

## プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 10: オブジェクト指向

## オブジェクト指向

オブジェクト指向プログラミングとは何か？

- Object Orientation
- Object Oriented (OO) Programming Languages

以下の教科書の 10 章を参考にすること .

- Programming Languages: Principles and Paradigms, Gabbrielli and Martini, Springer, 2006. (10 章 pp.277-332) (授業ウェブページから , 教科書の電子ファイルへのリンクあり)
- できるだけ一般的なオブジェクト指向言語の話 (実際には Java を念頭に置く)

## プログラミングスタイル(プログラムの構成方法)

コントロール指向のプログラミングスタイル:

- プログラム構成における主要な関心事が「制御」
- データに対する「操作」の観点でのプログラミング

データ指向のプログラミングスタイル:

- プログラム構成における主要な関心事が「データ」
- 操作される「データ」の観点でのプログラミング
- 例: スタックの操作をひとまとまりにしたもの (スタック抽象データ型の実装)

## 抽象データ型 (ADT) vs オブジェクト

ADT とオブジェクトに共通する事項

- 抽象化 (情報隠蔽, カプセル化)

オブジェクトの特徴 (ADT がない点)

- 継承 (inheritance)
- メソッドの動的選択 (dynamic lookup, dynamic dispatch)
- サブタイピング (subtyping)

```
class Point {
  private int x; インスタンス変数
  private int y;
  public int getX() { return x;} メソッド
  ...
class CPoint extends Point { 継承
  ...
```

クラス:

- オブジェクトたちの共通テンプレート .
- ひとつひとつのオブジェクトは、クラスをもとにして作成される (クラスのインスタンス) .
- cf. JavaScript は、クラスがなく、オブジェクトだけがある .

- Abstraction 抽象化
- Inheritance 継承
- Dynamic lookup 動的ルックアップ
- Subtyping サブタイピング (部分型付け)

J. C. Mitchell, "Concepts in Programming Languages", 2003.

## Abstraction

抽象データ型における Abstraction と同様。

- オブジェクトへのアクセスは、インタフェース関数 (メソッド) のみに限定される。
- 実装と仕様 (インタフェース) の分離を達成。

## Inheritance

継承によるコードの再利用

```
class Point {
  private int x = ...;
  ...
}
class CPoint extends Point {
  private int c;
  ...
}
```

再利用の利点:

- プログラムの見通しが良くなる
- プログラムの記述量の削減
- プログラムのコンパイルや解析・検証の手間, コード量の削減
- デバッグ, テストの容易さ

## Dynamic Method Lookup

### Method Lookup

- **メソッドの名前**から、実際に起動されるべき**メソッドの実装**を得る操作
- cf. 変数のルックアップ: 変数名から、現在の環境におけるその変数の値を得る操作

### ルックアップが動的 (dynamic)

- 名前と実体の対応が**実行時**に決まる。

## Dynamic Lookup

オブジェクト foo のメソッド add を e という引数で起動。

```
foo.add(e)
```

- 起動されるメソッドは、オブジェクトごとに決まる。
- 起動される add メソッドは、実行の時点ごとに (foo が変数の場合, その中身となるオブジェクトごとに) 異なり得る。

実例: Test.java ファイルの p.sum() の呼び出し

- p は Point クラスの変数だが, その中に CPoint クラスのオブジェクトが格納されているとき CPoint の sum メソッドが呼ばれる
- 多くのプログラミング言語の「関数」は静的束縛。(コンパイル時,あるいは,リンク時に,関数名がどの実体を指すかを決定)

## Dynamic Lookup

静的ではなく、動的なルックアップは、プログラミング上、極めて有用。

ColoredCircle クラスが Circle クラスを継承して作成された

- 「Circle オブジェクト」を draw するメソッド
- 「ColoredCircle オブジェクトを draw するメソッド

## Subtyping (A <: B)

クラス A がクラス B の subtype である (Aの方が子クラス)

- クラス B のオブジェクト (をあらわす式) を書くべきところに、クラス A のオブジェクト (をあらわす式) を書いても良い。[代入可能性, compatibility]

```
class Point {  
    ...  
    ... void move (int dx, int dy) { ...}  
}  
class Circle extends Point {  
    ...  
    ... void move (int dx, int dy) { ...}  
}
```

Point クラスのオブジェクトに対する操作は、その子クラスである Circle クラスのオブジェクトに対しても適用できる。

## Subtyping と多相型

OO 言語では:

- move メソッドが Point オブジェクトにも Circle オブジェクトにも適用可能。
- move メソッドは、Point クラスを継承した任意のクラスのオブジェクトに対して適用可能。
- 一種の多相性 (subtyping polymorphism ML 言語の parametric polymorphism)

## Subtyping vs Inheritance

これらの違いは何か?

- サブタイピング: 2つのオブジェクトのインタフェースの間の関係。
- 継承: 2つのオブジェクト (やクラス) の実装の間の関係。

クラス B がクラス A を継承すれば, B は A のサブタイプになる (B オブジェクトを A オブジェクトとして使って良い) ことが多いが, 概念としてはこれら 2 つは必ずしも一致しない.

## OO 言語たち

- Simula [1960 年代, K. Nygaard]
- Smalltalk [1970 年代, Xerox PARC 研究所, Alan Kay]
- C++ [1984-, Stroustrup]
- Java [1990-, Gosling]
- Ruby [1993-, Matsumoto]
- JavaScript [2005-, Eich]
- Scala [2003-, Odersky]

## ここまでのまとめ

- オブジェクト指向の 4 つの基本概念
- ADT・モジュールとの共通点、相異点

質問 1. 「動的ルックアップ」とは何か、説明せよ。  
 質問 2. Java では、変数束縛は静的である一方で、method のルックアップは動的である。なぜそのような設計が良いのか、考えなさい。

```
class Point { ...
  public String toString () {
    return "Point...";
  }
}
class ColoredPoint extends Point { ...
  public String toString () {
    return "ColoredPoint...";
  }
}
```

Override (上書き):

- 親クラス (Point) を継承した子クラス (ColoredPoint) では、メソッド toString の定義 (実装) をそのままもらうのではなく、違うものを書きかえている。
- toString の引数の個数、型、返すものの型は、まったく同じ。

```
class Test1 {
  public static void main(String args[]) {
    Point p = new Point(10.0, 20.0);
    ColoredPoint cp = new ColoredPoint(10.0, 20.0, 3);

    System.out.println(p.toString()); => 親の toString
    System.out.println(cp.toString()); => 子の toString
```

(1) 親クラスの変数に、子クラスのオブジェクトを代入してもよい。

```
class Test1 {
  public static void main(String args[]) {
    ColoredPoint cp = new ColoredPoint(10.0, 20.0, 3);
    Point p = cp;
    System.out.println(p.toString()); => 子の toString
```

cp.toString() について、動的にルックアップしている。

(2) 子クラスの変数に、親クラスのオブジェクトを代入するのはいけない。

```
class Test1 {
  public static void main(String args[]) {
    Point p = new Point(10.0, 20.0);
    ColoredPoint cp = p; => コンパイルエラー
```

## Override in Java

親クラスから子クラスへ (Java における extends キーワード) :

- 子クラスのメソッド等のインタフェース (引数の個数、引数の型、戻り値の型) は、親と同じ。(サブタイピング)
- 子クラスのメソッド等の実装は、親クラスのものと同じでもよいし (継承)、違うものを書き換えてよい。(オーバーライド、上書き)

## Overload in Java

Overload (オーバーロード) は Overwrite とは別の仕組み :

```
class Test1 {...
    public static void foo(Point p) {
        System.out.println("foo-1:" + p.toString());
    }
    public static void foo(ColoredPoint cp) {
        System.out.println("foo-2:" + cp.toString());
    }
    public static void foo(Point p, ColoredPoint cp) {
        System.out.println("foo-3:" + p.toString() + ":" + cp.toSt
    }}
```

Overload されたメソッド:

- 1つのクラスの同一メソッド名に対して、複数の実装を持つ。
- インタフェース (引数の個数、引数の型、戻り値の型) が異なる。

インタフェースに従ってどの実装を使うか静的に決定。

## Overload with Cast in Java

親クラスの変数に、子クラスのオブジェクトを代入した場合 :

```
Point p = new ColoredPoint(...);
foo(p);      ==> foo-1 が呼ばれる。
```

変数 p の中身は、子クラス ColoredPoint のオブジェクトであるが、foo のどの実装が呼ばれるかは、静的に (インタフェースで) 決定される。ここでは foo-2 でなく foo-1 が呼ばれる。

一方、foo の中で呼ばれる toString が、どの実装であるかは (override なので) 動的に決定され、子クラスの toString が呼ばれる。

## Override and Overload in Java

```
class Test4 {
    public static void foo(Point p) {
        System.out.println("foo-1:" + p.toString());
    }
}
class Test5 extends Test4 {
    public static void main(String args[]) {
        Point p = new Point(10.0, 20.0);
        ColoredPoint q = new ColoredPoint(10.0, 20.0, 5);
        Point r = q;
        foo(r);      ==> 何が返るか?
    }
    public static void foo(ColoredPoint cp) {
        System.out.println("foo-2:" + cp.toString());
    }
}
```

全ページの説明：

Test5 クラスにおけるメソッド foo は、親クラス Test4 から継承したものと、子クラス Test5 で定義したものの2つがあり、インタフェースが異なる。(引数の型が Point か ColoredPoint かが違う)。

この場合、overload なので、静的に解決され、変数 r が Point 型であることから、御クラス Test4 の foo の実装が呼ばれる。(変数 r には、実際には子クラス ColoredPoint のオブジェクトがはいっているのだが、それは動的な情報。)

Test4 の foo から呼ばれる toString は、Point と ColoredPoint の2か所で定義されている実装をもつが、これは override なので、動的に解決され、foo の引数 p にはいるオブジェクトが ColoredPoint クラスのものであるから、ColoredPoint の toString の実装が呼ばれる。(p の型が Point であることは関係ない。)

Java は静的型付きオブジェクト指向言語

- Override: 親クラスと子クラス (あるいはその子孫のクラス) の間で、同じインタフェース (名前・引数の個数、型) で、異なる実装をもつメソッドを持つこと。
  - 「どの型 (クラス) の変数か」ではなく、「実行時に、その変数にどのクラスのオブジェクトがはいっているか」によって、使われるメソッドが決まる (動的ルックアップ)。
- Overload: 同一クラス内で、1つのメソッド名が、異なるインタフェース (名前・引数の個数、型) を持つ複数の実装を持つこと。
  - どのメソッド定義が使われるかは、メソッド呼び出しがどのインタフェースに合致するかにより、静的に決定される。

## 次週

- Java generics (と C++ template)
- その他の話題
- 授業全体のまとめ
- (授業アンケート)