

# 第9章

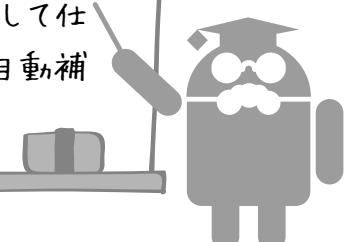
## ゲームによる実践

著：高橋憲一

# 9-1 ゲームによる実践（1）

著：高橋憲一

ここまで"の章で、Androidのアプリを構成するさまざまな要素について、パート単位で学びました。ここでは、それらを組み合わせてゲームというジャンルで、ひとつのアプリとして仕上げることを実践します。テキストの手順にしたがって自動補完機能をうまく活用しながら進めてみてください。



## この節で学ぶこと

- ・これまで学んで来たことを組み合わせて1つのアプリを作成
- ・ゲームならではの実装方法
- ・高速描画のための手法

この節で出て来るキーワード、Android SDKのクラス、外部ライブラリーの一覧

ゲーム  
高速描画  
スプライト  
クラスの継承  
Android SDKのクラス  
SurfaceView  
SurfaceHolder  
Canvas  
Thread

## 9-1-1 ゲームに必要な要素

### 開発で使われるツール

3Dゲームであれば最近は「Unity」等のゲームエンジン、2Dのゲームなら「Cocos2d-x」や「And Engine」等（2DでもUnityを使用するケースも増えています）の

ライブラリーを使用することが多いのですが、ここでは基本的なしくみを学ぶため、AndroidのSDKのみを使用して開発します。

## リソース作成（画像、音）

ゲームで使用する2Dの画像のことを「スプライト」と呼びます。自分で動かすキャラの画像や敵キャラの画像、背景画像などの画像ファイルが必要となります。また、実際のプレイを盛り上げるために効果音も重要な要素ですので、音声ファイルが必要となります。



### 9-1-2 基本の仕様を考える

スマートフォンで操作するゲームであるという観点から、ここではシンプルな操作で片手でプレイできるものを考えてみます。横長の画面に自分が操作するキャラがあって、横から流れて来る障害物をよけるゲームにします。キャラの操作は、画面を指でタッチしている間はロケットをブーストして上昇し、指を離すと下降していくという2つの操作にします。



### 9-1-3 ゲームを描画する「View」の作成 (SurfaceView)

#### 新規プロジェクト作成

EclipseのFileメニューから、「New」→「Android Application」とたどると、図1のようなダイアログが表示されますので、「Application Name」に「FlyingDroid」と入力します。

「Project Name」には自動で「Syllabus」、「Package Name」には自動で「com.example.flyingdroid」と入力されていることを確認して下さい。ここでPackage Nameは「jp.techinstitute.flyingdroid」と変更します。

com.exampleの部分のみ選択して反転させてからjp.techinstituteと入力すると必要な部分のみの書き換えで済みます。

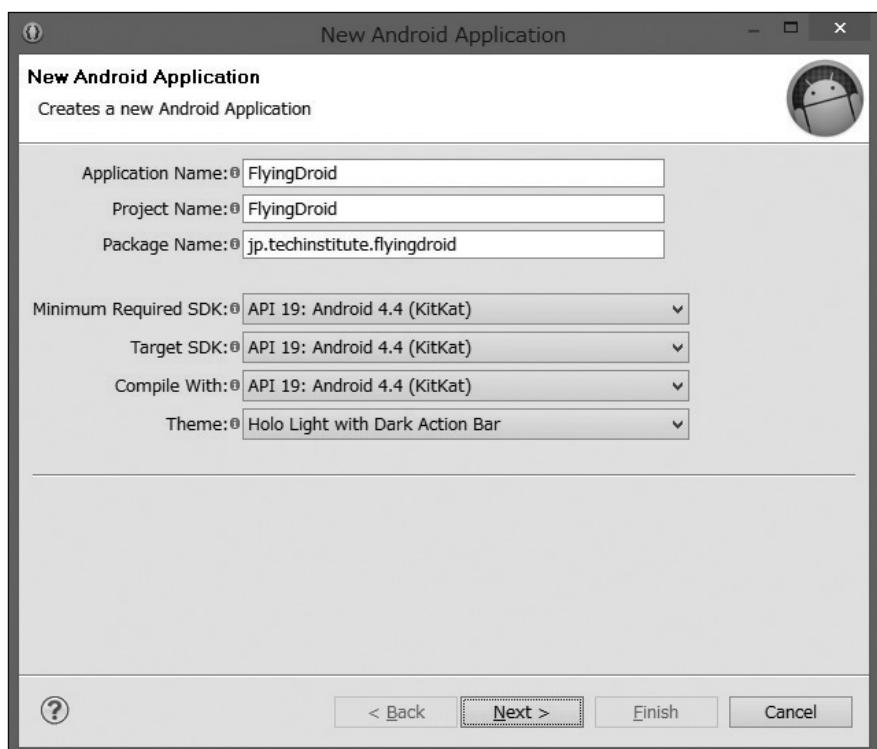


図1:New Android Applicationダイアログ

図1のように「Minimum Required SDK」「Target SDK」「Compile with」の3項目のすべてを「API 19:Android4.4(KitKat)」に設定します。「Theme」は「Holo Light with Dark Action Bar」を選択して「Next」ボタンを押します。

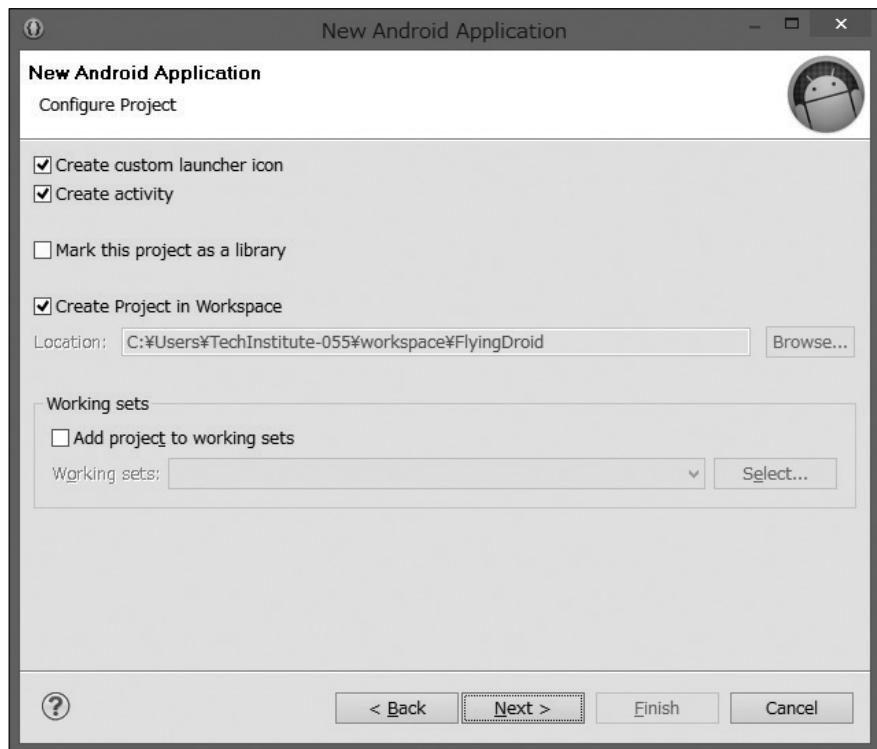


図2:New Android Application - Configure Project

続いて「New Android Application, Configure Project」というダイアログに切り替わります(図2)。「Create custom Launcher Icon」「Create activity」にチェックが入っていて、「Mark this project as a library」にはチェック

が入っていないことを確認し「Next」ボタンを押します。

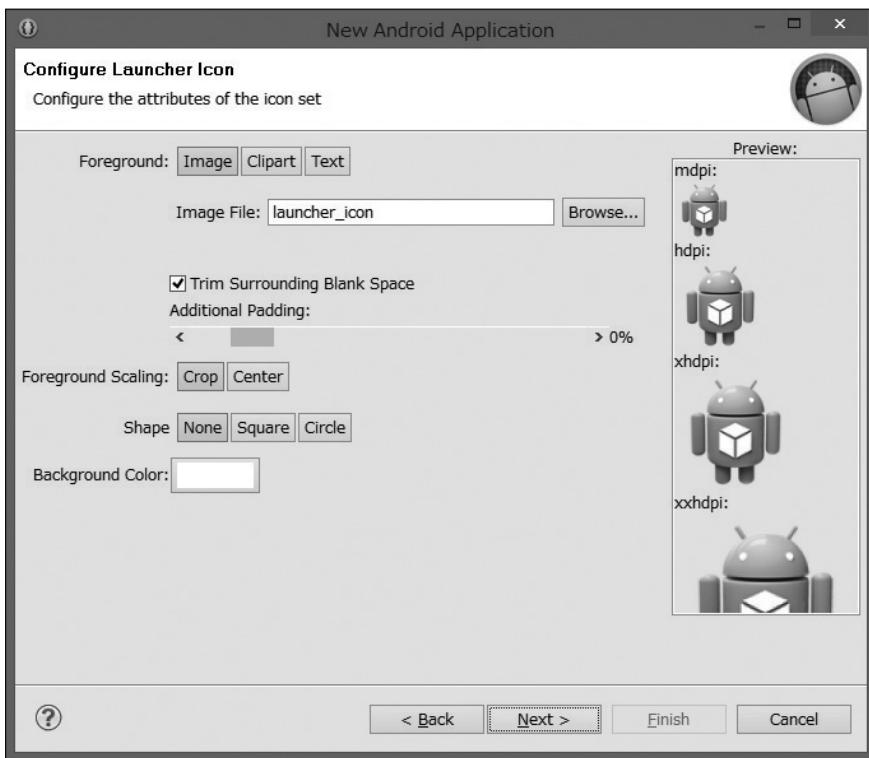


図3:New Android Application - Configure Launcher Icon

とくにアイコンを変更する必要がなければ、この画面ではそのまま「Next」ボタンを押します(図3)。

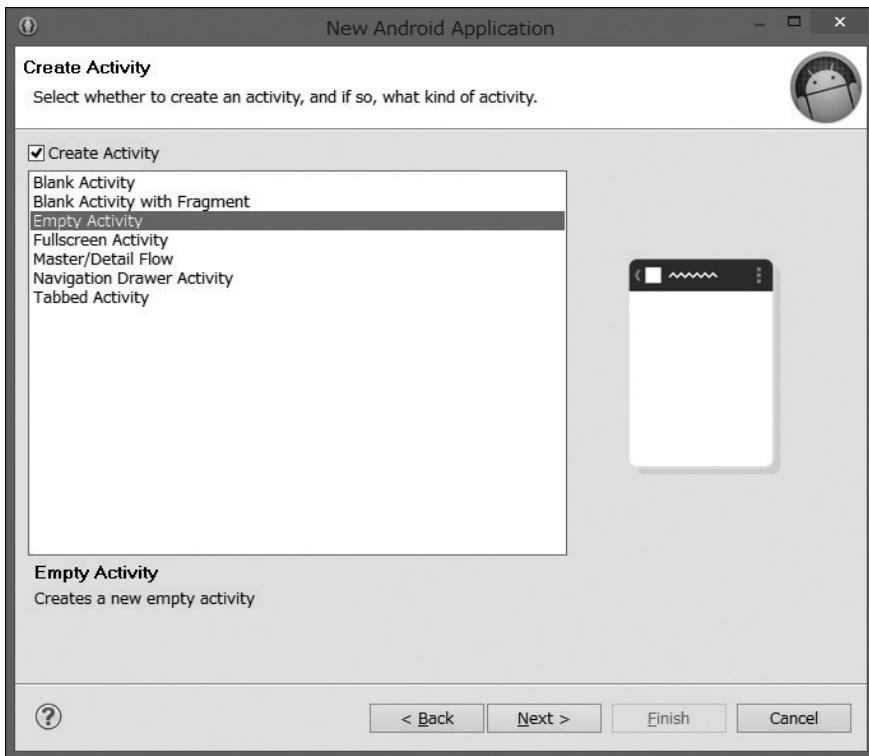


図4:New Android Application - Create Activity

図4では「Create Activity」にチェックが入っていることを確認し、「Empty Activity」を選んで「Next」ボタンを押します。

Action Barは使わないのでEmpty Activityを使います。

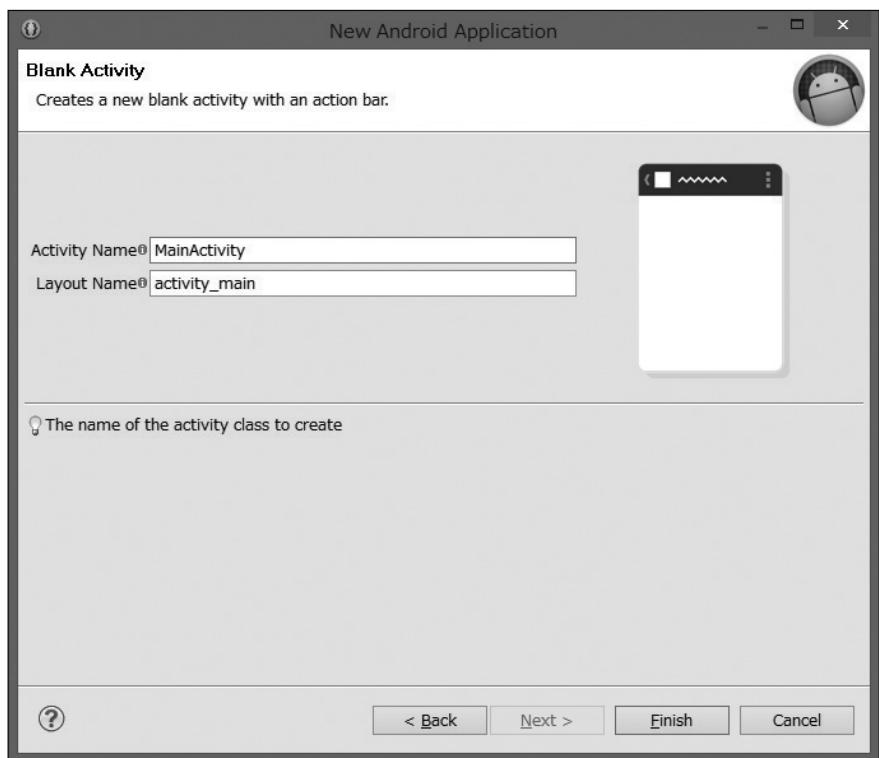


図5:New Android Application - Blank Activity

図5のように「Activity Name」には「MainActivity」、「Layout Name」には「activity\_main」と自動で入力されているはずです。確認して「Finish」ボタンを押します。

## 作成されたプロジェクトの確認

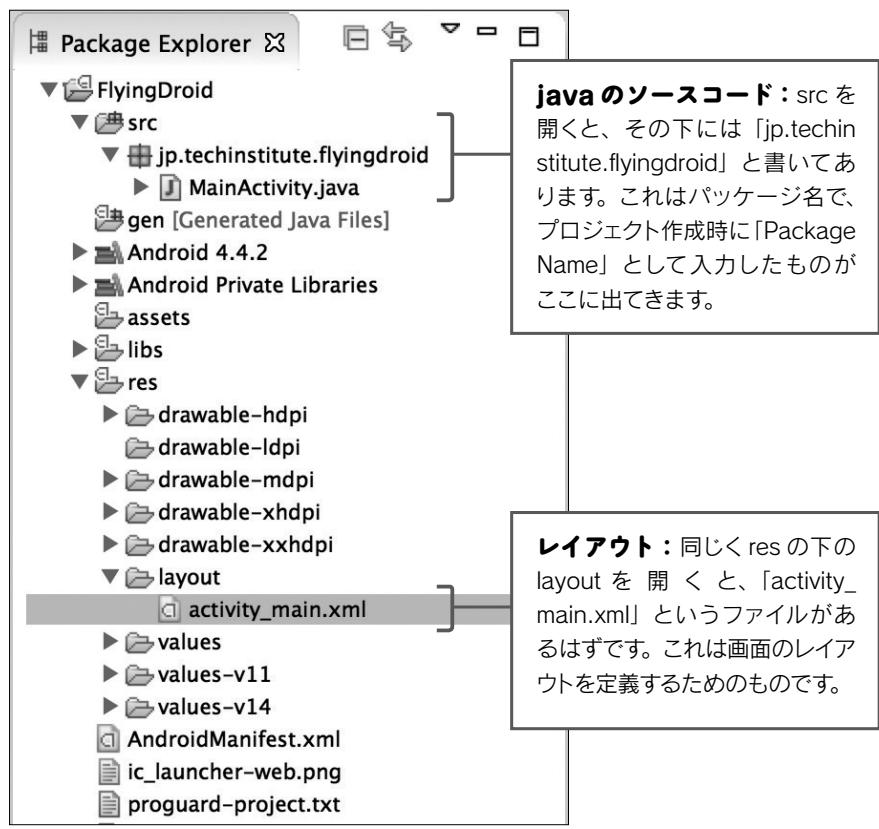
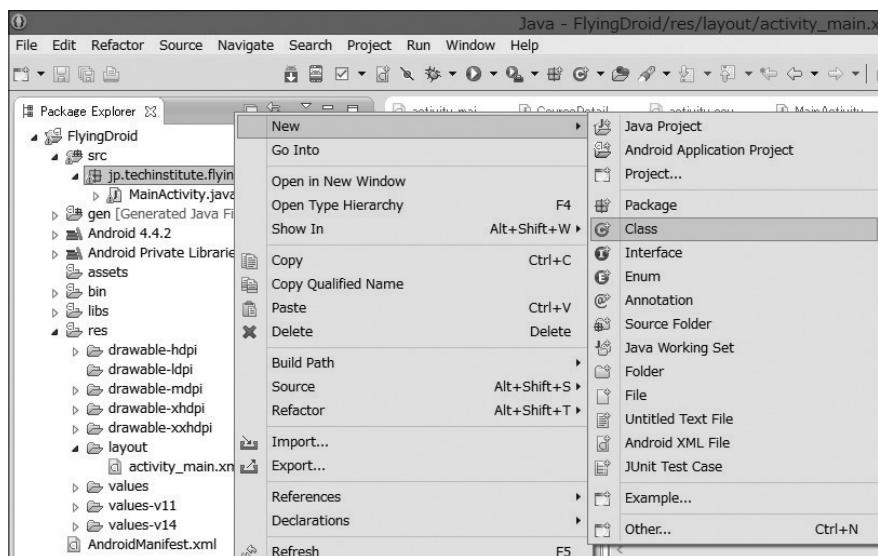


図6:EclipseのPackage Explorer

ここまでこの過程で、「FlyingDroid」というプロジェクトが作成されているはずです。作成されたプロジェクトの中を、復習を兼ねて確認します。「FlyingDroid」という名前のプロジェクトの下の階層が開かれていない場合は、左側の三角をクリックして開いてください。

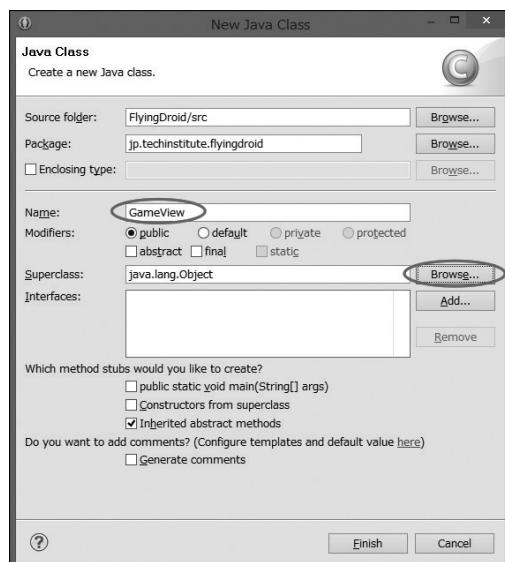
## カスタムViewクラスの作成

ゲーム画面をカスタムViewとして作成します。ここでは高速描画のために「SurfaceView」を使います。カスタムViewを作成するにはまずクラスの追加を行います。**図7**のように「Package Explorer」→「FlyingDroid」→「src」の下のパッケージ名「jp.techinstitute.flyingdroid」の部分で右クリックすると出て来るメニューから「New」→「Class」の順に選択します。



**図7:**Classの追加

**図8**の「New Java Class」ダイアログが開きます。「Name」欄に「GameView」と入力します(これがカスタムViewの名前になります)。次に継承元のクラスを設定するために「Browse...」ボタンを押します。



**図8:**New Java Classダイアログ

カスタムViewについては、第6章「ためしてわかるAndroidのしくみ」の6-2-4「独自のViewを利用してみよう」も読み返してみてください。

継承元クラスを選択する「Superclass Selection」ダイアログが開きます(図9)。「Choose a type」欄の「java.lang.Object」を消して、その代わりに「surface」と入力します。

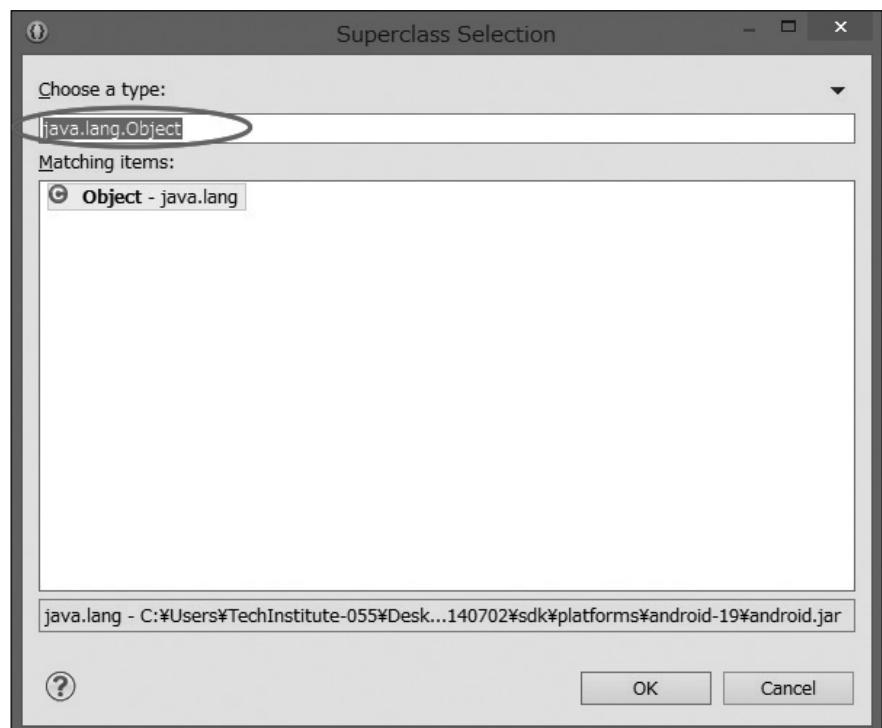


図9:Superclass Selectionダイアログ

「surface」と入力すると、図10のようにいくつか候補が表示されるので、「SurfaceView - android.view」を選択して「OK」を押します。

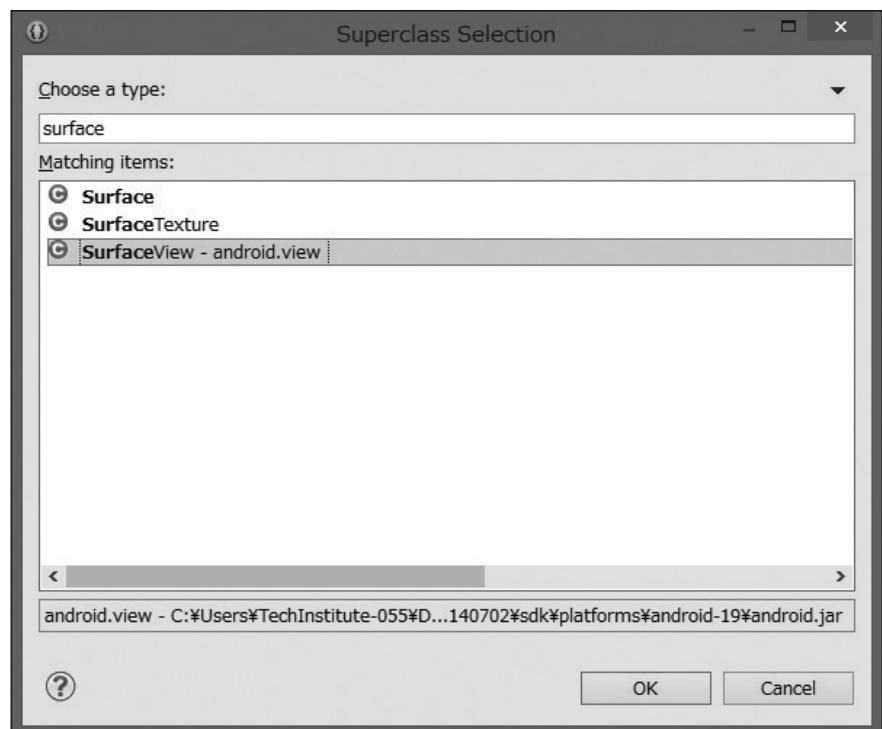


図10:Superclass selection - SurfaceView

「New Java Class」ダイアログに戻るので、「Superclass」が「android.view.SurfaceView」になっていることを確認して「Constructors from superclass」「Inherited abstract methods」「Generate comments」の3つに

チェックを入れて「Finish」ボタンを押します(図11)。

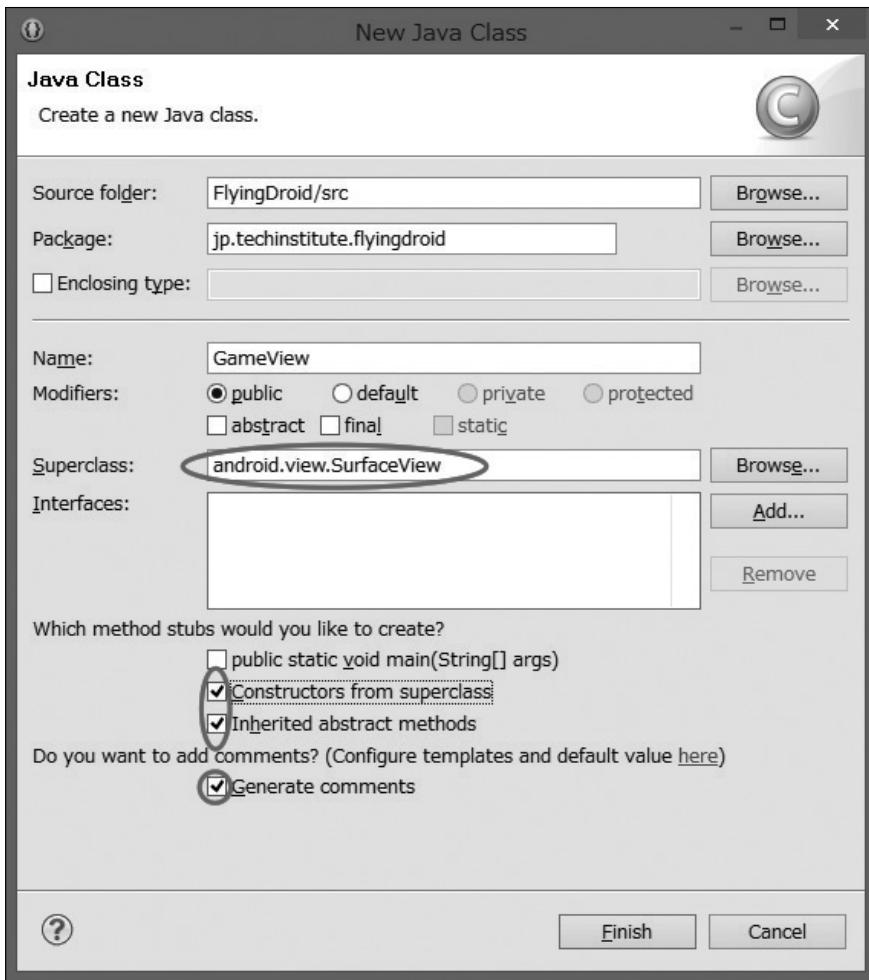


図11:New Java Classダイアログ - SurfaceView

「src」の「jp.techinstitute.flyingdroid」の下に「GameView.java」ができるていることを確認します(図12)。

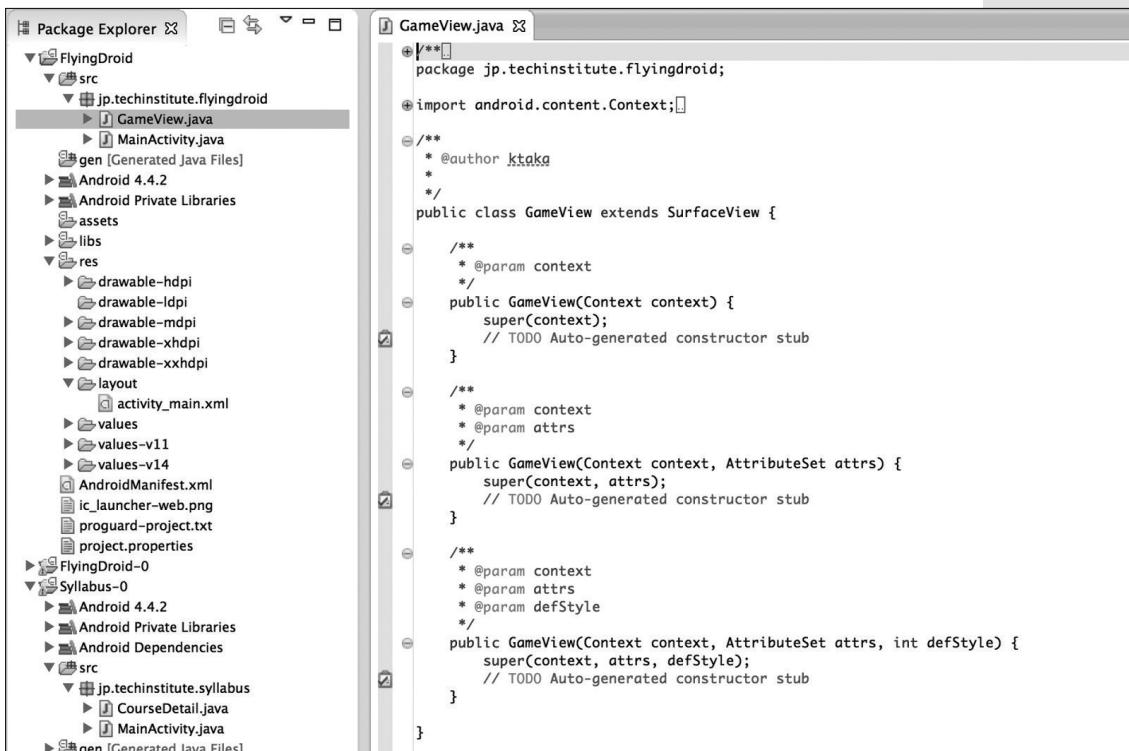


図12:GameView追加後の構成

check!

## クラスの継承を動物にたとえると……

ここでクラスの継承についておさらいしてみましょう。

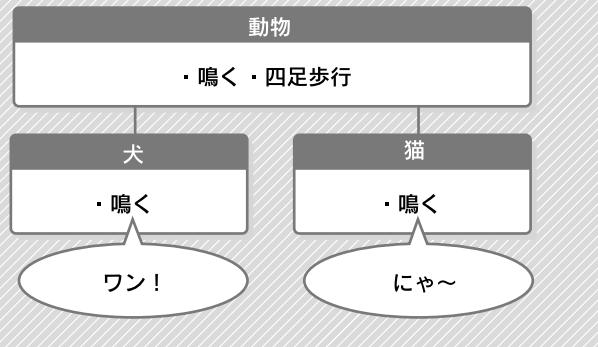
今回はandroid.view.SurfaceViewを継承してGameViewという新しいクラスを作成しました。ベースとなるクラスを継承して新しいクラスを作るという概念を、動物と犬、そして猫の関係で見てみます。

犬も猫も動物としての基本性質を備えています。たとえばどちらも「鳴く」という行為をしますよね。ただし、鳴き声には違いがあります。また、四本足で歩く行為は両方に共通しています。これをベースとなるクラス「Superclass」を継承して新しいクラスを作るという関係に当てはめて考えてみましょう。

動物は「Superclass」で、犬や猫はそれを継承して作る新しいクラスと捉えることができます。犬と猫のクラスは「四本足で歩く」という行為(メソッド)については、新たに実装する必要はありません。Superclassである動物が持っているものを利用できるからです。一方、それぞれでやるべきこと(処理)が異なる「鳴く」というメソッドは、それぞれが継承した「先」で新たに実装します。

これをふまえて、「android.view.SurfaceView」を継承し「Ga

meView」というクラスを作ることを考えると、「SurfaceView」が持つ多くの機能をそのまま活用しつつ、今回のゲームの描画機能やタップに応答する部分の処理を独自に実装していくべきということになります。これはandroid.app.Activityを継承してMainActivityを作ることにも当てはまります。



## GameView.javaの編集

追加された直後のGameView.javaは図13のようになっています。

```
① /**
 * @author ktaka
 */
public class GameView extends SurfaceView {
    ② /**
     * @param context
     */
    public GameView(Context context) {
        super(context);
        // TODO Auto-generated constructor stub
    }

    /**
     * @param context
     * @param attrs
     */
    ③ public GameView(Context context, AttributeSet attrs) {
        super(context, attrs);
        // TODO Auto-generated constructor stub
    }

    /**
     * @param context
     * @param attrs
     * @param defStyle
     */
    ④ public GameView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        // TODO Auto-generated constructor stub
    }
}
```

図13:GameView.java

図13の②の部分に次のコードを追加します。

GameView.java の②に追加する内容

```
class GameThread extends Thread {  
    SurfaceHolder surfaceHolder;  
    boolean shouldContinue = true;  
  
    public GameThread(SurfaceHolder surfaceHolder, Context context,  
        Handler handler) {  
        this.surfaceHolder = surfaceHolder;  
    }  
  
    @Override  
    public void run() {  
        while(shouldContinue) {  
            Canvas c = surfaceHolder.lockCanvas();  
            draw(c);  
            surfaceHolder.unlockCanvasAndPost(c);  
        }  
    }  
  
    public void draw(Canvas c) {  
        c.drawRGB(255, 0, 0);  
    }  
}  
  
GameThread gameThread;
```

「SurfaceView」は高速に描画するのに適していますが、このようにして描画スレッドを独自に実装する必要があります。図13の③の部分に次のコードを追加します。

GameView.java の③に追加する内容

```
SurfaceHolder holder = getHolder();  
holder.addCallback(this);  
  
gameThread = new GameThread(holder, context, new Handler() {  
    @Override  
    public void handleMessage(Message msg) {  
        super.handleMessage(msg);  
    }  
});
```

入力すると上記コードの2行目の「addCallback」の部分にエラーを示す赤線が付くので、マウスカーソルをその上に置くと出て来る黄色いウインドウの「Let "GameView" implement "Callback"」をクリックしてエラーを解決します。すると図13の①で示した部分に「implements Callback」という定義が自動的に追加されて次のようになります。

自動的に追加された GameView.java の先頭部分

```
public class GameView extends SurfaceView implements SurfaceHolder.Callback {
```

「GameView」に赤線が付くので、同様に黄色いウインドウの選択肢から「Add unimplemented methods」を選択します。すると、自動的に図13の④の部分に次のコードが追加されます。

```
@Override  
public void surfaceChanged(SurfaceHolder holder, int format, int width,  
    int height) {  
    // TODO Auto-generated method stub  
}  
  
@Override  
public void surfaceCreated(SurfaceHolder holder) {  
    // TODO Auto-generated method stub  
} ①  
  
@Override  
public void surfaceDestroyed(SurfaceHolder holder) {  
    // TODO Auto-generated method stub  
} ②
```

図14:SurfaceHolder.Callbackのメソッド

ここで①の矢印が示す部分に次のコードを追加します。

図14の①に追加する内容

```
gameThread.start();
```

これにより、「SurfaceView」が生成された時に描画用スレッドを開始することができます。

②の矢印が示す部分には次のコードを追加します。

図14の②に追加する内容

```
gameThread = null;
```

**check!**

## インターフェース (Interface) について

先ほどのコラムに出て来た継承とはまた別の方法で、同じ名前のメソッドを複数のクラスで実装することを強要できるしくみがJavaにあります。

強要と書くとあまり良いイメージではないかもしれません、同じ名前で同じ引数を取るメソッドがあるという前提条件が満たされることにより、オブジェクトのやり取りをおこなう場合に「この名前のメソッドを実装してもらえば、適切なタイミングでこちらからそれを呼び出しますよ」ということを明確にできます。

今回のSurfaceViewを使う場合で言うと、「SurfaceHolder.Callback」というインターフェースに「surfaceChanged」、「surfaceCreated」、「surfaceDestroyed」の3つのメソッドがあり、「GameView」というクラスでそのインターフェースの各メソッドを実装しておくと、「SurfaceView」が生成された時にはsurfaceCreated、サイズが変更された時はsurfaceChanged、破棄される時はsurfaceDestroyedが呼び出されるようになります。

## レイアウト作成

レイアウトファイル「activity\_main.xml」に、作成したGameViewを組み込みます。

プロジェクトのresフォルダーの中、layoutの下にあるactivity\_main.xmlをダブルクリックして開きます。この時、XML表示ではなくグラフィカルレイアウトになっている場合は、activity\_main.xmlのタブをクリックして切り替えます。

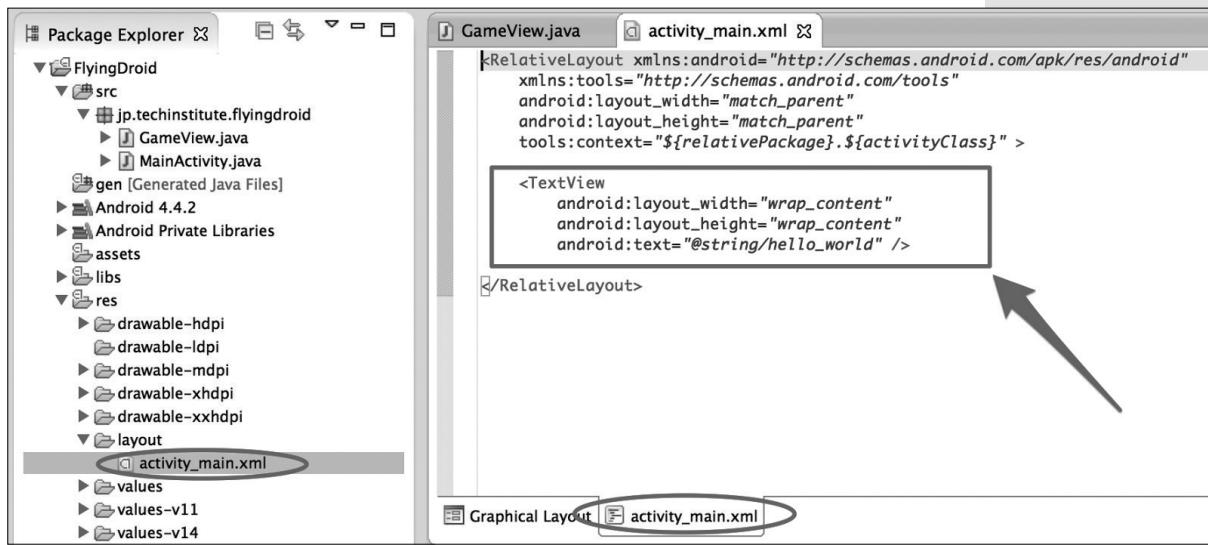


図15:activity\_main.xml

図15の矢印が示す部分を次のように書き換えます。

activity\_main.xmlで書き換える内容

```
<jp.techinstitute.flyingdroid.GameView
    android:id="@+id/gameView1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_centerHorizontal="true" />
```

## フルスクリーンの設定

次に、画面を横方向(landscape)に固定して、フルスクリーン表示の設定を追加します。

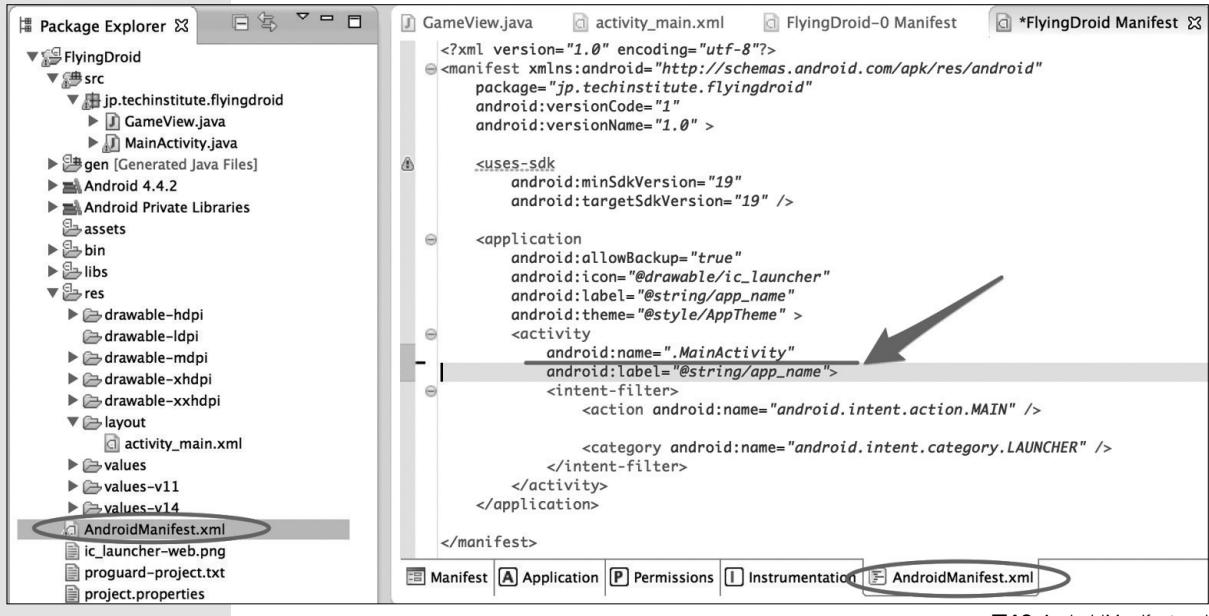


図16:AndroidManifest.xml

図16のようにプロジェクトの下にある「AndroidManifest.xml」をダブルクリックして開き、AndroidManifest.xmlのタブを押してXML編集の画面にして、図の矢印が示す部分に次の定義を追加します。1行目がフルスクリーン、2行目が横向きの設定です。

AndroidManifest.xml に追加する内容

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
android:screenOrientation="landscape"
```



## 9-1-4 スプライト描画

画像はAndroidでゲームを開発するお手本としてAndroidがこの世に出で間もない頃からあるRelica Islandのものを利用させてもらっています(URL <https://code.google.com/p/replica-island/>)。このゲームのソースコード、画像とともにApache 2.0 Licenseというオープンソースのライセンスで公開されているため、ここで利用することも可能になっています。

### リソースをダウンロードする

この後の作業に使用する画像ファイルを、ZIP形式にまとめてあるので、それをダウンロードします。ダウンロードするファイルのURLは「<http://goo.gl/jI1AjJ>」です。小文字のjの次は、大文字のI(アイ)です。小文字のl(エル)ではありません。

ダウンロードしたらZIP形式のファイルを展開し、中に入っている各PNG形式のファイルを「res」→「drawable-hdpi」内にコピーします。

## クラスの追加

ここから先では、障害物やビームなどの描画オブジェクトが増えることを踏まえて、共通する機能は「AbstractGameObject」クラスとして作成し、それを継承して自分で動かすキャラである「Droid」クラスを作成します。

## AbstractGameObjectクラスの作成

「Package Explorer」から「FlyingDroid」→「src」→「jp.techinstitute.flyingdroid」を開いて右クリックし、メニューから「New」→「Class」と選択します（図17）。

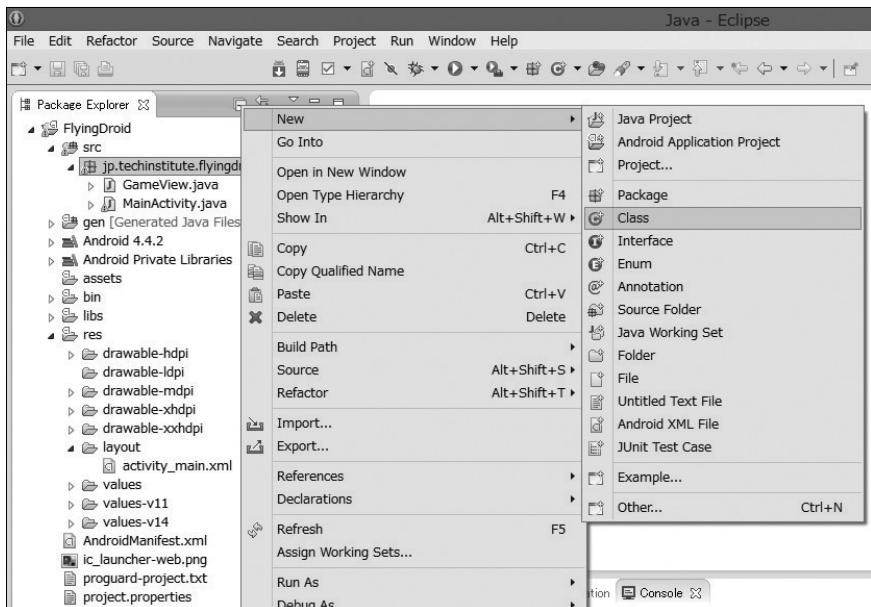


図17:新しいクラスの作成

「New Java Class」ダイアログが開いたら、「Name」欄に「AbstractGame Object」と入力。「abstract」にチェックを入れてから「Finish」ボタンを押します（図18）。ここで「abstract」にするのは、「AbstractGameObject」そのものでは新しいオブジェクトを生成できないようにして、かならずこのクラスを継承して作ったクラスでオブジェクトを生成するようにしたいためです。

先に説明した動物と犬、猫の関係に置き換えると、ベースとなる動物だけでは「鳴く」というメソッドを持つことが決まってはいても実装を持っていないので、それだけではオブジェクトを生成できず、それを継承した犬や猫は「鳴く」というメソッドの実装を持っているのでオブジェクトを生成できる…ということになります。

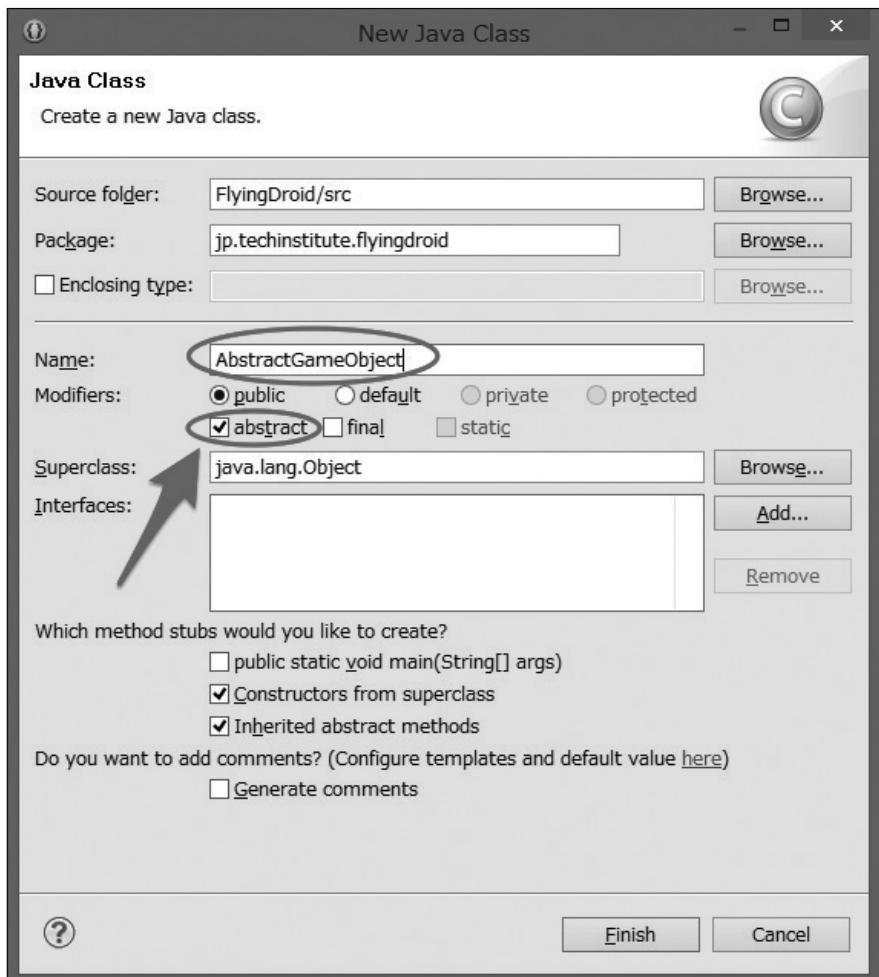


図18:New Java Classダイアログ - AbstractGameObject

作成したAbstractGameObject.javaを開くと図19のようになっています。

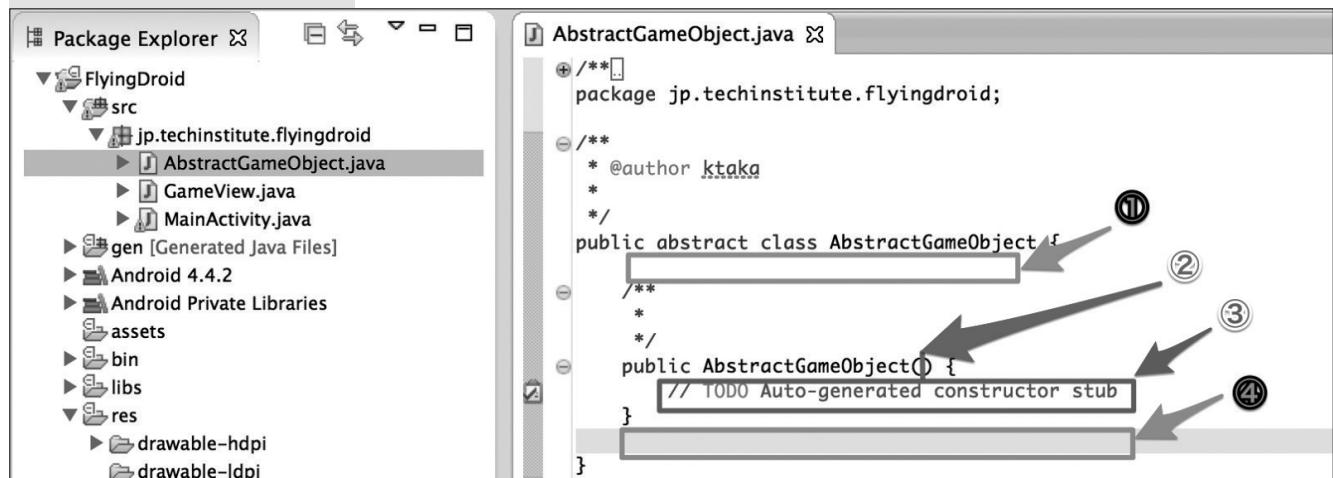


図19:AbstractGameObject.java

図19の①が示す部分にはメンバ変数を追加します。

AbstractGameObject.java の①に追加するメンバ変数

```
protected Drawable drawableImg;
protected int width;
protected int height;
protected int x;
protected int y;
```

それ順に、画像描画用の「Drawable」オブジェクトを保持するためのもの、画像の幅と高さ、描画位置のX座標、Y座標を保持するためのものです。**図19** の②が示す部分に引数(Context context, int resourceId, int width, int height)を追加します。それぞれ次のような意味です。

**context**… リソースを読み出すために必要なContextオブジェクト

**resourceId**… 画像を示すリソースID

**width, height**… 表示の際に使用する縦、横のサイズ

追加した結果は次のようになります。

#### ②に追加した内容

```
public AbstractGameObject(Context context, int resourceId, int width, int height) {
```

③の部分には次のコードを追加します。

#### ③に追加する内容

```
drawableImg = context.getResources().getDrawable(resourceId);
this.width = width;
this.height = height;
```

1行目で画像から描画のためのDrawableオブジェクトを生成し、2行目と3行目で表示用の幅と高さをメンバ変数に保存しています。④の部分には描画のためのメソッドを追加します。

#### ④に追加する内容

```
public void draw(Canvas c, int x, int y) {
    drawableImg.setBounds(x, y, x + width, y + height);
    drawableImg.draw(c);
}
```

上のコードでは、メソッドの中の1行目で描画位置とサイズを設定し、2行目で実際に描画しています。

ContextとCanvasの下についた赤線で示されるエラーは、マウスカーソルをその上に置いて表示されるエラー解決の選択肢からImportを選択して解決する。

## Droidクラスの作成

**図20**のように「Package Explorer」にある「FlyingDroid」→「src」→「jp.techinstitute.flyingdroid」の部分で右クリックすると出て来るメニューから「New」→「Class」の順に選択します。

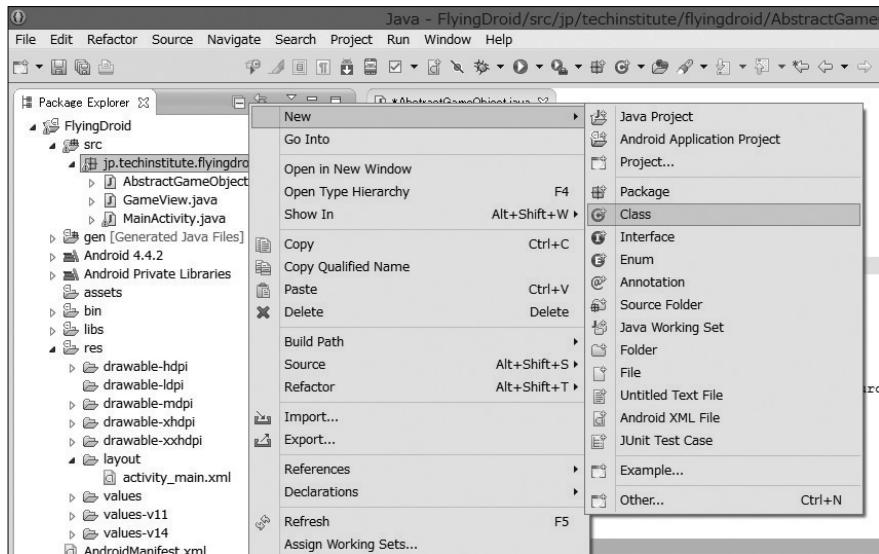


図20:新しいクラスの作成

「New Java Class」ダイアログが開くので、機能印をつけたように「Name」の枠に「Droid」と入力します。次に継承元のクラスを設定するために「Browse」ボタンを押します(図21)。



図21:New Java Class ダイアログ

継承元クラスを選択する「Superclass Selection」ダイアログが開かれるので「Choose a type」の枠に入っている「java.lang.object」を消して、代わりに「abstractga」と入力します。するとこの場合には1つだけ候補が表示されるので、「AbstractGameObject -jp.techinstitute.flyngdroid」を選択して「OK」を押します(図22)。

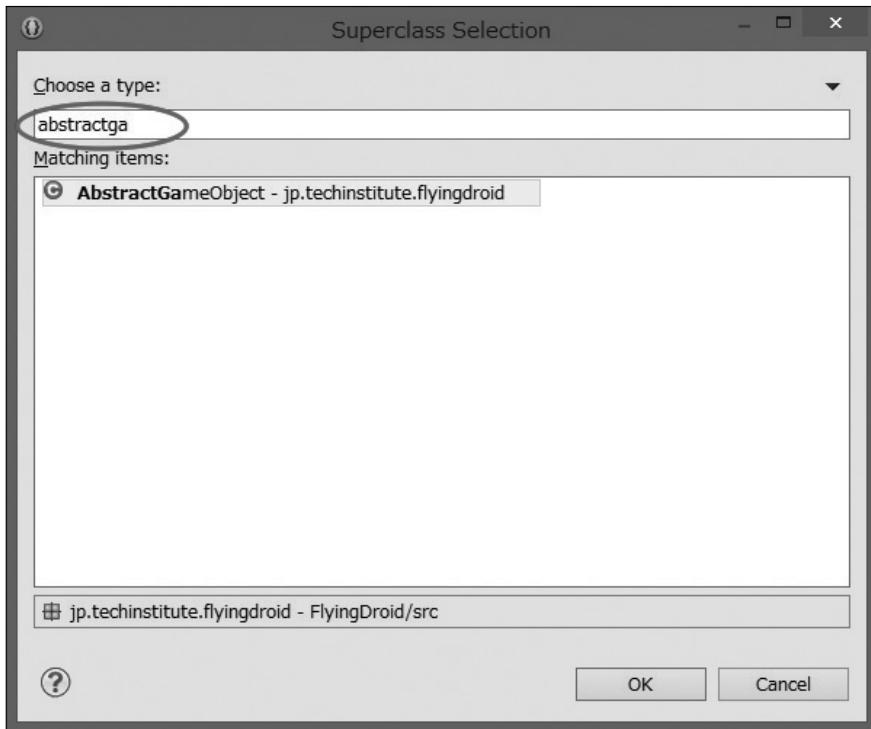


図22:Superclass Selectionダイアログ

New Java Classダイアログに戻ってきます(図23)。「Superclass」が「jp.techinstitute.flyngdroid.AbstractGameObject」になっていることを確認し、「Constructors from superclass」「Inherited abstract methods」「Generate comments」の3つにチェックを入れて、「Finish」ボタンを押します。

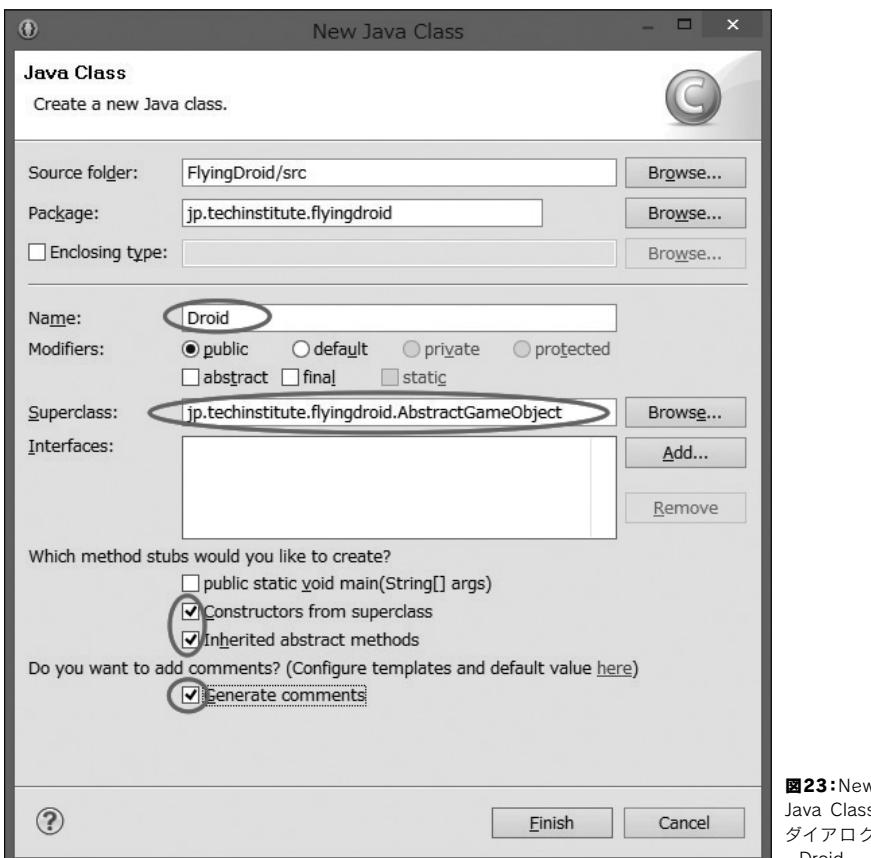


図23:New Java Classダイアログ - Droid

作成したDroid.javaを開くと図24のようになっています。

```
package jp.techinstitute.flyingdroid;  
import android.content.Context;  
/** * @author ktaka */  
public class Droid extends AbstractGameObject {  
    /** * @param context * @param resourceId * @param width * @param height */  
    public Droid(Context context, int resourceId, int width, int height) {  
        super(context, resourceId, width, height);  
        // TODO Auto-generated constructor stub  
    }  
}
```

図24:Droid.java

図24の①で示される引数は削除します。その代わりに、継承元を呼び出す行（superと書いてある部分）の②で示される引数を、直接Droidオブジェクトで表示する画像のリソースID「R.drawable.andou\_diag01」で書き換えます。その結果、次のようにになります。

図24で書き換える内容

```
public Droid(Context context, int width, int height) {  
    super(context, R.drawable.andou_diag01, width, height);  
}
```

## GameView.javaの修正

作成したDroidクラスを使って描画を行うよう修正します。図25のようにGameView.javaを開きます。

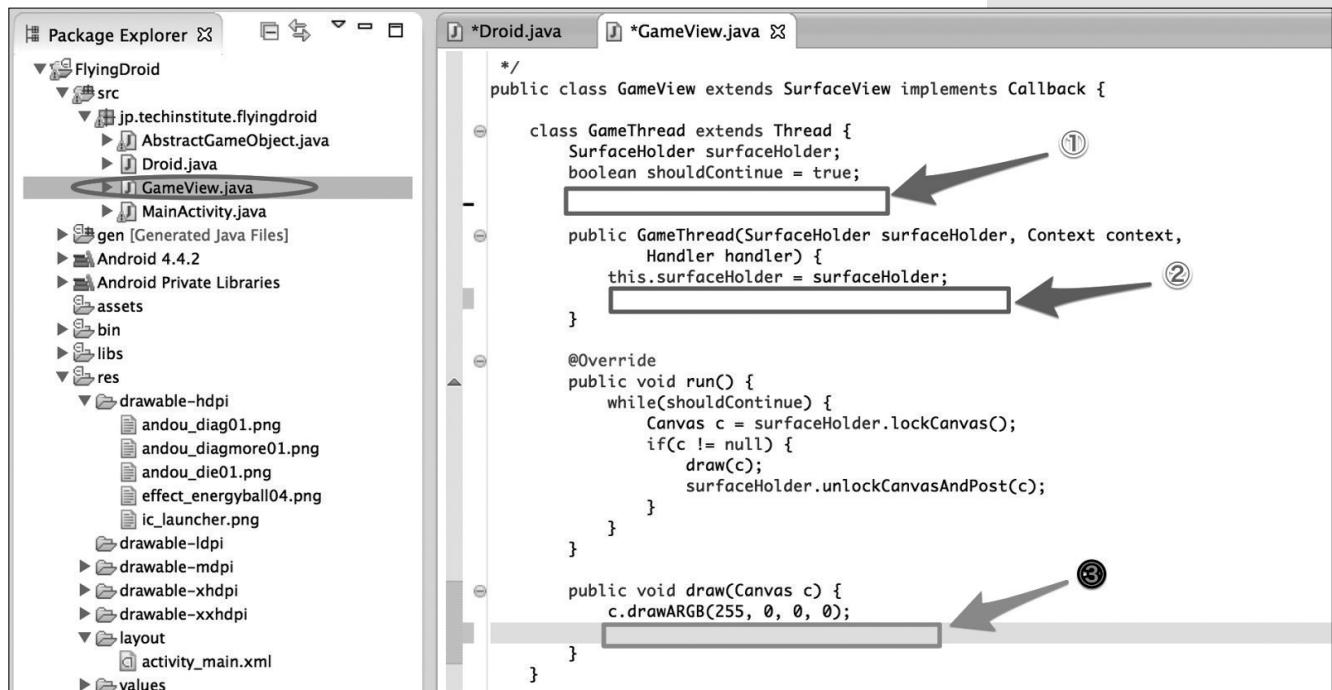


図25:GameView.java

図25の①で示す部分にはメンバ変数として定義を追加します。

GameView.java の①に定義を追加

```

Droid droid;
static final int droidSize = 200;

```

1行目は「Droid」オブジェクトを保持するためのもの、2行目は表示の際のサイズを定義するための定数です。図25の②にはメンバ変数にnewで生成したオブジェクトを代入するコードを追加します。

②にコードを追加

```

droid = new Droid(context, droidSize, droidSize);

```

③にはドロイドを描画するためのコードを追加します

③にコードを追加

```
droid.draw(c, 100, 100);
```

引数で渡す「c」は描画するための「Canvas」、2つ目と3つ目の「100」は仮置きのためのX,Y座標です。

ここまでできたら、実行してみましょう。正しくできていれば、図26のように表示されるはずです。

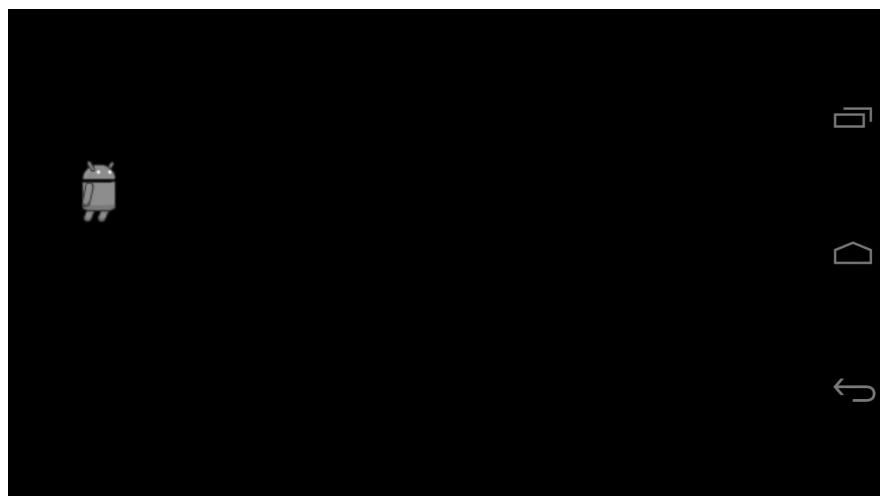


図26:表示結果

# 9-2 ゲームによる実践（2）

著：高橋憲一

ゲームというからには、ユーザーの操作によって画面上のオブジェクトが反応するというインタラクティブな要素が必要です。この節ではそれを実装していきます。



## この節で学ぶこと

- ・これまで学んで来たことを組合わせてゲームアプリを作成
- ・自分のキャラを操作する
- ・障害物を表示させて自分のキャラとぶつかったか判定する

この節で出て来るキーワード、Android SDKのクラス、外部ライブラリーの一覧

当たり判定

スレッド

Android SDKのクラス

SurfaceView

SurfaceHolder

Canvas

Thread



## 9-2-1 Graphical Layout を使ってレイアウトを作ってみよう

自分で操作するキャラ（スプライト）の動きを考えてみます。操作をしなければスプライトが少しづつ下降していくようにします。画面をタップをすれば上昇し、離すとまた下降するようにしてみましょう。

## スプライトの下降

自分のキャラとなるDroidオブジェクトに、初期表示位置を設定します。その後は画面の下に落ちていくようにする機能を追加します。まずは「Droid.java」を開きましょう(図1)。

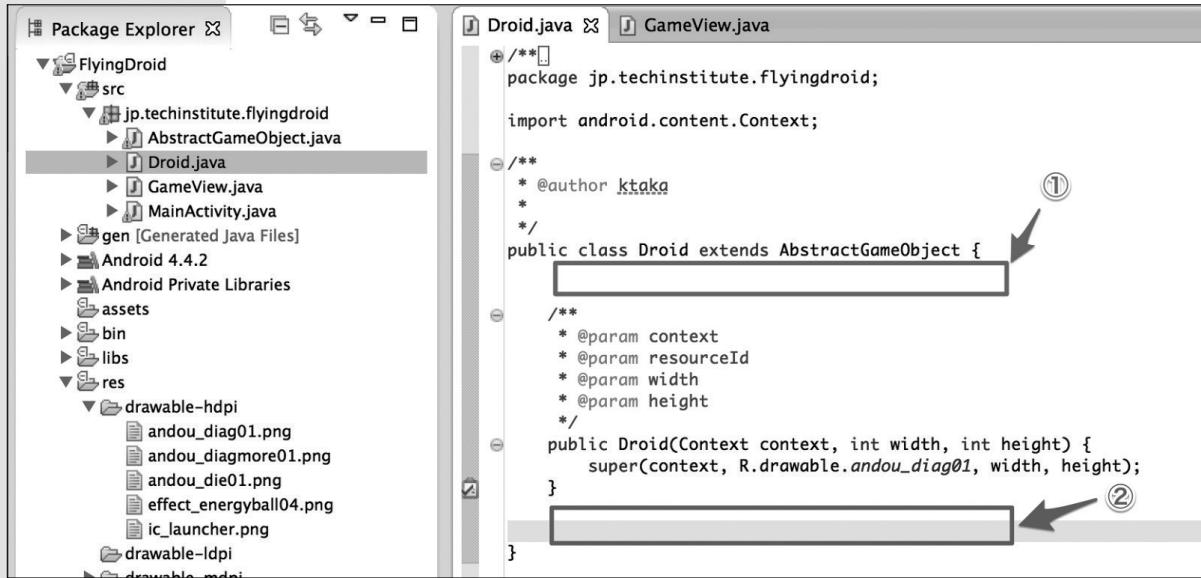


図1:Droid.java

図1の①で示した部分にはメンバ変数を定義します。

Droid.java の①へ追加するメンバ変数の定義

```
private int defaultX;
private int defaultY;
private float velocity = 2;
```

図1の「defaultX」と「defaultY」は初期表示位置のX座標とY座標を保持するためのもの、yは現在のY座標、「velocity」は移動速度を保持するための変数です。

②で示した部分には、以下の2つのメソッドを追加します。

②に追加するメソッドの内容

```
public void setInitialPosition(int x, int y) {
    defaultX = x;
    defaultY = y;
    this.x = defaultX;
    this.y = defaultY;
}

public void draw(Canvas c) {
    draw(c, defaultX, y);
    y += velocity;
}
```

1つ目の「setInitialPosition」は、初期表示の位置をX座標とY座標で指定するためのものです。このとき、初期表示位置を代入するだけでなく、Y座標の現在地を保持するyにも初期表示位置を代入しておきます。2つ目の「draw(Canvas c)」は、毎フレーム呼び出されて現在のY座標で指定した位置で描画し、次のフレームのためのY座標を、速度の値を足すことで計算します。

ここで画面の座標系を確認しておきます。図2のように左上が原点(X,Yが0,0)、横方向はX、縦方向はYで、Xの値を増やすと右に移動し、Yの値を増やすと下に移動します。

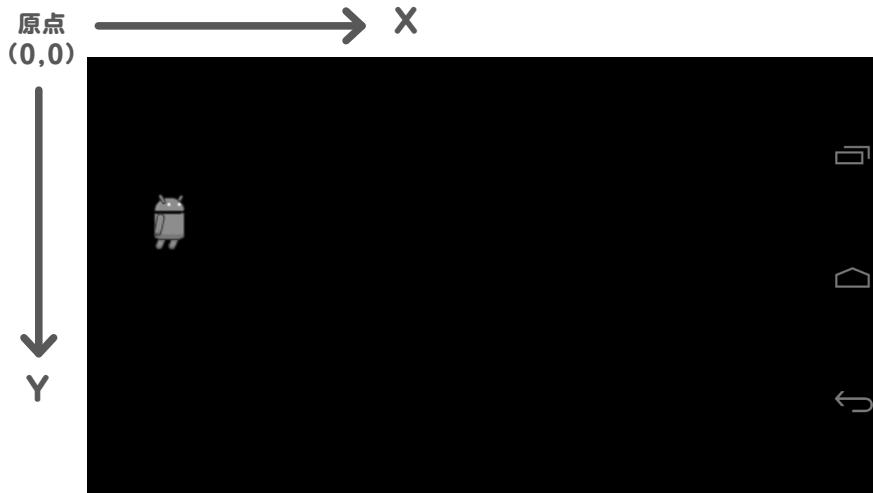


図2:画面の座標系

次に「GameView.java」を開いて「Droid.java」に追加した機能を呼び出すためのコードを書き入れます(図3)。

```

/*
 * 
 */
public class GameView extends SurfaceView implements Callback {
    class GameThread extends Thread {
        SurfaceHolder surfaceHolder;
        boolean shouldContinue = true;

        Droid droid;
        static final int droidSize = 200;

        public GameThread(SurfaceHolder surfaceHolder, Context context,
                Handler handler) {
            this.surfaceHolder = surfaceHolder;
            droid = new Droid(context, droidSize, droidSize);
        }

        @Override
        public void run() {
            while(shouldContinue) {
                Canvas c = surfaceHolder.lockCanvas();
                if(c != null) {
                    draw(c);
                    surfaceHolder.unlockCanvasAndPost(c);
                }
            }
        }

        public void draw(Canvas c) {
            c.drawRGB(255, 0, 0);
            droid.draw(c, 100, 100);
        }
    }
}

```

図3:GameView.java

メソッド名や変数名には、その機能や目的を類推しやすいような名前をつけることが望ましいです。

図3の①が示す部分には初期表示位置を設定するメソッドを呼び出すコードを追加します。

①に追加するメソッドを呼び出すためのコード

```
droid.setInitialPosition(100, 0);
```

図3の②が示す部分のdroidオブジェクトの「draw」メソッドは、直接描画位置を設定するものではなく、先ほど追加した描画するための「Canvas」のみを渡すものに書き換えます（描画位置の指定は、droidオブジェクトの中での計算に任せます）。

②に追加する Canvas のみを渡すためのコード

```
droid.draw(c);
```

ここで一度実行してみましょう。正しくできていれば、ドロイド君の画像が下に移動していくはずです。

## タッチで上昇させる

タッチ操作を検出し、画面に指が触れている間はドロイド君が上昇するようにします。逆に、指を画面から離すと再び下降を始めるようにしてみます。図4のようにDroid.javaを開きます。

```
*Droid.java  GameView.java  GameView.java
import android.content.Context;
/*
 * @author ktaka
 */
public class Droid extends AbstractGameObject {
    private int defaultX;
    private int defaultY;
    private int y;
    private float velocity = 2;
    /**
     * @param context
     * @param resourceId
     * @param width
     * @param height
     */
    public Droid(Context context, int width, int height) {
        super(context, R.drawable.andou_diag01, width, height);
    }
    public void setInitialPosition(int x, int y) {
        defaultX = x;
        defaultY = y;
        this.y = defaultY;
    }
    public void draw(Canvas c) {
        draw(c, defaultX, y);
        y += velocity;
    }
}
```

図4:Droid.java

図4の①が示す部分の変数の定義を次のように修正します。

図4 ①の変数の定義

```
private static final float DefaultVelocity = 2;
private float velocity = DefaultVelocity;
```

1行目でデフォルトの速度の値を定数として定義しておき、2行目では実際のY座標を計算する時の処理で使用する「velocity」の値の初期値として代入します。

②の示す部分には次のメソッドを追加します。

②に追加するメソッド

```
public void uplift(boolean on) {
    if (on) {
        velocity = -DefaultVelocity;
    } else {
        velocity = DefaultVelocity;
    }
}
```

このメソッドを引数「on」に「true」をセットして呼ぶことで、「velocity」がマイナスの値になり、図4の③で示される行の計算「y += velocity」でY座標が減少します。動作としてはドロイド君は上昇していくことになります。逆に「on」に「false」をセットして呼び出すと、「velocity」は正の値になり、Y座標は増えていてドロイド君は下降することになります。あとは「GameView.java」の方でタッチ操作に合わせてこの「uplift」メソッドを呼び出すようにすれば、期待する動作になります。

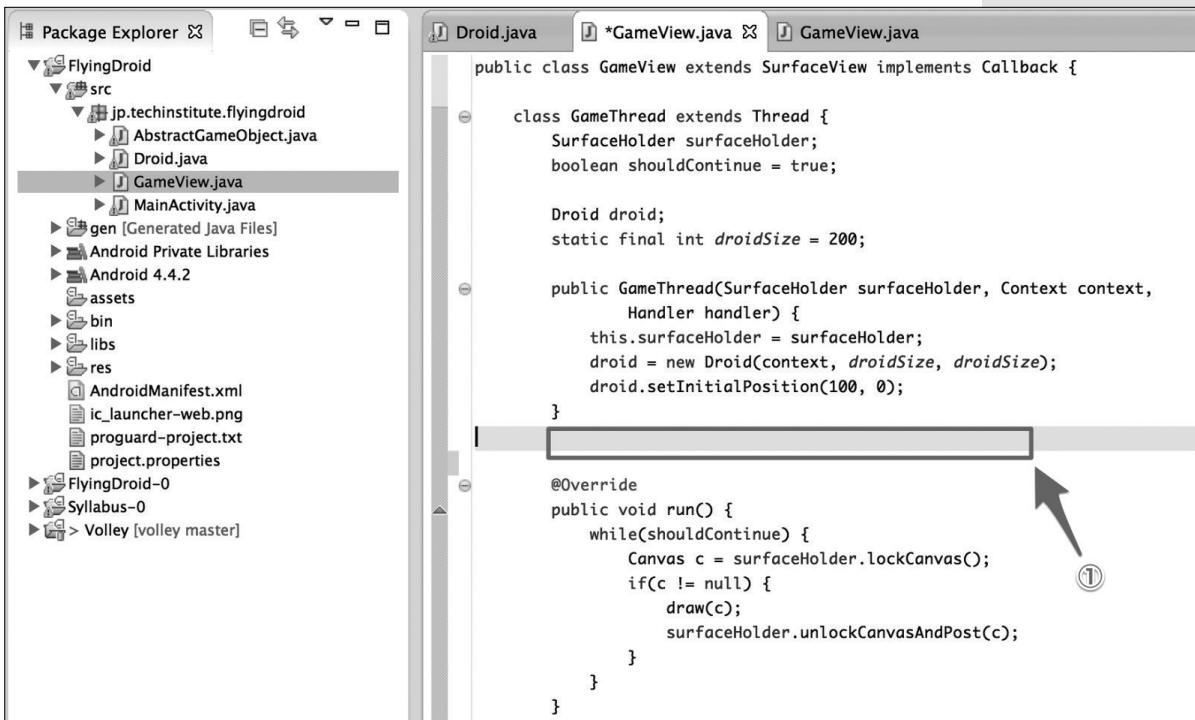


図5:GameView.java(前半)

図5の①が示す部分に次のメソッドを追加します。

①に追加するメソッド

```
public void upliftDroid(boolean on) {  
    droid.uplift(on);  
}
```

ここで、自分が動かすキャラ(ドロイド君のオブジェクト)の「uplift」というメソッドを呼び出しています。さらにこの「upliftDroid」メソッドを、タッチ操作のイベントが発生した際に呼び出すようにします。図6のように「GameView.java」の後半にスクロールします。

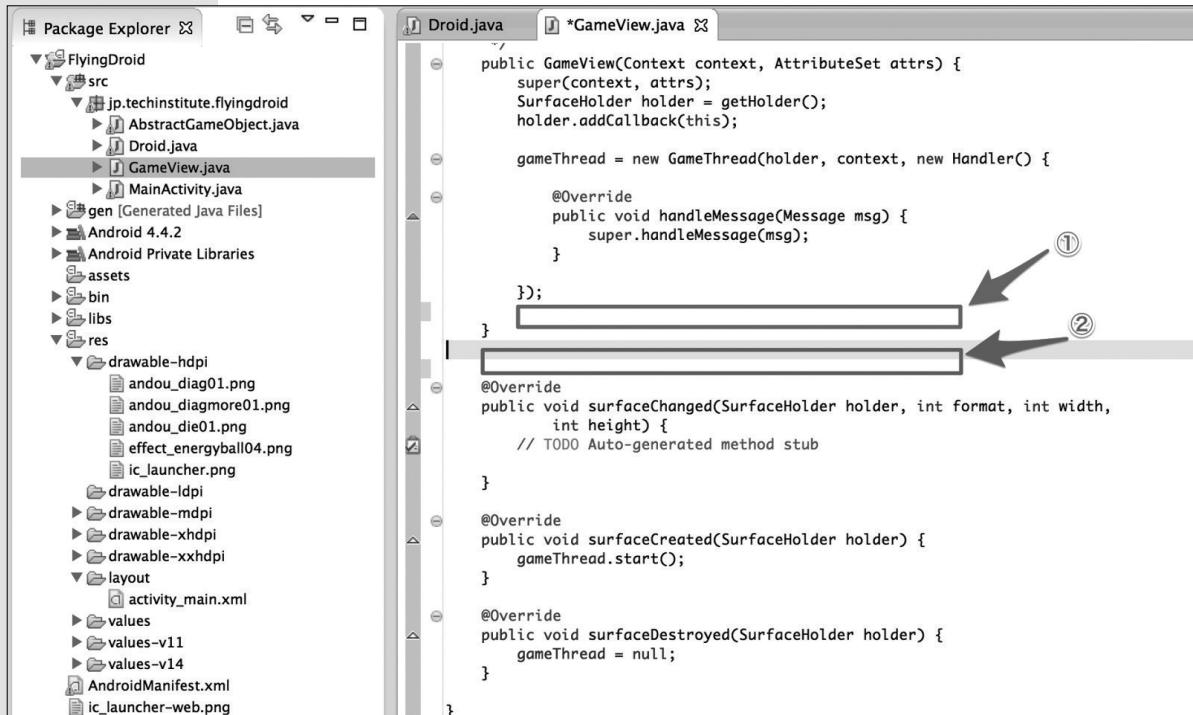


図6:GameView.java(後半)

まず先に図6の②が示す部分に次のメソッドを追加します。

②に追加する内容

```
private boolean dispatchEvent(MotionEvent event) {  
    switch(event.getAction()) {  
        case MotionEvent.ACTION_DOWN:  
            gameThread.upliftDroid(true);  
            return true;  
        case MotionEvent.ACTION_UP:  
            gameThread.upliftDroid(false);  
            return false;  
        default:  
            return false;  
    }  
}
```

ここでは「switch」を使って、「event.getAction()」で得られたアクションの種別によって処理を分岐させています。「MotionEvent.ACTION\_DOWN」の場

合は指を画面にタッチした時なので、「upliftDroid」メソッドに「true」をセットして呼び出して上昇させるようにします。「MotionEvent.ACTION\_UP」の場合は指を画面から離した時なので、「upliftDroid」メソッドに「false」をセットして呼び出し、下降するようにします。

次に、図6の①が示す部分に以下のコードを追加します。自動補完を使うことで、大部分の入力を省略できます。「setOnTouchListener(new View.OnTouchListene」まで入力すると、補完候補として「View.OnTouchListener()」が表示されます(図7)ので、これを選択します。補完候補が表示されない場合は、[Ctrl]+[Space]キーを押してみて下さい。

①に追加するコード

```
setOnTouchListener(new View.OnTouchListener() {

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        dispatchEvent(event);
        return false;
    }
});
```

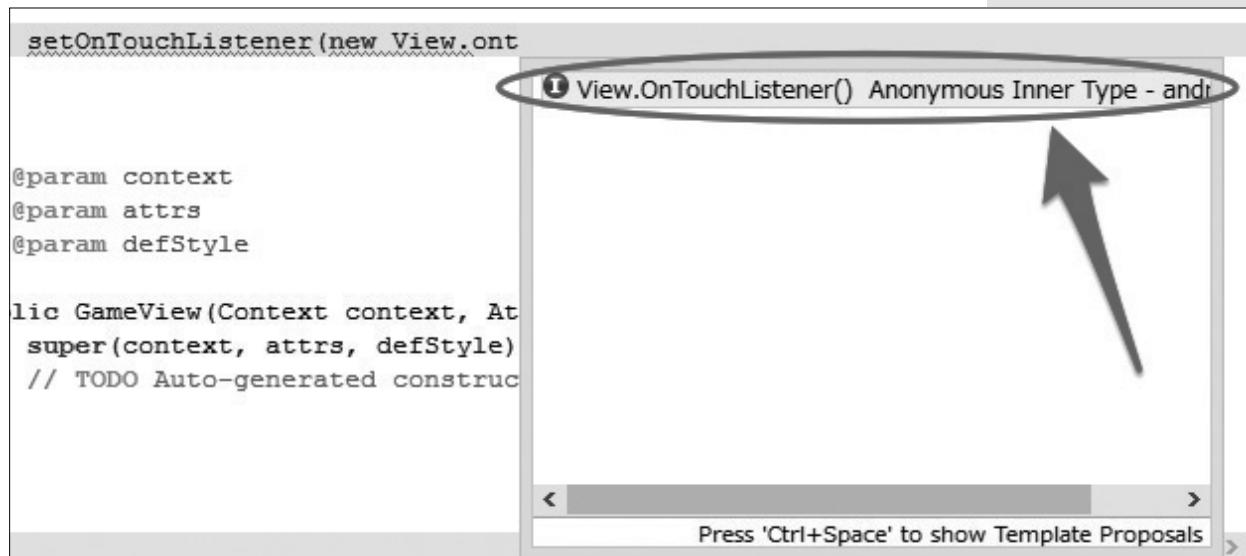


図7:View.OnTouchListenerの補完

自動補完された直後は図8のようになっています。

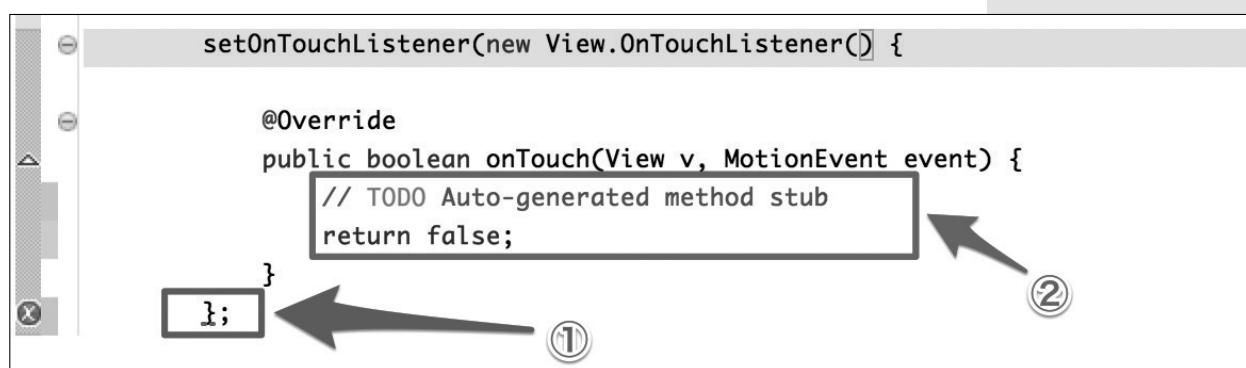


図8:OnTouchListener自動補完後

まず図8の①で示される部分のエラーは括弧の対応の問題なので、「});」のようにしてセミコロンの前に閉じる括弧を追加すると解決します。次に②で示す部分は次の1行に修正します。

#### ②の修正内容

```
return dispatchEvent(event);
```

この部分(View.onTouchListenerのonTouchメソッド)が、画面のタッチ操作のイベントが発生した時に呼び出される部分になります。ここまでできたら実際に動かして、タッチによる反応を確かめてみましょう。

## 移動範囲の制限

ここでひとつ問題に気付くと思います。ドロイド君の上昇、下降ともに画面の外に出た際の対策をまだしていません。方法としては、ドロイド君(Droidクラス)に画面の座標範囲の設定メソッドを追加すればいいのですが、このメソッドはこの後に追加する障害物のクラスでも必要になるはずです。そのため、継承元となるクラス「AbstractGameObject」に追加して、共通で使用できるようにしておきます。

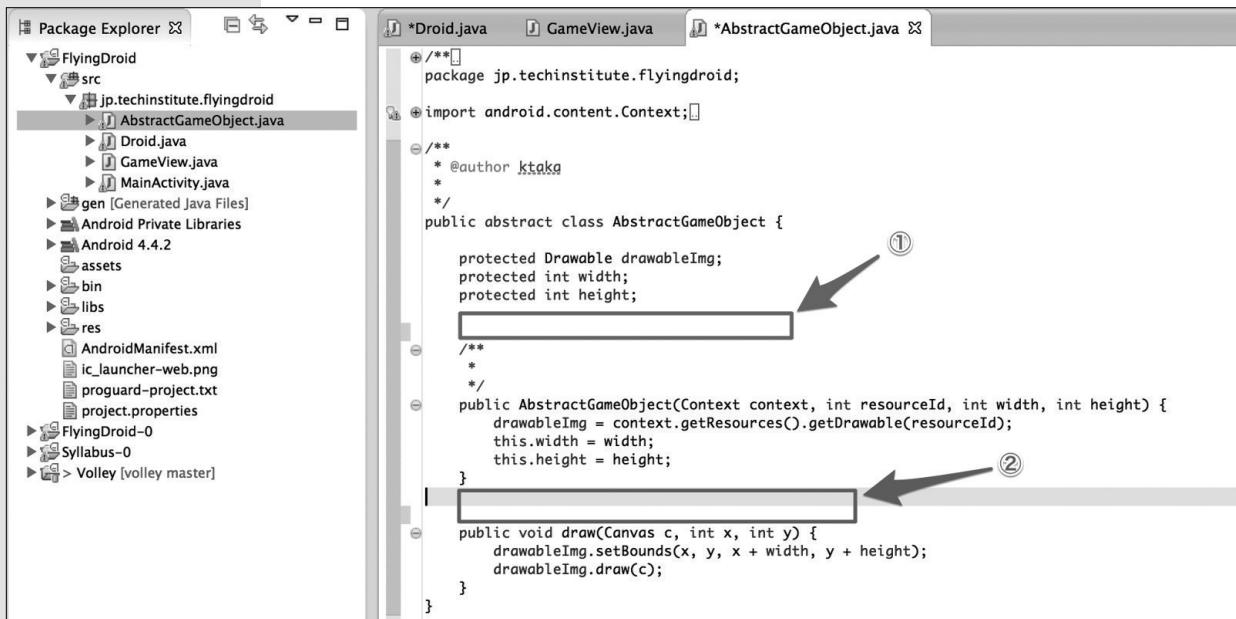


図9:AbstractGameObject.java

図9の①が示す部分には、画面の範囲を保持するメンバ変数を追加します。

#### ①に追加するメンバ変数

```
protected int left;
protected int top;
protected int right;
protected int bottom;
```

図9の②が示す部分には、画面の範囲の値をセットするためのメソッドを追加します。

②に追加するメソッド

```
public void setMovingBoundary(int left, int top, int right, int bottom) {
    this.left = left;
    this.top = top;
    this.right = right;
    this.bottom = bottom;
}
```

描画時の範囲チェックは「Droid」クラスの中で行います。

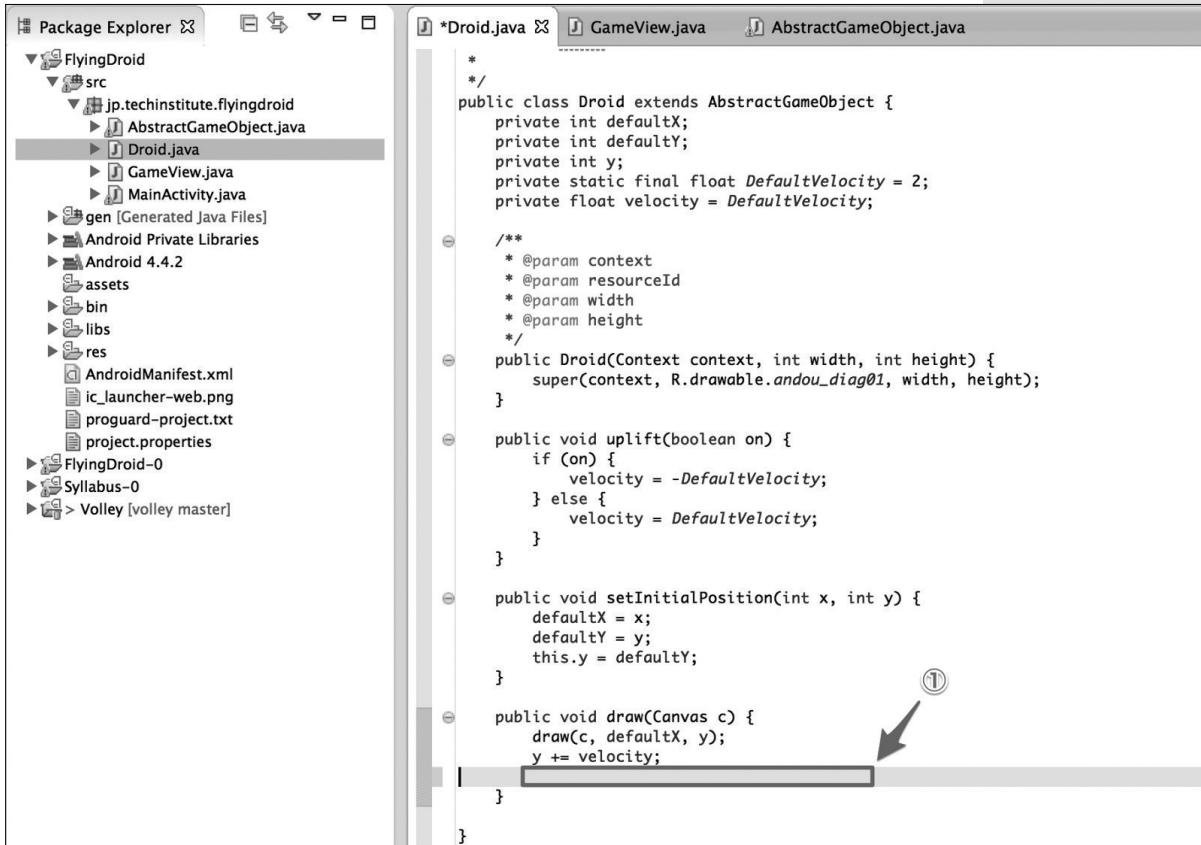


図10:Droid.java

図10の①が示す部分に次のコードを追加します。

①に追加するコード

```
if (y < top) {
    y = top;
} else if (y > bottom) {
    y = bottom;
}
```

Y座標の計算後に上方向(top)、および下方向(bottom)の範囲を超える可能性があるなら、それぞれの限界値を代入しておきます。そうすれば、範囲外に描画されて見えなくなる現象を防げます。次に「GameView」側で範囲の値をDroidオブジェクトにセットする処理を追加します(図11)。

```

    /*
     * 
     */
    public class GameView extends SurfaceView implements Callback {
        ...
        class GameThread extends Thread {
            SurfaceHolder surfaceHolder;
            boolean shouldContinue = true;
            ...
        }
        Droid droid;
        static final int droidSize = 200;

        public GameThread(SurfaceHolder surfaceHolder, Context context,
                          Handler handler) {
            this.surfaceHolder = surfaceHolder;
            droid = new Droid(context, droidSize, droidSize);
            droid.setInitialPosition(100, 0);
        }
        ...
        public void upliftDroid(boolean on) {
            droid.uplift(on);
        }
    }

```

図11:GameView.java前半

図11の①で示される部分には、画面の幅と高さを保持するメンバ変数を追加します。

①へのメンバ変数の追加

```

int width;
int height;

```

図11の②で示される部分には、この幅と高さをセットするメソッドを追加します。

②へのメソッドの追加

```

public void setViewSize(int width, int height) {
    this.width = width;
    this.height = height;

    droid.setMovingBoundary(0, 0, width, height);
}

```

メンバ変数に値をセットしたあとは、droidオブジェクトの「setMovingBoundary」メソッドを呼び出して、原点(0,0)と幅、高さをセットします。さらに、画面のサイズが変更されたときや初期化時に「setViewSize」を呼び出すようにします(図12)。

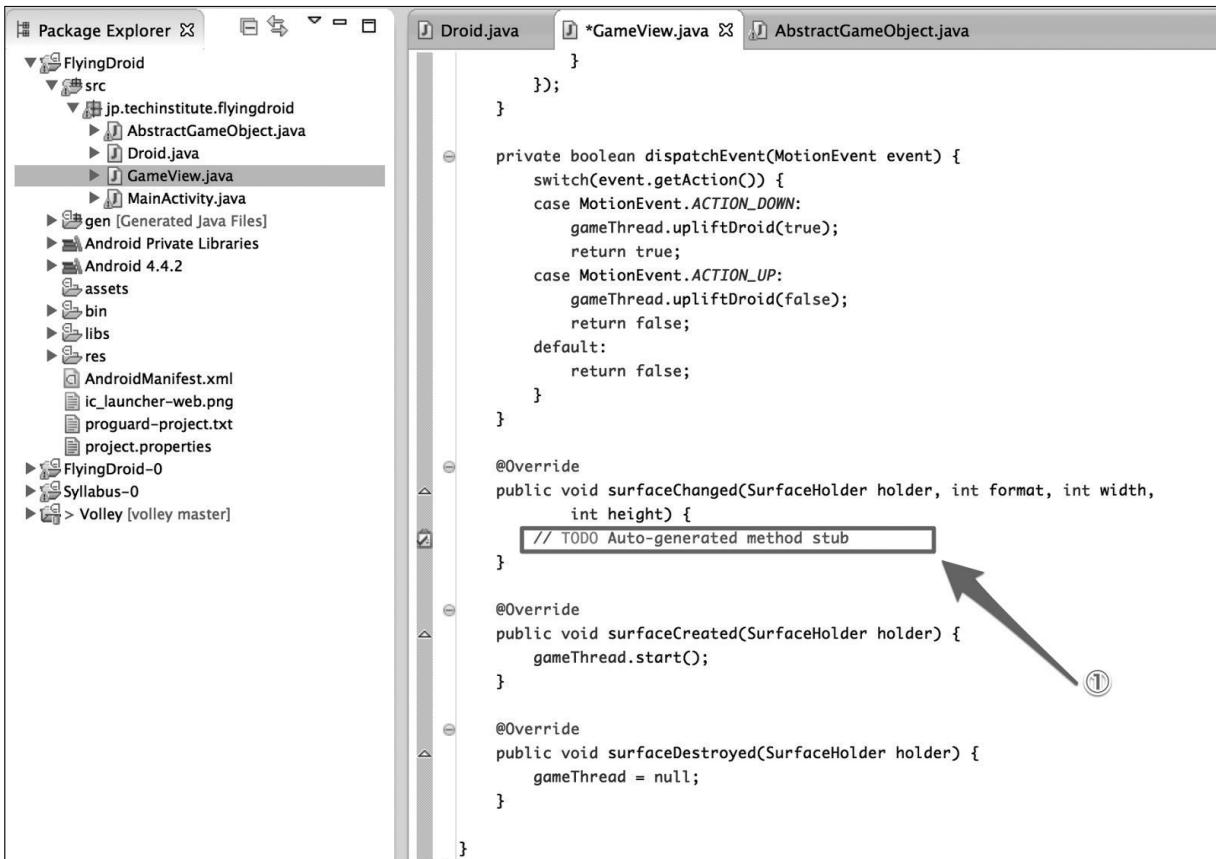


図12:GameView.java後半

図12の①で示される部分、`surfaceChanged`というメソッドの中は、画面(`SurfaceView`)のサイズが変更された時に、その幅と高さの引数を伴って呼び出される部分です。ここに次のコードを追加します。

①に追加する内容

```
gameThread.setViewSize(width, height);
```

ここで一度実行してみましょう。上方向、下方向ともに画面の端でドロイド君が止まってくれるでしょうか。正しくコードを入力できていれば上方向は問題ないはずです。しかし、下方向は画面の外に出てしましますね。これはミスではなく処理がひとつ足りないためです。「`Droid.java`」にコードを追加して、問題を修正してみましょう。図13のように「`Droid.java`」を開きます。

```

/*
 * 
 */
public class Droid extends AbstractGameObject {
    private int defaultX;
    private int defaultY;
    private int y;
    private static final float DefaultVelocity = 2;
    private float velocity = DefaultVelocity;

    /**
     * @param context
     * @param resourceId
     * @param width
     * @param height
     */
    public Droid(Context context, int width, int height) {
        super(context, R.drawable.andou_diag01, width, height);
    }

    public void uplift(boolean on) {
        if (on) {
            velocity = -DefaultVelocity;
        } else {
            velocity = DefaultVelocity;
        }
    }

    public void setInitialPosition(int x, int y) {
        defaultX = x;
        defaultY = y;
        this.y = defaultY;
    }

    public void draw(Canvas c) {
        draw(c, defaultX, y);
        y += velocity;
        if (y < top) {
            y = top;
        } else if (y > bottom) {
            y = bottom;
        }
    }
}

```

図13:Droid.java

ここでコードを手で入力するのではなく、「AbstractGameObject」の持っている「setMovingBoundary」メソッドをオーバーライドしてみましょう。「Droid.java」を開いている状態でEclipseのメニューバーから「Source」→「Override/Implement Methods」を選択します(図14)。

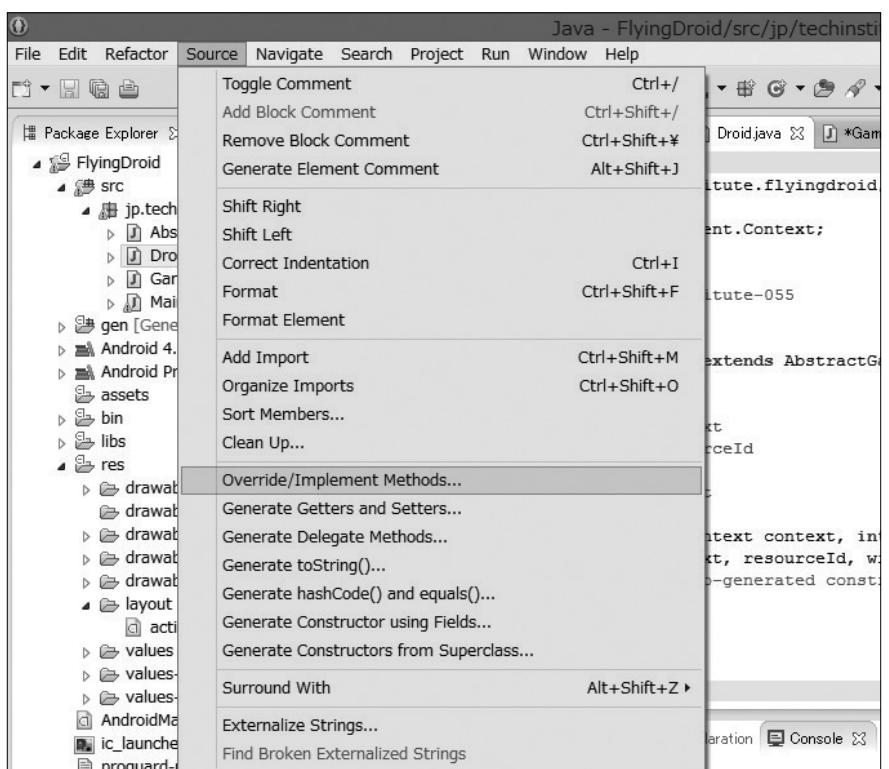


図14:EclipseのSourceメニュー

すると「Override/Implement Methods」ダイアログが開きますので、オーバーライドするメソッドを指定します(図15)。

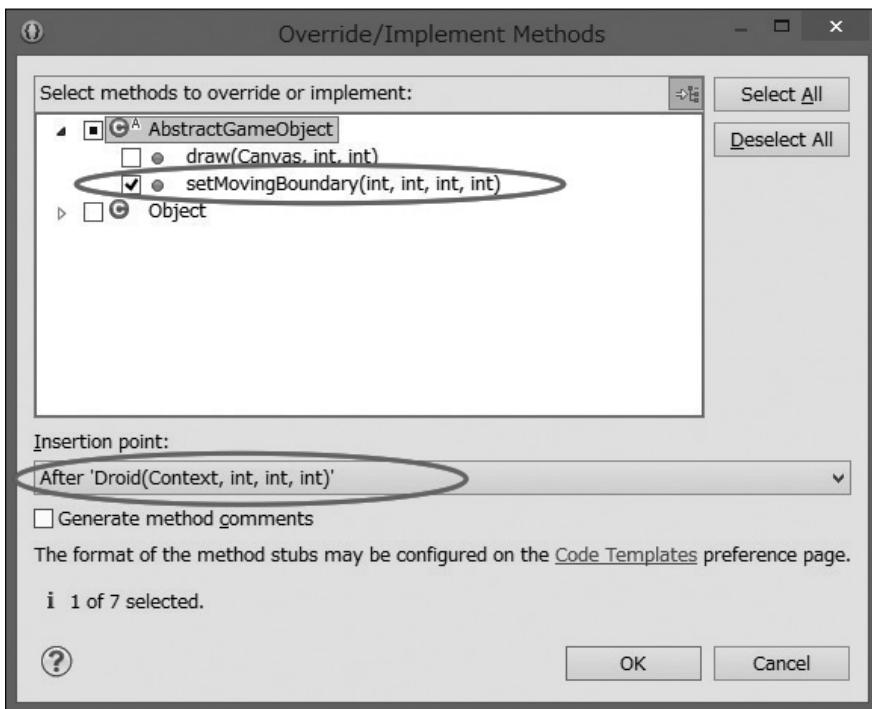


図15:Override/Implement Methodsダイアログ

図15の「setMovingBoundary」にチェックを入れて「Insertion Point」欄を「After‘Droid(Context,int,int,int)」にして「OK」ボタンを押します。すると図16のように「setMovingBoundary」をオーバーライドするコードが自動で追加されます。

```

public Droid(Context context, int width, int height) {
    super(context, R.drawable.andou_diag01, width, height);
}

@Override
public void setMovingBoundary(int left, int top, int right, int bottom) {
    // TODO Auto-generated method stub
    super.setMovingBoundary(left, top, right, bottom);
}

public void uplift(boolean on) {
    if (on) {
        velocity = -DefaultVelocity;
    } else {
        velocity = DefaultVelocity;
    }
}

```

A large black arrow points from the 'OK' button in the dialog box to the 'super.setMovingBoundary(left, top, right, bottom);' line in the code editor. A small circled number '1' is located in the bottom right corner of the code editor window.

図16:setMovingBoundary

図16の①で示す部分に、画面の下方向の限界値を補正するコードを追加します。

①に追加する内容限界値を補正するコード

```
this.bottom -= height;
```

図16①の上の行にある「super.setMovingBoundary(left,top,right,bottom);」で継承元の同名メソッドを呼び出していく、その後に上記のような処理を加えることでDroidクラス独自の値にすることが可能になります。

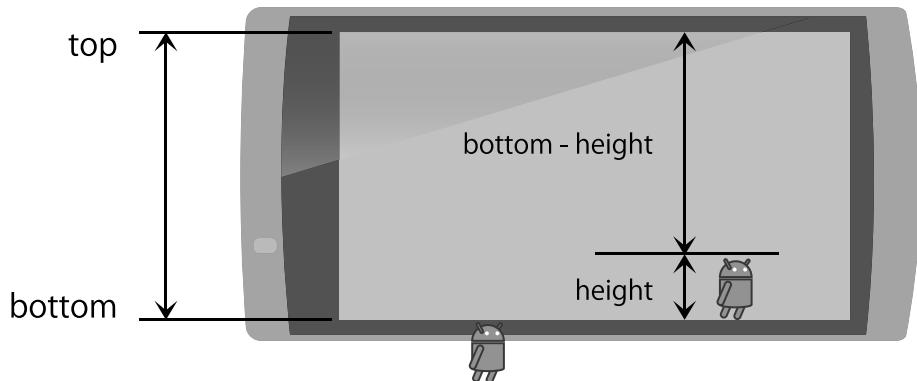


図17:描画範囲の設定

図17のように限界値をbottomに指定すると左側のドロイド君のように画面からはみでてしまいます。そのため、画像の高さであるheight分を引いた値(bottom - height)を限界値にすることで、右側のドロイド君のようにちょうど画面の下端で止まるようにします。実行して確かめてみましょう。



## 9-2-2 障害物を表示して動かす

続いて、障害物を表示するためのクラスを作成します。

### Enemyクラスの作成

Package Explorerの「FlyingDroid」→「src」の下にある「jp.techinstitute.flyingdroid」を右クリックして、メニューから「New」→「Class」とたどります（図18）。

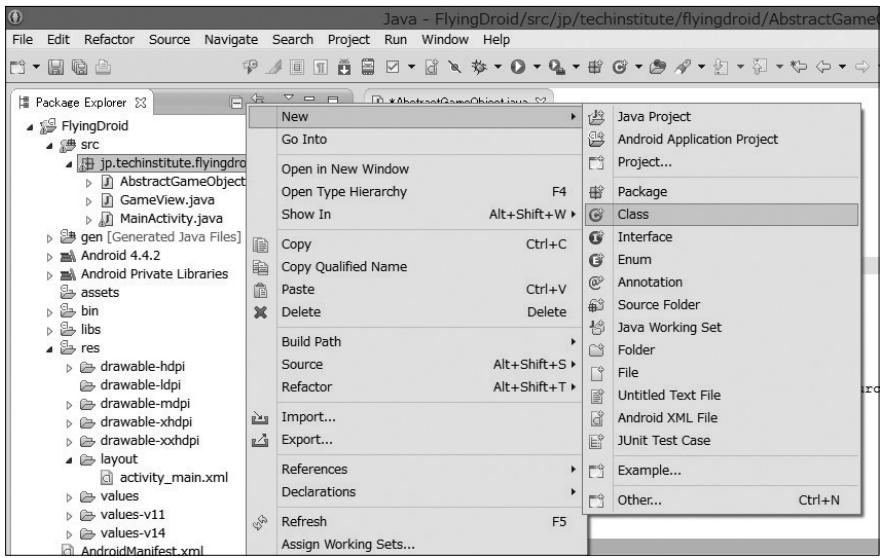


図18:新しいクラスの作成

「New Java Class」ダイアログが開いたら「Name」欄に「Enemy」と入力します。次に、継承元のクラスを設定するために「Browse」ボタンを押します(図19)。

図19:New Java Class  
ダイアログ

継承元クラスを選択する「Superclass Selection」ダイアログが開かれるので、「Choose a type」欄の「java.lang.object」を消して「abstractga」と入力します。すると、ここでは候補が1つだけ表示されるので、「AbstractGameObject - jp.techinstitute.flynigdroid」を選択して「OK」を押します(図20)。

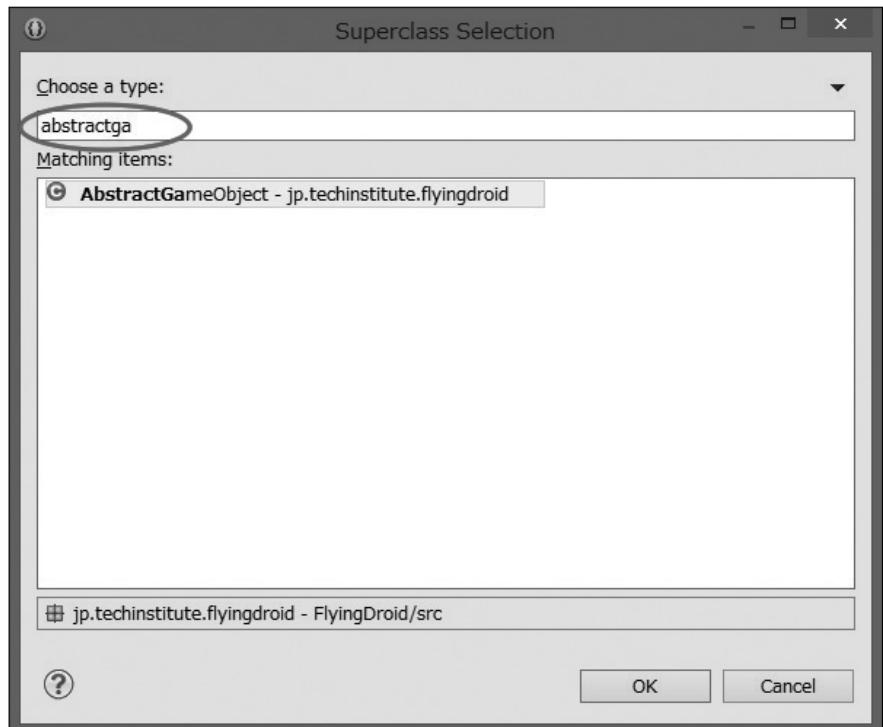


図20:Superclass Selectionダイアログ

「New Java Class」ダイアログに戻ります。「Superclass」が「jp.techinstitute.flynigdroid.AbstractGameObject」になっていることを確認して「Constructors from superclass」「Inherited abstract methods」「Generate comments」の3つにチェックを入れて「Finish」ボタンを押します(図21)。

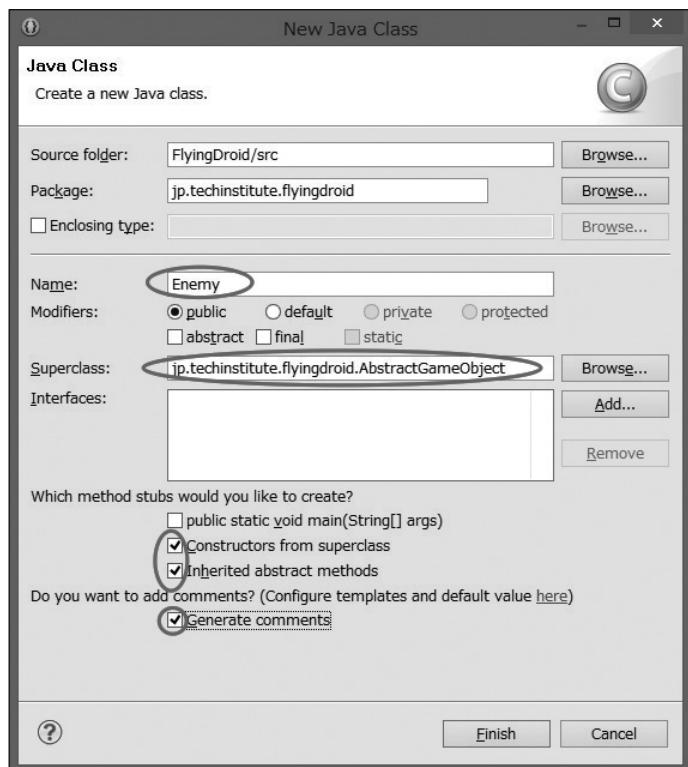


図21:New Java Class  
ダイアログ -Enemy

ダイアログが閉じたら、図22の①で示される引数は削除しておきます。

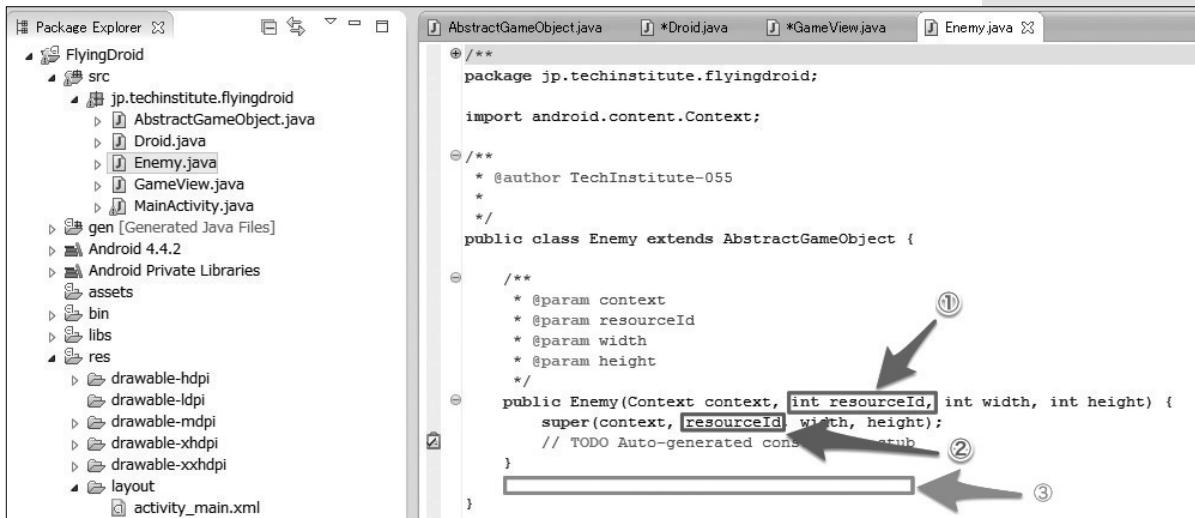


図22:Enemy.java

その代わり、継承元を呼び出す行(superと書いてある部分)の②で示される引数を直接「Enemy」オブジェクトで表示する画像のリソースID「R.drawable.enemy\_pinkdude\_jump」で書き換えます。その結果、次のようにになります。

#### ②を書き換えた内容

```

public Enemy(Context context, int width, int height) {
    super(context, R.drawable.enemy_pinkdude_jump, width, height);
}

```

図20の③で示される部分に、次のメソッドを追加します。

#### ③に追加するメソッド

```

public void draw(Canvas c) {
    draw(c, x, y);
    x -= 5;
    if (x < left) {
        x = right;
    }
}

```

このメソッドでは、X,Y座標を指定した描画メソッドを呼び出し、次のフレームのためのX座標を求めて、もし左端の限界値を超えるならその限界値以上にはならないようにしています。ここで「AbstractGameObject」の持っている「setMovingBoundary」メソッドをオーバーライドします。

「Enemy.java」を開いている状態でEclipseのメニューバーから「Source」→「Override/Implement Methods」を選択します(図23)。

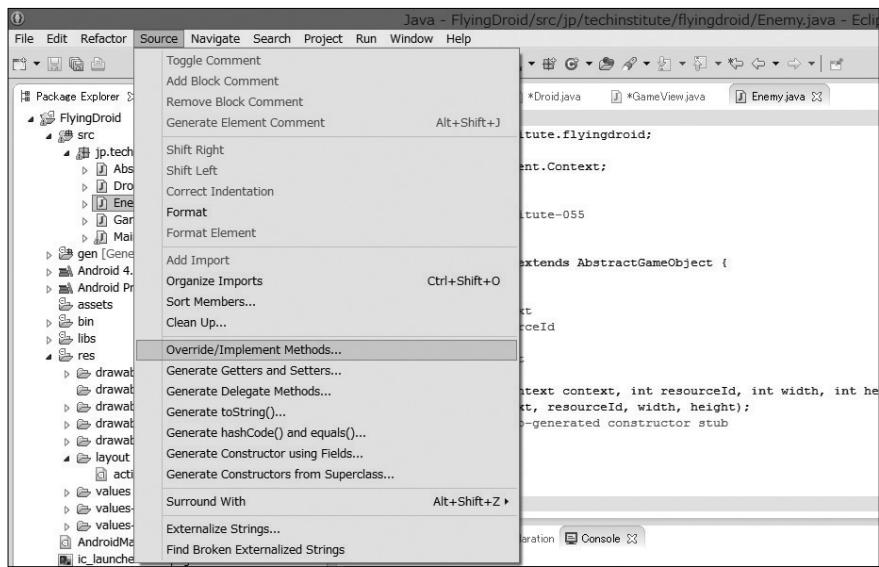


図23:EclipseのSourceメニュー

すると図24のような「Override/Implement Methods」ダイアログが開かれますので、オーバーライドするメソッドを指定します。

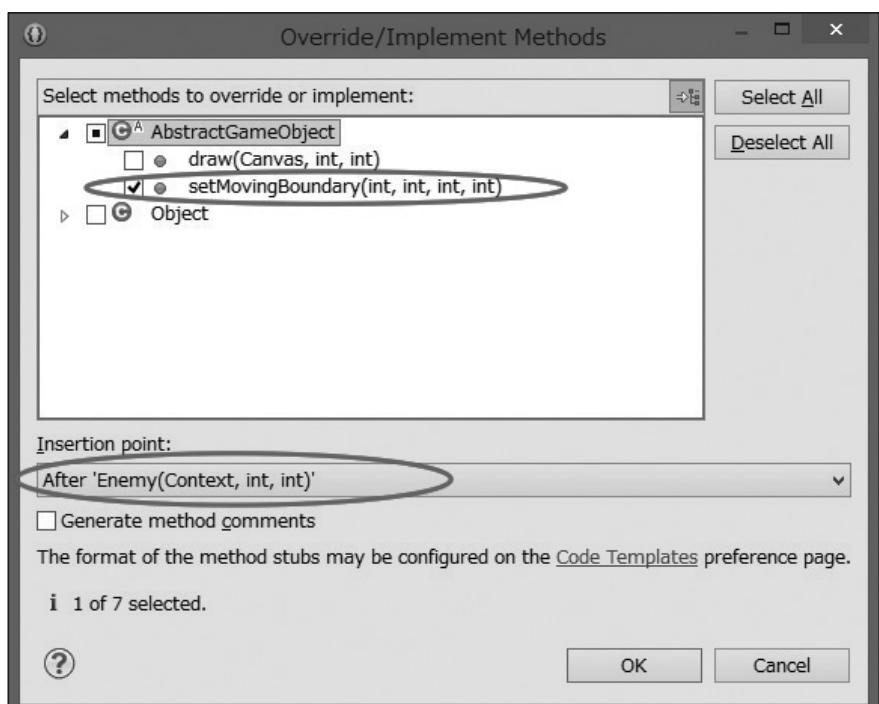


図24:Override/Implement Methodsダイアログ

「setMovingBoundary」にチェックを入れ、「Insertion Point」欄は「After 'Enemy(Context, int, int, int)'」を選択してから「OK」ボタンを押します(図24)。するとsetMovingBoundaryをオーバーライドするコードが自動で追加されます(図25)。

```
public class Enemy extends AbstractGameObject {  
  
    /**  
     * @param context  
     * @param resourceId  
     * @param width  
     * @param height  
     */  
    public Enemy(Context context, int width, int height) {  
        super(context, R.drawable.enemy_pinkdude_jump, width, height);  
    }  
  
    @Override  
    public void setMovingBoundary(int left, int top, int right, int bottom) {  
        // TODO Auto-generated method stub  
        super.setMovingBoundary(left, top, right, bottom);  
    }  
  
    public void draw(Canvas c) {  
        draw(c, x, y);  
        x -= 5;  
        if (x < left) {  
            x = right;  
        }  
    }  
}
```

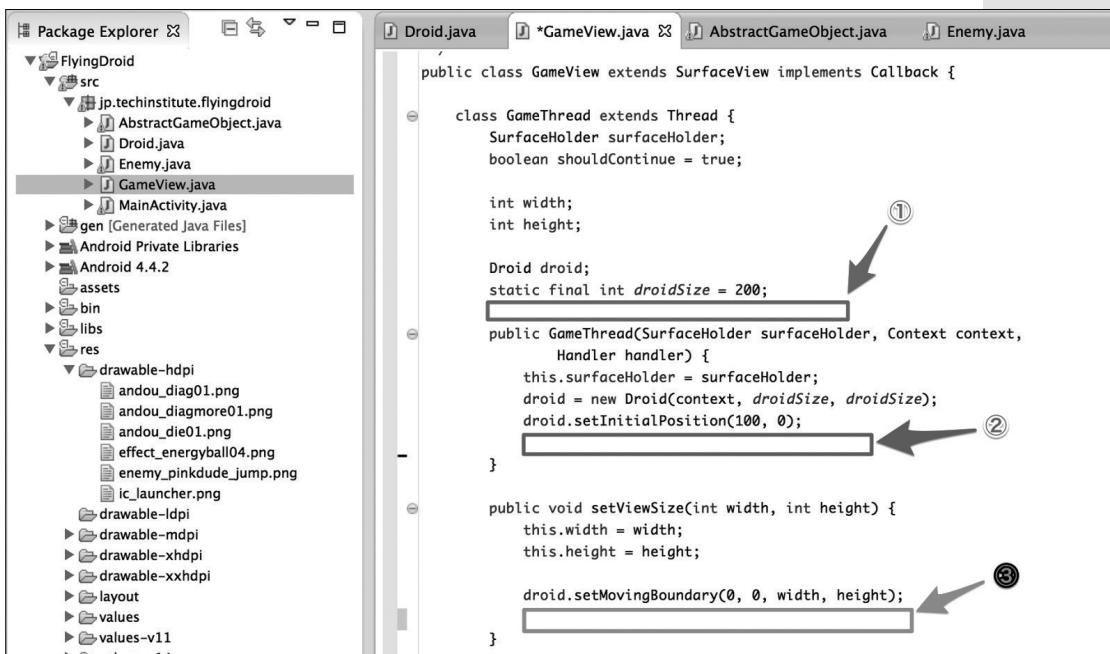
## ■ 25: setMovingBoundary

図25の①で示す部分に画面の左方向の限界値を補正するコードと、X, Y座標の初期値を設定するコードを追加します。

## ①に追加する内容

```
left -= width;  
x = right;  
y = 300;
```

①の上の行にある「super.setMovingBoundary(left, top, right, bottom);」は継承元の同名メソッドを呼び出していることになり、その後に上記のような処理を加えることで、「Enemy」クラス独自の処理にできます。次に「GameView.java」にEnemyの処理を加えます。GameView.javaを開きます(図26)。



## ■ 26: GameView.java

図26の①で示される部分にはメンバ変数としての定義を追加します。

①に追加する内容

```
Enemy enemy;  
static final int enemySize = 200;
```

②で示される部分でEnemyオブジェクトを生成するコードを追加します。

②に追加する内容

```
enemy = new Enemy(context, enemySize, enemySize);
```

③で示される部分には画面の範囲を設定するコードを追加します。

③に追加する内容

```
enemy.setMovingBoundary(0, 0, width, height);
```

次に「GameView.java」を開いたままで、少し下にスクロールします(図27)

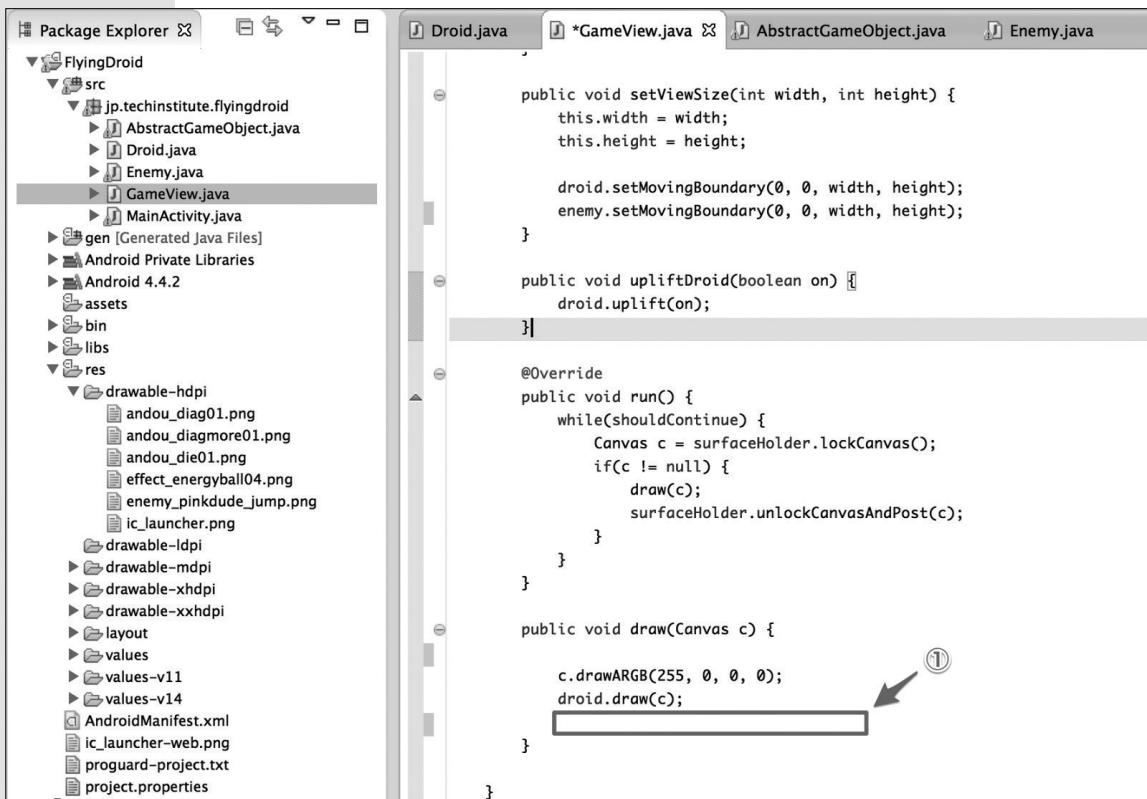


図27:GameView.java 続き

図27の①で示される部分にenemyの描画のためのコードを追加します。

①に追加する enemy の描画

```
enemy.draw(c);
```

ここで一度実行してみましょう。

正しくできていれば、図28のように障害物が表示され、右から左へ移動しているはずです。



図28:障害物の表示

### 9-2-3 当たり判定

オブジェクト同士の当たり判定にはさまざまな手法がありますが、ここでは円と円との衝突を判定する方法で行います。やや大雑把な判定にはなりますが、しくみが簡単で計算量も少ないためコード量も少なくて済みます。

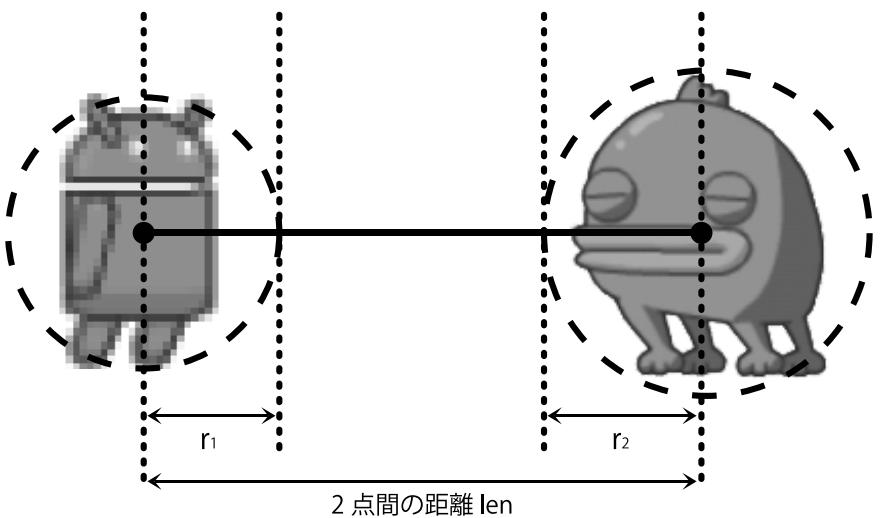


図29:当たり判定のしくみ1

図29のように、2つのオブジェクトを包み込む円があると考えます。その2つの円の中心と中心の2点間の距離( $len$ )と、2つの円の半径の和( $r_1+r_2$ )を比較することで、当たりを判定します。この図の場合は「 $len > (r_1+r_2)$ 」となり、当たっていないと判定されます。

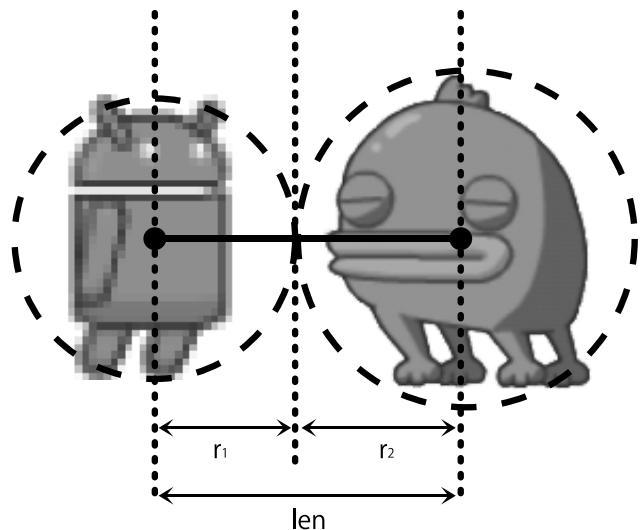


図30:当たり判定のしくみ2

一方、図30の場合は「 $len \leq (r_1 + r_2)$ 」となり、当たっていると判定されます。

## 当たり判定コードの追加

当たり判定のコードは「AbstractGameObject」に追加して、継承されたオブジェクトから共通で使用できるようにします。

まずは「AbstractGameObject.java」を開きます(図31)。

```

package jp.techinstitute.flyingdroid;

public abstract class AbstractGameObject {
    protected Drawable drawableImg;
    protected int width;
    protected int height;

    protected int left;
    protected int top;
    protected int right;
    protected int bottom;

    protected int x;
    protected int y;
}

public AbstractGameObject(Context context, int resourceId, int width, int height) {
    drawableImg = context.getResources().getDrawable(resourceId);
    this.width = width;
    this.height = height;
}

public void setMovingBoundary(int left, int top, int right, int bottom) {
    this.left = left;
    this.top = top;
    this.right = right;
    this.bottom = bottom;
}

public void draw(Canvas c, int x, int y) {
    drawableImg.setBounds(x, y, x + width, y + height);
    drawableImg.draw(c);
}
}

```

図31:AbstractGameObject.java

①の部分に円の半径を保持するためのメンバ変数の定義を追加します。

①に追加するメンバ変数の定義

```
protected double radius;
```

次に、②で示される部分に半径を求めるコードを追加します。ここでは幅として設定した値を2分の1にした値を用いることにします。

②に追加する半径を求めるコード

```
radius = width * 0.5;
```

最後に、③で示した部分に当たり判定のメソッドを追加します。

③に追加する当たり判定のメソッド

```
public boolean isHit(AbstractGameObject obj) {
    double xlen = (x + radius) - (obj.x + obj.radius);
    double ylen = (y + radius) - (obj.y + obj.radius);
    double len = Math.sqrt((xlen * xlen) + (ylen * ylen));
    double radiusSum = radius + obj.radius;
    if (len <= radiusSum) {
        return true;
    } else {
        return false;
    }
}
```

処理の流れは次のようになります。

- ・引数で自分自身と当たり判定の対象となるオブジェクトを受け取ります。
- ・最初の2行ではX, Y成分それぞれの距離を求めます。
- ・次の行でX, Yの成分から三平方の定理で2点間の距離を計算しています（`Math.sqrt`は平方根を求める関数です）。
- ・自分自身と当たり判定の対象となるオブジェクトの半径の和を求めます。
- ・2点間の距離が半径の和よりも小さいか等しければ当たつていると判定し、大きければ当たっていないと判定します。

この当たり判定メソッドをGameViewの中から呼び出す処理を追加します。図

32のように「GameView.java」を開きます。

The screenshot shows the Eclipse IDE interface. On the left is the Package Explorer view, displaying the project structure for 'FlyingDroid'. The 'src' folder contains several Java files: AbstractGameObject.java, Droid.java, Enemy.java, GameView.java (which is selected), and MainActivity.java. It also includes generated files in the 'gen' folder and various resource folders like 'assets', 'bin', 'libs', and 'res' with subfolders for different screen densities (hdpi, ldpi, mdpi, xhdpi, xxhdpi) containing PNG images such as 'andou\_diag01.png', 'andou\_diagramore01.png', 'andou\_die01.png', 'effect\_energyball04.png', 'enemy\_pinkdude\_jump.png', and 'ic\_launcher.png'. On the right is the Droid.java code editor, showing methods like setViewSize, upliftDroid, and run. Below it is the GameView.java code editor, showing the draw(Canvas c) method. A red circle labeled '①' is placed over the opening brace of the draw(Canvas c) method in GameView.java. An arrow points from this circle to the corresponding closing brace in the code.

```
public void setViewSize(int width, int height) {  
    this.width = width;  
    this.height = height;  
  
    droid.setMovingBoundary(0, 0, width, height);  
    enemy.setMovingBoundary(0, 0, width, height);  
}  
  
public void upliftDroid(boolean on) {  
    droid.uplift(on);  
}  
  
@Override  
public void run() {  
    while(shouldContinue) {  
        Canvas c = surfaceHolder.lockCanvas();  
        if(c != null) {  
            draw(c);  
            surfaceHolder.unlockCanvasAndPost(c);  
        }  
    }  
}  
  
public void draw(Canvas c) {  
    c.drawRGB(255, 0, 0, 0);  
    droid.draw(c);  
    enemy.draw(c);  
}
```

図32:GameView.java

図32の①で示される部分に次のコードを追加します。

①に追加する内容

```
if (enemy.isHit(droid)) {  
    droid.setImageResource(R.drawable.andou_die01);  
}
```

enemyとdroidが当たっていたら、「enemy.isHit」は「true」を返します。その場合はdroidのスプライト画像を置き換える処理をしています。スプライト画像を置き換えるメソッドは「AbstractGameObject」に追加します。まず、AbstractGameObject.javaを開きます(図33)。

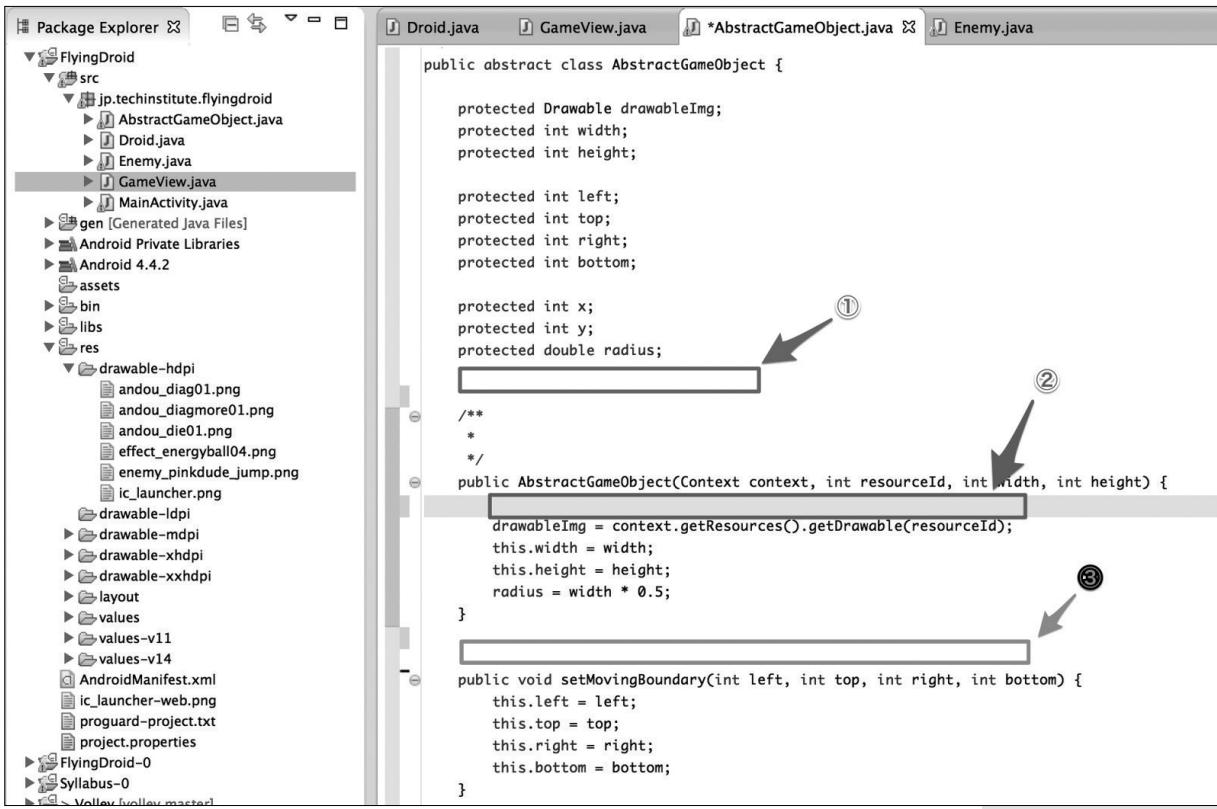


図33:AbstractGameObject.java

図33の①で示される部分には、リソースの読み込みに必要なContextを保持するメンバ変数を追加します。

①に追加するメンバ変数

```
protected Context context;
```

②で示される部分で「Context」をメンバ変数に代入します。

②に追加する内容

```
this.context = context;
```

③で示される部分に、スプライト画像を置き換えるメソッドを追加します。

③に追加する画像を置き換えるメソッド

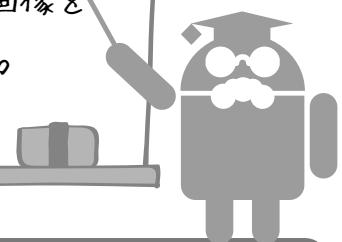
```
public void setImageResourceId(int resourceId) {
    drawableImg = context.getResources().getDrawable(resourceId);
}
```

ここまでできたら実行してみましょう。障害物は1つで、出る位置も固定ですが、当たり判定もついて、これで最低限のインタラクティブな要素が揃いました。

# 9-3 ゲームによる実践（3）

著：高橋憲一

前の節までで必要なクラスはほぼ揃いました。ここでは複数の障害物を発生させたり、操作によってスプライト画像を切り替えるなど、よりゲームの面白みを高めていきます。



## この節で学ぶこと

- ・これまで学んで来たことを組み合わせてひとつのアプリを作成する
- ・ゲームの実装を通して配列を使った複数のオブジェクトの生成と管理
- ・フレーム番号を意識したタイミングの制御

この節で出て来るキーワード、Android SDKのクラス、外部ライブラリーの一覧

配列

乱数

Hit Point

Random

SoundPool

MediaPlayer

SurfaceView

SurfaceHolder

Canvas

Thread



## 9-3-1 ワンパターンではすぐ飽きる！ ランダムに障害物を発生させるには

毎回同じ場所から障害物が発生してもゲームとしては面白みがありません。そこで、障害物の出現位置をランダムにします。

## ゲームで重要な「乱数」とは

Javaには「Random」というクラスがあり、サイコロを振ると毎回違う目が出てくるようにランダムな数(乱数)を発生させることができます。障害物を発生させる時のY座標をこれを使って求めることで、ゲーム性を高められます。

## Enemy.javaの修正

「Enemy.java」に乱数発生のコードを追加します。まず、図1のように「Enemy.java」を開きます。

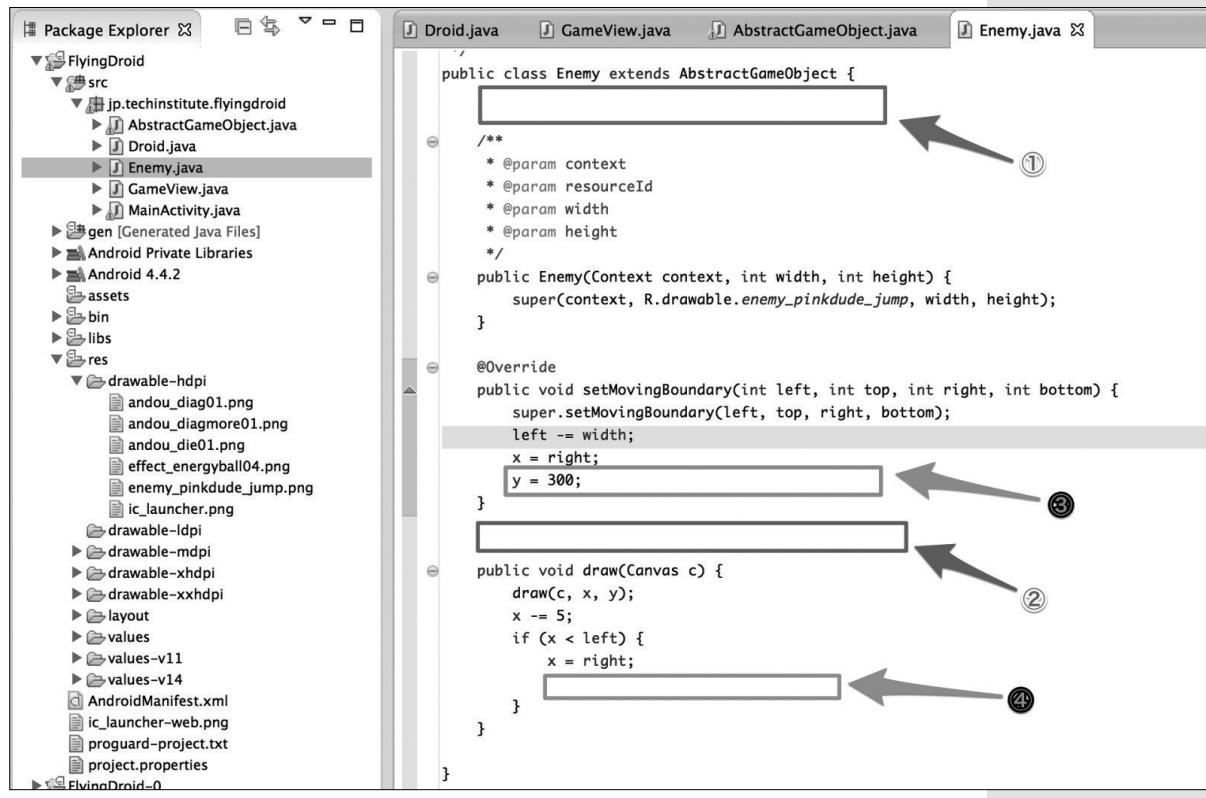


図1:Enemy.java

図1の①が示す部分に、乱数を発生させるためのRandomの定義と初期化のコードを追加します。ここでstaticをつけているのは、この後にEnemyを複数発生させることを踏まえてのものです。

①に追加する初期化のコード

```
static Random random;

static {
    random = new Random(System.currentTimeMillis());
}
```

「new Random()」の引数に「System.currentTimeMillis()」を使って現在の時間を渡しているのは、アプリを起動するたびに違う数を発生させる目的で、乱数のシードに毎回違うものを与えるためです。②が示す部分には次のメソッドを追

加します。

②に追加する追加するメソッド

```
private int getY() {  
    return random.nextInt(this.bottom);  
}
```

「random.nextInt()」とするたびに乱数を得ることができるので、引数に縦方向の最大値を与えることで、画面の縦方向の範囲での数を得ることができます。

③が示す部分は次のように修正します。

③を修正する最大値の設定

```
this.bottom -= height;  
y = getY();
```

④が示す部分に次のコードを追加します。

④に追加する縦方向の指定

```
y = getY();
```

ここで一度実行してみましょう。障害物が画面の左端で消えて再び右端から出て来るたびに、異なる場所から出て来ているはずです。



## 9-3-2 障害物を複数発生させる

障害物が1つだけ消えては出て来るというだけでは、まだまだゲームとしての面白みが足りません。画面上に複数の障害物が発生するようにしましょう。

### GameView.javaの修正

これまでGameView.javaの中で1つのEnemyオブジェクトを生成しているだけでしたが、この中で配列を定義して複数のEnemyオブジェクトを生成、管理できるようにします。

図2のようにGameView.javaを開きます。

```

public class GameView extends SurfaceView implements Callback {
    class GameThread extends Thread {
        SurfaceHolder surfaceHolder;
        boolean shouldContinue = true;

        int width;
        int height;

        Droid droid;
        static final int droidSize = 200;
        Enemy enemy;
        static final int enemySize = 200;

        public GameThread(SurfaceHolder surfaceHolder, Context context,
                          Handler handler) {
            this.surfaceHolder = surfaceHolder;
            droid = new Droid(context, droidSize, droidSize);
            droid.setInitialPosition(100, 0);

            enemy = new Enemy(context, enemySize, enemySize);
        }

        public void setViewSize(int width, int height) {
            this.width = width;
            this.height = height;

            droid.setMovingBoundary(0, 0, width, height);
            enemy.setMovingBoundary(0, 0, width, height);
        }

        public void upliftDroid(boolean on) {
            droid.uplift(on);
        }
    }
}

```

図2:GameView.java 1

図2の①が示す行は削除して、代わりに次の定義を追加します。

①への定義の追加

```

long frameNo = 0;
long nextGenFrame = 100;

Context context;

static final int EnemyNum = 5;
Enemy[] enemys;

```

②が示す部分は次のように修正します。ここでは1つだけEnemyオブジェクトを生成して、配列の0番目に入れておきます。

②を修正する内容

```

for (int i = 0; i < EnemyNum; i++) {
    if (enemys[i] != null) {
        enemys[i].setMovingBoundary(0, 0, width, height);
    }
}

```

次に、図3のように「GameView.java」を開いたままで少し下にスクロールします。

```

    }
    public void upliftDroid(boolean on) {
        droid.uplift(on);
    }
    @Override
    public void run() {
        while(shouldContinue) {
            Canvas c = surfaceHolder.lockCanvas();
            if(c != null) {
                draw(c);
                surfaceHolder.unlockCanvasAndPost(c);
            }
        }
    }
    public void draw(Canvas c) {
        if (enemy.isHit(droid)) {
            droid.setImageResource(R.drawable.andou_die01);
        }
        c.drawRGB(255, 0, 0, 0);
        droid.draw(c);
        enemy.draw(c);
    }
}

```

図3:GameView.java 2

図3の①が示すコードは、当たり判定をしている部分です。次のように配列内の要素をループして、すべての障害物の判定をするよう修正します。

#### ①を修正する障害物の判定

```

for (int i = 0; i < EnemyNum; i++) {
    if (enemys[i] != null && enemys[i].isHit(droid)) {
        droid.setImageResource(R.drawable.andou_die01);
    }
}

```

②が示す描画をしている部分も同様にして、配列内のすべての要素をループして描画するよう修正します。

#### ②を修正して描画を繰り返す

```

for (int i = 0; i < EnemyNum; i++) {
    if (enemys[i] != null) {
        enemys[i].draw(c);
    }
}

if (frameNo == nextGenFrame) {
    for (int i = 0; i < EnemyNum; i++) {
        if (enemys[i] == null) {
            enemys[i] = new Enemy(context, enemySize, enemySize);
            enemys[i].setMovingBoundary(0, 0, width, height);
            nextGenFrame += 100;
            break;
        }
    }
}
frameNo++;

```

このコード中の「if (frameNo == nextGenFrame)」の行から始まる部分は、時間差でEnemyオブジェクトを出現させるためのものです。

ここで一度実行してみましょう。図4のように複数の障害物が発生するようになっています。

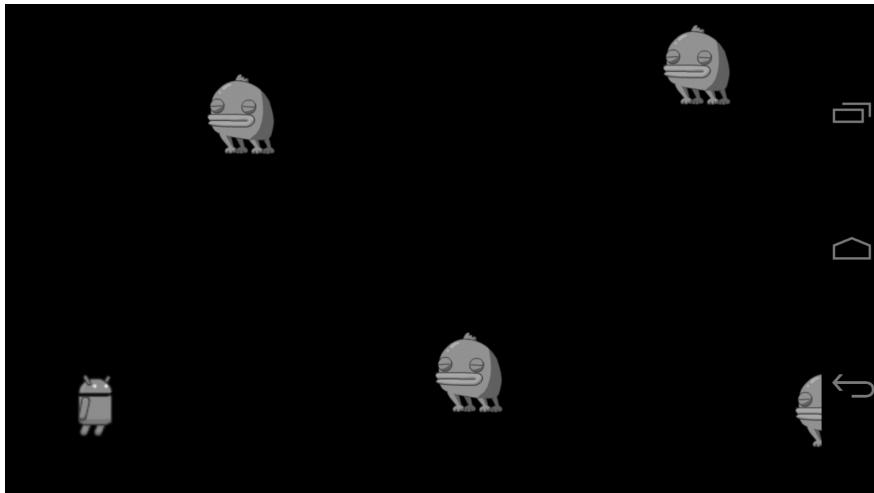


図4:複数の障害物の発生結果

ここで「Droid.java」に少し修正を加えます。障害物に当たった後はスプライト画像を切り替えるようにしていましたが、複数の障害物が飛んで来る状況では一度切り替えた後に何かしらのタイミングで元に戻すようにします。タイミングとしては上昇と下降でスプライト画像を切り替える処理を入れて、そこで元の画像に戻るようにします。「Droid.java」を開きます(図5)。

図5:Droid.java

```

Package Explorer Droid.java GameView.java AbstractGameObject.java Enemy.java
▼ FlyingDroid
  ▼ src
    ▼ jp.techinstitute.flyingdroid
      ▶ AbstractGameObject.java
      ▶ Droid.java
      ▶ Enemy.java
      ▶ GameView.java
      ▶ MainActivity.java
    ▶ gen [Generated Java Files]
    ▶ Android Private Libraries
    ▶ Android 4.4.2
      ▶ assets
    ▶ bin
    ▶ libs
  ▶ res
    ▶ drawable-hdpi
      andou_diag01.png
      andou_diagramore01.png
      andou_die01.png
      effect_energyball04.png
      enemy_pinkdude_jump.png
      ic_launcher.png
    ▶ drawable-ldpi
    ▶ drawable-mdpi
    ▶ drawable-xhdpi
    ▶ drawable-xxhdpi
    ▶ layout
    ▶ values
    ▶ values-v11
    ▶ values-v14
  ▶ AndroidManifest.xml
  ▶ ic_launcher-web.png
  ▶ proguard-project.txt
  ▶ project.properties

public class Droid extends AbstractGameObject {
    private int defaultX;
    private int defaultY;
    private static final float DefaultVelocity = 2;
    private float velocity = DefaultVelocity;

    /**
     * @param context
     * @param resourceId
     * @param width
     * @param height
     */
    public Droid(Context context, int width, int height) {
        super(context, R.drawable.andou_diag01, width, height);
    }

    @Override
    public void setMovingBoundary(int left, int top, int right, int bottom) {
        // TODO Auto-generated method stub
        super.setMovingBoundary(left, top, right, bottom);
        this.bottom -= height;
    }

    public void uplift(boolean on) {
        if (on) {
            velocity = -DefaultVelocity;
        } else {
            velocity = DefaultVelocity;
        }
    }
}

```

図5の①と②で示している部分ではドロイド君の上昇、下降を切り替えるための速度を設定していることを覚えているでしょうか。ここでスプライト画像の切り替えをするように修正します。下記のコードをそれぞれに入力します。

①に追加する内容

```
setImageResourceId(R.drawable.andou_diagmore01);
```

②に追加する内容

```
setImageResourceId(R.drawable.andou_diag01);
```

ここで再度実行してみましょう。だいぶゲームらしくなってきたのではないかでしょうか。



### 9-3-3 効果音とBGMを付ける

ゲームを盛り上げる要素の1つとして「音」があります。このゲームにも効果音とBGMを追加してみましょう。

#### 音声ファイルの追加

プロジェクトに音声ファイルをリソースとして追加します。スプライトの画像ファイルを追加した際にダウンロードしたZIP形式のファイルの中に、「raw」というフォルダーがあるので、それを図のように「res」の下に追加します(図6)。

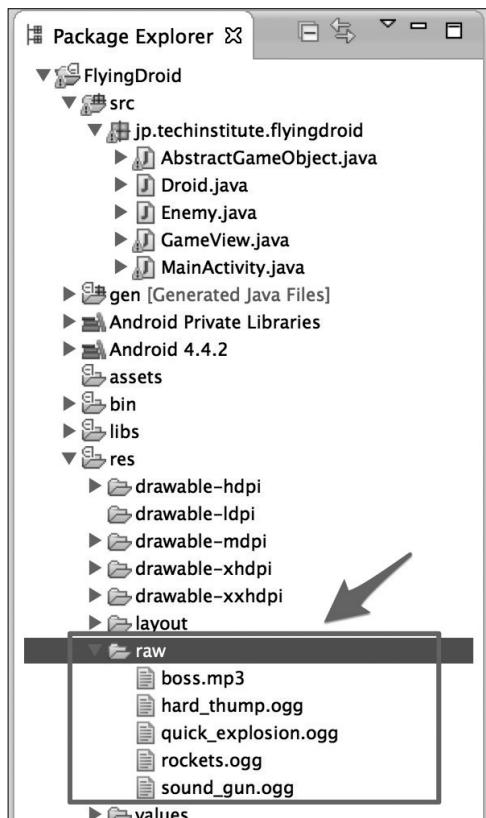


図6:音声ファイルの追加

## 効果音

障害物とドロイド君がぶつかった時の衝撃音、ドロイド君が上昇している時のロケットの噴射音を入れます。効果音のように短めの音を出す場合は「SoundPool」というクラスが適しています。図7のように「GameView.java」を開いて「SoundPool」を使って音を出す処理を追加していきます。

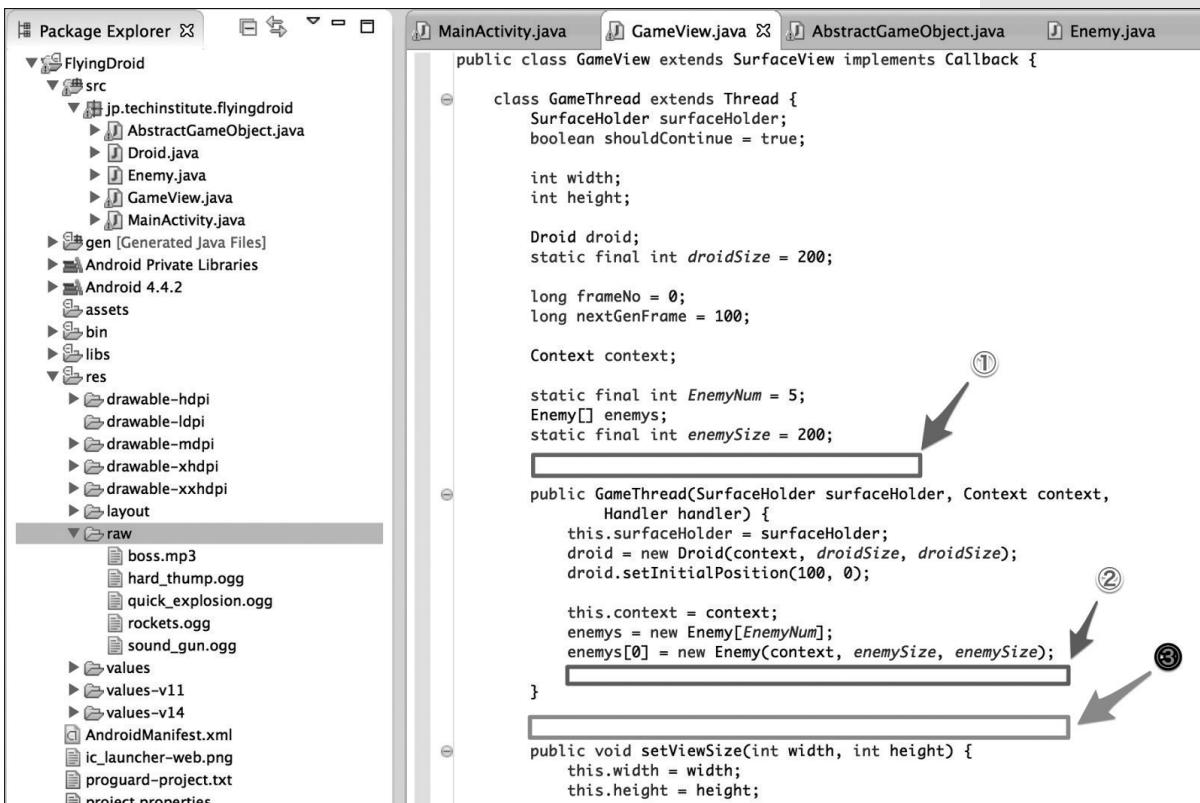


図7:GameView.java 1

図7の①が示す部分に「SoundPool」関連のメンバ変数を追加します。

①にメンバ変数を追加

```

SoundPool sound;
int hitSoundId;
int rocketSoundId;
int rocketStreamId;

```

上から順に「SoundPool」のオブジェクト、障害物と当たったときの音声用ID、上昇時のロケット音の音声用ID、上昇時のロケット音のストリームID(再生中の音を止めるのに必要)となっています。②が示す部分には初期化メソッドを呼び出す次のコードを追加します。

②に追加する内容

```
setupSoundPool();
```

③が示す部分には「SoundPool」関連の初期化メソッドとリリースメソッドを追加します。

③に初期化とリリースメソッドの追加

```
public void setupSoundPool() {  
    sound = new SoundPool(2, AudioManager.STREAM_MUSIC, 0);  
    hitSoundId = sound.load(context, R.raw.quick_explosion, 1);  
    rocketSoundId = sound.load(context, R.raw.rocket, 1);  
}  
  
public void releaseSoundPool() {  
    sound.release();  
}
```

初期化メソッドでは「SoundPool」オブジェクトを生成し、障害物と当たったときの音と上昇時のロケット音をロードしています。リリースメソッドでは「SoundPool」オブジェクトをリリースするメソッドを呼び出しています。つづいて、図8のように「Game View.java」を開いたまま少し下にスクロールします。

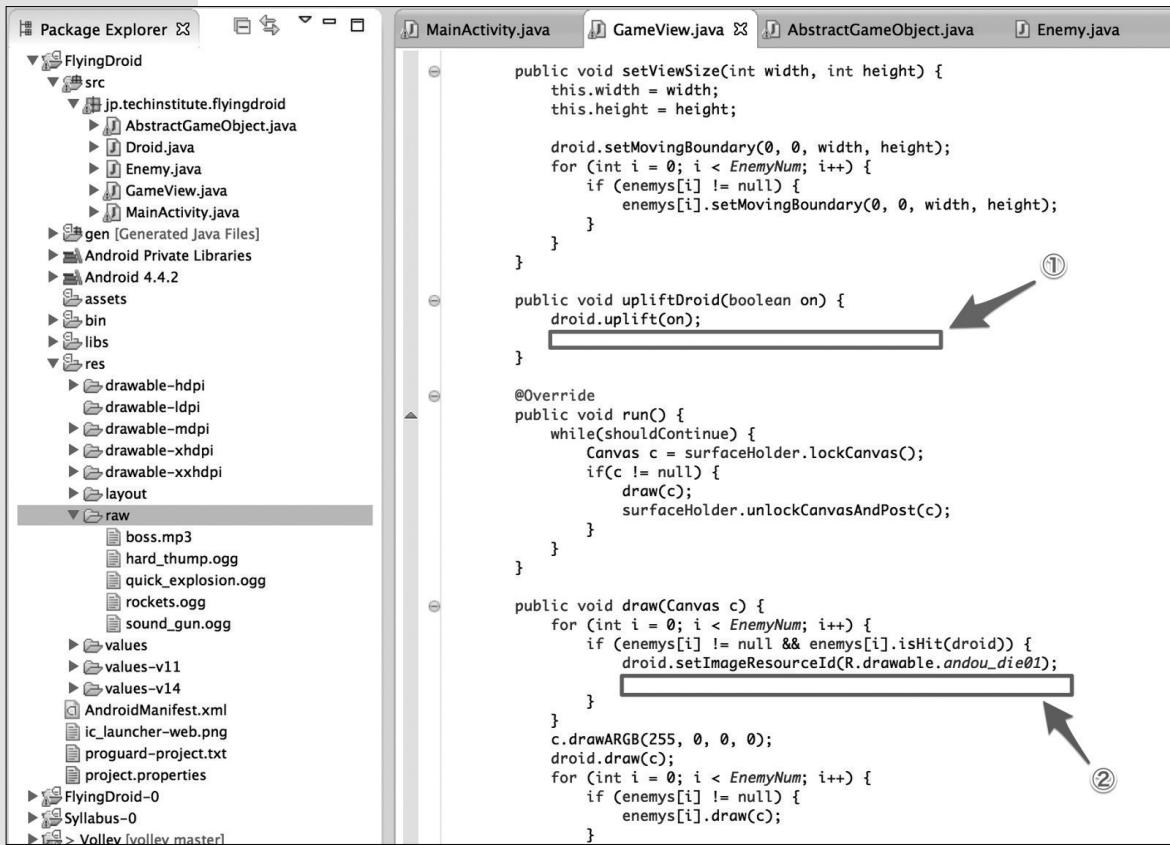


図8:GameView.java 2

図8の①が示す部分にロケット上昇時の音の再生を行うコードを追加します。

①で上昇時の効果音を再生

```
if (on) {  
    rocketStreamId = sound.play(rocketSoundId, 0.5f, 0.5f, 0, -1, 1.0f);  
} else {  
    sound.stop(rocketStreamId);  
}
```

「on」が「true」ならば上昇するとして再生を開始し(この時に返されるストリームIDを保持しておく)、それ以外ならば下降するとして保持しておいたストリームIDで再生を止めます。②が示す部分には、障害物と当たった時の音の再生を行うコードを追加します。

②で当たり判定時に効果音を再生

```
sound.play(hitSoundId, 1.0f, 1.0f, 0, 0, 1.0f);
```

## 当たり判定処理の修正

これまでの処理方法では、当たり判定でtrue(当たっている)という結果が出た後も、毎フレームで当たっているという判定が繰り返されていました。この状況のまま効果音を出すと、毎フレーム音を鳴らすよう処理が行われてしまいます。そこで、一度当たっているという判定が出た障害物とは次のフレーム以降は判定が出ないよう修正します。図9のようにAbstractGameObject.javaを開きます。

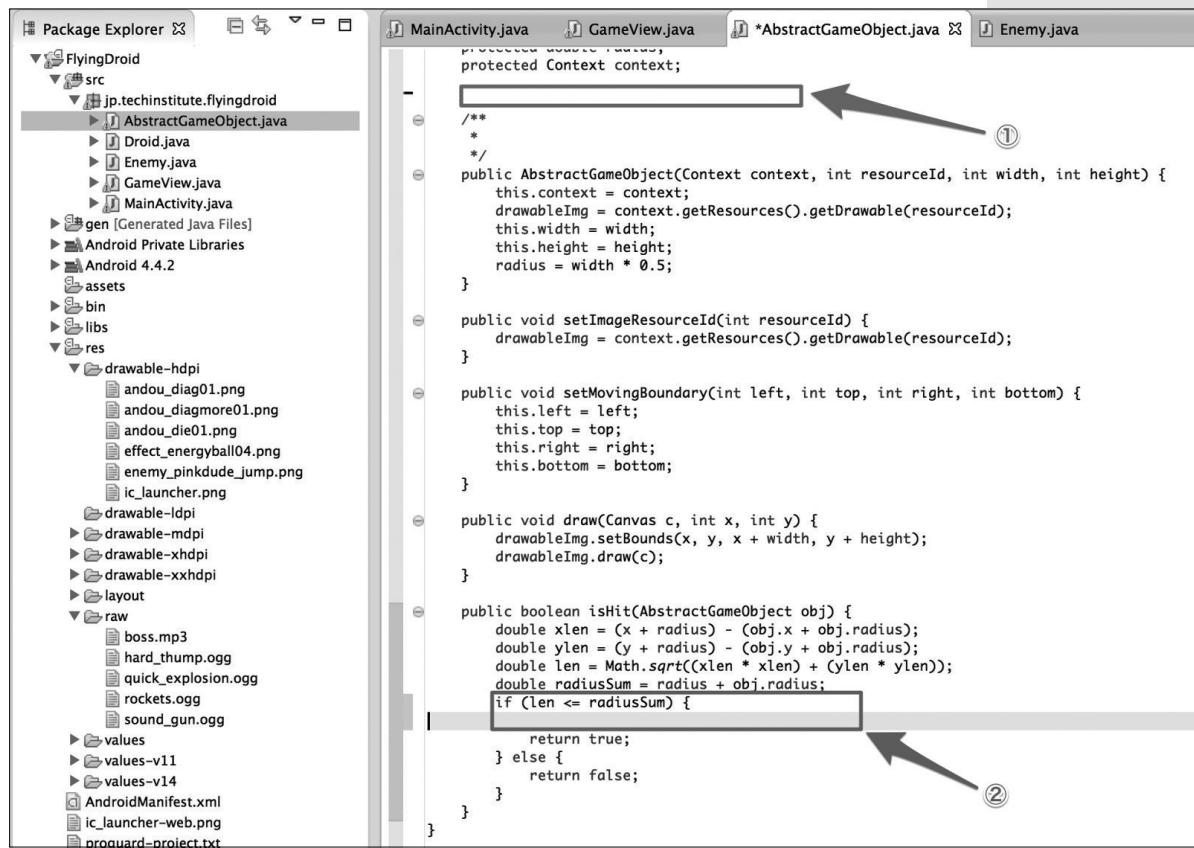


図9:AbstractGameObject.java

図9の①が示す部分に「当たり判定済み」フラグのメンバ変数を追加します。

①にメンバ変数の追加

```
protected boolean alreadyHit = false;
```

②が示す部分を次のように修正します。

②の修正内容

```
if (len <= radiusSum && !alreadyHit) {  
    alreadyHit = true;
```

当たっているかどうかを判定する条件式に、「当たり判定済み」フラグがOFFならばという条件を加えます。2つの条件が両方満たされる場合は当たっているとして、「当たり判定済み」フラグをONにします。

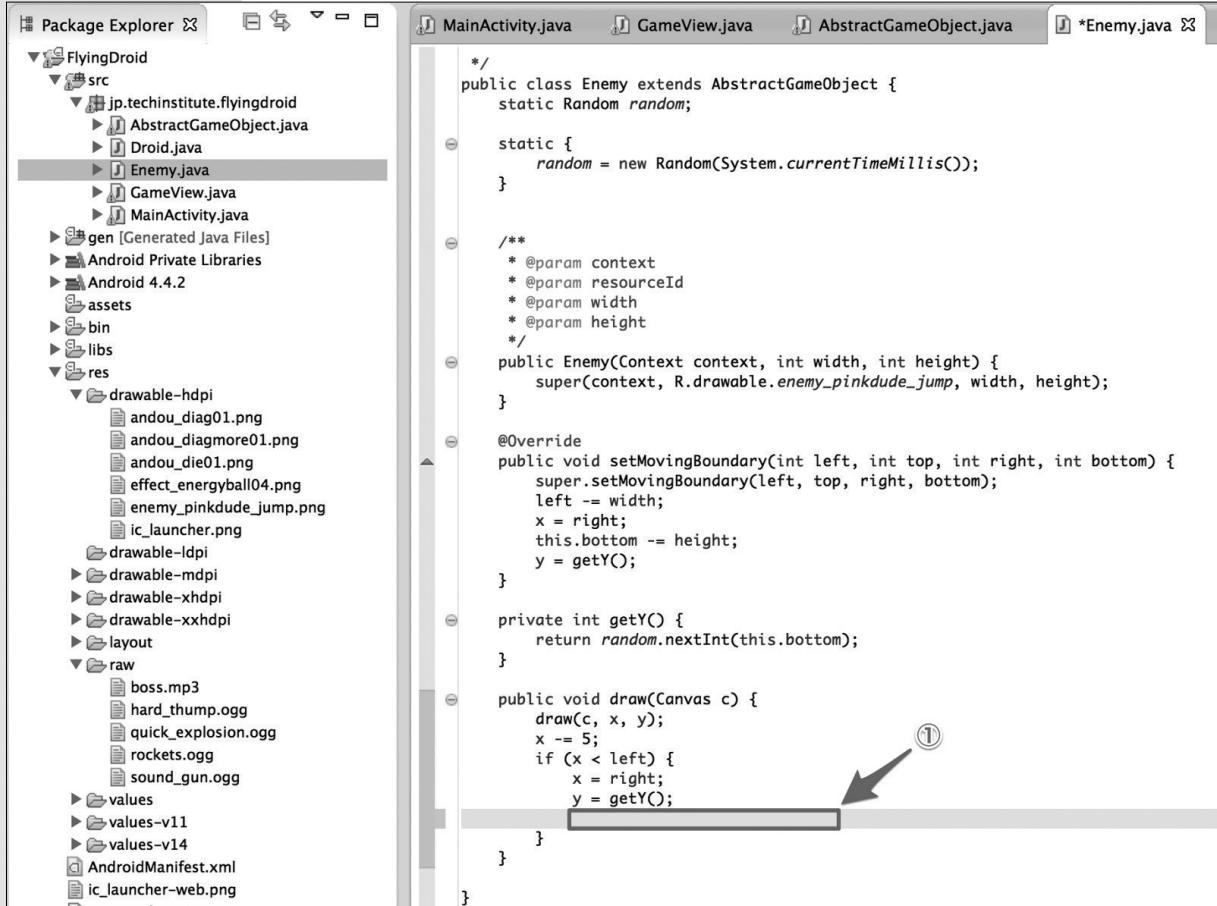


図10:Enemy.java

図10の①が示す部分に次のコードを追加します。これで画面の右端から再度画面に出るときは「当たり判定済み」フラグがOFFになります。

当たり判定済みのフラグ

```
alreadyHit = false;
```

## BGM

効果音だけでは淋しいので、プレイ中に流れるBGMを追加します。図11のように「MainActivity.java」を開きます。

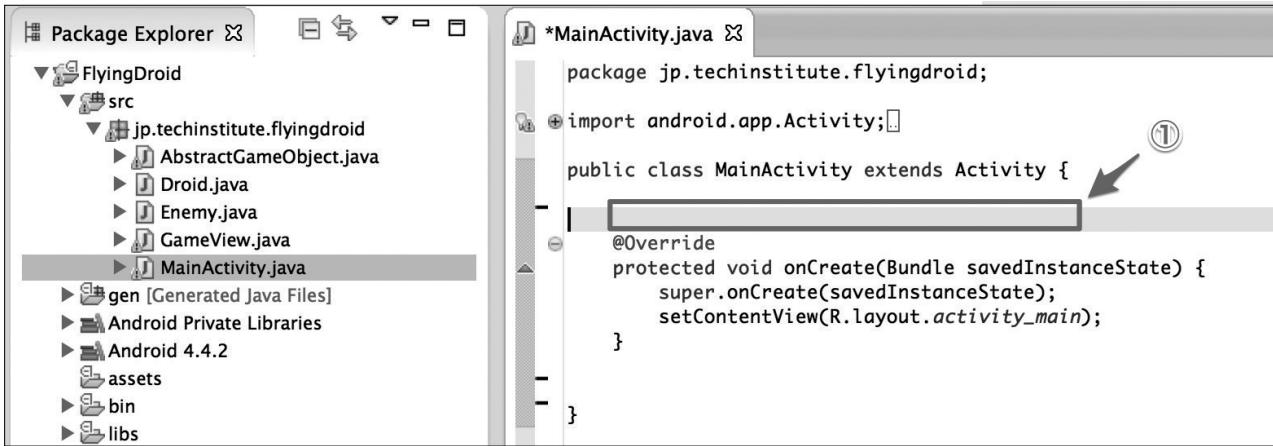


図11:MainActivity.java

図11の①が示す部分に次の定義を追加します。

①に追加する定義内容

```
MediaPlayer mediaPlayer;
```

BGMのように長い音声ファイルの再生には「MediaPlayer」を使います。続いて「MainActivity」で必要なメソッドをオーバーライドします。図12のようにして「MainActivity」を開いている状態でメニューバーから「Source」→「Override/Implement Methods」を選択します。

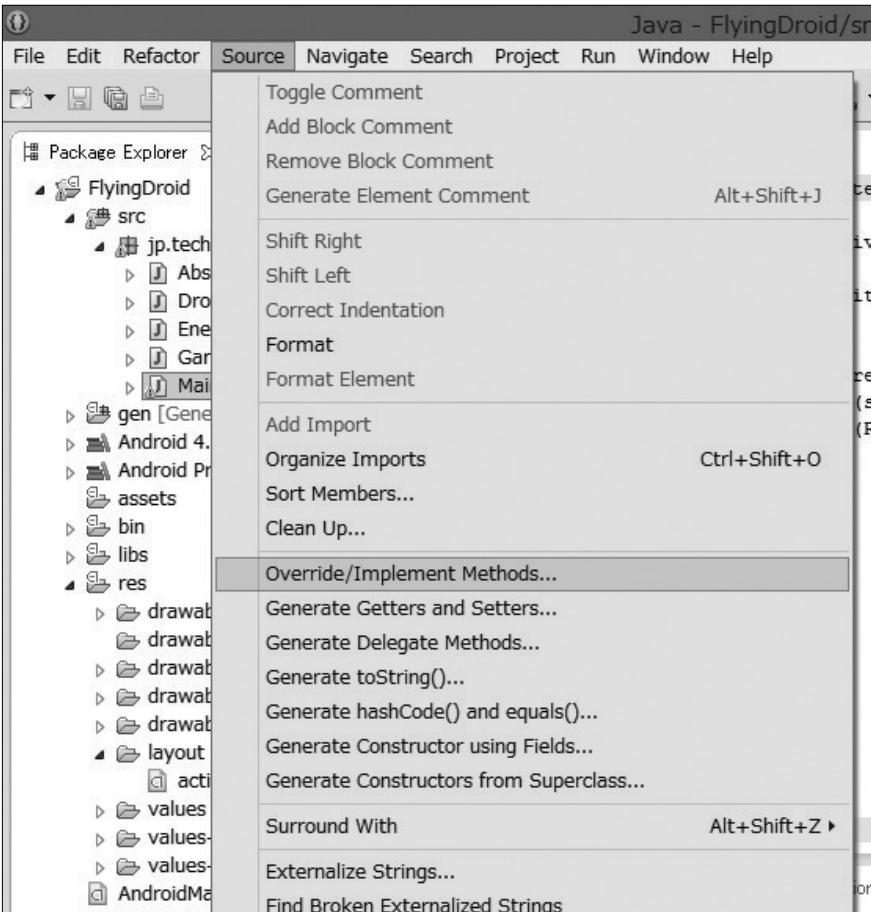


図12:EclipseのSourceメニュー

「Override/Implement Methods」ダイアログが開くので、オーバーライドするメソッドの一覧から「onPause」「onResume」「onStart」「onStop」の4つにチェックを入れて「OK」を押します(図13)。

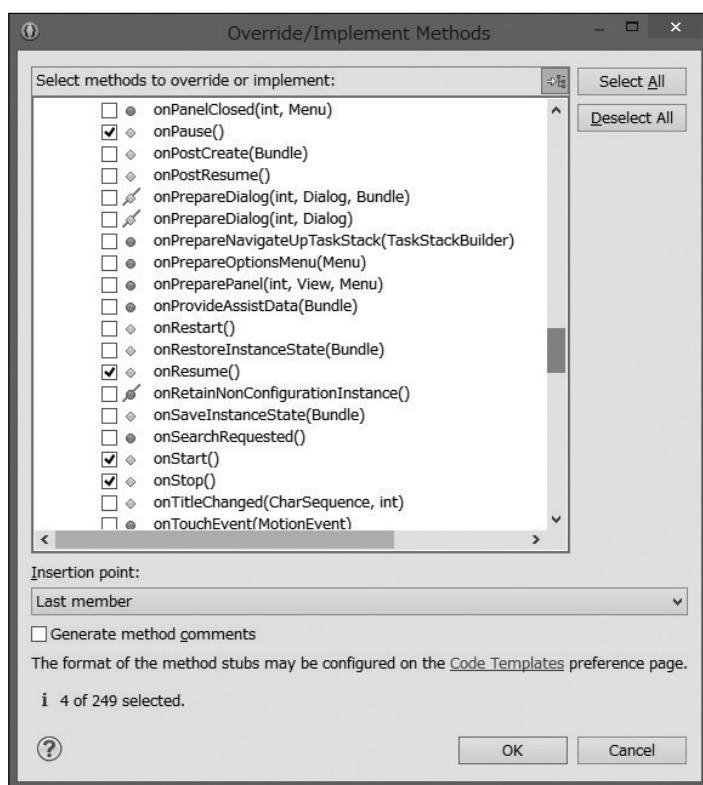


図13:Override/Implement Methodsダイアログ

すると次の図14のようにコードが自動補完で入力されます。

The screenshot shows the code editor for MainActivity.java. It contains the following code with annotations:

```

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);

@Override
protected void onPause() {
    // TODO Auto-generated method stub
    super.onPause();
} ①

@Override
protected void onResume() {
    // TODO Auto-generated method stub
    super.onResume();
} ②

@Override
protected void onStart() {
    // TODO Auto-generated method stub
    super.onStart();
} ③

@Override
protected void onStop() {
    // TODO Auto-generated method stub
    super.onStop();
} ④
}

```

Annotations ① through ④ are circled numbers pointing to the four generated method stubs.

図14:MainActivity.java - 補完後のコード

①～④には、以下のコードを追加します。

①に追加する内容

```
super.onPause();  
mediaPlayer.pause();
```

②に追加する内容

```
mediaPlayer.start();  
mediaPlayer.setLooping(true);
```

③に追加する内容

```
mediaPlayer = MediaPlayer.create(this, R.raw.boss);
```

④に追加する内容

```
mediaPlayer.stop();  
mediaPlayer.release();
```

ここで実行してみましょう。アプリが動作を開始するとBGMが流れ始め、動作に合わせた効果音が鳴ることを確認してみて下さい。



## 9-3-4 Hit Point 管理

何らかの制限がある方がゲーム性が高まります。自分のキャラであるドロイド君にHit Pointの概念を取り入れ、障害物に当たるたびにそれが減っていくようにして、0になると操作不能になるようにしてみましょう。

### Droid.javaの修正

図15のように「Droid.java」を開きます。

```
public class Droid extends AbstractGame0bject {
    private int defaultX;
    private int defaultY;
    private static final float DefaultVelocity = 2;
    private float velocity = DefaultVelocity;

    /**
     * @param context
     * @param resourceId
     * @param width
     * @param height
     */
    public Droid(Context context, int width, int height) {
        super(context, R.drawable.andou_diag01, width, height);
    }

    @Override
    public void setMovingBoundary(int left, int top, int right, int bottom) {
        // TODO Auto-generated method stub
        super.setMovingBoundary(left, top, right, bottom);
        this.bottom -= height;
    }

    public void uplift(boolean on) {
        if (on) {
            velocity = -DefaultVelocity;
            setImageResourceId(R.drawable.andou_diagmore01);
        } else {
            velocity = DefaultVelocity;
            setImageResourceId(R.drawable.andou_diag01);
        }
    }
}
```

図15:Droid.java

図15の①が示す部分には Hit Pointを保持するためのメンバ変数を追加します。また、Hit Pointの満タン時の値として100という定数も定義しておきます。

①で HitPoint の満タン時の値を指定

```
private static final int FullHitPoint = 100;
private int hitPoint = FullHitPoint;
```

②が示す部分にはHit Point関連のメソッドを追加します。

## ②でメソッドを追加

```

public void hit() {
    if (hitPoint > 0) {
        hitPoint -= 10;
    }
    if (hitPoint <= 0) {
        velocity = DefaultVelocity;
        setImageResourceId(R.drawable.andou_explode10);
    }
}

public int getHitPoint() {
    return hitPoint;
}

public void resume() {
    hitPoint = FullHitPoint;
    setImageResourceId(R.drawable.andou_diag01);
}

```

上から順に説明すると、「hit()」は障害物と当たったと判定された際に呼ばれるメソッドで、Hit Pointの値を10ずつ減らし、もし0以下になった場合は速度を下降用の値にしてスプライト画像を焦げた色のものに変更します。「getHitPoint()」はこのクラスで持っているHit Pointの値を返すメソッドです。「resume()」は一度Hit Pointが0になって操作不能になった後に、再びプレイ開始する際に呼び出すメソッドです。Hit Pointを満タンにしてスプライト画像を元に戻しています。

③が示す部分は、次のようにしてHit Pointが0より大きい場合はという条件で囲んで、Hit Pointが0になった場合は操作ができないようにします。

## ③でゲームオーバー判定

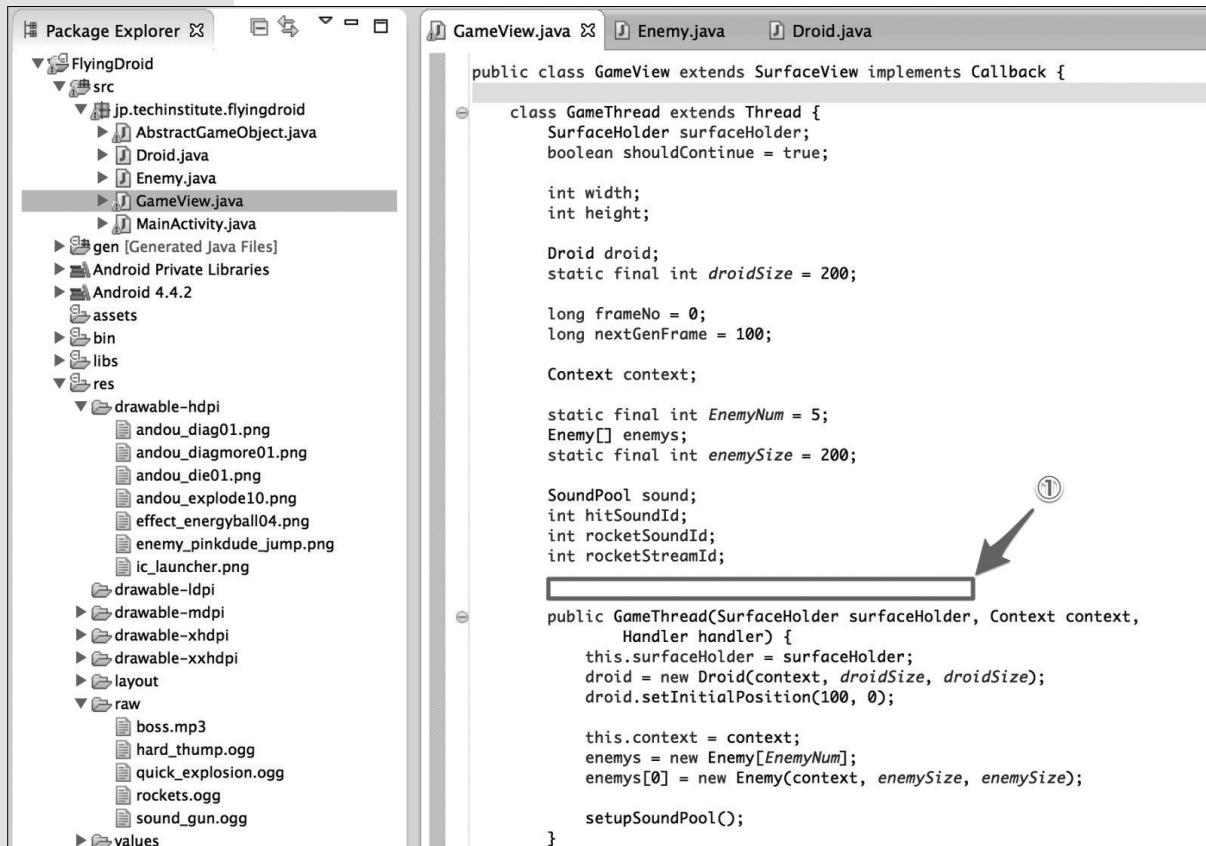
```

if (hitPoint > 0) {
    if (on) {
        velocity = -DefaultVelocity;
        setImageResourceId(R.drawable.andou_diagmore01);
    } else {
        velocity = DefaultVelocity;
        setImageResourceId(R.drawable.andou_diag01);
    }
}

```

## GameView.javaの修正

図16のように「GameView.java」を開きます。



```
public class GameView extends SurfaceView implements Callback {
    class GameThread extends Thread {
        SurfaceHolder surfaceHolder;
        boolean shouldContinue = true;

        int width;
        int height;

        Droid droid;
        static final int droidSize = 200;

        long frameNo = 0;
        long nextGenFrame = 100;

        Context context;

        static final int EnemyNum = 5;
        Enemy[] enemys;
        static final int enemySize = 200;

        SoundPool sound;
        int hitSoundId;
        int rocketSoundId;
        int rocketStreamId;
    }

    public GameThread(SurfaceHolder surfaceHolder, Context context,
                      Handler handler) {
        this.surfaceHolder = surfaceHolder;
        droid = new Droid(context, droidSize, droidSize);
        droid.setInitialPosition(100, 0);

        this.context = context;
        enemys = new Enemy[EnemyNum];
        enemys[0] = new Enemy(context, enemySize, enemySize);

        setupSoundPool();
    }
}
```

図16:GameView.java 1

図16の①が示す部分には、操作不能の後に復活する際のタイミングのフレーム番号を保持するメンバ変数を追加します。

①にメンバ変数の追加

```
long resumeFrame = -1;
```

次に、図17のようにGameView.javaを開いたままで少し下にスクロールします。

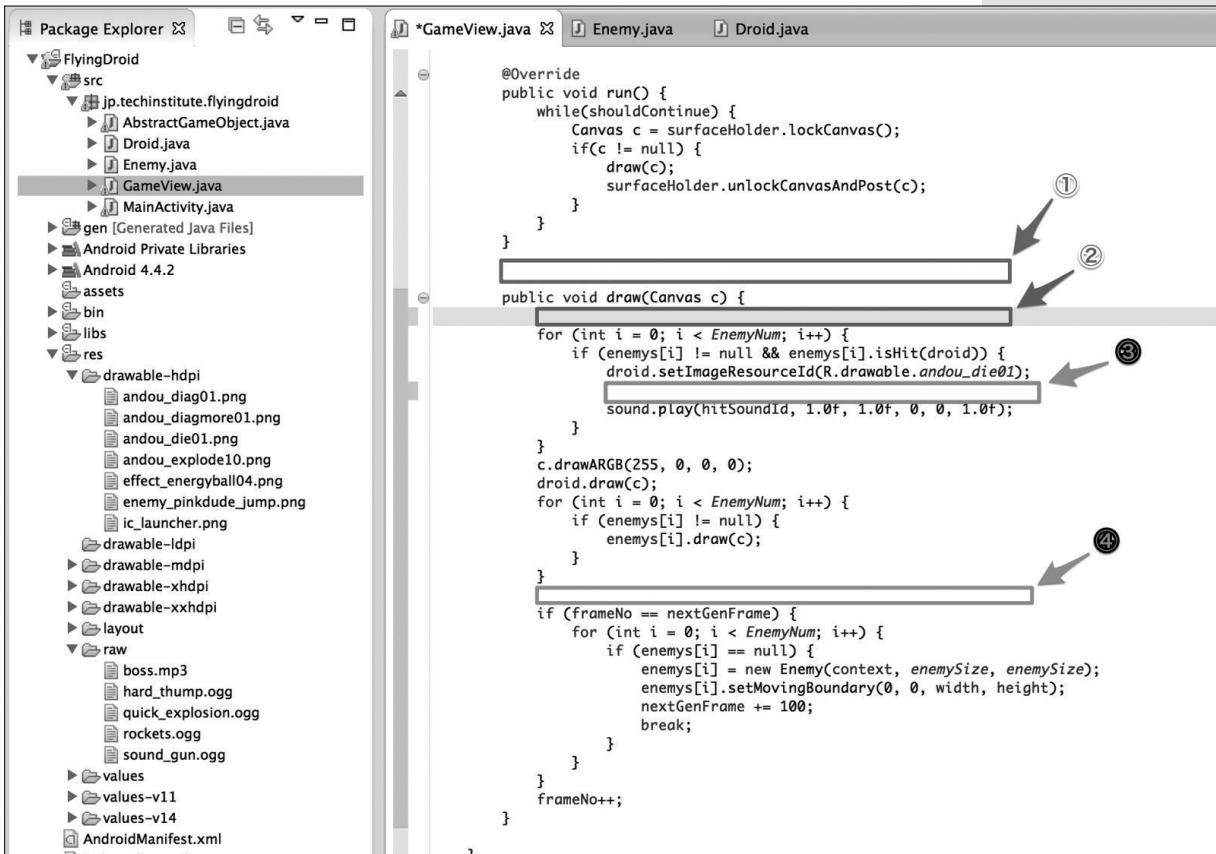


図17:GameView.java 2

図17の①が示す部分には、Hit Pointの現在値を描画するメソッドを追加します。

#### ①に追加するメソッド

```

private void drawHitPoint(Canvas c, int point) {
    Paint paint = new Paint();
    paint.setColor(Color.WHITE);
    paint.setTextSize(80);
    c.drawText("HP: " + point, 20, 100, paint);
}

```

文字の色を「setColor」で、大きさを「setTextSize」で指定して、「HP:」という文字列と引数で渡されたHit Pointの値を結合して描画します。②が示す部分には現在のHit Pointの値を取得するコードを追加します。

#### ②で Hit Point の値を取得

```
int hitPoint = droid.getHitPoint();
```

③が示す部分には障害物と当たったと判定された場合の処理を追加します。

③に追加する当たり判定後の処理

```
droid.hit();
hitPoint = droid.getHitPoint();
if (hitPoint == 0) {
    resumeFrame = frameNo + 300;
}
```

droidオブジェクトの「hit()」メソッドを呼び出すことでHit Pointを減算し、現在のHit Point値を取得して、もし0ならば復活時のフレーム番号を現在のフレーム番号から300フレーム後に設定します。④が示す部分には次のコードを追加します。

④でフレーム番号の指定

```
drawHitPoint(c, hitPoint);
if (frameNo == resumeFrame) {
    droid.resume();
}
```

Hit Pointを描画するメソッドを呼び出し、現在のフレーム番号が復活時のフレーム番号と一致する場合は、復活処理を行うメソッドを呼び出します。ここまでできたら完成です。実行してみましょう。図18のように、Hit Pointの値が描画され、障害物と当たるたびに値が減っていくのが見えるはずです。

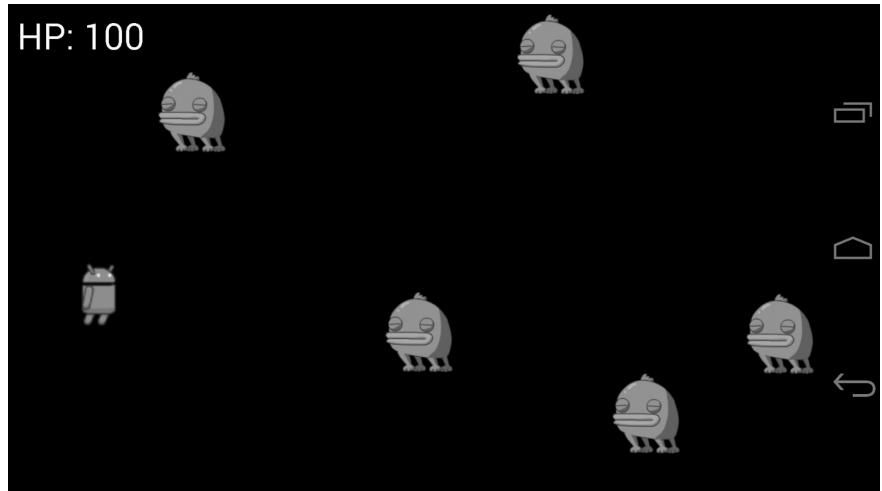


図18:Hit Pointの描画



## 演習問題

障害物を撃つ処理を追加してみましょう。

Beamクラスを追加して、Enemyと逆に左から右へ移動するようにして当たり判定を行えます。

