# Effective Java

[Java]

Effective Java 第 2 版 から、メモ

## コンストラクタの代わりに static ファクトリーメソッドを検討する

・static ファクトリーメソッドのほうが、public コンストラクタより好ましいので、最初に static ファクトリーメソッドを検討する。

### ■ static ファクトリーメソッドの一般的な名前

・vlueOf
・of
・getInstance
・newInstance
・getType
・newTyep

## 数多くのコンストラクタパラメータに直面したときにはビルダーを検討する

・コンストラクタや static ファクトリーメソッドが多くのパラメータを持つクラスを設計する際には、ビルダーパターンはよい選択。

### ■例

```
public class Foo {
  private final int a, b, c;
  public static class Builder {
    // 必須
    private final int a;
    // オプション
    private int b = 0;
    private int c = 0;

    public Builder(int a) { this.a = a; }

    public Builder putB(int b) {
      this.b = b; return this;
    }
    public Builder putC(int c) {
      this.c = c; return this;
    }

    pubcli Foo build() {
      return new Foo(this);
    }
  }

  private Foo(Builder builder) {
    this.a = builder.a;
    this.b = builder.b;
    this.c = builder.c;
  }
}
```

## private のコンストラクタか enum 型でシングルトン特性を強制する

### ■例

public final フィールド

```
public class Foo {
  public static final Foo INSTANCE = new Foo();
  private Foo() {}
  public void doSomething() {...};
}
```

static ファクトリーメソッド

```
public class Foo {
  private static final Foo INSTANCE = new Foo();
  private Foo() {}
  public static Foo getInstance() {
    return INSTANCE;
  }
  public void doSomething() {...};
}
```

単一要素を持つ enum 型

・現在 ( リリース 1.5 以降 ) では最善

```
public enum Foo {
  INSTANCE;
  public void doSomething() {...};
}
```

## private のコンストラクタでインスタンス化不可能を強制する

・static メソッドと static フィールドのみからなる ユーティリティクラスは、引数なしの private コンストラクタを宣言することで、インスタンスかを不可能にする。

## 不必要なオブジェクトの生成を避ける

### ■再利用を行う

・オブジェクトが不変 (immutable) であれば常に再利用できる
・変更されないとわかっている可変 (mutable) オブジェクトも再利用できる

### ■遅延初期化は勧められない

・実装が複雑になる
・顕著なパフォーマンス改善が得られない

### ■アダプター

・アダプターは状態を持たないので、複数インスタンスは不要

### ■ボクシング

・ボクシングされた基本データ型よりも基本データ型を選ぶ
・意図しない自動ボクシングに注意

### ■防御的コピー

・新たなオブジェクトを生成すべき場合、再利用は行わない

## 廃れたオブジェクト参照を取り除く

・廃れた参照 (obsolete reference) 決してそれを通してオブジェクトが参照されることのない単なる参照。
・使い終えるとあらゆる参照に null を設定するのは過度な対処。プログラムを不必要に乱雑にしてしまう。null 設定は例外であるべき
・最良の方法は、参照が含まれていた変数をスコープの外に出すこと。

### ■注意

## ファイナライザを避ける

・ファイナライザ (finalizer) は、予測不可能で、たいていは危険であり、一般には不要。* セーフティネットとして使用 ( ロギングを伴い ) する、ネイティブピア ( ネイティブメソッドを委譲を通して利用、GC の対象外 ) で使用するの 2 つが正当な利用方法。

## equals をオーバーライドするときは一般契約に従う

・一般契約を厳守すること

### ■一般契約

| 契約 | 内容 |
|---|---|
| 反射的 | x.equals(x) は true |
| 対称的 | y.equals(x) が true の場合のみ、x.equals(y) も true |
| 推移的 | x.equals(y) 、y.equals(z) が true ならば、<br>x.equals(z) も true |
| 整合的 | 情報が変更されなければ、x.equals(y) の結果は不変 |

抽象化の恩恵をあきらめずに、インスタンス化可能なクラスを拡張して、equals 契約を守ったまま、値要素を追加する方法はない。

## equals をオーバーライドする時は、常に hashCode をオーバーライドする

### ■ hashCode の実装例

1. ゼロではない定数 ( たとえば 17) を result(int 型 ) に保存
2. 意味のあるフィールドに対して、次を行う
    1.boolean なら (f?1:0)
    2.byte,char,short,int なら、(int)f
    3.long なら、(int)(f^(f>>32))
    4.float なら、Float.floatToIntBits(f)
    5.double なら、Double.doubleToLongBits(f) を行い、long と同様の変換
    6. 参照なら、オブジェクトの hashCode() null なら 0
    7. 配列なら、各要素を フィールドとして扱う
    8.result = 31 * result + c // c= 計算されたハッシュコード

```
public int hashCode() {
  int result = 17;
  result = 31 * result + foo;
  result = 31 * result + bar;
```

```
    return result;
}
```

Eclipse でオブジェクトのコンテキストメニュー - Source - Generate hashCode() and equals() で自動生成出来る

### toString() を常にオーバーライドする

・優れた toString() を実装するとクラスが使いやすくなる
・オブジェクトに含まれる興味ある情報をすべて出力
・意図をドキュメントに書く

### clone を注意してオーバーライドする

インターフェースは Cloneable を拡張すべきではなく、継承されるように設計されたクラスでは、Cloneable を実装すべきでない

・コピーを行う代替手段を提供するか、コピーしないほうがよい
・上手い方法は、コピーコンストラクタか、コピーファクトリーを提供すること。

コピーコンストラクタ

```
public Foo(Foo foo);
```

コピーファクトリー

```
public static Foo newInstance(Foo foo);
```

### Comparable の実装を検討する

equals メソッドの一般契約に似た契約

・sgn(x.compareTo(y) == -sgn(y.compareTo(x)) であること
・x.compareTo(y) > 0 && y.compareTo(z) は、x.compareTo(z) > 0 であること
・x.compareTo(y) ＝＝ 0 が、sgn(x.compareTo(z) ＝＝ sgn(y.comareTo(z)) であること
・x.compareTo(y) なら、x.equals(y) であることを推奨

equals と同様、オブジェクト指向の抽象化による恩恵をあきらめずに、インスタンス化可能クラスを拡張して、compareTo 契約を守ったまま、値要素を追加する方法はない。

## クラスとインターフェース

### クラスとメンバーへのアクセス可能性を最小限にする

・インスタンスフィールドは決して public にすべきではない
・public の可変フィールドをクラスは、スレッドセーフではない
・クラスが、public static final の配列フィールドやそのようなフィールドを返すアクセサを持つのはほとんど常に誤り。

### public のクラスでは public のフィールドではなく、アクセッサーメソッドを使う

・クラスがパッケージプライベート、あるいは、private のネストしたクラスの場合、データフィールドを直接公開するのに、本質的な問題はない

## 可変性を最小限にする

・不変クラスは、設計、実装、使用が可変クラスより容易、本質的にスレッドセーフ

### ■クラスを不変にする 5 つの規則

1. 状態を変更するミューテイターメソッドを提供しない
2. クラスを拡張させない ( 一般的には final クラスとするが、別の方法もある )
3. すべてのフィールドを final にする
4. すべてのフィールドを private にする
5. 可変コンポーネントに対する独占的アクセスを保障
    ・可変オブジェクトを参照するフィールドを持つ場合、そのクラスのクライアントが参照を取得できないように、もしくは防御的コピーを利用する

## 継承よりコンポジションを選ぶ

・インターフェースの継承は、対象外

### ■コンポジション

・継承する代わりに、既存のクラスのインスタンスを参照する private フィールドを持つ * 新たなクラスの各インスタンスメソッドは、保持している既存クラスの対応するインスタンスメソッドを呼び出し、結果を返す
・これは転送 (forwarding) と呼ばれ、新たなクラスのメソッドは転送メソッド (forwarding method) と呼ばれる
・ラッパークラス、デコレーターパターン、としても知られる。コンポジションと転送の組み合わせを委譲 (delegation) とも呼ぶ

## 継承のために設計および文書化する、でなければ継承を禁止する

・クラスはオーバーライド可能なメソッド (public または protected) を呼び出すいかなる状況もドキュメントに記述しなければいけない
・継承のために設計されたクラスをテストする唯一の方法はサブクラスを書くこと
・安全にサブクラス化されるための設計と文書化がなされていないクラスのサブクラス化を禁止する。( クラスを final に、または、コンストラクタを private かパッケージプライベートにして、static ファクトリーメソッドを追加する )

## 抽象クラスよりインターフェースを選ぶ

・既存のクラスを新たなインターフェースを実装するようにすることは簡単にできる
・インターフェースはミックスインを定義するのには理想的
・インターフェースは階層を持たない型フレームワークを構築することを可能にする
・ラッパークラスイデオムを通じて、インターフェースは安全で強力な機能拡張を可能にする
・抽象 骨格実装クラス (skeletal implementation) を外部に公開する重要なインターフェースごとに提供することで、インターフェースと抽象クラスの長所を組み合わせることができる。

## 型を定義するためだけにインターフェースを利用する

・定数インタフェース (static final フィールドのみからなる ) は下手な使い方
・enum 型か、インスタンス化不可なユーティリティクラスを使うべき
・インターフェースは型を定義するためだけに使用する

## タグつきクラスよりクラス階層を選ぶ

### ■タグつきクラス

・インスタンスが 2 つ以上の特性を持っていて、その特性を示すタグフィールドを持つクラス

例 - 円か四角かを表現できるクラス

・冗長で誤りやすく非効率
・クラス階層の面白みのない模倣に過ぎない

```
class Figure {
  enum Shape = { RECTANGLE, CIRCLE };
  final Shape shape;
        :
```

## 戦略を表現するために関数オブジェクトを使用する

・java は関数ポインタを提供していないが、同じ効果を得るためにオブジェクト参照を利用できる
・他のオブジェクトを操作するメソッドを 1 つだけ公開するようなオブジェクトは関数オブジェクトと呼ばれる
・具象戦略クラスは、典型的には、オブジェクトは状態を持たないので、シングルトンにするのが適している。