

Java講座

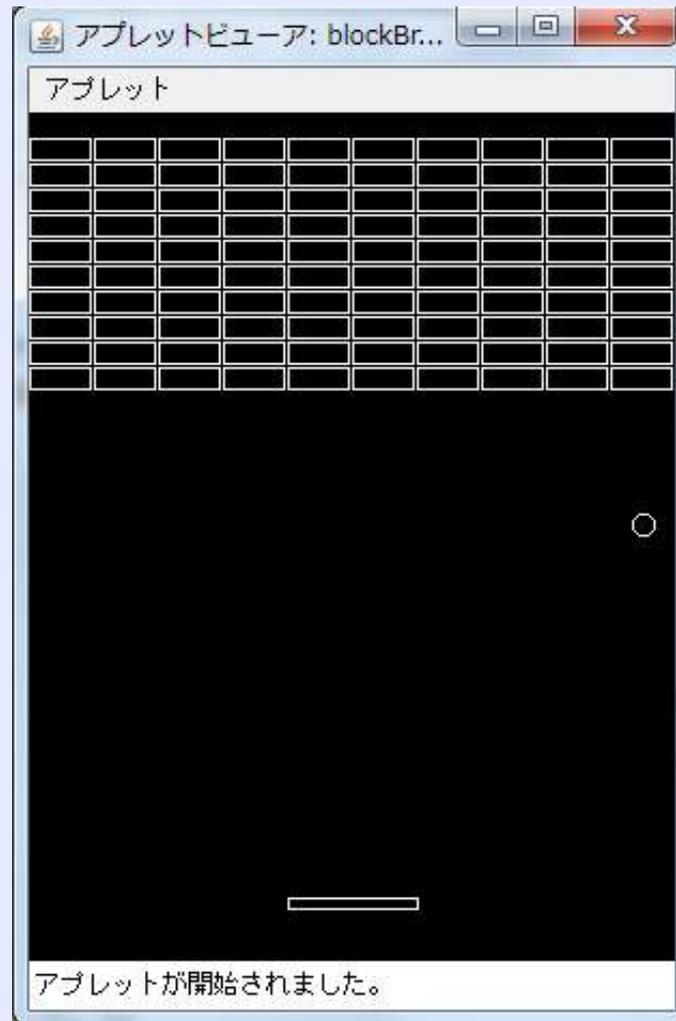
ブロック崩し

情報科学部コンピュータ科学科
2年 竹中 優

今回の内容

- ◆ 完成をイメージしよう
- ◆ ブロック崩しに必要なクラスを考えよう
 - ◆ クラス構造を考える
 - ◆ クラスを設計する
 - ◆ 実行してみる
- ◆ 当たり判定
- ◆ 実行クラスを完成させる

完成をイメージしよう



ブロック崩しに必要なクラスを考えよう

- ◆ とりあえず、ボールとバーとブロックを表すクラスが必要である。
- ◆ あとは、それらをまとめる実行クラス(Appletクラスのサブクラス)が必要である。
- ◆ クラス名は、それぞれBall, Bar, Block, Mainクラスで良いだろう。

クラス構造

既にあるクラス, インターフェイス

Appletクラス

Runnable
インターフェイス

KeyListener
インターフェイス

Mainクラス

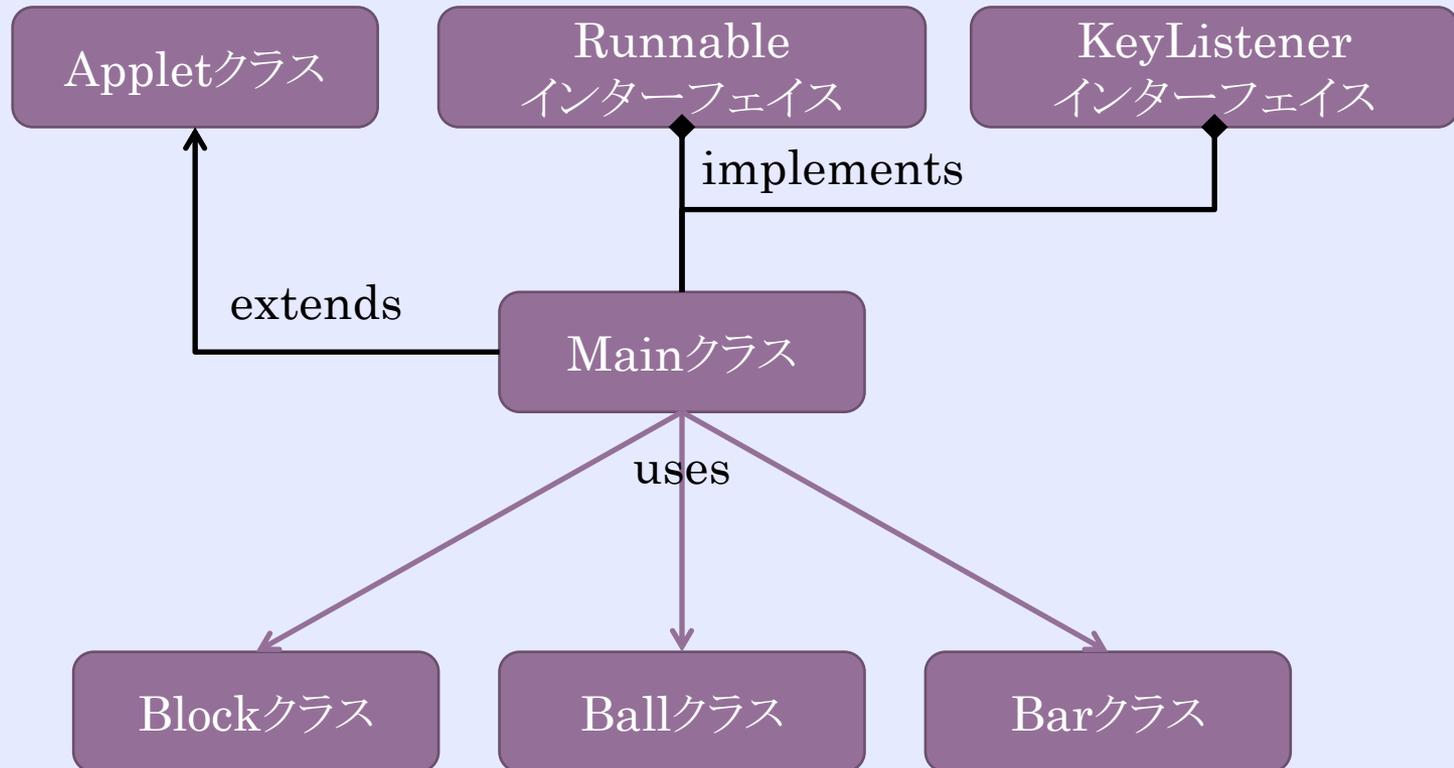
Blockクラス

Ballクラス

Barクラス

自分で定義するクラス

クラス構造



クラス設計 Mainクラス

Mainクラスについて

- ◆ スーパークラス: Applet
実装(implements)するインターフェイス:
KeyListener, Runnable
- ◆ オーバーライドするAppletのメソッド
public void init()
public void paint(Graphics g)
- ◆ 実装しなければならないメソッド
public void keyTyped(KeyEvent e)
public void keyPressed(KeyEvent e)
public void keyReleased(KeyEvent e)
public void run()

Mainクラスのフィールド

- ◆ フィールドは、
 - ◆ 再描画ごとに動くボール `Ball ball`
 - ◆ 左右矢印キーで動くバー `Bar bar`
 - ◆ 画面に配置されるブロックの2次元配列
`Block[][] stage`
 - ◆ 一定ミリ秒間隔で`repaint`メソッドを呼び、再描画するスレッド
`Thread repaintThread`
 - ◆ キーが押されているかのフラグ
`boolean leftKeyFlag, rightKeyFlag`
 - ◆ 描画するシーンを表す変数 `int scene`

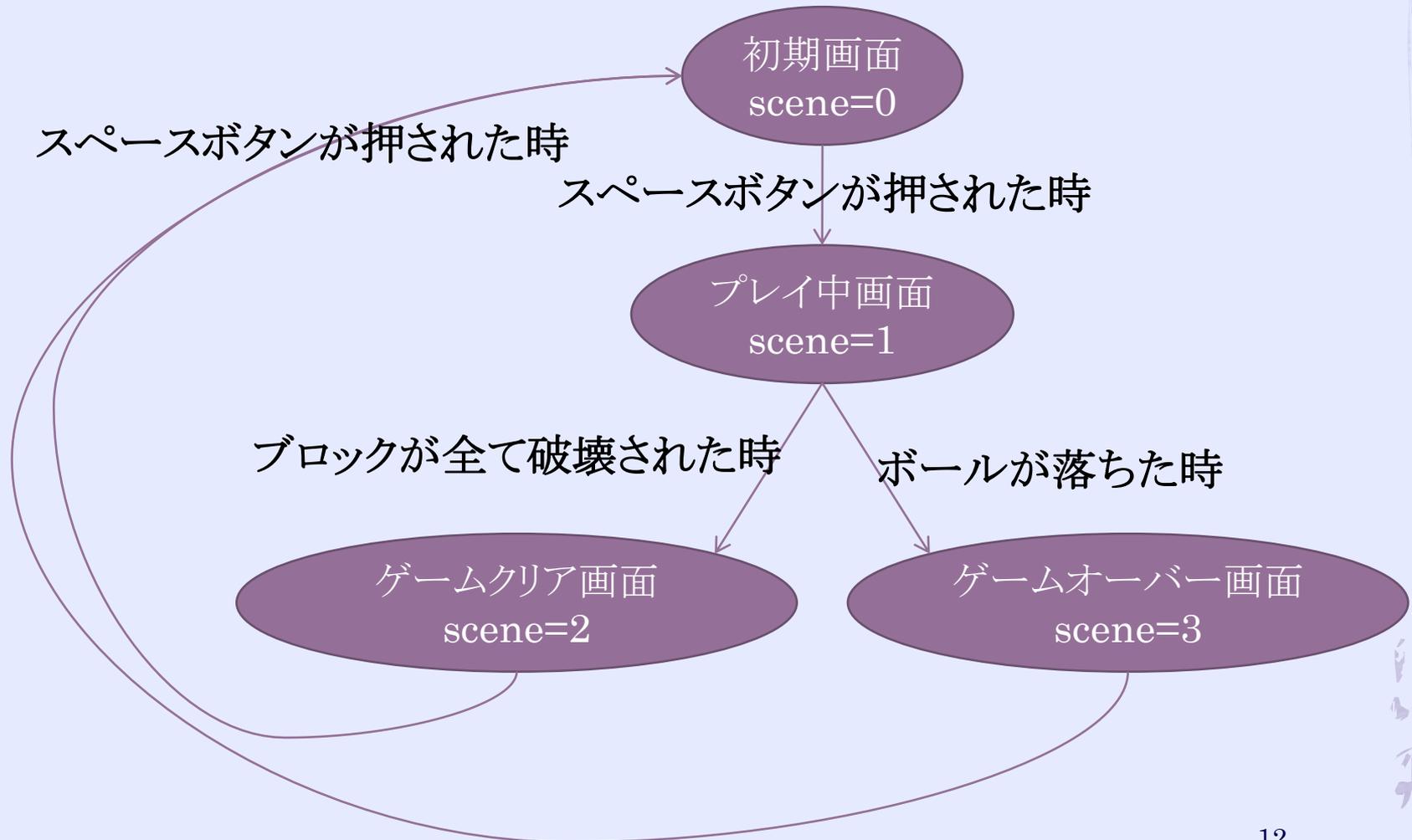
sceneフィールドについて

- ◆ sceneについて
 - 0:初期画面
 - 1:プレイ中画面
 - 2:ゲームクリア画面
 - 3:ゲームオーバー画面をそれぞれ表す。
- ◆ paint(Graphics g)メソッドの中でsceneの値に対応する画面を描画するメソッドを呼び出す。

paintメソッドについて

```
53 @Override
54 public void paint(Graphics g){
55     switch(scene){
56     case 0:
57         ready(g); //初期画面
58         break;
59     case 1:
60         playing(g); //プレイ中画面
61         break;
62     case 2:
63         gameclear(g); //ゲームクリア画面
64         break;
65     case 3:
66         gameover(g); //ゲームオーバー画面
67         break;
68     }
69 }
```

sceneフィールドのフローチャート



Mainクラスのメソッド

- ◆ メソッドは、
 - ◆ Appletクラスのメソッド
 - ◆ `init()`
 - ◆ `paint(Graphics g)`
 - ◆ `KeyListener`インターフェイスの抽象メソッド
 - ◆ `keyTyped(KeyEvent e)`
 - ◆ `keyPressed(KeyEvent e)`
 - ◆ `keyReleased(KeyEvent e)`
 - ◆ `Runnable`インターフェイスの抽象メソッド
 - ◆ `run()`
 - ◆ `scene=0`の時に`paint`メソッドが呼ぶメソッド`ready(Graphics g)`
 - ◆ `scene=1` // `playing(Graphics g)`
 - ◆ `scene=2` // `gameover(Graphics g)`
 - ◆ `scene=3` // `gameclear(Graphics g)`
- ◆ それぞれ処理は後で記述する(名前だけ宣言しておこう)

クラス設計 Ballクラス

Ballクラスのフィールド

- ◆ フィールドは、
 - ◆ 中心座標 `double x, y`
 - ◆ 再描画ごとの座標の変化量 `double dx, dy`
 - ◆ 半径 `int r`
 - ◆ ボールが落ちるかどうか `boolean isFallen`

が必要となるだろう。

そのほかは書いていくうちに随時追加していく。

Ballクラスのメソッド

- ◆ メソッドは、
 - ◆ ボールを座標 x , y を中心に描画するdrawメソッド
 - ◆ ボールの座標を移動させるmoveメソッド
 - ◆ ボールの当たり判定処理reflectメソッド

が必要となるだろう。

そのほかは書いていくうちに随時追加していく。

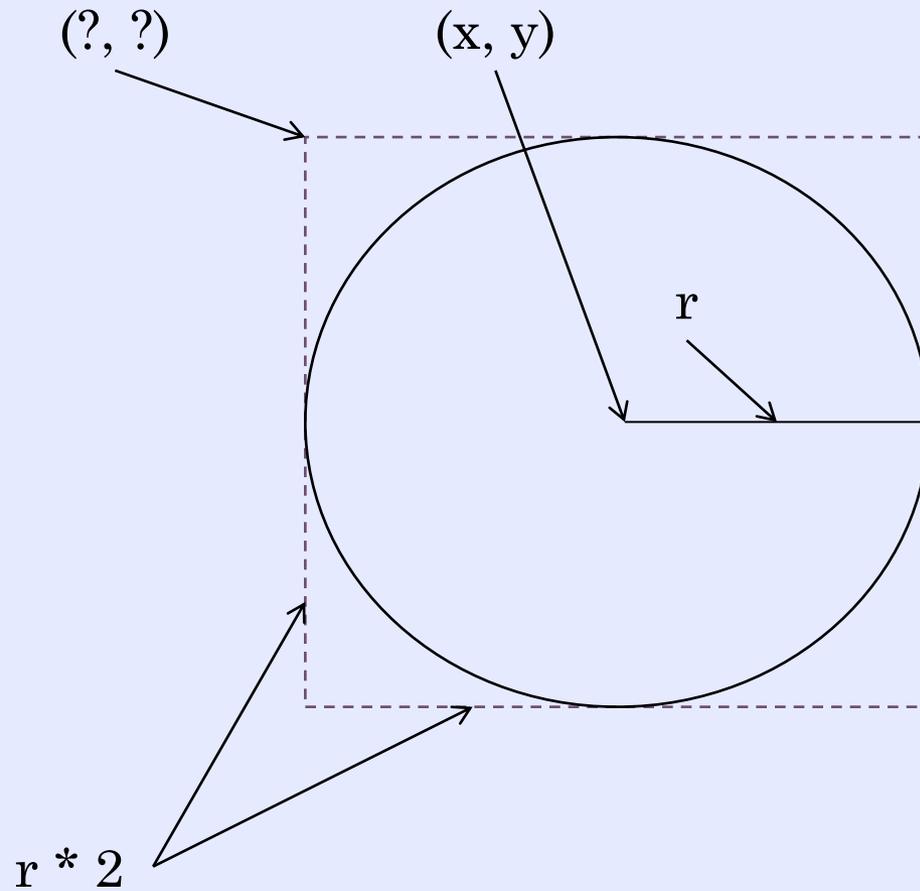
- ◆ 次項以降でdraw, move, reflectメソッドを定義していく。

Ballクラスのdrawメソッド

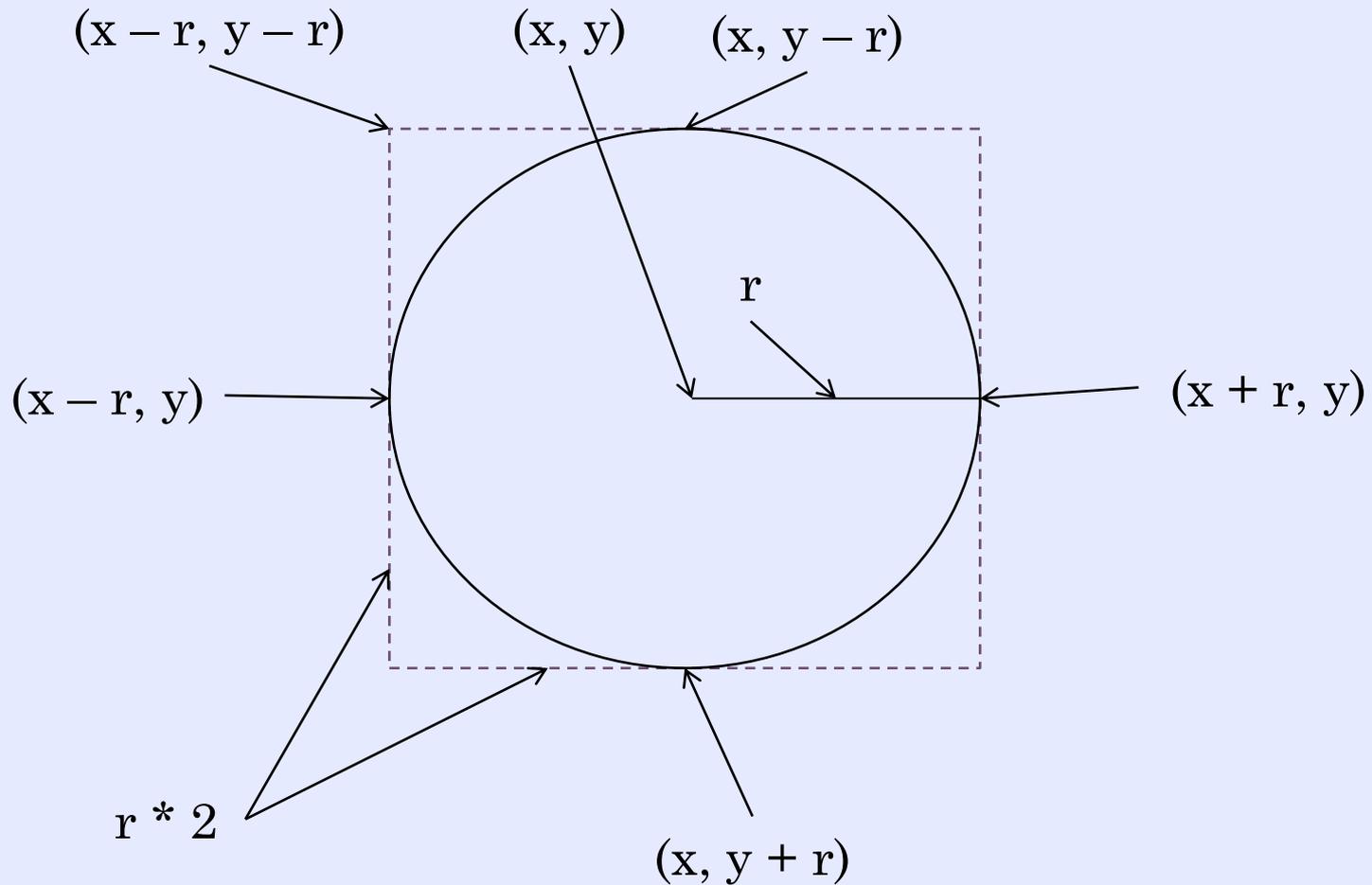
```
public void draw(Graphics g){  
    g.drawOval((int)x - r,(int)y - r,r * 2,r * 2);  
}
```

- ◆ Graphicsオブジェクトを引数として受け取り、それを使って円を描画する。
- ◆ 今回はfillOvalでもdrawOvalどちらでも良い。
- ◆ 次項で座標指定の解説をする

Ballクラスの座標指定について



Ballクラスの座標指定について



Ballクラスのmoveメソッド

```
public void move(){  
    x += dx;  
    y += dy;  
}
```

- ◆ x, yにそれぞれ再描画ごとの変化量dx, dyを加算する
- ◆ 当たり判定処理もこのあとに追加しても良い。

Ballクラスのreflectメソッド

```
public void reflect(int screen_w, int screen_h,  
                    Bar bar, Block[][] blocks){  
    /*壁との当たり判定を記述*/  
    /*バーとの当たり判定を記述*/  
    /*ブロックとの当たり判定を記述*/  
}
```

- ◆ 壁、バー、ブロックとの当たり判定は、後に解説する

クラス設計 Barクラス

Barクラスのフィールド

- ◆ フィールドは、
 - ◆ 左上の座標 `double x, y`
 - ◆ 幅 `int width`
 - ◆ 高さ `int height`
 - ◆ 横移動する速度(変化量) `double dx`

Barクラスのメソッド

- ◆ メソッドは、
 - ◆ バーを描画するdrawメソッド
 - ◆ バーを移動するmoveメソッド
- ◆ 次項以降でdraw, moveメソッドを定義していく。

Barクラスのdrawメソッド

```
public void draw(Graphics g){  
    g.drawRect((int)x, (int)y, width, height);  
}
```

- ◆ Ballクラスと同様にGraphicsオブジェクトを引数として受け取り、それを使って長方形を描画する。

Barクラスのmoveメソッド

```
public void move(boolean isRight){  
    if(isRight)  
        x += dx;  
    else  
        x -= dx;  
}
```

- ◆ 引数isRightがtrueの場合、バーを右に移動させ、falseの場合、バーを左に移動させる。
- ◆ すなわち、isRightは矢印キーが「→」の場合、true「←」の場合、falseとする。

クラス設計 Blockクラス

Blockクラスのフィールド

- ◆ 同じ長方形なので、Barクラスとほぼ同じ
- ◆ フィールドは、
 - ◆ 左上の座標 `double x, y`
 - ◆ 幅 `int width`
 - ◆ 高さ `int height`
 - ◆ 既に壊されたかどうか `boolean isBroken`

Blockクラスのメソッド

- ◆ メソッドは、
ブロックを描画するdrawメソッド
- ◆ 次項以降でdrawメソッドを定義していく。

Blockクラスのdrawメソッド

```
public void draw(Graphics g){  
    if(!isBroken){  
        g.drawRect((int)x, (int)y, width, height);  
    }  
}
```

- ◆ Ball, Barクラスと同様にGraphicsオブジェクトを引数として受け取り、それを使って長方形を描画する。
- ◆ フィールドisBrokenがfalseの場合は描画し、trueの場合は描画しない。

Mainクラス フィールドの宣言

```
16     /*-----オブジェクト関連のフィールド-----*/
17     private Ball ball;
18     private Bar bar;
19     private Block[][] stage;
20     private Thread repaintThread;
21     /*-----オブジェクト関連のフィールド-----*/
22
23
24     /*-----フラグ関連のフィールド-----*/
25     /** 現在、左矢印キーが押されているかどうかのフラグ */
26     private boolean leftKeyFlag = false;
27     /** 現在、右矢印キーが押されているかどうかのフラグ */
28     private boolean rightKeyFlag = false;
29     /*-----フラグ関連のフィールド-----*/
30
31
32     /*-----シーン制御関連のフィールド-----*/
33     /**
34      * 現在、どのシーンかを表す<br>
35      * 0:初期画面<br>
36      * 1:プレイ中画面<br>
37      * 2:ゲームクリア画面
38      * 3:ゲームオーバー画面<br>
39      */
40     private int scene = 0;
41     /*-----シーン制御関連のフィールド-----*/
42
```

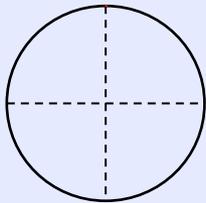
当たり判定

当たり判定

- ◆ Ballクラスのreflect()メソッドに記述する。
- ◆ 今回は、円と長方形の当たり判定のみ
(壁との当たり判定は第6回のMovingBallで実装済み)
- ◆ 次項からはブロックとボールの当たり判定を考える。

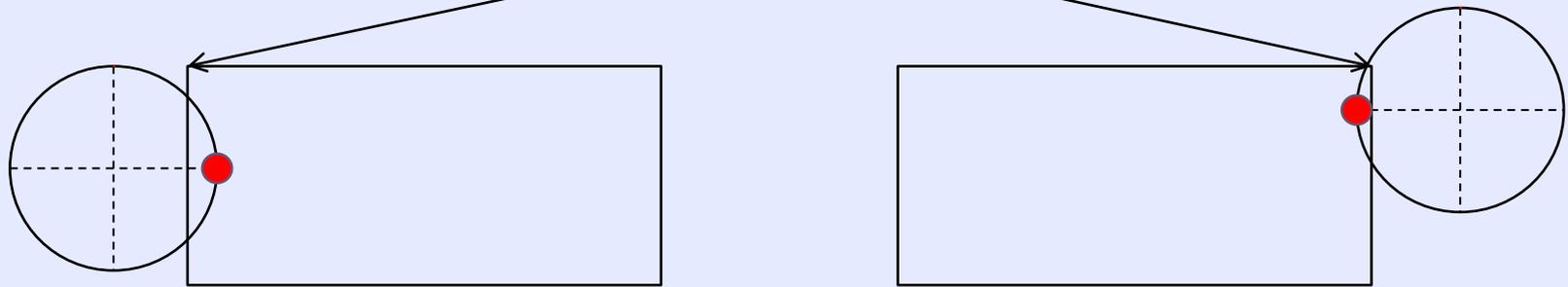
ブロックとボールの当たり判定の図

- ◆ 円=ボール
- ◆ 長方形=ブロック



当たり判定の図(左右)

補正值 = 線のx座標

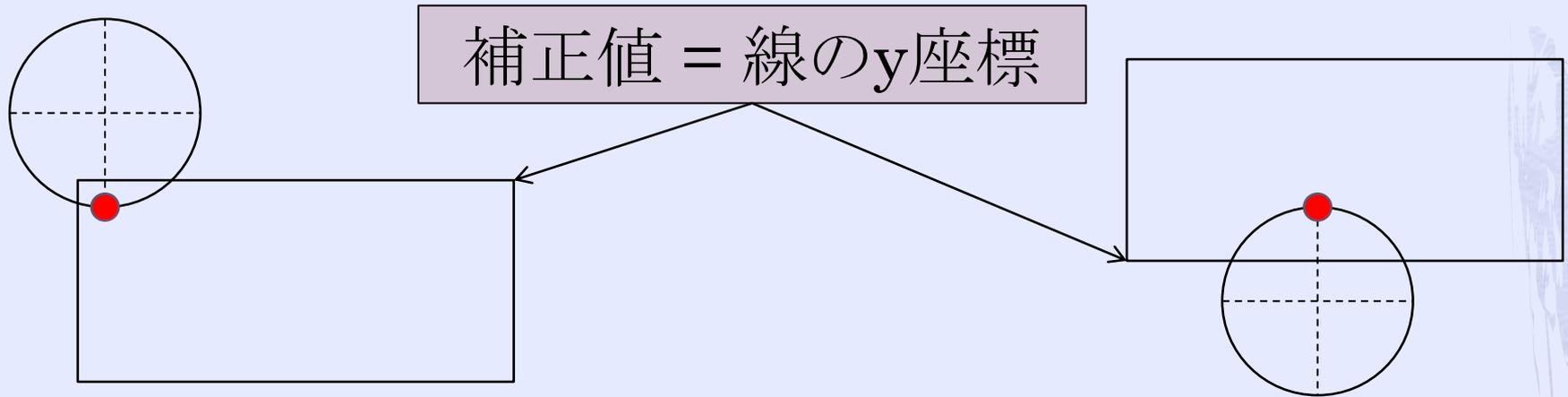


- ◆ 赤い点が長方形の左右どちらかに接する、または入り込んでいる場合、

`ball.dx *= -1;`

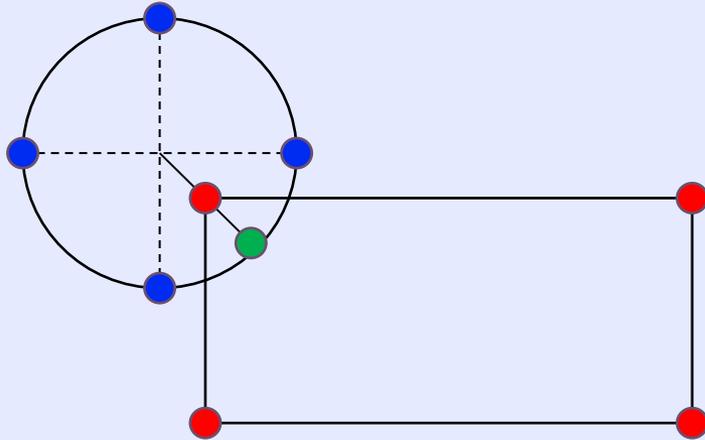
`ball.x = 補正值;`

当たり判定の図(上下)



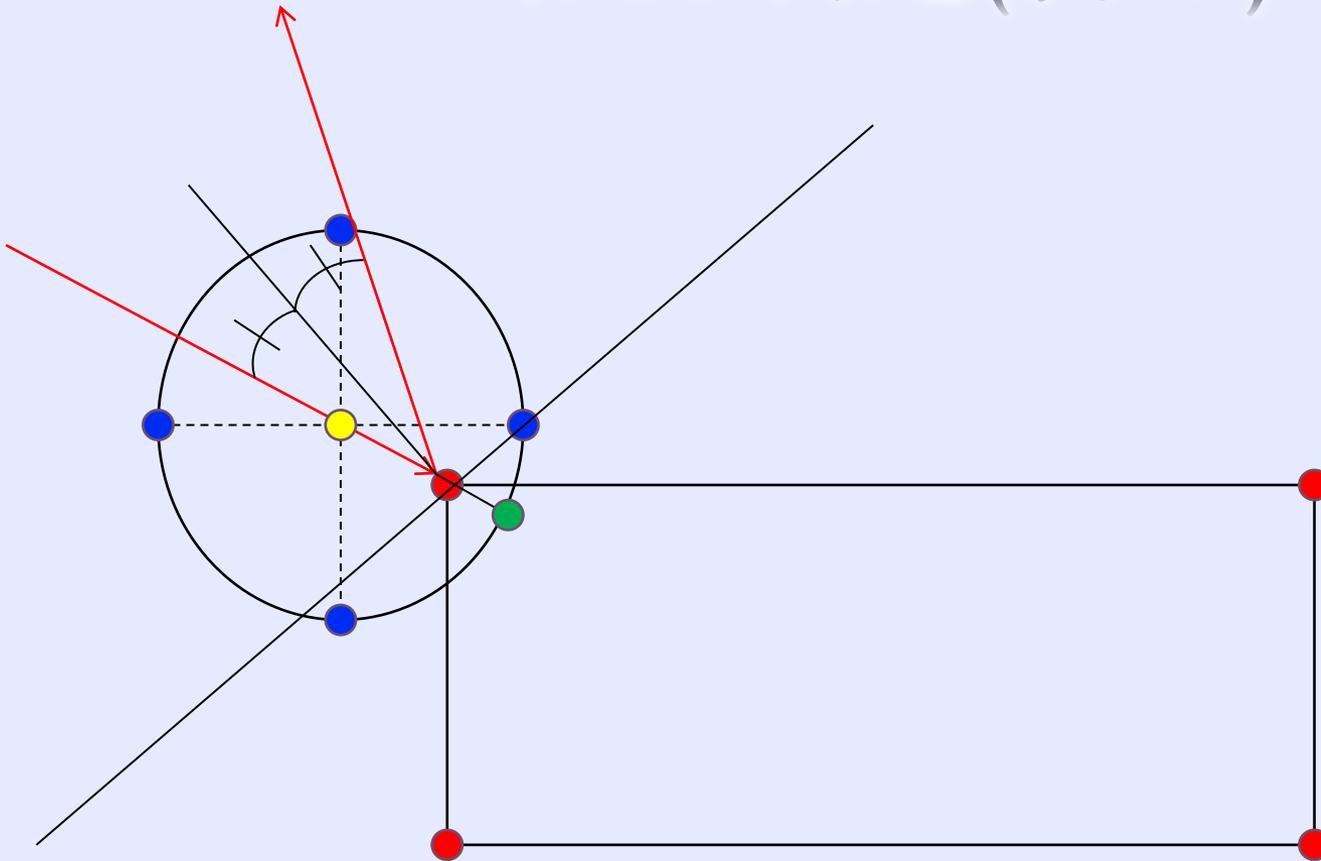
- ◆ 赤い点が長方形の上下どちらかに接する、または入り込んでいる場合、
 `ball.dy *= -1;`
 `ball.y = 補正值;`

当たり判定の図(斜め)



- ◆ 赤い点が円に接する、または入り込んでいる場合、ボールの座標を赤い点と緑の点に接するように補正し、反射する。

当たり判定(斜め)



これは難しい。。。。

当たり判定(斜め)

- ◆ 前項の反射を実装するのは難しいので、とりあえず、

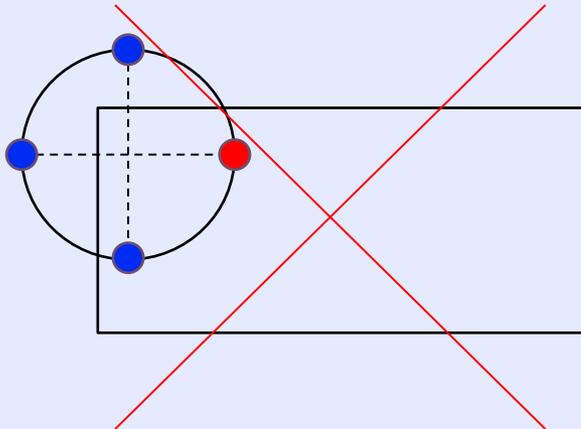
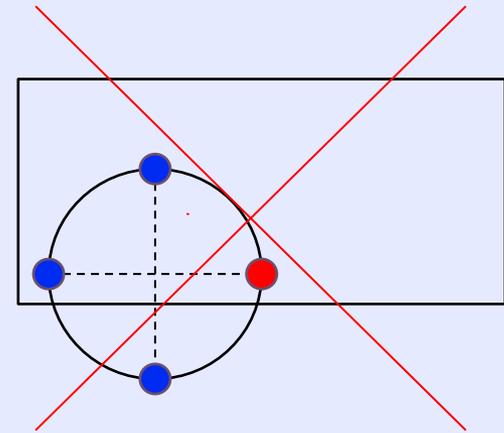
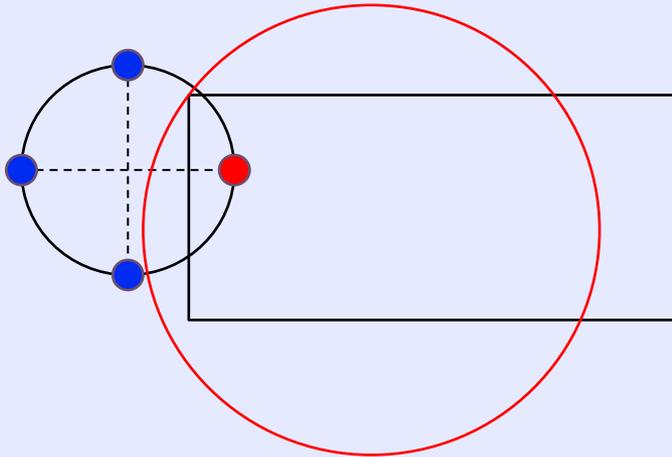
```
ball.dx *= -1;
```

```
ball.x = 補正值;
```

としておけば、それらしく見える。

当たり判定の図(左)

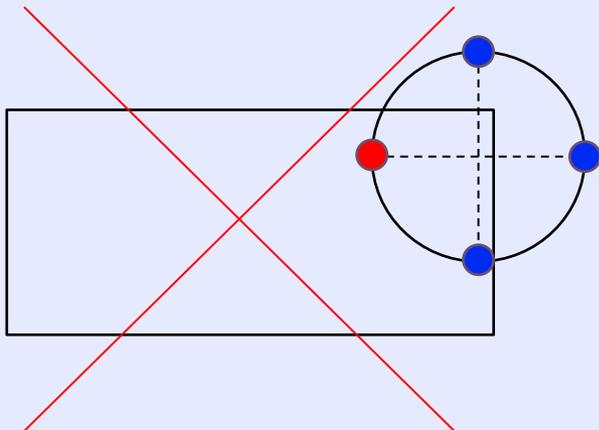
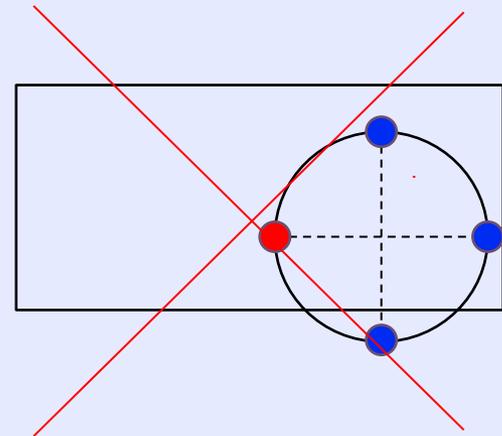
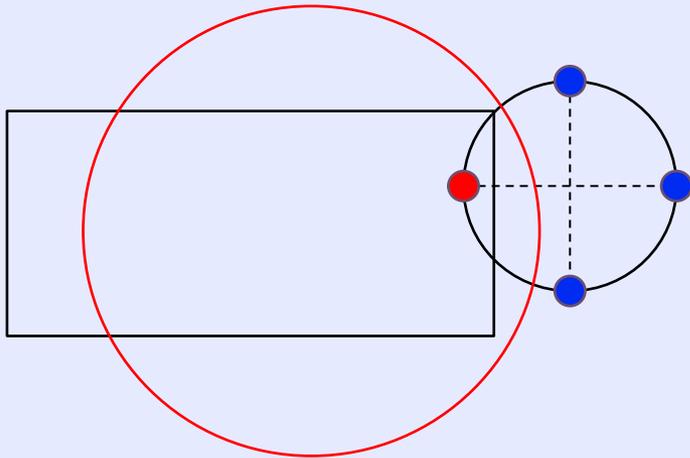
- ◆ 左から入り込んだ場合を考える。



赤い点のみが入り込んでいる場合、○
青い点も入り込んでいる場合、×

当たり判定の図(右)

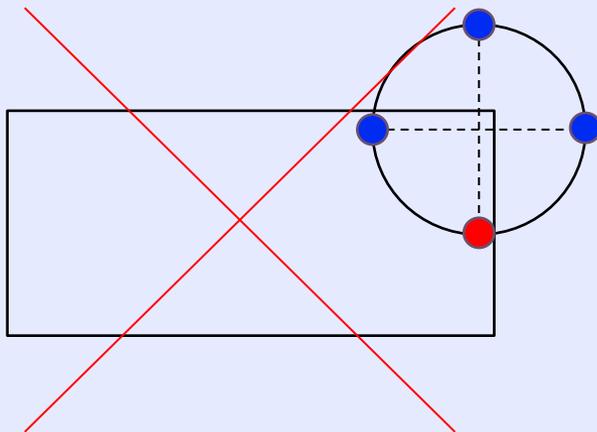
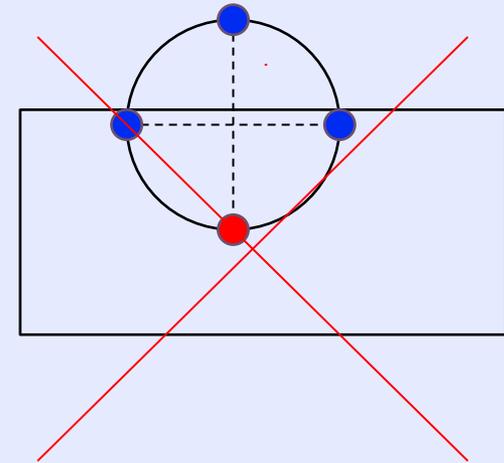
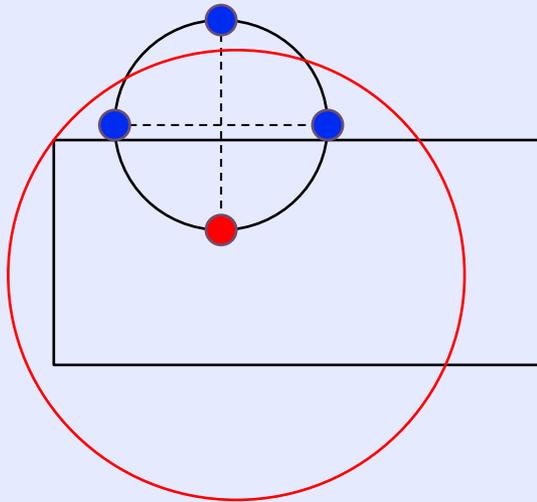
- ◆ 右から入り込んだ場合を考える。



赤い点のみが入り込んでいる場合、○
青い点も入り込んでいる場合、×

当たり判定(上)

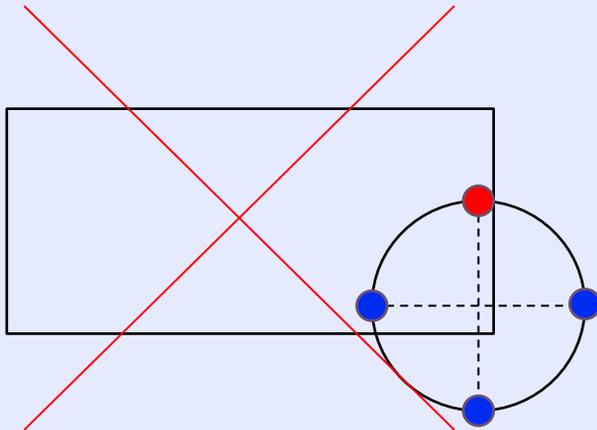
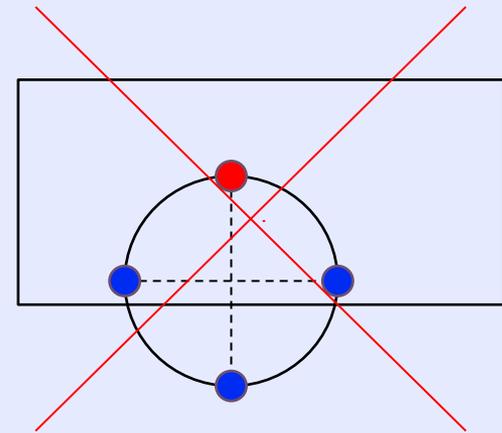
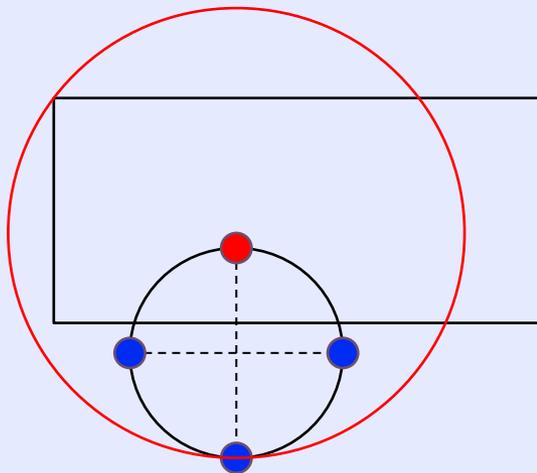
- ◆ 上から入り込んだ場合を考える。



赤い点のみが入り込んでいる場合、○
青い点も入り込んでいる場合、×

当たり判定の図(下)

- ◆ 下から入り込んだ場合を考える。



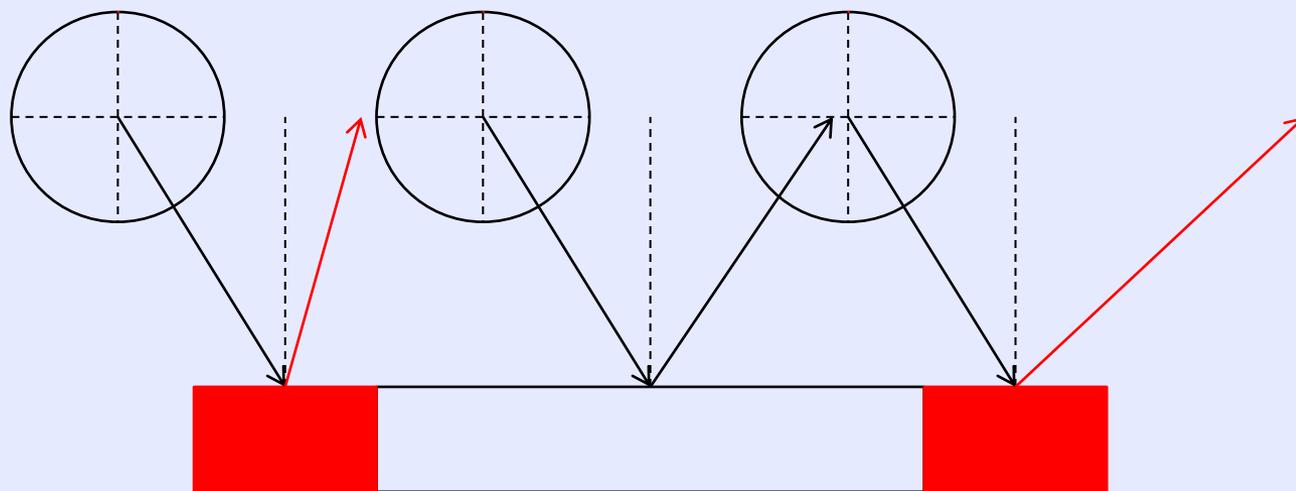
赤い点のみが入り込んでいる場合、○
青い点も入り込んでいる場合、×

バーとボールの当たり判定

- ◆ バーとボールの当たり判定を考える
- ◆ 長方形と円の当たり判定であるので、ブロックの場合とほぼ同じ。
- ◆ 次項からはバーとボールの当たり判定を考える。

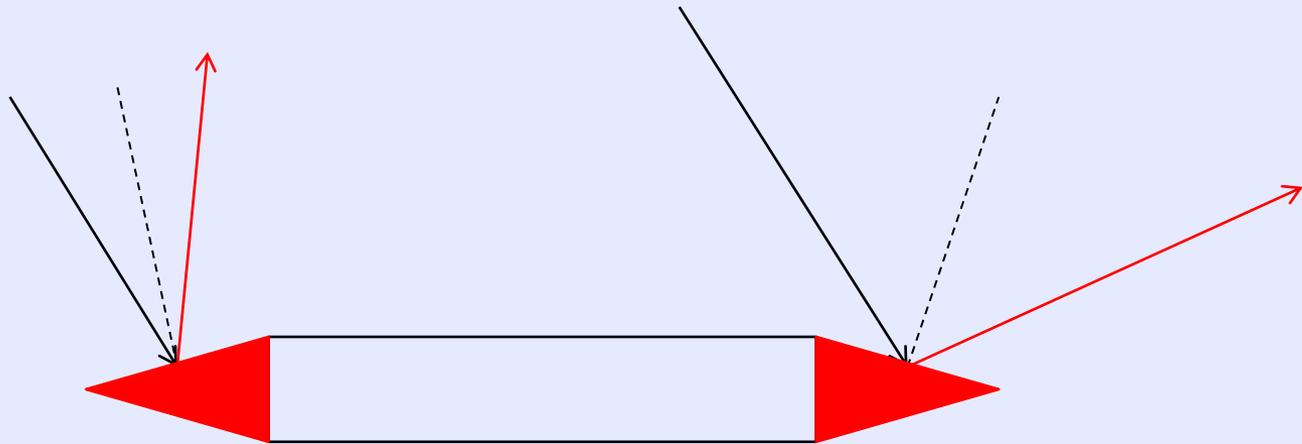
バーの反射パターン

- ◆ ブロック崩しのバーの当たり判定は下図のようになる。
- ◆ ボールの反射パターンが一定にならないようにするため



バーの反射パターン

- ◆ こんな感じのバーと考えると反射させる



入射角と反射角

- ◆ 入射角を 225° とすると、反射角は 315° である。

しかし、前項の左側の赤い部分の場合は反射角は、

$$315^\circ - 20^\circ = 295^\circ$$

右側の赤い部分の場合は反射角は、

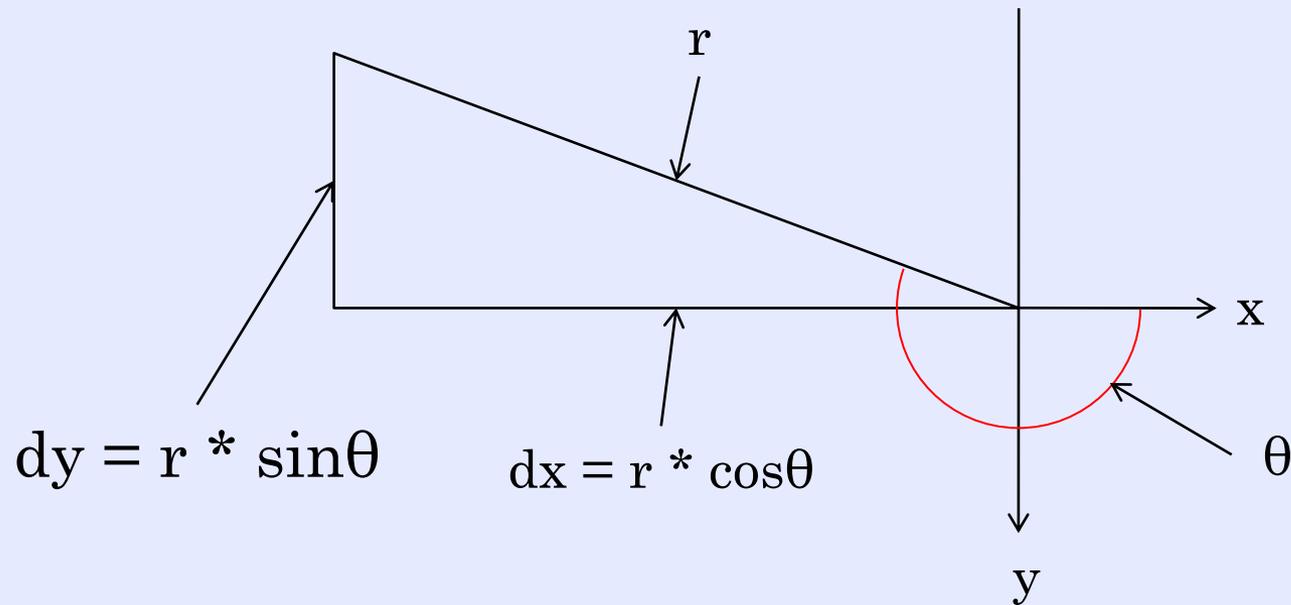
$$315^\circ + 20^\circ = 335^\circ$$

くらいにするといいかな。

当たり判定

- ◆ 角度からdx, dyを求める

角度からdx, dyを求める(事前知識)



◆ $r = \text{Math.sqrt}(dx * dx + dy * dy);$

反射の考え方

- ◆ 反射前の dx , dy で、
 $dx = -dx$;
とすると、通常の反射後の dx , dy が求まる。
- ◆ バーの左側部分の場合、
 1. その dx , dy から角度を求め、
 2. その角度に -20 する。
 3. そこから、また dx , dy を求めれば良い。
- ◆ バーの右側部分の場合、 $+20$ する。

実際の書き方

```
double r = Math.sqrt(dx * dx + dy * dy);  
dx = -dx;  
double radian = Math.atan2(dy, dx); // dy/dxの傾きの角度をラジアンで返す  
double angle = Math.toDegrees(radian) - 20; // ラジアンを弧度法に変換  
if(180 <= angle)    angle -= 360;  
else if(angle <= -180)  angle += 360;  
radian = Math.toRadians(angle); // 弧度法をラジアンに変換  
dx = Math.cos(radian) * r; // dxを求める  
dy = Math.sin(radian) * r; // dyを求める
```

- ◆ 上の例はボールがバーの左側部分に当たったと判定した時に反射後のdx, dyを求めるコードである。
- ◆ `Math.atan2(y, x)`はy / xの傾きの角度を-PI~PIの間で返す。
- ◆ `Math.toDegrees(radian)`は-PI~PIのradianを $^{\circ}$ に変換した値を返す。
- ◆ `Math.toRadians(angle)`は-180~180のangleをラジアンに変換した値を返す。

※PI.....パイ

バーの当たり判定

- ◆ したがって、左側部分、中央部分、右側部分に分けて判定を行うことになる。
- ◆ `reflect()`メソッドに記述してみよう。

実行クラスを完成させる

各画面を作成する

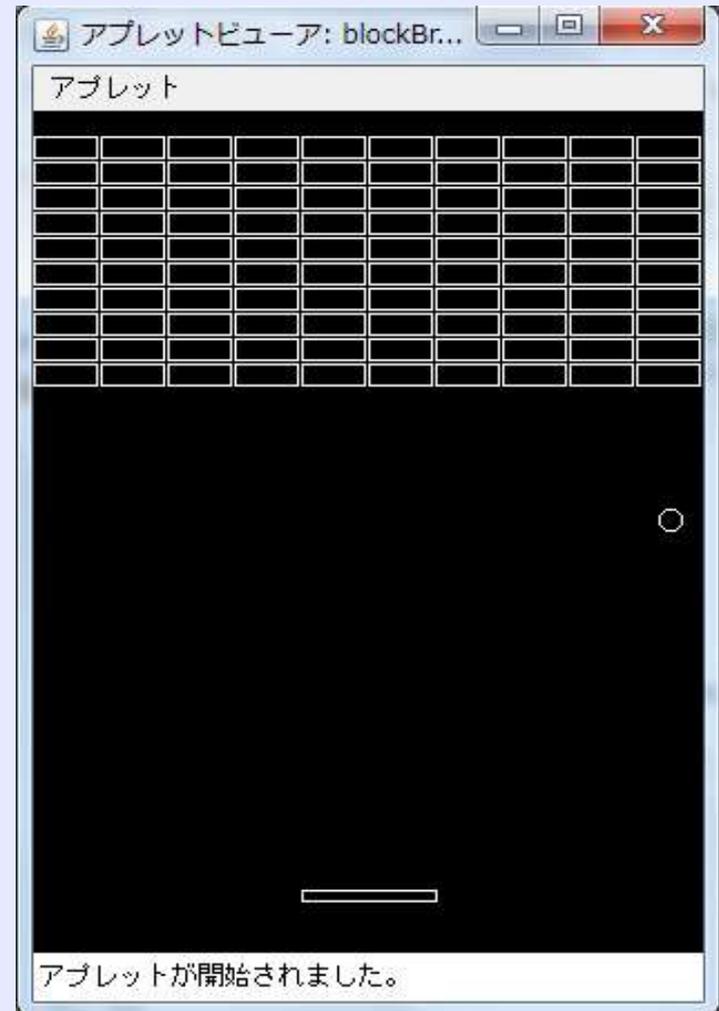
初期画面(scene = 0)

- ◆ 初期画面はこんな感じ。
- ◆ `ready()`メソッドに記述する
- ◆ この画面でスペースキーが押された場合、
`scene = 1;`



プレイ中画面(scene = 1)

- ◆ プレイ中画面は、ボール、バー、ブロックを描画する。
- ◆ `playing()`メソッドに記述する
- ◆ ボールが落ちたら、`scene = 2;`
- ◆ ブロックが全て破壊された場合、`scene = 3;`



ゲームクリア画面(scene = 2)

- ◆ クリアタイムは無くても良い
- ◆ `gameclear()`メソッドに記述する
- ◆ この画面でスペースキーが押された場合、
`scene = 0;`



ゲームオーバー画面(scene = 3)

- ◆ プレイタイムは無くても良い
- ◆ `gameover()`メソッドに記述する
- ◆ この画面でスペースキーが押された場合、`scene = 0;`



各画面を作成する

各画面の作成の流れとしては、

1. 描画色を指定する

(実例)`g.setColor(Color.WHITE);`

2. 描画する文字のフォントを指定する

(実例)`Font f = new Font(Font.DIALOG, Font.ITALIC, 40);
g.setFont(f);`

3. 文字等を描画する

次項以降で画面作成に必要なと思われるメソッドやその使い方を紹介していく。

g.setColor()について

- ◆ GraphicsクラスのsetColor()メソッドは引数にColorオブジェクトを渡さなければならない
- ◆ Colorオブジェクトは、
 - 「Color.色名」
 - 「new Color(r, g, b)」
 - 「new Color(r, g, b, a)」
(aは0~255で透明度を表す)以上の取得方法がある。

g.setFont()について

- ◆ GraphicsクラスのsetFont()メソッドは引数にFontオブジェクトを渡さなければならない。
- ◆ Fontオブジェクトは、「new Font(フォント名, スタイル名, サイズ)」で生成できる。
- ◆ フォント名はFont.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED, Font.SANS_SERIF, Font.SERIFのうちいずれか。
- ◆ スタイル名はFont.PLAIN, Font.BOLD, Font.ITALICのうちいずれか。
- ◆ サイズは文字の大きさ(整数値)を表す。
- ◆ フォント名やスタイル名は色々試してみよう。

画像を描画する

- ◆ `g.drawImage(Imageオブジェクト, x, y, width, height, null);`
`drawRect()`メソッドとほぼ同じで、四角形の領域にImageオブジェクト(画像)を描画する。

Imageクラスの扱い方

- ◆ 画像を扱うクラスとして、JavaにはImageクラスというものがある。
- ◆ Imageクラスを使うと、前項のようにGraphicsクラスのdrawImage()メソッドで簡単に画像が描画できる。
- ◆ Image image =
 new ImageIcon(“ファイルパス”).getImage();
とすると簡単にImageオブジェクトを取得できる。
- ◆ ファイルパスは相対パスで指定したほうが良い(srcから見た相対パスとなる)。
画像をプロジェクトの直下に保存した場合、
ファイルパスは、「../画像名」となる。

キーボード入力を受け取る

KeyListenerを登録する

- ◆ `init()`メソッドに、
`addKeyListener(this);`
という記述を追加する。
これはKeyListenerを実装したクラスを引数として渡すことでキーリスナーを登録できる。
するとキーが押された時、`keyPressed()`メソッドが呼ばれるようになる。
- ◆ 試しに、`keyPressed()`メソッドに
`systrace → [Ctrl+Space]`で出力を追加して、呼び出されているかどうか確かめてみよう。
※最初にアプレットのウィンドウをクリックして、フォーカスを合わせてから、キーを押す必要がある。

KeyEventを受け取る

- ◆ `keyPressed()`メソッドはKeyEventオブジェクトを引数として受け取っている。
- ◆ KeyEventオブジェクトには、どのキーが押されたか等の情報が格納されている。
- ◆ 次項に`keyPressed()`メソッドの例を示す

KeyEventを受け取る

```
public void keyPressed(KeyEvent e){
    int keyCode = e.getKeyCode();
    switch(keyCode){
    case KeyEvent.VK_LEFT:
        break;
    case KeyEvent.VK_RIGHT:
        break;
    case KeyEvent.VK_SPACE:
        break;
    }
}
```

- ◆ こんな感じで分岐できる。(if文でもOK)
- ◆ e.getKeyCode()で取得した整数値とKeyEvent.VK_キーの名前を比べることでどのキーが押されたかを判別することになる。

KeyEventを受け取る

- ◆ 前項でkeyPressed()メソッドの雛形を示したが、具体的な処理は自分で考えてみよう。
- ◆ keyPressed()メソッドでやらなければならないことは、
 - ◆ スペースボタンが押されたとき、
 - ◆ scene = 0ならば、scene = 1
 - ◆ scene = 2 または scene = 3ならば、scene = 0
 - ◆ 左矢印キーが押されたとき、
 - ◆ scene = 1ならば、barを左へ移動させるための処理
 - ◆ 右矢印キーが押されたとき、
 - ◆ scene = 1ならば、barを右へ移動させるための処理

プレイ中画面からの画面切り替え

- ◆ 「プレイ中画面(scene = 1)」からは
「ゲームクリア画面(scene = 2)」
「ゲームオーバー画面(scene = 3)」
へそれぞれ画面が切り替わる。
- ◆ 「ゲームオーバー画面(scene = 2)」
ボールが落ちたとき
- ◆ 「ゲームクリア画面(scene = 3)」
ブロックが全て破壊されたとき
- ◆ これらを実装しなければならない

プレイ中画面からの画面切り替え

自分でどこに記述すればよいかを考えてみよう。

終わり