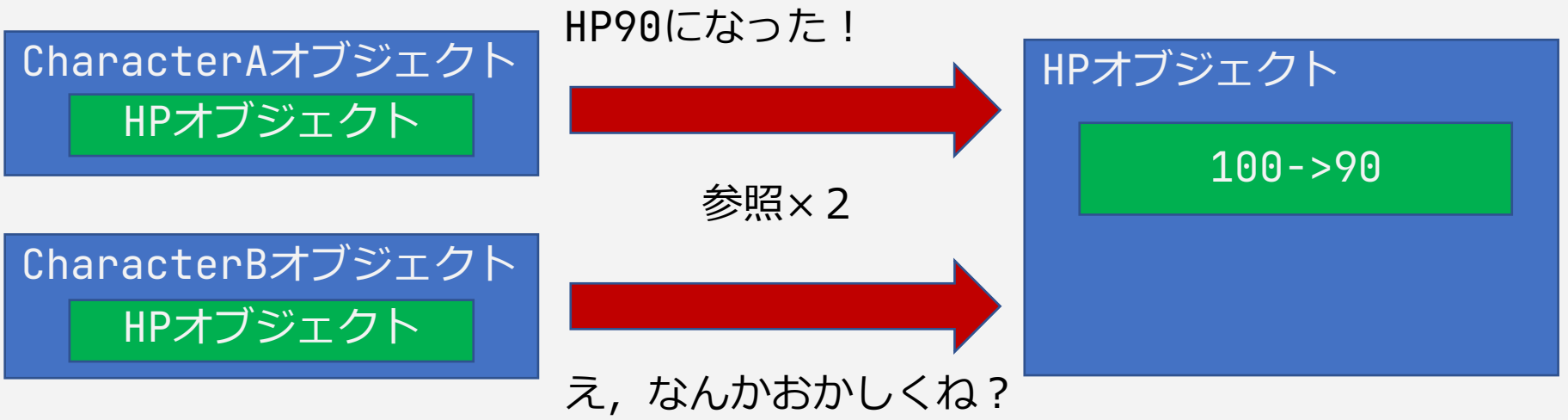
◆ケース１：可変インスタンスの使い回し

■あ，やっぱりキャラクターAだけHP90にしたい…

```
Character characterA = new Caracter(hitPoint);
Character characterB = new Caracter(hitPoint);

characterA.hitPoint.value = 90;
```

あ，キャラAのHP
やっぱ90で！



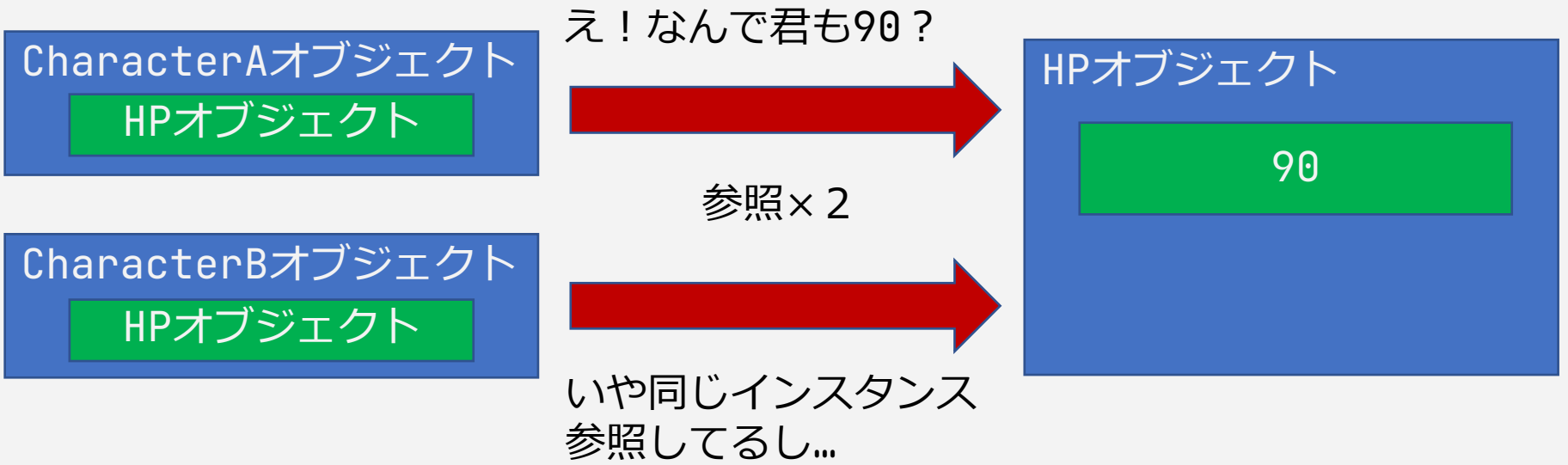
可変がもたらす意図せぬ影響

◆ケース１：可変インスタンスの使い回し

■なぜかキャラクターBもHP90に！

```
Character characterA = new Caracter(hitPoint);
Character characterB = new Caracter(hitPoint);
characterA.hitPoint.value = 90;
System.out.println(characterB.hitPoit.value);
```

90と出力される！

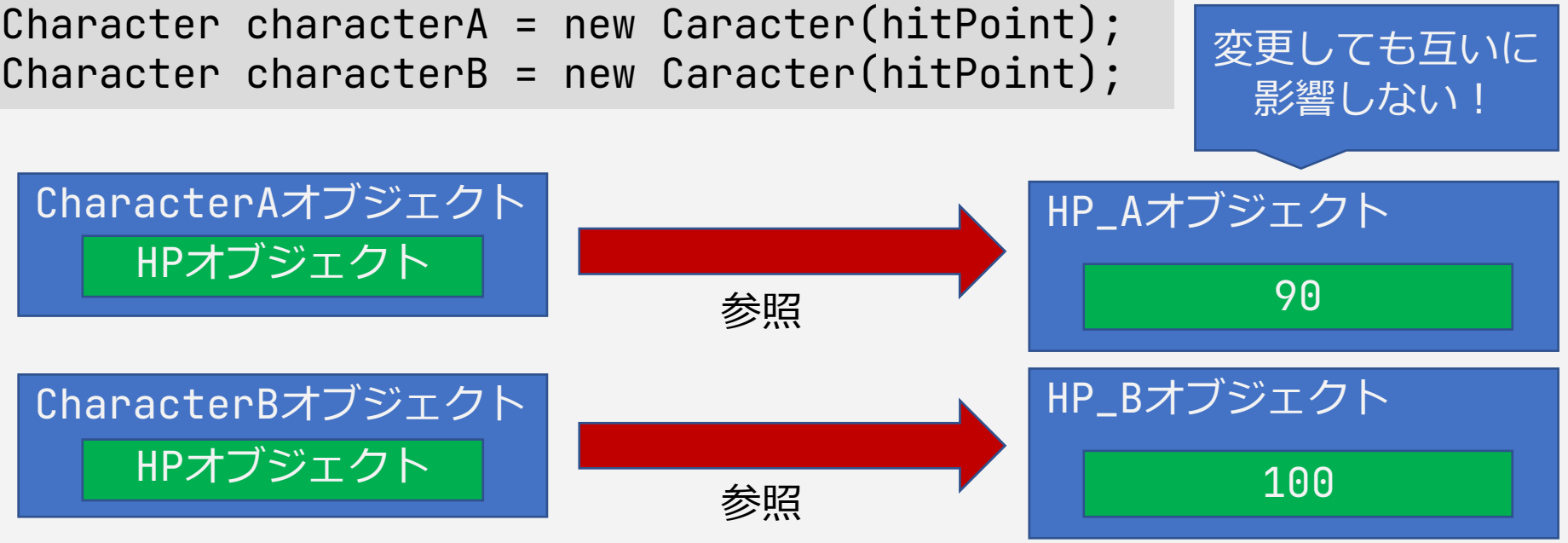


可変がもたらす意図せぬ影響

◆ケース１：可変インスタンスの使い回し

■(とりあえず)インスタンスの使い回しは避けよう！

```
HitPoint hitPointA = new HitPoint(90);
HitPoint hitPointB = new HitPoint(100);
Character characterA = new Caracter(hitPoint);
Character characterB = new Caracter(hitPoint);
```



可変がもたらす意図せぬ影響

ゲームを例に，不変の重要性について考えてみる．

◆ケース2：メソッドによる可変インスタンスの操作

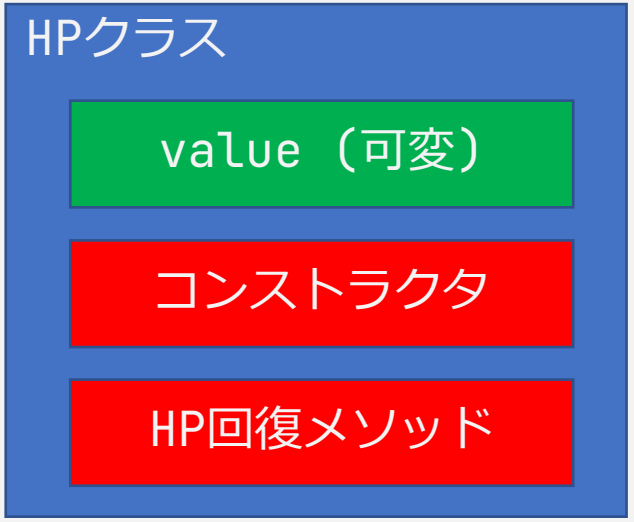
可変がもたらす意図せぬ影響

◆ケース２：メソッドによる可変インスタンスの操作

■HPを表現するクラスにHPをいじるメソッドを追加その1

```
class HitPoint {
 static final int MIN = 0;
 int value; // 可変

 HitPoint(int value) {
  if (value < MIN) {
   throw new IlligalArgumentException()
   }
    this.value = value;
 }
 public void recoverHP(int value) {
  this.value += value;
 }
}
```



先ほどのクラスにHPを回復するメソッドが追加された

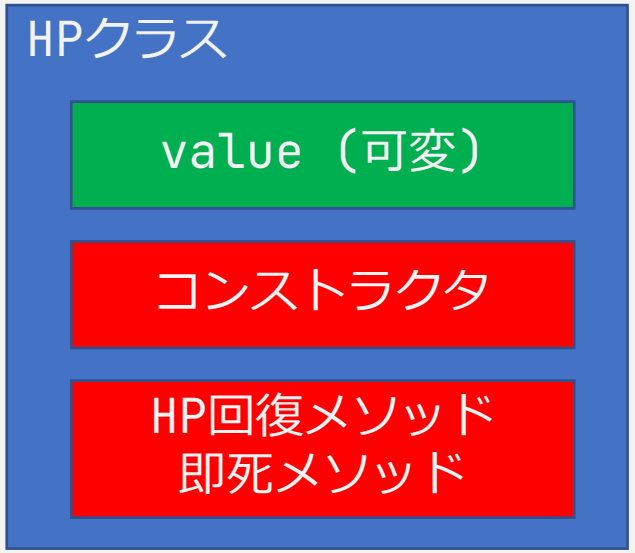
可変がもたらす意図せぬ影響

◆ケース２：メソッドによる可変インスタンスの操作

■HPを表現するクラスにHPをいじるメソッドを追加その2

```
class HitPoint {
 static final int MIN = 0;
 int value; // 可変
 // コンストラクタ省略
 public void recoverHP(int value) {
  this.value += value;
 }

 public void instantDeath() {
  this.value = MIN;
 }
}
```



先ほどのクラスに即死するメソッドが追加された

可変がもたらす意図せぬ影響

◆ケース２：メソッドによる可変インスタンスの操作

最初は上手くいくが…

```
HitPoint hitPoint = new HitPoint(40);
htiPoint.recoverHP(50); // HPを回復

…

System.out.println(hitPoint.value);

…

hitPoint.instantDeath(); // ここで強制的にHPを0に
System.out.println(hitPoint.value);
```

（出力）90

（出力）0

よしよしいい感じ

可変がもたらす意図せぬ影響

◆ケース２：メソッドによる可変インスタンスの操作

しばらくすると勝手にHPが0になるように！

```
HitPoint hitPoint = new HitPoint(40);
htiPoint.recoverHP(50); // HPを回復

…

System.out.println(hitPoint.value);

…

hitPoint.instantDeath(); // ここで強制的にHPを0に
System.out.println(hitPoint.value);
```

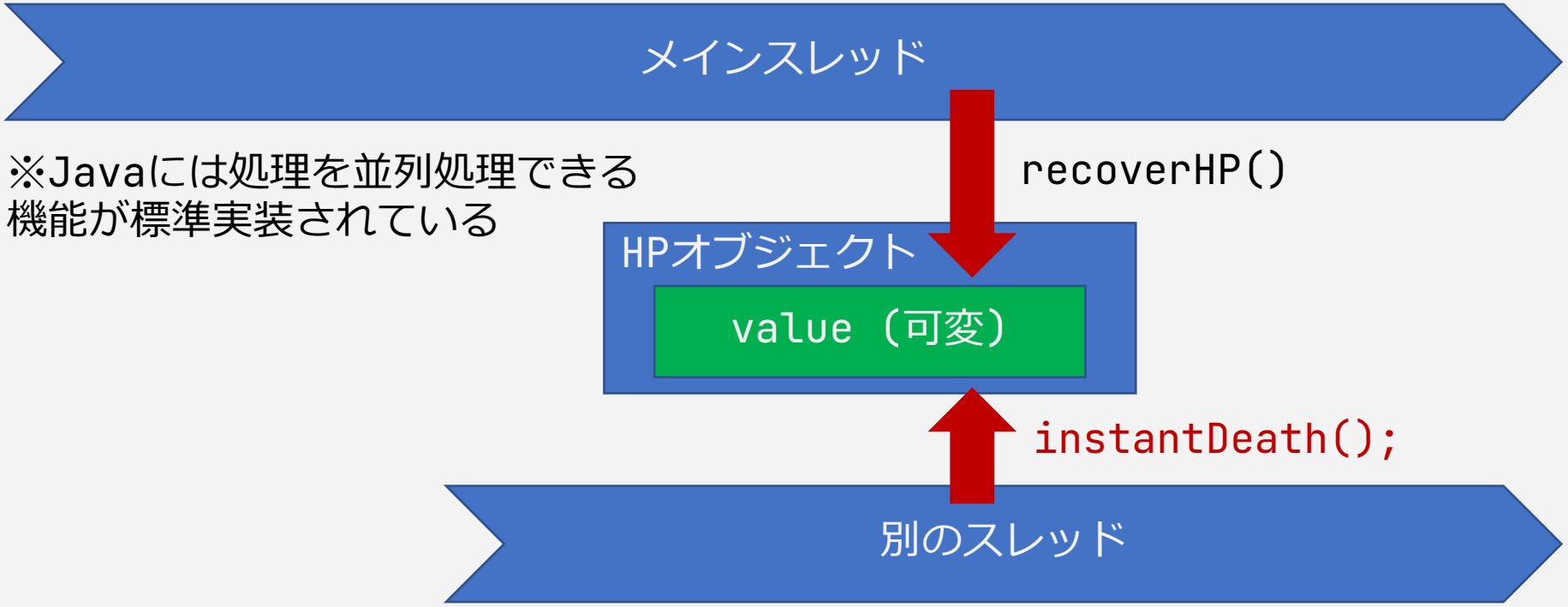
（出力）0

（出力）0

え，え？なんで？

可変がもたらす意図せぬ影響

◆ケース２：メソッドによる可変インスタンスの操作
なんと別のスレッドが同じインスタンスを操作していた
（インスタンスの使い回し）



可変がもたらす意図せぬ影響

◆このHitPointクラスは問題を抱えている

```
class HitPoint {
 static final int MIN = 0;
 int value; // 可変
 // コンストラクタ省略
 public void recoverHP(int value) {
  this.value += value;
 }

 public void instantDeath() {
  this.value = MIN;
 }
}
```

副作用と不変

副作用とは

副作用とは，外部の状態（変数など）を変更することを指す.

◆メソッドには以下の様な作用がある.

■主作用

- メソッドが引数を受け取り，値を返すこと.

■副作用

- 主作用以外に状態変更すること.
- 例えば，インスタンス変数の変更，グローバル変数の変更
  引数の変更，ファイルの読み書きなどのI/O操作

副作用とは

副作用とは，外部の状態（変数など）を変更することを指す．

◆このHitPointクラスは問題を抱えている

```
class HitPoint {
 static final int MIN = 0;
 int value; // 可変
 // コンストラクタ省略
 public void recoverHP(int value) {
  this.value += value;
 }

 public void instantDeath() {
  this.value = MIN;
 }
}
```

この変数が可変であり，なおかつメソッドが副作用を持つため，値がコロコロ変わり結果が処理の実行順に依存してしまう！

副作用を減らすためには

理想のメソッドは，引数で状態を受け取り，状態を変更せず，値を返すだけ．

◆メソッドの影響範囲を限定しよう

メソッドが以下の条件を満たすように設計する

■データ（状態）を引数で受け取る

■状態を変更しない

■値はメソッドの戻り値として返す

◆注意！

じゃあインスタンス変数って触ったらダメやんね？

インスタンス変数を絶対に触ってはいけないわけではない！

影響がクラス内に留まるならOKな場合もある（後述）

ここで不変を活用する

◆値を変更する場合は新しいインスタンスを作る.

■HPを表現するHitPointクラス

```
class HitPoint {
 static final int MIN = 0;
 final int value; // 不変
 // コンストラクタ省略
 public HitPoint recoverHP(final int value) {
  return new HitPoint(this.value + value);
 }

 public HitPoint instantDeath() {
  return new HitPoint(MIN);
 }
}
```

値は変更されない！

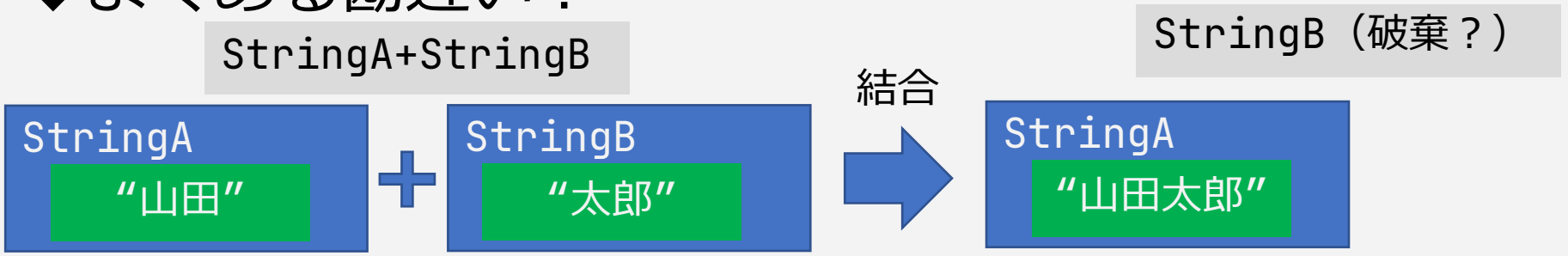
変更後のオブジェクトを戻り値にする.

これで値が勝手に変更されなくなりましたね.

(ちょっと豆知識)JavaのStringクラス

実はString（文字列）はプリミティブ型ではなく参照（リファレンス）型で，イミュータブル（不変）な型です．

◆よくある勘違い？



◆実際のStringクラス

実際のStringクラスはさらに他にも特殊な仕様がある…

デフォルトは不変に！

標準的には不変で設計しよう.

◆不変のメリットは以下の通り

■変数の意味が変化しなくなり，混乱を抑えられる

■挙動が安定し，結果を予測しやすくなる

■コードの影響範囲が限定的になり，保守がしやすくなる

◇Java
final修飾子で不変

◇JavaScript
const修飾子で不変

◇Kotlin/Scala
val句で不変

◇Rust
そもそも不変

final修飾子を付けたからと行ってコードは冗長にならない！
近年のプログラミング言語はより不変をより導入しやすいように！

じゃあ可変にするときってどんなとき？

◆基本的には不変が望ましいが，以下の場合は可変のほうがふさわしい

■パフォーマンスを重視する処理

- 例えば大量のデータの高速処理や画像処理，リソース制約の厳しい組み込みソフトウェアなどで可変を使用する.
- 不変はいちいちインスタンスを作るため少し時間がかかる.

■スコープが局所的

- ループカウンタなど，かなり限定的なスコープでしか使用されないことが確実なローカル変数などは可変にしてもよい.

次の章に向けて

不変の重要性について学びました.

◆基本的に不変で設計する

◆コード外とのやりとりは局所化する

◆次の章では第1章でも取り上げた低凝集について詳しく学習する

おまけ

このクラスは不変でより強固になったが，まだ問題を抱えている．それはどこだろうか？

```
class HitPoint {
    static final int MIN = 0;
    final int value;
    HitPoint(int value) {
        if (value < MIN) {
            throw new IlligalArgumentException()
        }
    this.value = value;
    }

    public HitPoint recoverHP(final int value) {
        return new HitPoint(this.value + value);
    }

    public HitPoint instantDeath() {
        return new HitPoint(MIN);
    }
}
```