オブジェクト指向の考え方

平成 26 年度 情報論理

目次

1 図形を描く 3

1.1 四角を描く 3
1.2 円を描く 4
1.3 四角と円の位置を合わせる 4
1.4 四角と円の共通点と相違点 4
1.5 図形を動かす 5
1.6 たくさん動かす 5

2 オブジェクト指向 8

2.1 処理中心からオブジェクト中心へ 8
2.2 クラスという考え方 8
2.3 クラスは「型」 9
2.4 属性（データメンバ・メンバ変数・フィールド） 9
2.5 操作（メソッド・メンバ関数） 9
2.6 自律分散 10
2.7 カプセル化 10

3 クラスを作る 10

3.1 四角クラス 11
3.2 円クラス 14

4 関連 15

4.1 has-a の関係 15
4.2 part-of の関係 17
4.3 継承 17
4.4 多態性 18

5　　抽象クラス　18

1　図形を描く

processing を使って図形を描きながら、考え方を確かめる。

※ プログラムは論理の組み立てである

以下が processing のスケルトン。エディタに書いて実行（Ctrl + R）すると 800 * 600 の大きさのカンバスが現れる。

```
/*
 * prim00.pde
 * 図形を描く 1
 */

/************************************
 * 最初に 1 回だけ動く関数（メソッド）
 *     おもに初期設定に使う
 ************************************/
void setup() {
  size(800, 600);    // 描画領域
  frameRate(20);     // 1 秒間に描写するフレーム数
  smooth();
}

/*****************************************
 * frameRate に応じて何度も動く（ループする）
 * 座標などを変えると絵を動かすことができる
 *****************************************/
void draw() {
  background(130, 160, 180);     // 背景の色 (Red, Green, Blue)

  }
}
```

- Processing 2 Reference : https://www.processing.org/reference/
  - Processing 2 日本語(非公式): https://dl.dropboxusercontent.com/u/120287171/Processing

1.1　四角を描く

```
    // 四角を描く rect(x, y, w, h)
    // x, y : 座標
    // w, h : 幅, 高さ
    rect(0, 0, 200, 100);  // processing の組込関数
```

- 描画領域を確認せよ
- 座標の位置を確認せよ

1.2 円を描く

```
// 円を描く ellipse(x, y, w, h)
// x, y : 座標
// w, h : 幅, 高さ
ellipse(0, 0, 200, 100);
```

- 「四角を描く rect(x, y, w, h)」と違うのは？

1.3 四角と円の位置を合わせる

次の図 1 のようにするにはどうする？

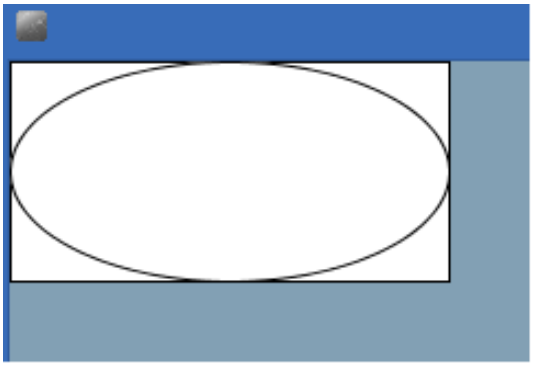

図 1

コード例

1.3.1 ポイント

- `void draw() {}` の中にデータを直打ちするのはどうか？
  - `rect(0, 0, 200, 100);`
  - `ellipse(0, 0, 200, 100);`
- たとえば四角の大きさが変わったとき、それに合わせて円の大きさを変更するには何個数字を書き換えることになる？
- データ（数値）を「変数」に置き換えて設計できるのは**抽象化**の能力

1.4 四角と円の共通点と相違点

- 共通点

- 相違点

※ あとで「クラス」を作るときに参照する

1.5 図形を動かす

- x の値を変えれば x 軸方向に動く
  x += 1;
- y の値を変えれば y 軸方向に動く
  y += 1;
- x, y 両方の値を変えれば斜め方向に動く

※ 問題はそのコードを「どこに」「どう」書くか
コード例

- 「`x += moveX;  y += moveY;`」 の行を
  「`// 円を描く ellipse(x, y, w, h)`」の前に移動したら、どうなると思う？
  – 四角は動かずに円だけが動いていくだろうか？

1.6 たくさん動かす

※ それぞれの図形がそれぞれ勝手に動くようにするには？

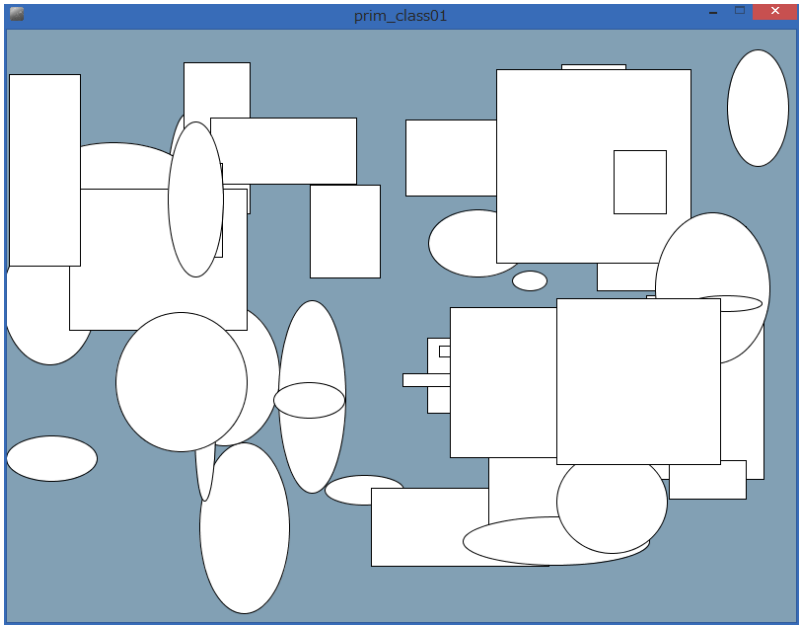

図 2

いきなり四角と円の両方に対処するのは無理があるので、とりあえず四角だけを複数作っ

てみよう。

次のようにして、円の描画部分をコメントアウトしておく。

```
  // 円を描く ellipse(x, y, w, h)
  // x, y : 座標　円の中心の位置
  // w, h : 幅, 高さ
//  ellipse(x + w / 2, y + h /2, w, h);
```

描く図形ごとに x, y, w, h,moveX, moveY は異なるから、これらを「配列」に作っておく。

```
float x[] = new float[MAX];
float y[] = new float[MAX];
float w[] = new float[MAX];
float h[] = new float[MAX];
float moveX[] = new float[MAX];
float moveY[] = new float[MAX];
```

図形をたくさん描くには for 文でループさせればよい。

```
for(int i=0; i<MAX; i++){
  rect(x[i], y[i], w[i], h[i]);  // processing の組込関数
  // 移動量
  x[i] += moveX[i];  y[i] += moveY[i];
}
```

この場合 for 文の中でカウントしている「 i 」が、描画する図形の番号を表している。

- 図形 [0] の座標が x[0], y[0]、大きさが w[0], h[0]
  X 軸方向の移動量が moveX[0]、 Y 軸方向の移動量が moveY[0]
- 図形 [1] の座標が x[1], y[1]、大きさが w[1], h[1]
  X 軸方向の移動量が moveX[1]、 Y 軸方向の移動量が moveY[1]
- 図形 [2] の座標が x[2], y[2]、大きさが w[2], h[2]
  X 軸方向の移動量が moveX[2]、 Y 軸方向の移動量が moveY[2]

random() 関数を使うと乱数を発生させられる。

```
random(100);      // 0 から 100 までの数値をランダムに発生
random(10, 20);   // 10 から 20 までの数値をランダムに発生

random(width);  // width は描画領域の横幅を表す
random(height); // height は描画領域の横幅を表す
```

これを使ってすべてのデータ（変数）をランダムに設定できる。

```
//変数初期化
for(int i=0; i<MAX; i++){
  x[i] = random(width); y[i] = random(height);
  w[i] = random(10,200); h[i] = random(10,200);
  moveX[i] = random(1,5); moveY[i] = random(1,5);
}
```

おまけ。

このままだとせっかく描画した図形がカンバスの外に出ていったままいなくなってしまうので、次のようにしてみよう。よく読めば何をやっているかわかるね？

```
// 跳ね返り
if(x[i] < 0 || x[i] + w[i] > width) moveX[i] *= -1;
if(y[i] < 0 || y[i] + h[i] > height) moveY[i] *= -1;
```

コード例

ただ、これで描画するとおかしな動きをする奴がたまに出現する。実行してみるとわかる。なぜおかしな動きになるか、理由を考えて対処してみて。コード例

※ 四角はこれでいいとして、円を同様にたくさん作って動かすにはどうする？

円は円で、四角とは別に操作するわけだから、新たに円用の変数配列を作ることになるか。たとえば `x[i]`, `y[i]`, `w[i]`, `h[i]` にしても

- 四角用：`bx[i]`, `by[i]`, `bw[i]`, `bh[i]`（b は box の略）
- 円用：`cx[i]`, `cy[i]`, `cw[i]`, `ch[i]`（c は circle の略）

のようにするかな。

配列は同じタイプの変数を一手に操作するには便利だが、これだけ多種類のものを要素番号で区別しながらコーディングするのは頭が痛いよね。

2　オブジェクト指向

2.1　処理中心からオブジェクト中心へ

ここまでの「目次」をリストアップしてみると次のようになる。

- 図形を描く
  - 四角を描く
  - 円を描く
  - 図形を動かす
  - たくさん動かす

作業は「動き（動詞）」で表現される。つまり、作業の中心は「処理」を手がかりに進めてきたことがわかる。

処理中心の考え方は、システムが「何をするか」から考えて全体を組み立てるものだ。システムの動きが「確定」していればとてもわかりやすいし、コードも書きやすくなる。

しかし、処理を中心に組み立てると、たくさん存在する図形が「それぞれ勝手に動くようにする」ような処理はとても煩雑になる。（たとえばそれぞれの図形の座標を、全て区別するために膨大な変数の判別をしなくてはならない）「処理」を正確に稼働させるために、天空にコントロールセンターを置いて、そこからトップダウン式に全体を制御しているようなものだ。

壁にぶつかって跳ね返る処理を追加したが、配列の要素番号（index）で区別しながら個々の図形に動きを施しているのがわかるだろう。

> 繰り返すが、処理中心の考え方は、これはこれで有効な考え方である。
>
> そのシステムが「どういう動きをするか」を基準にして分析する考え方だから、処理の仕様がかっちり定まっていて大きな改変がないようなシステムなら、分析も確定できるし、処理も書きやすい。（たとえば、会社の会計処理とか）
>
> だが、時代は変化が大きく常に新しいトレンドが生まれ、それに対応せざるをえなくなっている。

「処理」のまとまりを単位として切り出し（これを「モジュール」と呼ぶ）、それを構造的に組み上げることを**「構造化プログラミング」**という。「構造化プログラミング」の基本構造は

- 順次処理
- 条件分岐
- 繰り返し処理

だ。この 3 つの処理を組み立てながらプログラミングする。構造化プログラミングを実現する構造化言語の代表格が「C 言語」である。

2.2　クラスという考え方

動くものに目を向けるのではなく、その前に「存在」するもの（これを「オブジェクト」と呼ぶ）を見る。「モノ」中心に見るということ。

また、「動き」ではなく「データ」を中心に見る、ということもできる。先の「1.6 たくさん動か

す」処理でも、結局はそれぞれの図形が持つ（x[i], y[i], w[i], h[i]）の変化が動きをもたらしていた。

つまり、「動き」をつけるために「データ」を変化させた、と考えるのではなく、「データ」の変化が「動き」をつけた 、と考えるわけ。

すると、考え方が「動詞」から「名詞」中心になるのがわかるかな？そう、「クラス」はそのシステムを分析したときに名詞として表現されるものである。

次の仕様から「クラス」になり得る名詞を抜き出しなさい。

```
四角形は座標と大きさを持つ。
rect(x, y, w, h) が四角形を描く。
円は座標と大きさを持つ。
ellipse(x, y, w, h) が円を描く。
```

※ 名詞がすべてクラスになるわけではないことに注意

※ クラスが持つデータも名詞で表現される（「属性」と呼ぶ）

2.3 クラスは「型」

有名な比喩を挙げておく。

クラスはクッキーを作る時の「型」と同じだ。小麦粉を練った生地から型抜きをするための「型枠」（星形とかハート形とか）のことだ。この「型枠」は出来上がる形を規定しているが、実際に食べられない。あくまでも、クッキー（オブジェクト）の形を定義しているだけ。

この「型枠」からそれぞれの形に抜き出されたものが、実際に食べられるクッキーになる。これを「インスタンス」（実体）という。

また、プログラム的に見ても、いわゆる変数の「型」と同じようにクラスを扱う。次の例は「クッキー」クラスの変数 cookie を宣言して、同時にインスタンスも作っている。上の行の「int i;」と似た形になっているのがわかるだろう。

```
int i;          // int 型の変数 i を宣言
Cookie cookie;  // Cookie 型の変数 cookie を宣言
cookie = new Cookie();  // new 演算子はインスタンスを作る
```

2.4 属性（データメンバ・メンバ変数・フィールド）

クラスは「属性」を持つ。属性は「そのクラスの成り立ちを示すデータ」[1] である。

例えば、「生徒」クラスならば、「学年」「組」「出席番号」「名前」などの属性を持っている。上の「四角形」クラスなら「座標（x,y)」「大きさ（w,h)」がそれにあたる。

属性は「プログラム的にはデータメンバに該当」[1] する。データメンバはクラス内だけで通用するデータで、特に外部に公開するための仕組みを作らない限り、基本的に外部からはアクセスできない。「メンバ変数」「フィールド」とも呼ばれる。「クラス変数」や「インスタンス変数」とも。

2.5 操作（メソッド・メンバ関数）

クラス内で使う変数が「属性」なら、同じようにクラス内で使う「操作」も定義できる。

例えば、「生徒」クラスならば、「授業を受ける」「部活をする」など、そのオブジェクトが「何をするのか」が「操作」にあたる。当然、これは**動詞**である。

上の「四角形」クラスなら「四角形を描く」（プログラム的には「`rect()`」）がそれにあたる。

2.6 自律分散

それぞれのオブジェクトは、それぞれ個別に動く。自分がどう動くかは、自分が知っていて当たり前。誰かが「神」的な視座からコントロールする必要はない。そもそも、自然界にあるものは「自律分散」しながら、互いに関係を持ち合って存在しているのではないか？そういう自然の存在に合わせて、システムをシミュレーションするのが「オブジェクト指向」の考え方だ。

考えてみれば（普段そんなことを考えもしないけれど）私たちも、神様が命令するから行動するわけではなくて、あくまでも自分の意思で動いている（はず `(^0^)`）。けれども「処理」中心のプログラミングは、こういう「神の視座」でシステムを構築してきた。全体にくまなく目が行き届く範囲では、この方法は有効だ。しかし、運用開始後になんらかの仕様変更があったり、システム規模が膨大に大きくなったりするような場合には、とても対処できないという状況が現実に起こっている。

2.7 カプセル化

個人情報を持ち出すまでもなく、自分にとって大事なデータ（属性）はそうそうオープンにしないのが普通だ。システムやプログラムも同じように考える。これを「**隠蔽**」といい、クラスの大きな特徴の一つだ。

クラス内でもつ「属性」と「操作」を外側から隠すことができる。これを「カプセル化」といい、内部処理の独立性と安全性を保つことができる。

先の「四角形」の例でいえば、自分で自分を描くわけだから、自分の「座標」や「大きさ」はなにも外部に公開しなくていい。また「四角形を描く」操作にしても、他人から自分を描かせることがない限り、これも他者に「さらす」必要はないだろう。

現実にも、自分の身長や体重をラベルにぶら下げて歩いている人間はいない。

けれど、先に書いた「1.6 たくさん動かす」のソースコード（p.6）を見てみると、内部的な属性や操作が平気で公開されていることがわかるだろう。

もし、内部属性を外からアクセスさせる必要があるときは、そのための操作（メソッド）を特別に作り、「`public`」 という指定子をつけて）公開する。隠蔽関係で使う指定子には次のようなものがある。

- `private`
- `protect`
- `public`

3 クラスを作る

以下は『憂鬱なプログラマのためのオブジェクト指向開発講座』[1]（P72）から引用

順序としては、
- まずクラスの仕様の決定
- 「操作」の洗い出し
- そのために必要な「属性」を考える

「データ中心」とはいいながら、「ソフトウェアの構成部品としてクラスを考えると、内部に隠蔽されてしまう属性よりも、外部からの窓口となる操作のほうがはるかに重要」（前出）ということだ。

先に書いた四角を例にとると、

- `x, y, w, h, moveX, moveY` を初期化している部分
- `rect()` 関数を使って四角を描画している部分
- 壁への跳ね返り処理をしている部分

が「操作」にあたるだろう。

クラスの中身はメソッドから考える。

3.1 四角クラス

3.1.1 クラス図

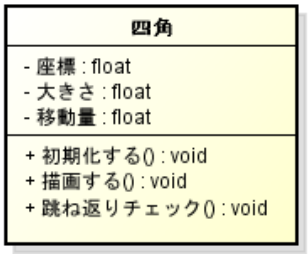

図 3

3.1.2 ソースコード

```
/*
 * 四角を描くクラス
 */
class Box{
  // メンバ変数
  float x, y;  // 座標
  float w, h;  // 幅, 高さ
  float moveX, moveY;  // 移動量

  // コンストラクタ  new でインスタンス（実体）が作られた時に
  // 最初に自動的に起動するメソッド
  Box(){
    //変数初期化
    x = random(width); y = random(height);
    w = random(10,200); h = random(10,200);
    moveX = random(1,10); moveY = random(1,10);
    // 位置調整
    if(x + w > width) x = width - (w + moveX);
    if(y + h > height) y = height - (h + moveY);
  }

  // 跳ね返りを判定するメソッド
  void chechDirection(){
    if(x < 0 || x + w > width) moveX *= -1;
    if(y < 0 || y + h > height) moveY *= -1;
  }

  // 図形を描くメソッド
  void drawPrim(){
    // 四角を描く
    rect(x, y, w, h);  // 四角を描く
    x += moveX; y += moveY;
    chechDirection();
  }
}
```

3.1.3 Processing に書くには

Processing ではエディタから保存したファイル名と同じタブが、必ずできる。（図 4 だと「prim_class00」）これがこの Processing プログラムがスタートするメイン・クラスだ。（Java ではクラス名とファイル名が同じになっている決まり）

Processing ではこれにさらにタブを追加して、新たなモジュールを追加できる。図 4 のようにタブの隣にある三角印をクリックして「New Tab」を選ぶ。ここでは、「PrimClass」というタブに「四角クラス」と「円クラス」の定義を書いている。

これで実行すると、「prim_class00」の方が実行されることになる。次のコードがそれだ。

先に「処理中心」で書いた配列たくさんのソースコードと比べると、ずいぶんシンプルに出来上がっていることがわかるだろう。先にも書いたように、クラス内のデータはクラスにお任せしてよい。

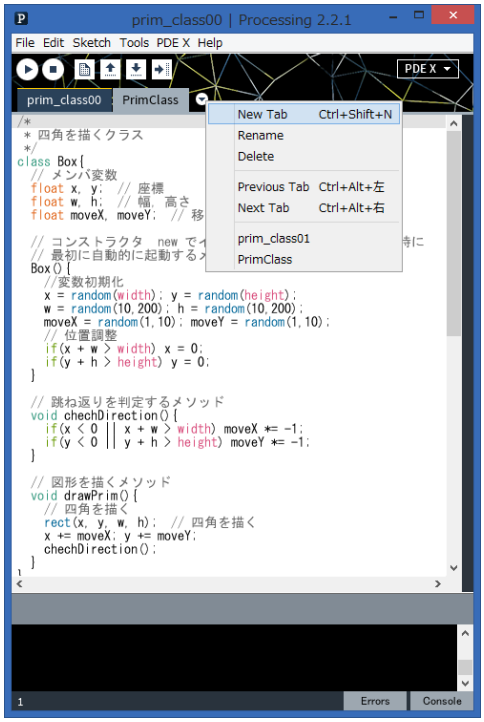

図 4

```
/**************************************
 * prim_class00
 **************************************/
// クラスを型にして変数を宣言
// ここでは入れ物ができただけで中身はまだ
Box box;
Circle en;

void setup() {
  size(800, 600);
  frameRate(20);
  smooth();
  // 四角形を作る
  // ここで一個ずつ中身が作られる
  box = new Box();
  // 円を作る
  en = new Circle();
}

void draw() {
  background(130, 160, 180);

  // 円を描く
  en.drawPrim();
  // 四角を描く
  box.drawPrim();
}
```

3.2 円クラス

3.2.1 クラス図

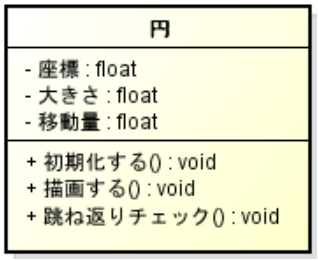

図 5

3.2.2 ソースコード

```
/*
 * 円を描くクラス
 */
class Circle{
  // メンバ変数
  float x, y;  // 座標
  float w, h;  // 幅, 高さ
  float moveX, moveY;  // 移動量

  // コンストラクタ　new でインスタンス（実体）が作られた時に
  // 最初に自動的に起動するメソッド
  Circle(){
    //変数初期化
    x = random(width); y = random(height);
    w = random(10,200); h = random(10,200);
    moveX = random(1,10); moveY = random(1,10);
    // 位置調整
    if(x + w > width) x = width - (w + moveX);
    if(y + h > height) y = height - (h + moveY);
  }

  // 跳ね返りを判定するメソッド
  void chechDirection(){
    if(x < 0 || x + w > width) moveX *= -1;
    if(y < 0 || y + h > height) moveY *= -1;
  }

  // 図形を描くメソッド
  void drawPrim(){
    // 円を描く
    ellipse(x + w / 2, y + h / 2, w, h);  // 四角を描く
    x += moveX; y += moveY;
    chechDirection();
  }
}
```

4　関連

ここまで特に注意せずに二つのタブを使い分けてきた。「prim00」と「PrimClass」だ。「prim00」はメイン・クラスとして特別な意味を持つことは前述したが、今設計している「図形を描く」システムとして考えると、「prim00」は絵を描くための「カンバス」の役割をしている。

そうするとこのシステムは「カンバスクラス (prim00)」を中心に「四角クラス (class Box)」と「円クラス (class Circle)」とがそれぞれ関連しあって動いていると分析できる。

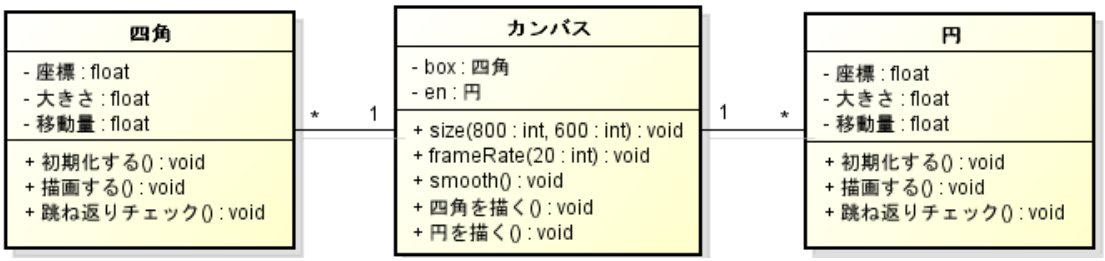

図 6

※ 注意

カンバスクラスの属性に「box 四角」「en 円」とあるのはそれぞれ「四角クラスのインスタンス box」と「円クラスの en」を表しているが、通常クラス図にインスタンスは書かない。ここでは説明のために書いただけ。

ふつう、クラス間にラインが引かれて関連が示されれば、それだけで「インスタンスを持つ」ことがわかる。

※ 四角と円がたくさん、それぞれに動き回るようにしてみなさい。コード例

4.1　has-a の関係

次の図 7 のように、四角と円がワンセットになって動くように変更してみなさい。

これは、

四角が円を持つ （四角 has-a 円）

ような形になっている。

四角が円を自分の中に取り込み、大きさを自分に合わせていると考えればよい。これを「**has-a の関係**」または「**包含**」という。

四角クラスの中に円クラスのインスタンスを持つようにする。円は四角の内部に取り込まれるので、外から見えるのは四角だけになる。クラス図にすると図 8 のようになる。

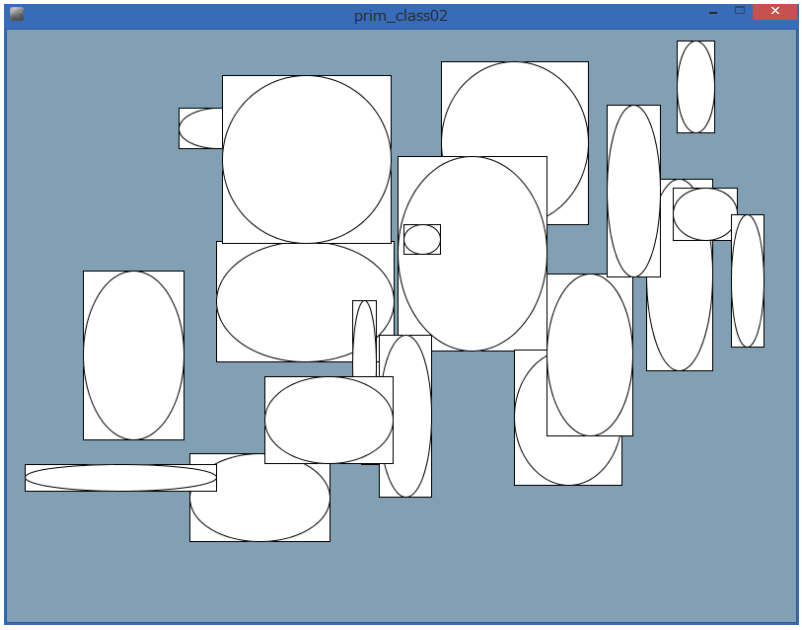
prim_class02

図 7

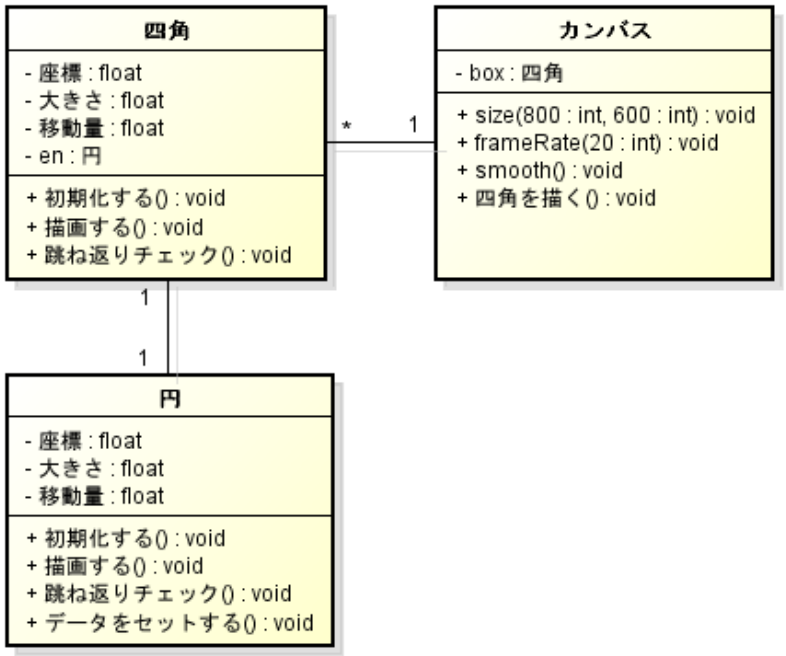
四角
- 座標 : float
- 大きさ : float
- 移動量 : float
- en : 円
+ 初期化する() : void
+ 描画する() : void
+ 跳ね返りチェック() : void
*
1
カンバス
- box : 四角
+ size(800 : int, 600 : int) : void
+ frameRate(20 : int) : void
+ smooth() : void
+ 四角を描く() : void
1
1
円
- 座標 : float
- 大きさ : float
- 移動量 : float
+ 初期化する() : void
+ 描画する() : void
+ 跳ね返りチェック() : void
+ データをセットする() : void

図 8

図 8 では、次の点が変わっているを確認せよ。

- カンバスから円クラス関連の属性・メソッドがなくなった
  - 円は四角の中に隠蔽された、ということもできる
- 四角クラスが円クラスのインスタンスを持つ
  ということは、以下の点が暗黙に了解される
  -「初期化する ()」の中では円のインスタンスも初期化する
  -「描画する ()」の中では円も同時に描画する
- 円クラスに「データをセットする ()」メソッドが追加された
  - 四角から円のデータ属性を書き換えるためのインターフェイスが必要になるため

コード例

以前のコードにほんの数行の変更を加えるだけで、これらの動きが実現できた。クラスにすると仕様変更が楽。これを最初にやっていたように、全ての図形を配列にして処理していたとしたらどうだろうか？

4.2 part-of の関係

4.2.1 （もう一度）共通点と相違点

ここまでコーディングしてきた「四角クラス」と「円クラス」のソース・コードを見比べてみよう。ほとんど同じコードになっている。それはそうだ。四角も円も座標や大きさなど持っている属性は同じだし、図形を描く操作も同じだ。

違うのは具体的に四角を描いたり、円を描いたりする、最下層の proccessing の関数の違いだけ。メソッドでいうと「`drowPrim()`」の中身がちょっと違うだけだ。

それなら、両方に共通する部分をまとめて書けないか？ということで、もう一段階上の抽象的なクラスを新たに作ることを考える。

（複数あるオブジェクトの共通点を抜き出し、さらに上位の概念でまとめなおすことを「**抽象化**」という。その逆は「**具体化**」だ。）

四角と円の上位にある概念はなんだろう？

かくして「図形クラス」というものが、四角クラスと円クラスの上位概念として見えてくる。

四角と円は図形の一部である（四角 and 円 are part of 図形）

というわけだ。

コード例

4.3 継承

クラス図にすると、図 9 のようになる。四角クラスと円クラスが抽象化されて「図形クラス」が出来上がっているのが分かるだろう。これを「**汎化**」という。

親クラスである「図形クラス」に、共通する属性と操作を組み込む。子クラスでは親クラスの属性と操作を「**継承**」するので、改めて書く必要はない。書いてなくても、それらは子クラスにも存在するのだ。

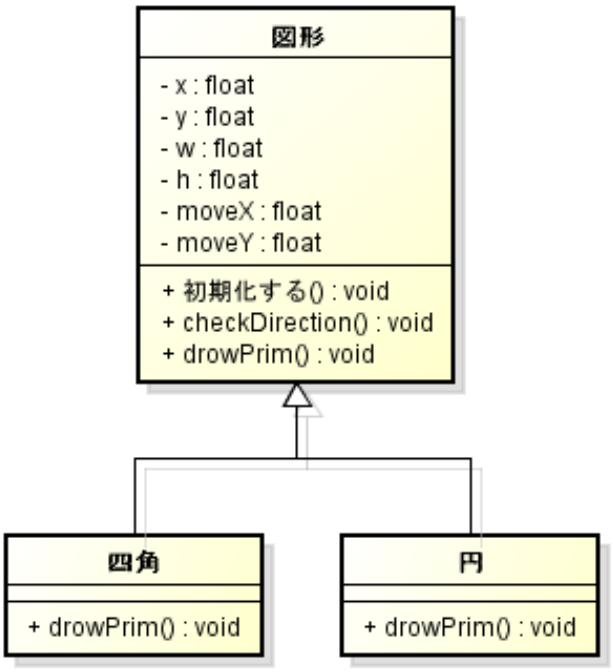

図 9

4.4　多態性

一方、親クラスと異なる部分は、あらためて子クラスで定義することができる。

たとえば、図 9 の「`drowPrim()`」は四角クラスと円クラスで動きが違っているので、それぞれで中身を書く。親クラスの「`drowPrim()`」を子クラスで上書きして使うことになるので「**オーバーライド**」という。

すると「`drowPrim()`」というメソッドは同じ名前でも、親クラスとそれを継承した子クラスとで振る舞いが違うことになる。

同じ名前なのに振る舞いが違うことを「**多態性（ポリモーフィズム）**」という。

C 言語などでは同じ名前の関数が存在してはいけないので、このような場合次のように関数名を変更して混在させることになる。

- drawPrimBox()：四角を描く関数
- drawPrimCiecle()：円を描く関数

これでも悪くはないが、「図形を描く」という、より抽象度の高いレベルで扱えることはシステムの設計には楽だ。

5　抽象クラス

「abstract」という指定子があってね...

「インターフェイス」とか「フレームワーク」とかに進んでいくんだけど、今のところはここまでにしておこうかな。

次のコード例は抽象クラスを使って書いてみた。

大きさと色も変化するように拡張してみた。親クラスの中だけを書き換えればいいので楽だった。こういう拡張性や保守性が、オブジェクト指向の目的である。コード例

『オブジェクト指向でなぜつくるのか』[3] には「クラスの効能」として次の 3 点を挙げている。

> クラスは「まとめて、隠して、たくさん作る」仕組み
> 1. サブルーチンと変数を「まとめる」
> 2. クラスの内部だけで使う変数やサブルーチンを「隠す」
> 3. 1 つのクラスからインスタンスを「たくさん作る」

ここまでの実習で以上のことがわかってもらえたら幸いである。

参考文献

[1] Tucker『憂鬱なプログラマのためのオブジェクト指向開発講座』 翔泳社, 1998

[2] ε π ι σ τ η μ η『ε π ι σ τ η μ η のオブジェクト指向的日常』 翔泳社, 1999

[3] 平澤章『オブジェクト指向でなぜつくるのか 第 2 版』日経 BP 社, 2011

[4] 平澤章『UML モデリングレッスン』日経 BP 社, 2013