

JAIST x IMLプログラミング講習会

OOP1:「オブジェクト指向入門」

2022年06月26日

佐藤俊樹@JAIST

本日の内容

1. オブジェクト指向プログラミング入門
 - クラスを使ったブロック崩しのキャラクターの再設計

オブジェクト指向プログラミング入門

Object Oriented Programming(OOP)

オブジェクト指向プログラミング(OOP)

- オブジェクト指向プログラミングとは？
「Object Oriented Programming(OOP)」
 1. プログラムに登場する「データ」や「機能(処理)」を抽象化し、それらを「オブジェクト」としてとらえる
 - 例え1) オブジェクトはプログラムを実現するためのパーツのようなもの
 2. 「オブジェクト」同士を組み合わせてプログラムを完成させる
 - 例え1) パーツを組み合わせることで、目的の処理を実現する

オブジェクト指向で大切なこと(後述)

- オブジェクトは**抽象度**に応じた階層関係(親子関係)で記述する

- 抽象度の高いオブジェクト

- 普遍的な概念だけを記述したオブジェクト
 - 普遍的な概念は変更されにくい

- 抽象度の低い(つまり具体度の高い)オブジェクト

- 具体的な実装が含まれるオブジェクト
 - 具体的な実装はこまごと変更されやすい

まだ意味が
わからなくてもOK！

- なるべく「抽象度の高いオブジェクト」同士だけで処理を記述する

- 「具体的な部分」は後に変更されやすい

- 変更されるとプログラムを書き直さないといけない
 - 変更されやすいオブジェクトはなるべく使わないようにプログラムを書きたい
 - そのために抽象度の高いオブジェクトのみを用いるようにする

オブジェクト指向が前提のプログラム言語

- 例
 - Java
 - ProcessingはJavaがベースとなっている
 - オブジェクト指向でも書ける(厳密にはちょっと緩い感じ)
 - C++
 - Cはオブジェクト指向言語ではない
 - C#
 - Unityを使うにはC#が必要
 - ほか多数
 - ほぼすべての言語がオブジェクト指向で書ける
- オブジェクト指向プログラミングは現代のプログラマの常識

実際にやってみましょう

- 理屈で説明すると難しいので
- ブロック崩しの例で感覚をつかんでください

(復習)ブロック崩しの3種類のキャラクター

• 「バー」

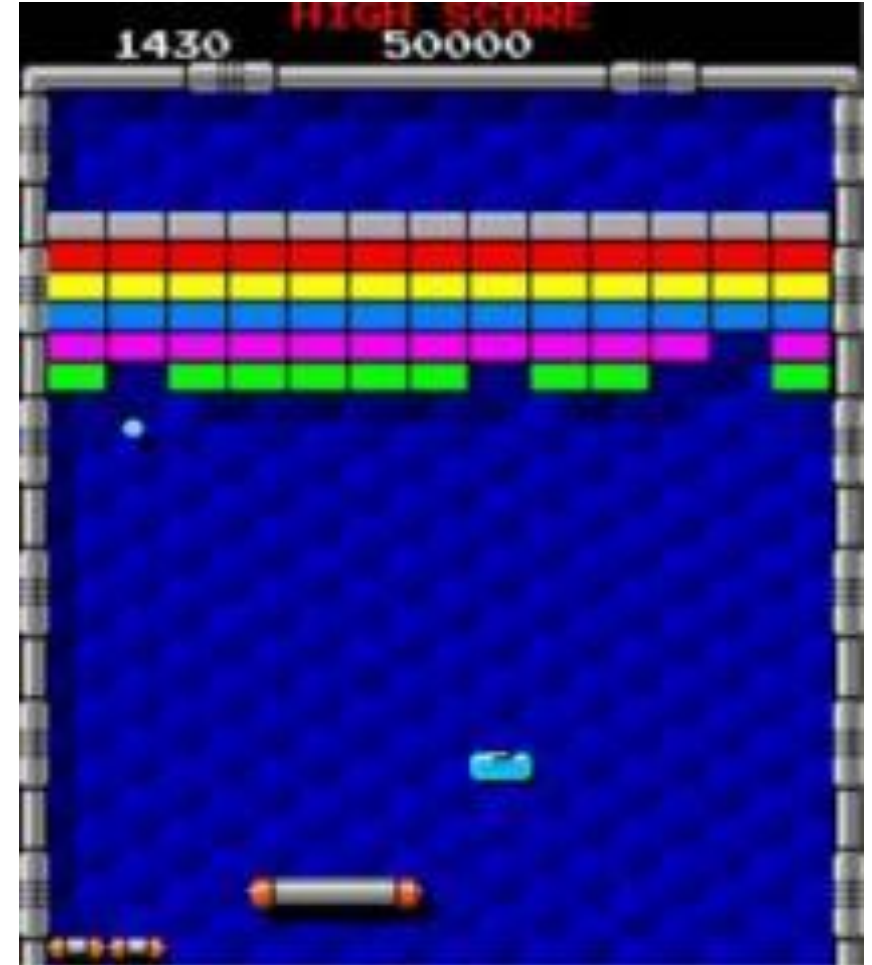
- ボールを跳ね返すことができる
- プレイヤーが左右に動かせる

• 「ボール」

- ブロックやバーに当たると跳ね返る
- 下に落ちたら負け

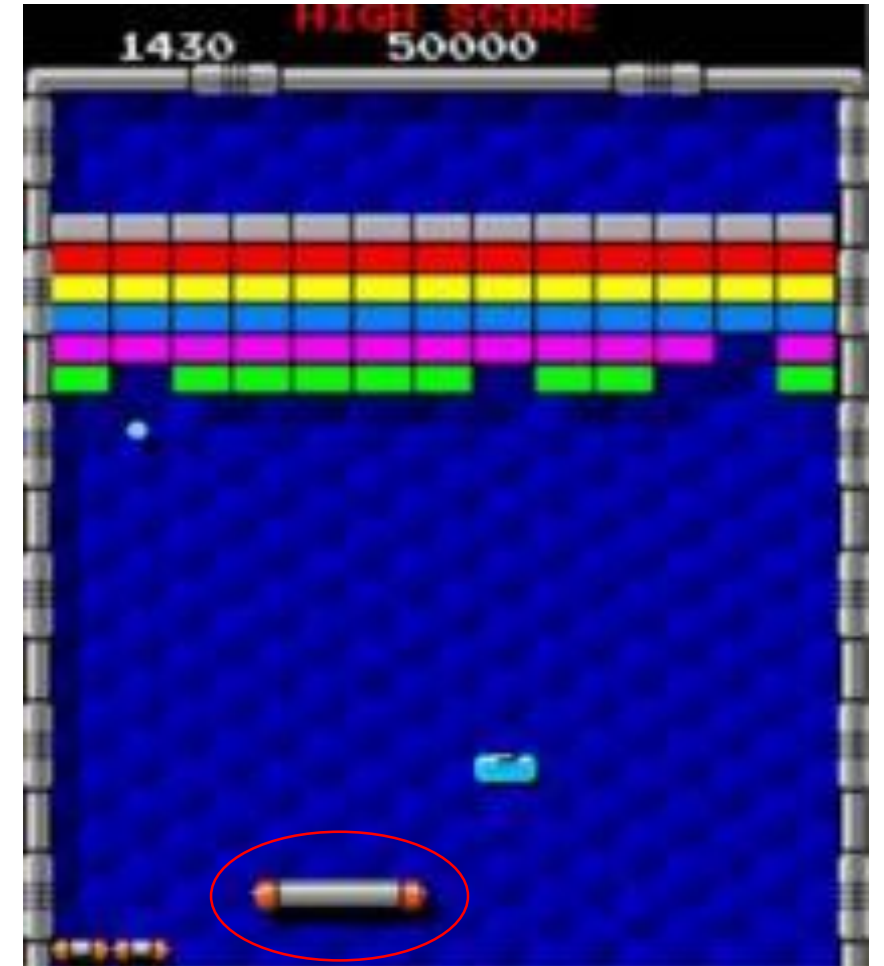
• 「ブロック」

- 画面上に複数個配置される障害物
- ボールが当たると消える
- 全部消すとゲームクリア



バーの要素の抽出

- バーに必要な情報
 - 位置情報
 - 速度情報
 - バーのサイズ情報
- バーが行うこと
 1. 1フレーム分移動する
 - ユーザが動かす
 2. 自分の位置にバーの絵(図形や画像)を表示



バーの要素の抽出

- バーに必要な情報

- 位置情報
- 速度情報
- バーのサイズ情報

```
float barX, barY;  
float barVX;  
float width, height;
```

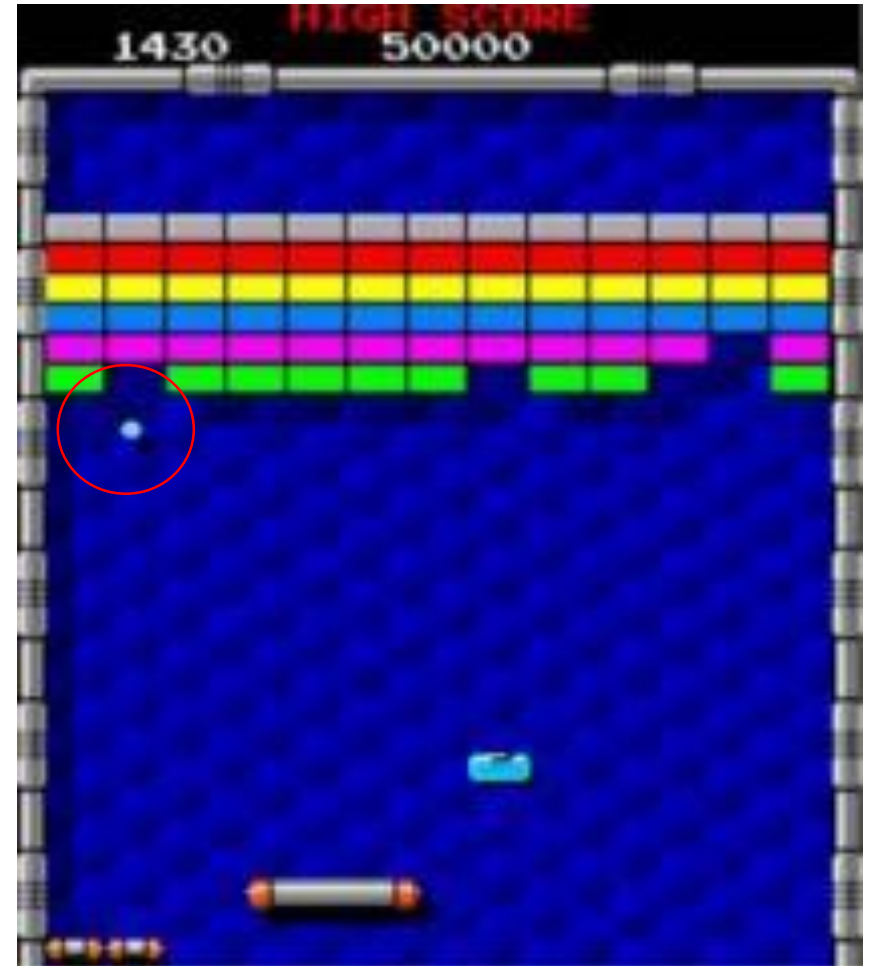
- バーが行うこと

1. 1フレーム分移動する
 - ユーザが動かす
2. 自分の位置にバーの絵を表示

```
void moveBar(){...}  
void drawBar(){...}
```

ボールの要素の抽出

- ボールに必要な情報
 - 位置情報
 - 速度情報
 - 大きさ (半径など)
- ボールが行うこと
 1. 1フレーム分移動する
 - 壁やブロック、バーに当たると跳ね返る
 2. 自分の位置にボールの絵を表示



ボールの要素の抽出

- ボールに必要な情報

- 位置情報
- 速度情報
- 大きさ (半径など)

```
float ballX, ballY;  
float ballVX, ballVY;  
float radius;
```

- ボールが行うこと

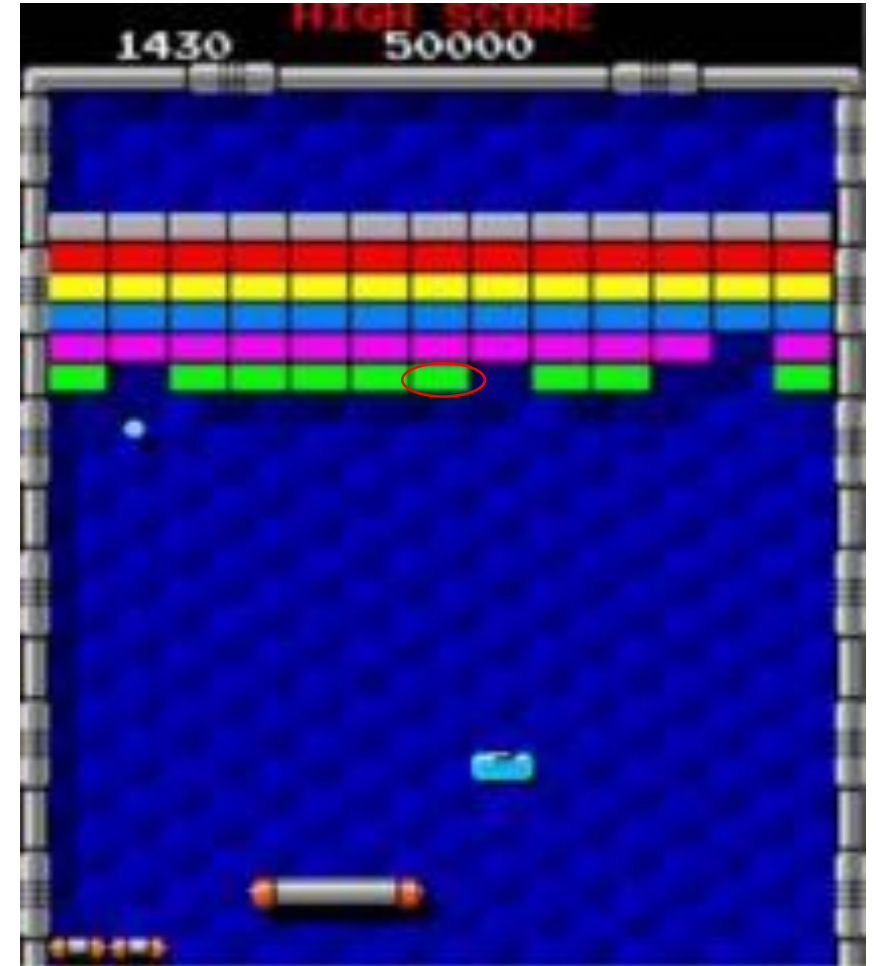
1. 1フレーム分移動する
 - 壁やブロック、バーに当たると跳ね返る
2. 自分の位置にボールの絵を表示

```
void moveBall(){...}
```

```
void drawBall(){...}
```

ブロックの要素の抽出

- ブロックに必要な情報
 - 位置情報
 - ブロックのサイズ情報
 - 消えたかどうか、のフラグ
- ブロックが行うこと
 - ボールが当たったら消える
 - 自分の位置にブロックの絵を表示



ブロックの要素の抽出

- ブロックに必要な情報

- 位置情報
- ブロックのサイズ情報
- 消えたかどうか、のフラグ

```
float blockX, blockY;  
float blockWidth, blockHeight;  
boolean hitFlag;
```

- ブロックが行うこと

- ボールが当たったら消える
- 自分の位置にブロックの絵を表示

```
void moveBlocks(){...}  
void drawBlocks(){...}
```

ここまではこれまでと同じ！

1. 「登場人物」を洗い出す

- ブロック崩しの場合: 「バー」、「ボール」そして「ブロック」

2. 登場人物の持つ「情報」の洗い出し

- それらを「変数」として記述してきた

3. 登場人物の持つ「仕事」の洗い出し

- それらを「関数」として記述してきた

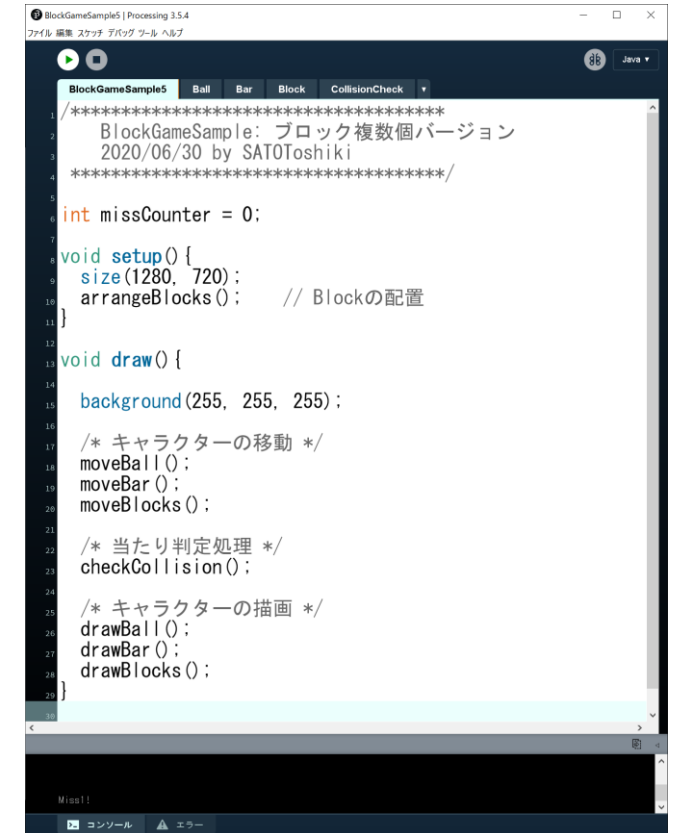
プログラムを見やすくするために
「タブ」に分けてソースコードを
書いてきた

ここからが新しいこと

- 以上の登場人物を「**クラス**」というもので記述する！
 - オブジェクト指向プログラミングでは**クラスによりオブジェクトを定義する**
- 「クラス(class)」とは？
 - **オブジェクトの設計図**のようなもの
 - クラスがあれば、オブジェクトが作れる
 - イメージ的には、**クラスは「変数の型」**のようなもの

実際にやってみましょう

- 理屈で説明すると難しいので
- ブロック崩しの例で感覚をつかんでください
- 準備: ブロック崩しのソースコードを手元に用意
 - バー、ボール、ブロック(複数個)まで実装されたもの
 - なければダウンロードしてください



```
BlockGameSample5 | Processing 3.5.4
ファイル 編集 スケッチ デバッグ ツール ヘルプ

BlockGameSample5  Ball  Bar  Block  CollisionCheck

1  /**
2   * BlockGameSample: ブロック複数個バージョン
3   * 2020/06/30 by SATOToshiki
4   */
5
6  int missCounter = 0;
7
8  void setup() {
9    size(1280, 720);
10   arrangeBlocks(); // Blockの配置
11 }
12
13 void draw() {
14   background(255, 255, 255);
15
16   /* キャラクターの移動 */
17   moveBall();
18   moveBar();
19   moveBlocks();
20
21   /* 当たり判定処理 */
22   checkCollision();
23
24   /* キャラクターの描画 */
25   drawBall();
26   drawBar();
27   drawBlocks();
28 }

Miss!!
コンソール エラー
```

ブロック崩しゲームの再設計

「クラス」を使って登場するキャラクターを設計しなおしてみる

バーの元のソースコード

```
/* バーの変数 */
```

```
float barX = 500.0f;
```

```
float barY = 600.0f;
```

```
float barVX = 15.0f;
```

```
float barWidth = 200.0f;
```

```
float barHeight = 50.0f;
```

```
/* バーの描画 */
```

```
void drawBar(){
```

```
    rect(barX, barY, barWidth, barHeight);
```

```
}
```

```
/* バーの移動 */
```

```
void moveBar(){
```

```
    if ( keyPressed ){
```

```
        if ( keyCode == RIGHT ){
```

```
            barX = barX + barVX;
```

```
        }else if ( keyCode == LEFT ){
```

```
            barX = barX - barVX;
```

```
        }
```

```
    }
```

```
}
```

バーのコードを下記のように書き換えてみよう

```
class Bar{  
    public float barX = 500, barY = 600;  
    public float barVX = 15;  
    public float barWidth = 200;  
    public float barHeight = 50;  
  
    public void moveBar(){...}  
    public void drawBar(){...}  
}
```



これが「Barクラス」！

変更箇所の説明

クラスの名前(クラスの名前は大文字で始めよう)

```
class Bar{  
    public float barX = 500, barY = 600.0f;  
    public float barVX = 15;  
    public float barWidth = 200.0f;  
    public float barHeight = 50.0f;  
  
    public void moveBar(){...}  
    public void drawBar(){...}  
}
```

- ・ class {...}の波カッコ「{}」の中に登場人物の変数と関数を並べる。
- ・ 変数・関数の頭にpublicをつける
 - 今は何も考えずにpubliと書くだけでOK

ここはこれまでと同じ

moveBar()とdrawBar()の中身

```
public void moveBar(){  
    if ( keyPressed ){  
        if ( keyCode == RIGHT ){  
            barX = barX + barVX;  
        }else if ( keyCode == LEFT ){  
            barX = barX - barVX;  
        }  
    }  
}
```

```
public void drawBar(){  
    rect(barX, barY, barWidth, barHeight);  
}
```

これらの関数の中身は皆さん自身の
ブロック崩しのソースコードをそのまま
使ってOKです。

これで「Barクラス」ができた！

- クラスはオブジェクトの設計図
 - この「Barクラス」があれば、オブジェクトを作ることができる！

ボールの元のソースコード

```
float ballX = 500.0f;  
float ballY = 100.0f;  
float ballVX = 5.0f;  
float ballVY = 5.0f;  
float ballRadius = 25.0f;
```

```
/* ボールの描画 */  
void drawBall(){  
    ellipse(ballX, ballY, ballRadius * 2, ballRadius * 2);  
} // drawBall
```

```
/* ボールの移動 */  
void moveBall(){  
  
    ballX = ballX + ballVX;  
    ballY = ballY + ballVY;  
  
    /* 壁での跳ね返し */  
    if ( ballX < 0 || ballX > width ){  
        ballVX = -ballVX;  
    }  
  
    /* ミス判定 */  
    if ( ballY < 0 ){  
        ballVY = -ballVY;  
    }else if ( ballY > height + 300 ){  
        missCounter++;  
        ballY = 100;  
        ballX = width / 2;  
        println("Miss" + missCounter + "!");  
    }  
} // moveBall
```


「ボール」についても同様に

```
class Ball{  
    public float ballX = 500;  
    public float ballY = 100;  
    public float ballVX = 5.0f;  
    public float ballVY = 5.0f;  
    public float ballRadius = 25;  
  
    public void moveBall(){...}  
    public void drawBall(){...}  
}
```



これが「Ballクラス」！

変更箇所の説明

クラスの名前(クラスの名前は大文字で始めよう)

```
class Ball{  
    public float ballX = 500;  
    public float ballY = 100;  
    public float ballVX = 5.0f;  
    public float ballVY = 5.0f;  
    public float ballRadius = 25;  
  
    public void moveBall(){...}  
    public void drawBall(){...}  
}
```

- ・ class {...}の波カッコ「{}」の中に登場人物の変数と関数を並べる。
- ・ 変数・関数の頭にpublicをつける
 - 今は何も考えずにpubliと書くだけでOK

ここはこれまでと同じ

moveBall()とdrawBall()の中身

```
/* ボールの動き */
```

```
public void moveBall(){
```

```
    ballX = ballX + ballVX;
```

```
    ballY = ballY + ballVY;
```

```
/* 壁での跳ね返し */
```

```
if ( ballX < 0 || ballX > width ){
```

```
    ballVX = -ballVX;
```

```
}
```

```
/* ミス判定 */
```

```
if ( ballY < 0 ){
```

```
    ballVY = -ballVY;
```

```
}else if ( ballY > height + 300 ){
```

```
    missCounter++;
```

```
    ballY = 100;
```

```
    ballX = width / 2;
```

```
    println("Miss" + missCounter + "!");
```

```
}
```

```
} // moveBall
```

```
/* ボールの描画 */
```

```
public void drawBall(){
```

```
    ellipse(ballX, ballY, ballRadius * 2, ballRadius * 2);
```

```
} // drawBall
```

これで「Ballクラス」ができた！

- クラスはオブジェクトの設計図
 - この「Ballクラス」があれば、オブジェクトを作ることができる！

ブロックの元のソースコード

/* ブロックの変数 */

```
final int MAX_BLOCKS = 100;
float[] blockX = new float[MAX_BLOCKS];
float[] blockY = new float[MAX_BLOCKS];
float[] blockWidth = new float[MAX_BLOCKS];
float[] blockHeight = new float[MAX_BLOCKS];
boolean[] blockHitFlag = new boolean[MAX_BLOCKS];
```

```
final int BLOCK_ROWS = 12;
final int BLOCK_GAP = 6;
```

/* ブロックの移動 */

```
void moveBlocks() {
    // 今のところブロックは動かない
} // moveBlocks
```

/* ブロックの描画 */

```
void drawBlocks() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        if (blockHitFlag[i] == false) {
            rect(blockX[i], blockY[i], blockWidth[i], blockHeight[i]);
        }
    }
}
```

/* ブロックの初期化・配置 */

```
void arrangeBlocks() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        blockWidth[i] = 100.0f;
        blockHeight[i] = 10.0f;
        blockHitFlag[i] = false;
        blockX[i] = BLOCK_GAP + i % BLOCK_ROWS * (blockWidth[i] + BLOCK_GAP);
        blockY[i] = BLOCK_GAP + i / BLOCK_ROWS * (blockHeight[i] + BLOCK_GAP);
    }
} // arrangeBlocks
```

「ブロック」クラスを書く

```
class Block{  
    public float blockX;  
    public float blockY;  
    public float blockWidth;  
    public float blockHeight;  
    public boolean blockHitFlag;  
  
    public void moveBlocks() {...}  
    public void drawBlocks() {...}  
  
    final int BLOCK_ROWS = 12;  
    final int BLOCK_GAP = 6;  
    public void arrangeBlocks() {...}  
}
```

- 「ブロック」クラスは「ブロック単体」の設計図
- ブロックが沢山必要だからといって
ブロック単体の設計に複数の変数はいらない

- 「ブロック」の配置は「ブロックを並べる人」が決めること
- ブロックの配置にまつわる変数・関数は
ブロック自体の設計図には不要(・・・と考えたとする)

変更箇所 of 解説

← クラスの名前(クラスの名前は大文字で始めよう)

```
class Block{  
    public float blockX;  
    public float blockY;  
    public float blockWidth;  
    public float blockHeight;  
    public boolean blockHitFlag;  
  
    public void moveBlock() {...}  
    public void drawBlock() {...}  
}
```

- class {...}の波カッコ「{}」の中に登場人物の変数と関数を並べる。
- 変数・関数の頭にpublicをつける
 - 今は何も考えずにpubliと書くだけでOK

moveBlock()とdrawBlock()の中身

```
public void moveBlock(){  
    // ブロックは動かない  
}
```

```
public void drawBlock (){  
    if (blockHitFlag == false) {  
        rect(blockX, blockY, blockWidth, blockHeight);  
    }  
}
```

- **arrangeBlocks()はどこに書くのか？**
 - **今回は、少なくともブロックの設計図の外でいい(…と考えた)**

これで各登場人物のクラス設計は終わり

- 次のステップ
 - クラス(設計図)から「オブジェクト(実体)」を作る
 - 作ったオブジェクトを使ってゲームを動かす
- これらの処理はクラスの外のメインのソースコードに書いていく

BarクラスからBarオブジェクトを作る

Bar bar;

```
void setup(){  
    bar = new Bar();  
}
```

この瞬間に
「barさん」が誕生！

```
void draw(){  
    ...  
}
```

- 手順:

1. 「Bar」型の変数を宣言する

- 変数名は自由に付けられる(変数の宣言と同様)
- 左の例では「bar」という名前になっている

2. 宣言した変数に **new** 演算子を使って
Barオブジェクトを新規作成し、代入する

- **オブジェクトはクラスからnewされることで
はじめて実体化され使用可能になる！！**
 - newされるまでは実体が存在しない
 - newされないままの変数は、ただの箱(中は空っぽ)

作ったBarオブジェクトに「動いてもらう」

```
Bar bar;
```

```
void setup(){  
    bar = new Bar();  
}
```

```
void draw(){  
    bar.moveBar();  
    bar.drawBar();  
}
```

- 生成したオブジェクトの持つ変数・関数は変数名にドット演算子「.」をつけることでアクセス可能(呼び出し可能)になる

- 「bar.moveBar();」により、barオブジェクトの持つmoveBar()関数が実行される

- 「bar.drawbar();」により、barオブジェクトの持つdrawBar()関数が実行される

この時のイメージ

```
Bar bar;
```

```
void setup(){  
    bar = new Bar();  
}
```

- ここで「bar」さんが誕生した

```
void draw(){  
    bar.moveBar();  
    bar.drawBar();  
}
```

- 「barさん、moveBar()お願いします！」

- 「barさん、drawBar()お願いします！」

同様にBallオブジェクトについても

```
Bar bar;  
Ball ball;  
  
void setup(){  
    bar = new Bar();  
    ball = new Ball();  
}  
  
void draw(){  
    bar.moveBar();  
    ball.moveBall();  
    bar.drawBar();  
    ball.drawBall();  
}
```

- 生成したオブジェクトの持つ変数・関数は変数名にドット演算子「.」をつけることでアクセス可能(呼び出し可能)になる
 - 「ball.moveBall();」により、ballオブジェクトの持つmoveBall()関数が実行される
 - 「ball.drawBall();」により、ballオブジェクトの持つdrawBall()関数が実行される

Blockについては、ここで複数個誕生させる

```
Bar bar;
Ball ball;
int MAX_BLOCKS = 100;
Block[] blocks = new Block[MAX_BLOCKS];

void setup(){
    bar = new Bar();
    ball = new Ball();
    for (int i = 0; i < MAX_BLOCKS; i++){
        blocks[i] = new Block();
    }
}
```

```
void draw(){
    bar.moveBar();
    ball.moveBall();
    bar.drawBar();
    ball.drawBall();
    for (int i = 0; i < MAX_BLOCKS; i++){
        blocks[i].drawBlock();
    }
}
```

なんで何度もnewをしているのか？

```
Bar bar;
```

```
Ball ball;
```

```
int MAX_BLOCKS = 100;
```

1回目

```
Block[] blocks = new Blocks[MAX_BLOCKS];
```

```
void setup(){
```

```
    bar = new Bar();
```

```
    ball = new Ball();
```

2回目

```
for (int i = 0; i < MAX_BLOCKS; i++){
```

```
    blocks[i] = new Block();
```

```
}
```

```
}
```

```
void draw(){
```

```
    bar.moveBar();
```

```
    ball.moveBall();
```

```
    bar.drawBar();
```

```
    ball.drawBall();
```

```
    for (int i = 0; i < MAX_BLOCKS; i++){
```

```
        blocks[i].drawBlock();
```

```
    }
```

```
}
```

なんで何度もnewをしているのか？

```
Bar bar;
```

```
Ball ball;
```

```
int MAX_BLOCKS = 100;
```

1回目

```
Block[] blocks = new Blocks[MAX_BLOCKS];
```

```
void setup(){
```

```
    bar = new Bar();
```

```
    ball = new Ball();
```

2回目

```
    for (int i = 0; i < MAX_BLOCKS; i++){
```

```
        blocks[i] = new Block();
```

```
    }
```

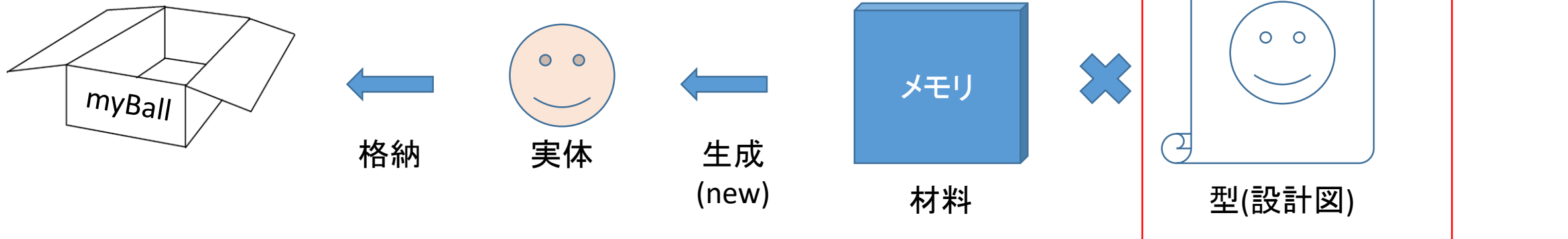
```
}
```

- 1回目のnewは、箱をつくるためのnew
 - 空箱を作って並べている感じ
 - 注意: この時点で箱の中身はまだ「空っぽ」である
- 2回目のnewは作った空の箱にBlockの実体を入れていくためのnew
 - 1個1個、並んだ箱全部にnewしていく
 - newすることで、中身が入る

クラスを使ってオブジェクトをnewするということ1

- クラスは設計図
- クラスを型として宣言した変数はただの箱
- クラスをnewして変数に代入することで、オブジェクトが誕生する

```
Ball myBall = new Ball();
```

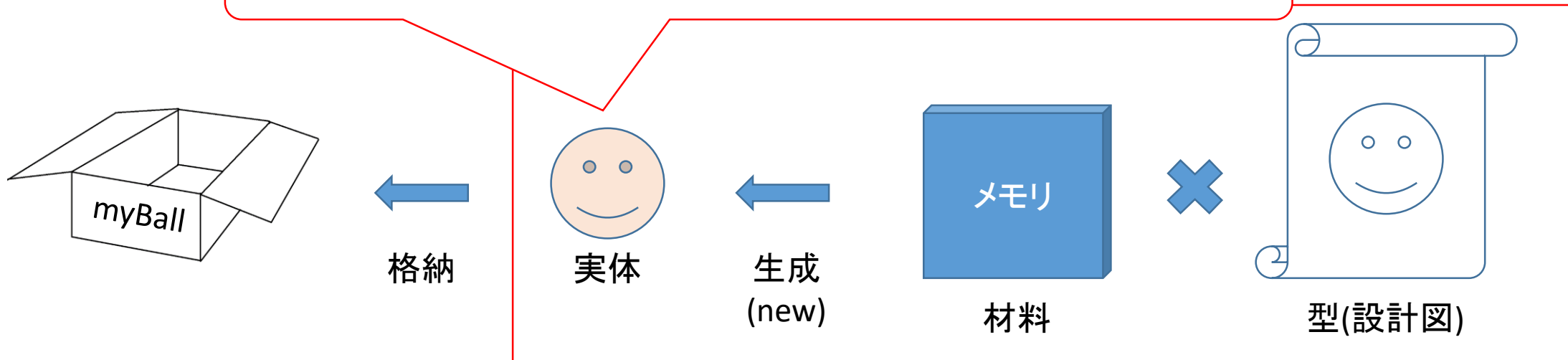


クラスを使ってオブジェクトをnewするということ2

- クラスは設計図
- クラスを型として宣言した変数はただの箱
- クラスをnewして変数に代入することで、オブジェクトが誕生する

```
Ball myBall = new Ball();
```

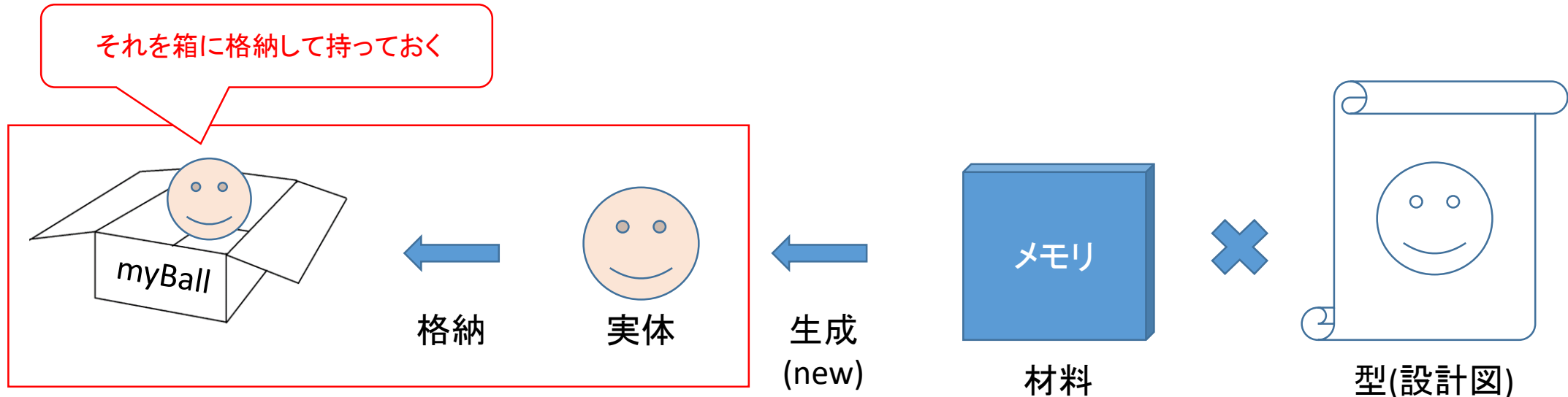
- 「newする」ことで、メモリ上にクラスを実体化させたオブジェクトが実体化する
- 「クッキー型(クラス)」で「生地(メモリ)」から「クッキー(オブジェクトの実体)」を作るイメージ



クラスを使ってオブジェクトをnewすること3

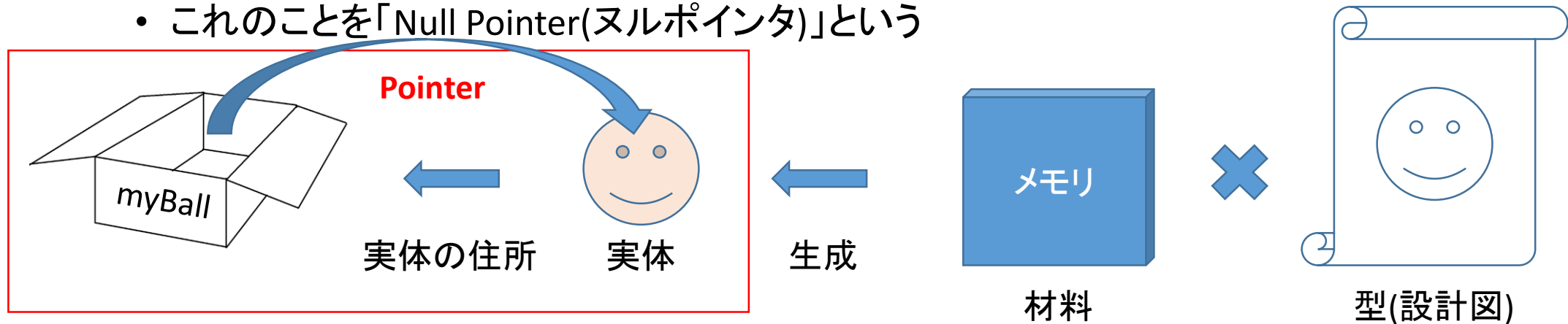
- クラスは設計図
- クラスを型として宣言した変数はただの箱
- クラスをnewして変数に代入することで、オブジェクトが誕生する

```
Ball myBall = new Ball();
```



実際、newによって箱には何が代入されるのか

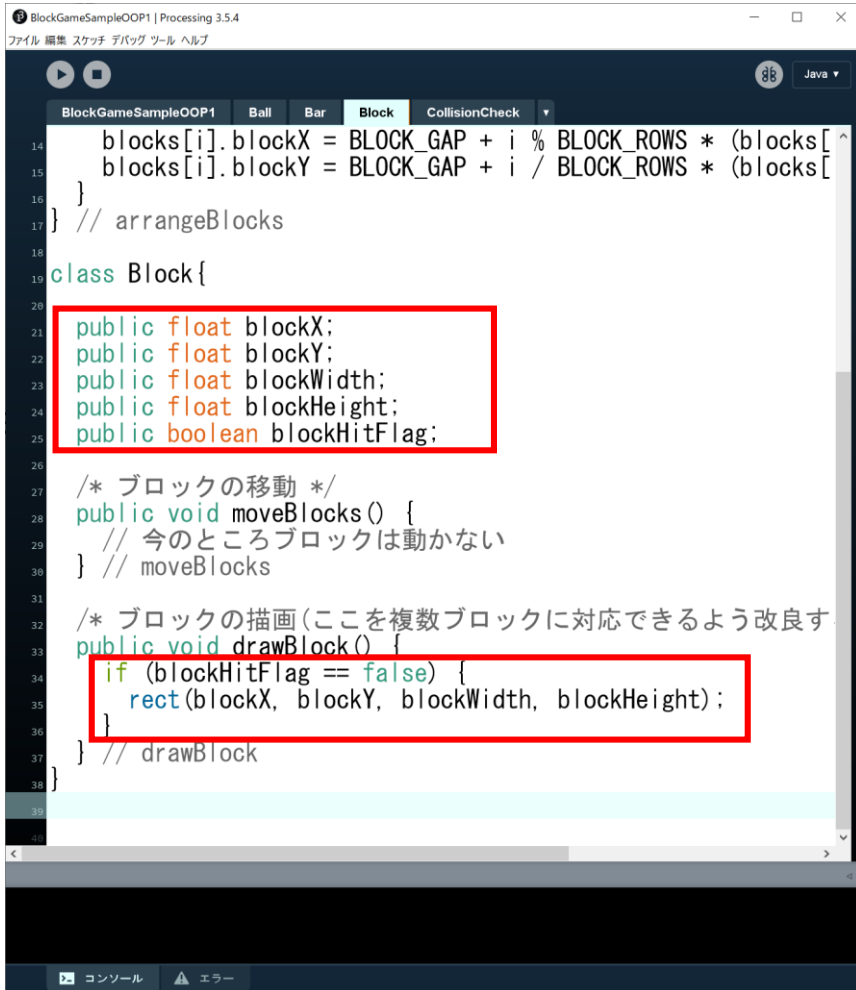
- newによって実体化したオブジェクトは・・・
 - 「メモリ空間上のどこか」に配置される
 - 広大な倉庫の中のどこに格納されるか、はその時々で変わる
(どこに置かれるか、なんてことは我々は気にしなくていい)
- newが返すのは、その配置された場所を示す「住所」
 - 「住所」とは、メモリ上のどこかの場所を指し示す値(アドレス、番地)
 - この住所の値を「参照」や「ポインタ(Pointer)」などという
- 宣言されただけの箱の中には、「どこも指し示していないポインタ」が入っている
 - このことを「Null Pointer(ヌルポインタ)」という



複数ブロック問題と配置問題はどうするの？

- 「複数ブロック問題」
 - Blockクラスはブロック自体の設計図
 - なのでブロック1個のことを考えればよい(という設計思想を今回は取った)
 - **なので、ブロックを増やしたいなら、必要な数だけnewすればいいこと**
- 「複数ブロックの配置問題」
 - そもそも「ブロックの設計」の外の問題
 - 「ブロック」自体が自分の配置について考える必要はない(という設計思想を今回は取った)
 - **なので、ブロックを配置する処理はクラスの外に別途書く必要がある**

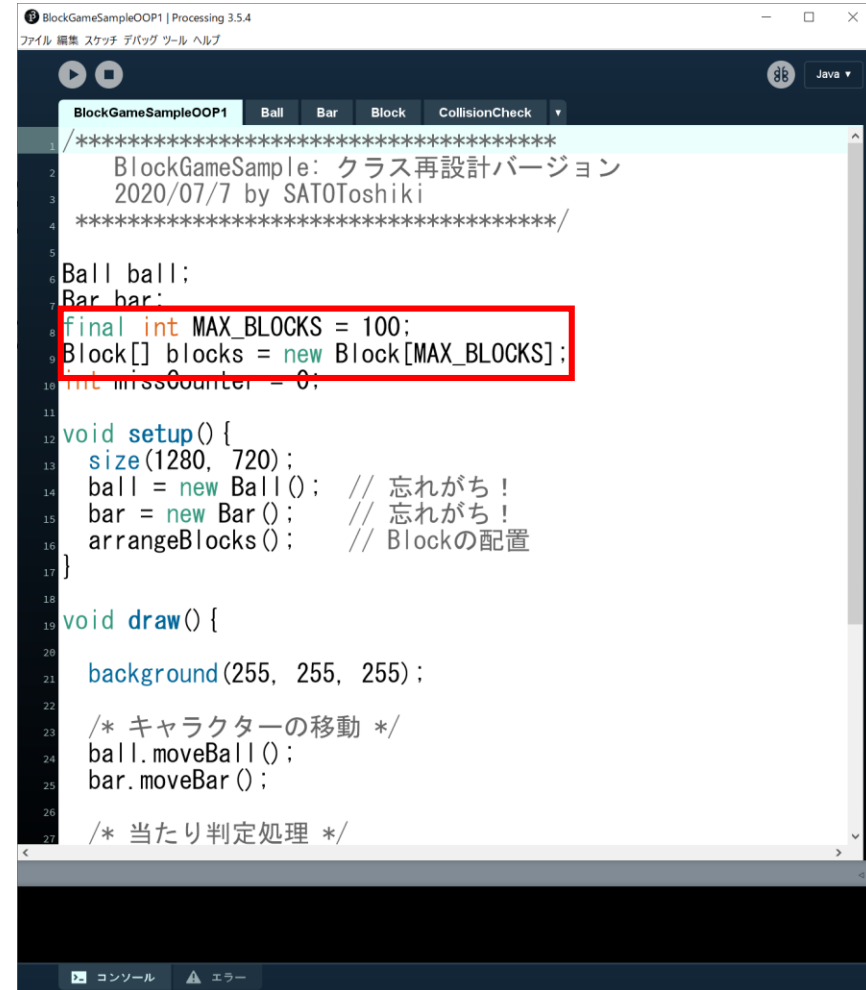
複数ブロック問題への対処



```
BlockGameSampleOOP1 | Processing 3.5.4
ファイル 編集 スケッチ デバッグ ツール ヘルプ

BlockGameSampleOOP1 Ball Bar Block CollisionCheck
14 blocks[i].blockX = BLOCK_GAP + i % BLOCK_ROWS * (blocks[
15 blocks[i].blockY = BLOCK_GAP + i / BLOCK_ROWS * (blocks[
16 }
17 } // arrangeBlocks
18
19 class Block{
20
21 public float blockX;
22 public float blockY;
23 public float blockWidth;
24 public float blockHeight;
25 public boolean blockHitFlag;
26
27 /* ブロックの移動 */
28 public void moveBlocks() {
29 // 今のところブロックは動かない
30 } // moveBlocks
31
32 /* ブロックの描画(ここを複数ブロックに対応できるよう改良す
33 public void drawBlock() {
34 if (blockHitFlag == false) {
35 rect(blockX, blockY, blockWidth, blockHeight);
36 }
37 } // drawBlock
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

クラスの中は単数でOK



```
BlockGameSampleOOP1 | Processing 3.5.4
ファイル 編集 スケッチ デバッグ ツール ヘルプ

BlockGameSampleOOP1 Ball Bar Block CollisionCheck
1 /* **** */
2 BlockGameSample: クラス再設計バージョン
3 2020/07/7 by SATOToshiki
4 /* **** */
5
6 Ball ball;
7 Bar bar;
8
9 final int MAX_BLOCKS = 100;
10 Block[] blocks = new Block[MAX_BLOCKS];
11 int missCounter = 0;
12
13 void setup() {
14 size(1280, 720);
15 ball = new Ball(); // 忘れがち!
16 bar = new Bar(); // 忘れがち!
17 arrangeBlocks(); // Blockの配置
18 }
19
20 void draw() {
21
22 background(255, 255, 255);
23
24 /* キャラクターの移動 */
25 ball.moveBall();
26 bar.moveBar();
27
28 /* 当たり判定処理 */
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

newするときに配列に格納

複数ブロックの配置問題への対処

```
BlockGameSampleOOP1 | Processing 3.5.4
ファイル 編集 スケッチ デバッグ ツール ヘルプ

BlockGameSampleOOP1 | Ball | Bar | Block | CollisionCheck

1 /******
2   Blockクラス
3   ******/
4
5 /* ブロックの初期化・配置を行う関数 */
6 final int BLOCK_ROWS = 12;
7 final int BLOCK_GAP = 6;
8 void arrangeBlocks() {
9   for (int i = 0; i < MAX_BLOCKS; i++) {
10     blocks[i] = new Block(); // 忘れがち!
11     blocks[i].blockWidth = 100.0f;
12     blocks[i].blockHeight = 10.0f;
13     blocks[i].blockHitFlag = false;
14     blocks[i].blockX = BLOCK_GAP + i % BLOCK_ROWS * (blocks[i].blockWidth + BLOCK_GAP);
15     blocks[i].blockY = BLOCK_GAP + i / BLOCK_ROWS * (blocks[i].blockHeight + BLOCK_GAP);
16   }
17 } // arrangeBlocks
18
19 class Block{
20
21   public float blockX;
22   public float blockY;
23   public float blockWidth;
24   public float blockHeight;
25   public boolean blockHitFlag;
26
27   /* ブロックの移動 */
```

ブロック自体が自分の配置を知っていて、自分で配置を決定する場合、配置処理は「ブロックの仕事」となるためBlockクラスの中を書くことになる。

- ・今回はそういう設計ではない

並べる処理はクラスの外に別途用意

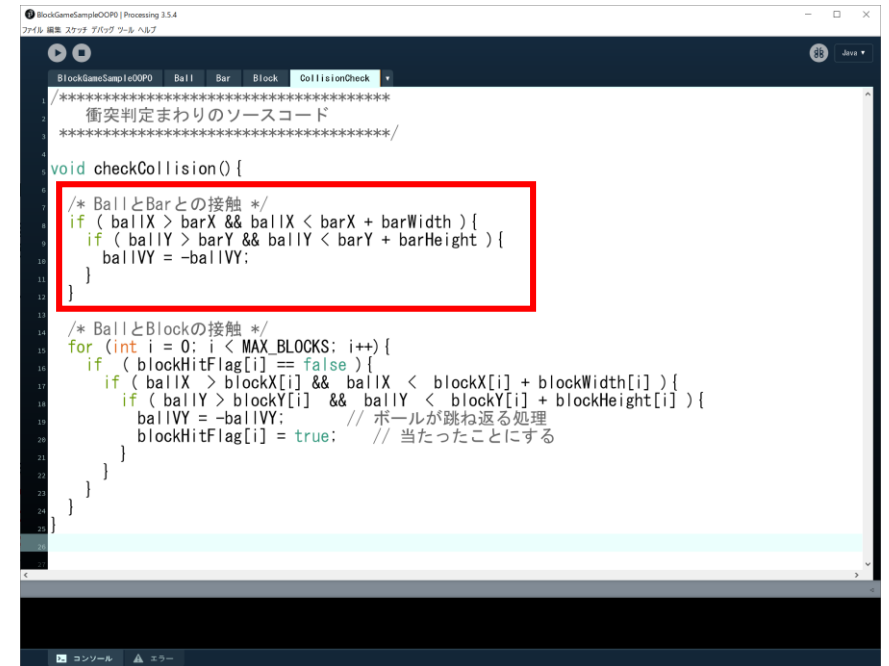
当たり判定処理の書き換え(Ball x Bar)

- Before

```
if ( ballX > barX && ballX < barX + barWidth ){  
    if ( ballY > barY && ballY < barY + barHeight ){  
        ballVY = -ballVY;          // ボールが跳ね返る処理  
    }  
}
```

- After

```
if ( ball.ballX > bar.barX && ball.ballX < bar.barX + bar.barWidth ){  
    if ( ball.ballY > bar.barY && ball.ballY < bar.barY + bar.barHeight ){  
        ball.ballVY = -ball.ballVY; // ボールが跳ね返る処理  
    }  
}
```



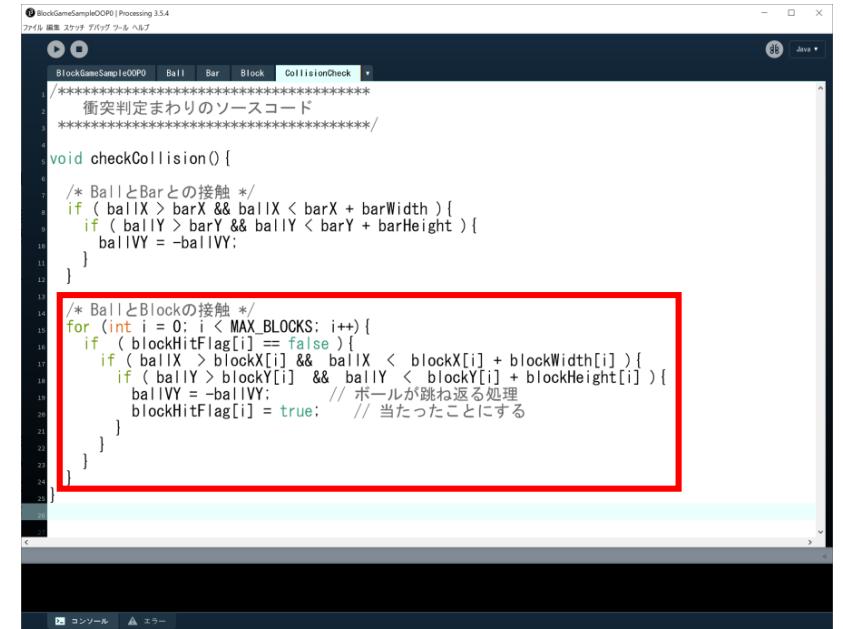
当たり判定処理の書き換え(Ball x Block)

- **Before**

```
for (int i = 0; i < MAX_BLOCKS; i++){
    if ( blockHitFlag[i] == false ){
        if ( ballX > blockX[i] && ballX < blockX[i] + blockWidth[i] ){
            if ( ballY > blockY[i] && ballY < blockY[i] + blockHeight[i] ){
                ballVY = -ballVY;           // ボールが跳ね返る処理
                blockHitFlag[i] = true;      // 当たったことにする
            }
        }
    }
}
```

- **After**

```
for (int i = 0; i < MAX_BLOCKS; i++){
    if ( blocks[i].blockHitFlag == false ){
        if ( ball.ballX > blocks[i].blockX && ball.ballX < blocks[i].blockX + blocks[i].blockWidth ){
            if ( ball.ballY > blocks[i].blockY && ball.ballY < blocks[i].blockY + blocks[i].blockHeight ){
                ball.ballVY = -ball.ballVY;           // ボールが跳ね返る処理
                blocks[i].blockHitFlag = true;          // 当たったことにする
            }
        }
    }
}
```



この接頭語、もういらなくない？

class Bar{

public float ~~bar~~X = 500, ~~bar~~Y = 600;

public float ~~bar~~VX = 15;

public float ~~bar~~Width = 200;

public float ~~bar~~Height = 50;

public void move~~Bar~~() {...}

public void draw~~Bar~~() {...}

}

ただ、Processingの
「width」と「height」と
ネーミングが被る
のは避けたい・・・

これもProcessingの「void draw() {...}」と
ネーミングが被る・・・

class Bar{

public float x = 500, y = 600;

public float vX = 15;

public float barWidth = 200;

public float barHeight = 50;

public void move() {...}

public void render() {...}

}

変数名がシンプルになった！

・・・ので、別の名前「render」にしてみた

この接頭語、もういらなくない？

class Ball{

public float ~~ball~~X = 500;

public float ~~ball~~Y = 100;

public float ~~ball~~VX = 5.0f;

public float ~~ball~~VY = 5.0f;

public float ~~ball~~Radius = 25;

public void move~~Ball~~() {...}

public void draw~~Ball~~() {...}

}

class Ball{

public float x = 500;

public float y = 100;

public float vX = 5.0f;

public float vY = 5.0f;

public float radius = 25;

public void move() {...}

public void render() {...}

}

この接頭語、もういらなくない？

```
class Block{  
    public float blockX;  
    public float blockY;  
    public float blockWidth;  
    public float blockHeight;  
    public boolean blockHitFlag;  
  
    public void moveBlock() {...}  
    public void drawBlock() {...}  
}
```

これも、Processingの
「width」と「height」と
ネーミングが被るのは
避けたい・・・

```
class Block{  
    public float x;  
    public float y;  
    public float blockWidth;  
    public float blockHeight;  
    public boolean hitFlag;  
  
    public void move() {...}  
    public void render() {...}  
}
```