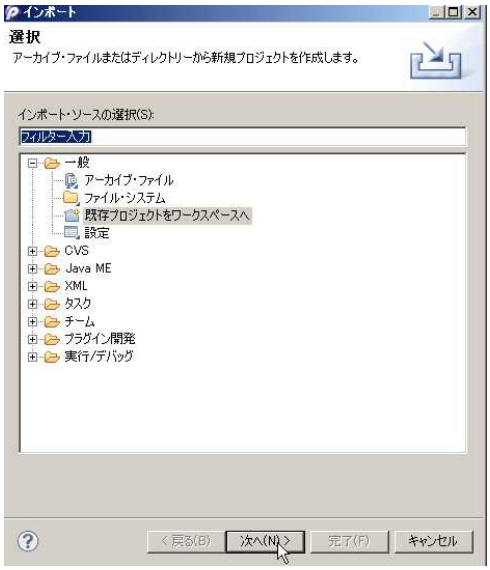
Android プログラミング【STEP】

1　実習方法　　Step の標準授業時数　２０時間（10 月中旬まで）

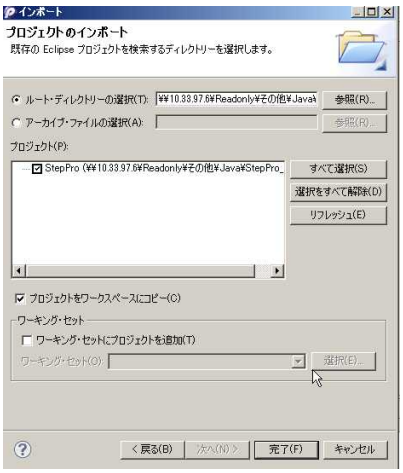
2　実習用サーバーの「R:¥その他¥Java¥StepPro_init」フォルダを「z:¥workspace」へインポートしてください。

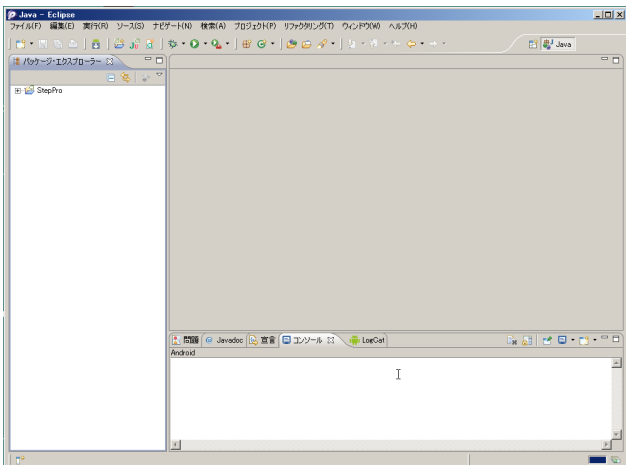
（1）Eclipse を起動して「ファイル」－「インポート」を選択してください。

（2）「既存プロジェクトをワークスペースへ」を選択してください。



（3）「ルートディレクトリの選択」の「参照」ボタンをクリックし、「R:¥その他¥Java¥StepPro_init」フォルダを選択してください。その際、「プロジェクトをワークスペースにコピー」をチェックしてください。





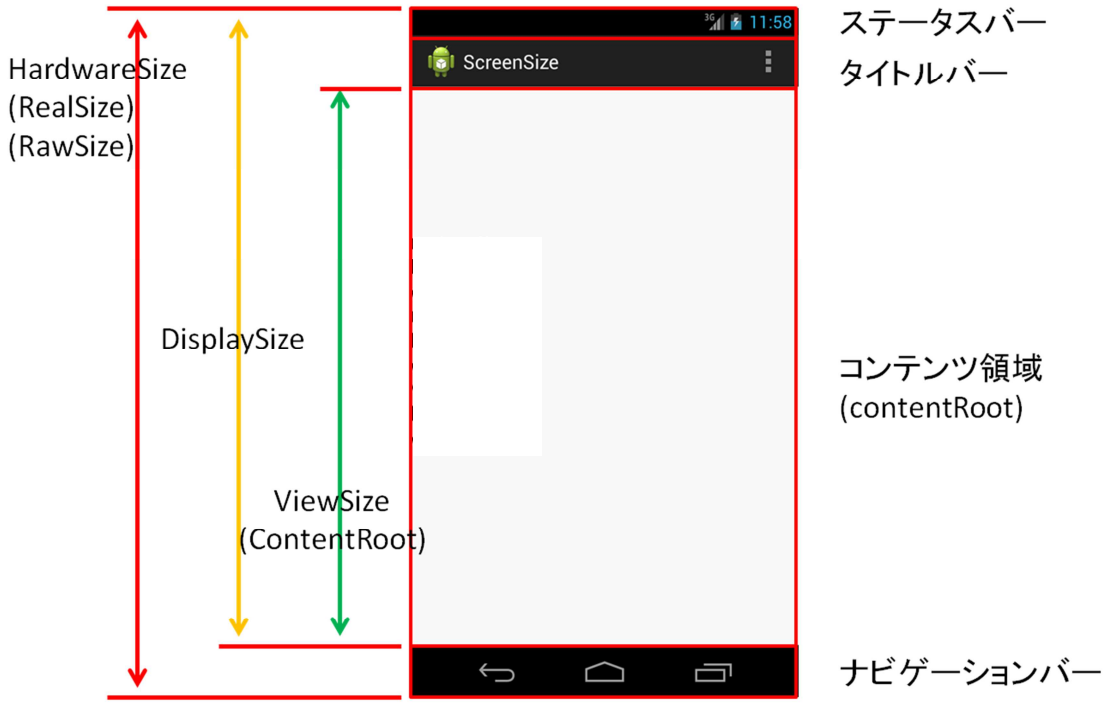
3　次のアプリケーションを新規作成し、①の箇所へ、それぞれの実習課題で指定されたソースを入力して、コンパイルし、実行結果を確認してください。

プロジェクト名：StepPro????(年組席)　　アプリケーション名：Step

```
/*    年    組    席   名前
 * Step
 */
package jp.edu.mie;
import android.app.Activity;
import android.os.Bundle;
import android.view.Window;

public class Step extends Activity
{
    public void onCreate(Bundle bundle)
    {
        super.onCreate(bundle);
        requestWindowFeature(Window.FEATURE_NO_TITLE);//タイトルの非表示
        setContentView(new      ①      (this));//実画面に表示するビューの指定
    }
}
```

4　画面サイズ



4.1　Android の画面を構成するパーツの名前

コンポーネント名	説明
ステータスバー	通知バーとも呼ばれます。端末によってサイズが変わります。アプリケーションから表示/非表示を指定できます。
タイトルバー	アプリケーション名を表示します。横幅や高さは適用する Style ファイル(アプリのデザイン)に依存します(標準であれば 48dp 最近の xhdpi 端末では 96px)。
コンテンツ領域	contentRoot。普段アプリを作る際にレイアウトファイルを(fill_parent などを指定して)配置する領域です。
ナビゲーションバー	ソフトウェアキーの領域です。端末によって(または縦/横表示によって)サイズが変わります。

4.2　取得できる画面サイズ

種類	説明
HardwareSize	液晶パネルそのもののサイズを取得できます。ただし、SDK が対応したのは Android 4.2 以降からです。
DisplaySize	アプリケーションが利用できる表示領域です
ViewSize	コンテンツを表示できるサイズです

4.3　今回の領域

requestWindowFeature(Window.FEATURE_NO_TITLE);でタイトルを非表示にしていますので、ディスプレイサイズ＝ステータスバー＋コンテンツ領域＋ナビゲーションバー、という状態にして解説します。

1　文字と画像の表示

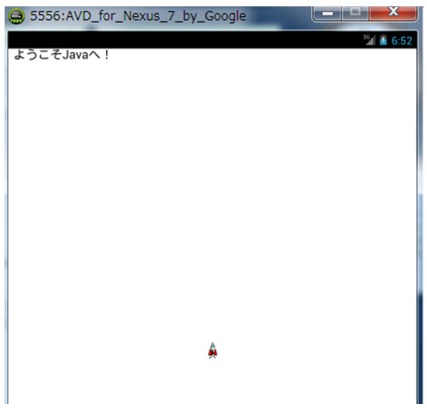
「ようこそ Java へ」という文字とロケットの画像を表示します。

1.1　ソースコード

(1)テンプレート　Step の①の箇所に　Step010View　を入力してください。

(2)次のアプリケーションを新規作成してください。

プロジェクト名：StepPro????(年組席)　　アプリケーション名：Step010View




```
/*    年    組    席   名前
* Step010View
*     文字と画像の表示
*/
package jp.edu.mie;
import android.content.Context;
import android.content.res.Resources;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class Step010View extends SurfaceView implements SurfaceHolder.Callback
{
    private SurfaceHolder holder;//サーフェイスホルダー
    private Bitmap image;
    private Canvas canvas;
    private Paint paint;
    pprivate final int TSIZE=24;
    private final String TEXT1="ようこそ Java へ！";
    //コンストラクタ
    public Step010View(Context context)
    {
        super(context);//context:アプリケーション環境の情報を保持する
        //画像の読み込み
        Resources r=context.getResources();//リソースオブジェクトの取得
        image=BitmapFactory.decodeResource(r, R.drawable.rocket);//読み込み
        //サーフェイスホルダの作成
        holder=getHolder();//サーフェイスフォルダの取得
        holder.addCallback(this);//サーフェイスフォルダの通知先の指定
        holder.setFixedSize(getWidth(), getHeight());//サイズの指定
    }
    //サーフェイスの生成
    public void surfaceCreated(SurfaceHolder holder)
    {
        //Canvas オブジェクトをロックして取得
        canvas=holder.lockCanvas();
        //描画の設定
        paint=new Paint();
```

```
        paint.setAntiAlias(true);//文字やラインを滑らかに見せる。
        paint.setTextSize(TSIZE);//文字サイズ
        //バッファの描画
        canvas.drawColor(Color.WHITE);//背景を塗りつぶす
        canvas.drawText(TEXT1, 10, 20, paint);//文字列の表示
        canvas.drawText("画面サイズ：縦(y)="+getHeight()+"　横(x)="+getWidth(),
                                                        250, 20, paint);
        canvas.drawBitmap(image, getWidth()/2-16, getHeight()/2-16, paint);
        //Canvas のロックを解除する。実画面に反映
        holder.unlockCanvasAndPost(canvas);
    }
    //サーフェイスの変更
    public void surfaceChanged(SurfaceHolder holder, int format, int w, int h){}
    //サーフェイスの破棄
    public void surfaceDestroyed(SurfaceHolder holder){}
}
```

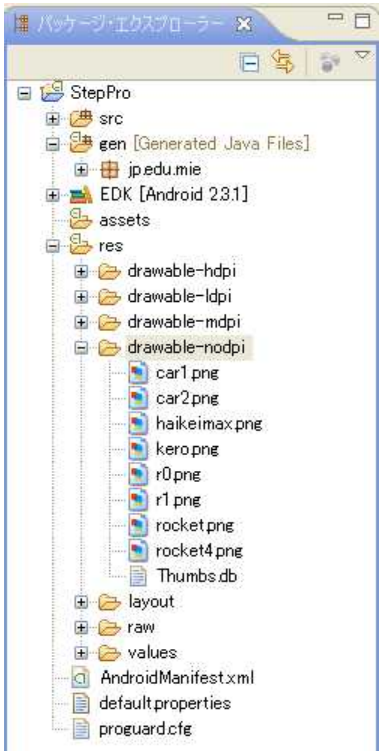
1.2　実行方法と実行結果

1.2.1　準備

「res」フォルダに画像ファイルを置きます。

1.2.2　実行方法

Step.java をクリックして、メニューから「実行」又は実行アイコンをクリックしてください。



1.3　SurfaceViewクラスとCanvasクラス

この画面サイズは、800×1183（ViewSize）です。

Android のアプリケーション上で画像を表示するには、まず Activity（画面）を生成して、そこに描画する領域を指定します。この描画指定には、Canvas と surfaceView の２つの方法があります。

Android のプログラムというのは、ユーザーインターフェイスを扱うための UI スレッドと呼ばれるスレッドから処理が呼び出され実行されるようになっています。スレッドとは、１つのアプリケーション(プロセス)内でメモリを共有する小さな実行単位のことで、最低１つのスレッドが実行されています。

UI スレッドは、文字通りユーザーインターフェイス関連のすべてのイベント等が実行されるもので、この単一のスレッド内からすべてのコンポーネント類のイベント処理などが呼び出されているわけです。

Android アプリの UI はシングル・スレッド モデルです。View による表示の更新（onDraw メソッド）なども、やはりこの UI スレッドからイベントの処理が呼び出されています。つまり、すべてのユーザーインターフェイス関係は、１つのスレッド内で順番待ちして自分が呼び出されるのを待っているのです。描画の更新だけが、順番を飛ばして優先的に処理されるわけではありません。

この「全部が１つのスレッドで順番待ちしながら処理していく」というやり方は、速度を要求されない場合はシンプルでいいのですが、高速描画を必要とするリアルタイムゲームなどでは足かせとなってしまいます。

こうしたときに用いられるのが、SurfaceView と呼ばれるコンポーネントです。これは View のサブクラスです。

SurfaceView では、UI スレッドとは無関係に描画の更新を行うことができます。

1.3.1　Canvas とは

表示する画面を生成するためのシステムです。表示以外の処理とともに動作するため、描画が遅くなりがちです。つまり、画像の処理に 100 ミリ秒かかったら、画面もそのタイミングで 100 ミリ秒止まってしまいます。 具体的に描画するには「onDraw メソッドをオーバーライドしておけば必要に応じて描画される」という処理をしています。

1.3.2　SurfaceView とは

サーフェイス(Surface：表面)とは、UI ビューから独立して描画を行うことで高速な連続描画をすることです。

SurfaceView クラスは、基本的にイベントが発生したときしか動かないイベント駆動型となっているので、そのままではリアルタイムに動作させることができません。

そこで、スレッドを利用します。複数のスレッドを実行させることをマルチスレッドといいます。スレッドを作成すれば、SurfaceView のイベント処理にとらわれることなく、並列してゲーム固有の処理をリアルタイムに実行させ続けることができます。

SurfaceView はマルチスレッドで処理するための専用クラスをもっており、それを使って UI スレッドとは別に独自にスレッドを使った処理を行わせることができます。

非常に便利ですが、その代りに使い方がちょっと面倒になっています。まず描画ですが、これは View のように「onDraw メソッドをオーバーライドしておけば必要に応じて描画される」というような形にはなっていません。必要に応じて描画の更新を自分で行わなければいけません。

マルチスレッド処理も、通常の Thread クラスなどではなく、専用のクラスを使うことになります。

【クラスの継承(extends)】
新しく拡張したクラスが既存のクラスのメンバを受け継ぐこと。もとになる既存のクラスをスーパークラス、新しいクラスをサブクラスと呼ぶ。
例　「車」クラス・・・・・・・・・スーパークラス
　　「レーシングカー」クラス・・・サブクラス

【オーバーライド(overriding)】
スーパークラスとまったく同じメソッド名・引数の数・型をもつメソッドを定義することができる。サブクラスのメソッドが、スーパークラスのメソッドに代わって機能すること。オーバーライドの利点としては、１つのメソッド名を使うことによって、そのオブジェクトのクラスに応じた適切な処理を行うことができる。

1.4　SurfaceViewの実装１

```
//①インタフェース SurfaceHolder の Callback を実装します
public class Step010View extends SurfaceView
                                    implements SurfaceHolder.Callback
{
    //②サーフェイスの生成
    public void surfaceCreated(SurfaceHolder holder){   ・・・   }

    //③サーフェイスの変更
```

```
    public void surfaceChanged
                        (SurfaceHolder holder, int format, int w, int h){}
    //④サーフェイスの破棄
    public void surfaceDestroyed(SurfaceHolder holder){}
}
```

SurfaceView を用いた描画を行うには、SurfaceView クラスそのものを利用するよりも、継承した独自のクラスを定義する方法が一般的です。SurfaceView は View のように自動的に onDraw などが呼ばれませんから、継承したクラス（ここでは、Step010View）では表示が更新されたときに何かをするように SurfaceView に起こったイベントを取得するインタフェース **SurfaceHolder** の Callback を実装します。(①)

実装といっても、SurfaceView に起きた３つのイベントを処理するために、以下のメソッドを定義するだけです。

・②surfaceCreated メソッド・・・SurfaceView の生成時

・③surfaceChanged メソッド・・・SurfaceView のサイズ等変更時

・④surfaceDestroyed メソッド・・SurfaceView の破棄時

これらのメソッドは、SurfaceView に該当するイベントが起きた際、システムが自動的に呼び出すようになっています。ゲームプログラムでは、SurfaceView が生成されたということは「画面描画ができるようになった」ことになるので、このタイミングでゲームを初期化し、メインループを開始させます。また、アプリケーション終了の際などに SurfaceView も破棄されますが、これは「画面描画ができない」ということになるので、このタイミングでメインループを終了させるようにします。

SurfaceView は自由なタイミングで画面の描画を行うことができ、マルチスレッドにも対応しています(View はマルチスレッド非対応)。そのため、システムが自動的に生成するメインスレッド以外の(アプリケーションが生成した)スレッドからも、SurfaceView に描画することができます。(View ではできません)

SurfaceView は、View の階層内での面を専門に描画する View クラスの特殊なサブクラスです。アプリケーションの 2 番目のスレッド (1 番目はメインスレッド) に面の描画を依頼することにより、アプリケーションがシステムの View 階層が描画の準備できるまで待たされることがなくなるというのがこのクラスのねらいです。そのかわり、SerfaceView に対するリファレンスを保持した 2 番目のスレッドが、自身の Canvas に対する描画を自分のペースで行えるようになります。

1.5　SurfaceViewの実装２

```
public class Step010View extends SurfaceView
                                    implements SurfaceHolder.Callback
{
    ・・・
    private SurfaceHolder holder;//サーフェイスホルダー
    ・・・
    public class Step010View(Context context)
    {
        ・・・
        //サーフェイスホルダの作成
        holder=getHolder();//①サーフェイスフォルダの取得
        holder.addCallback(this);//②サーフェイスフォルダの通知先の指定
        holder.setFixedSize(getWidth(), getHeight());//サーフェイスのサイズの指定
        ・・・
    //サーフェイスの生成
```

```
    public void surfaceCreated(SurfaceHolder holder)
    {
        //③Canvas オブジェクトをロックして取得
        Canvas canvas=holder.lockCanvas();
        //描画の設定
        Paint paint=new Paint();
        paint.setAntiAlias(true);//文字やラインを滑らかに見せる。
        paint.setTextSize(TSIZE);//文字サイズ
        //バッファの描画
        canvas.drawColor(Color.WHITE);//背景を塗りつぶす
        canvas.drawText(TEXT1, 10, 20, paint);//文字列の表示
        canvas.drawBitmap(image, getWidth()/2-16, getHeight()/2-16, paint);
        //④Canvas のロックを解除する。実画面に反映
        holder.unlockCanvasAndPost(canvas);
    }
    ・・・
}
```

①holder=getHolder();

Surface オブジェクトを直接ハンドリングする代わりに、SurfaceHolder を使ってそれをハンドリングする必要があります。

SurfaceView が初期化されたとき、 getHolder() を呼び出すことにより SurfaceHolder インスタンスを取得します。SurfaceHolder インスタンスの役割は、表示が更新されたときに何かをするように SurfaceView に起こったイベントを取得することです。

②holder.addCallback(this);

SurfaceHolder の Callback を呼び出して、コールバックに設定するインスタンスを指定します。これにより、サーフェイス上で何かの処理が行われると、コールバックに設定されたインスタンス内のメソッドが呼び出されるようになります。

③Canvas canvas=holder.lockCanvas();

SurfaceView では、通常の View の onDraw のように、引数で Canvas が渡されるわけではありません。ですから、最初に Canvas を取得するところから始めます。「lockCanvas」は、サーフェイスの表示をロックし、描画のための Canvas を返します。

④holder.unlockCanvasAndPost(canvas);

「unlockCanvasAndPost」は画面のロックを解除して表示を更新します。これでその面がそこに残したいグラフィックを描画するようになります。 この Canvas のロック、アンロックのシーケンスを再描画したいタイミングで毎回実行します。

SurfaceView では、このように「ロック→Canvas 取得→描画→アンロック＆更新」という一連の流れが描画の基本となります。

■注意:
SurfaceHolder から Canvas の取得処理を行うたびに、Canvas の前の状態が保持されたままになっています。適切にグラフィックを動かすには、面全体を再描画する必要があります。例えば、 drawColor() で、ある色に塗りつぶしたり、背景のイメージを drawBitmap() で設定したりして、Canvas の前の状態をクリアすることができます。それをしないと、それ以前に実行した描画の痕跡が残ってしまいます。

1.6　イメージの描画

1.6.1　画像ファイルの準備

画像ファイル形式は「PNG-8(256 色以下のインデックスカラーPNG)」「JPEG」「GIF」「BMP」です。プロジェクトの res フォルダに置いてください。

大きな画像がぼやける場合は、「AndroidManifest.xml」に次の項目を追加してください。

```
<supports-screens android:anyDensity="true" />
```

ゲームで多く使用される透明色(キャラクターの周辺など画像には存在するものの画面には描画しない色)は、アルファ情報付きの PNG 形式もしくは透過 GIF で行います。

リソースへの登録は、res フォルダの drawable フォルダにファイルをドラッグ＆ドロップで追加し、プロジェクトをリフレッシュすれば、自動的にコンパイルされて認識されます。

drawable フォルダは、密度にあわせて３つのフォルダがあります。XPERIA や GALAXY などの高密度端末は、「drawable-hdpi」フォルダに登録します。「drawable-mdpi」は中密度、「drawable-ldpi」は低密度用、「drawable-nodpi」は全ての密度用です。

Android 1.6 以上は複数の密度対応となっています。密度に合わせた画像を用意したり、プログラムを工夫することにより、あらゆる密度で同じように動作するアプリケーションを作成することができます。なお、リソースフォルダに「Thumbs.db」があるとコンパイルエラーになりますので削除してください。

1.6.2　内容

リソースオブジェクトを作って表示させるには次のように記述します。

```
Resources r=context.getResources();//①
image=BitmapFactory.decodeResource(r, R.drawable.rocket);//②
・・・
//③イメージの表示
canvas.drawBitmap(image, getWidth()/2-16, getHeight()/2-16, paint);
・・・
```

①Resources r=context.getResources();

リソースオブジェクトは、Context クラスの getResources()メソッドで取得します。

②image=BitmapFactory.decodeResource(r, R.drawable.rocket);

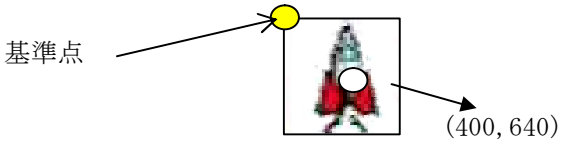
リソースの画像ファイルを読み込むには、BitmapFactoryクラスのdecodeResource()メソッドを使います。このメソッドの引数は、リソースオブジェクトとリソースIDを指定します。リソースIDは、res/drawableフォルダに配置したファイルを「R.drawable.ファイル名（拡張子なし）」の書式で指定します。戻り値としてBitmapオブジェクトが返ってきます。

③canvas.drawBitmap(image, getWidth()/2-16, getHeight()/2-16, paint);

ここでは、imageという変数に画像ファイル（rocket.png）のデータを読み込んでいます。しかし、データを読み込んだだけでは表示はされません。表示させるには、CanvasクラスのdrawBitmap()メソッドを使います。

書式１
canvas.drawBitmap(Bitmap bitmap, float left, float top, Paint paint);
　bitmap・・・描画するビットマップ
　left　・・・描画先の x 座標
　top　・・・描画先の y 座標
　paint　・・・描画時に使用する paint クラス。使用しない場合は null。

x 座標、y 座標の基準点は画像の左上になります。画面のサイズは幅が 800、高さが 1280　です。画面の真ん中に画像を表示させる座標は(400,640)になりますが、画像サイズが、32 ピクセル×32 ピクセルのため、次のような計算式の座標となります。



例　canvas.drawBitmap(image, getWidth()/2-16, getHeight()/2-16, paint);

書式２
canvas.drawBitmap(Bitmap bitmap, Rect src, Rect dst, Paint paint);
　bitmap・・・描画するビットマップ
　src　・・・描画元の領域。　null にするとビットマップ全体が描画される。
　dst　・・・描画先の領域。　null は不可
　paint　・・・描画時に使用する paint クラス。使用しない場合は null。

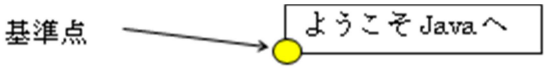
書式３
canvas.drawBitmap(Bitmap bitmap, Rect src, RectF dst, Paint paint);
　bitmap・・・描画するビットマップ
　src　・・・描画元の領域。　null にするとビットマップ全体が描画される。
　dst　・・・描画先の領域。 float 型で指定できる。　null は不可
　paint　・・・描画時に使用する paint クラス。使用しない場合は null。

キャンバスの(120, 240)から(360, 480)の領域にビットマップ全体を伸縮して描画する例

```
Rect dst = new Rect(120, 240, 360, 480);
canvas.drawBitmap(charaBmp, null, dst, null);
```

1.7　文字列の表示

文字列を表示するには、Canvas クラスの drawText()メソッドを使います。
X 座標、Ｙ座標の基準点は文字列の左下になります。



書式
canvas.drawText(文字列, int 表示位置の x 座標,
　　int 表示位置の y 座標, Paint インスタンス);

1.8　ダブルバッファリング

描画命令を実行ごとに画面に反映させないで、複数の描画命令をまとめて反映させ、画面のちらつきを防ぐための技術です。

一般的な描画は、画面を一度クリア（もしくは塗りつぶし）してから背景、キャラクターと描いていきます。キャラクターを動かすなどのアニメーションを行うときは

これを繰り返し行います。

通常の描画の場合、画面に直接描画を行っています。そのため、画面をクリアした際に何も無い画面が一瞬見えてしまうために画面がちらついて見えてしまうことがあります。

ダブルバッファリングでは、表示されている画面とは別に、裏にもう一つの画面（オフスクリーンバッファ）を持ちます。裏の画面で描画を行い、描画が完成したら表の画面と入れ替えることで描画している姿を見せないようにしています。これによりクリアされた画面が見えないためにちらつきを防ぐことができます。

Java SE ではオフスクリーンバッファで行っていましたが、Android ではフレームワークが内部でダブルバッファリングを行ってくれるため、オフスクリーンバッファを持たなくてもよくなりました。

ダブルバッファリングの使い方は、描画前に SurfaceHolder クラスの lockCanvas()メソッドで描画用の Canvas オブジェクト取得し、描画処理後、unlockCanvasAndPost()メソッドを呼ぶだけです。unlockCanvasAndPost()メソッドを呼んだときに、Canvas オブジェクトに描画した内容が実画面に反映されます。

1.9　演習

ソースの中に演習番号と組席名前を明記してください。

【演習１１】

「宇宙船」の画像と「はるかなる未知の惑星へ！」の文字を表示するプログラムを作成してください。

・プロジェクト名　　：StepPro????（年組席）

・アプリケーション名　：Step011View

・画像ファイル名：r0.png

・[res]フォルダへ r0.png ファイルを入れる。

※Step010View をコピー＆ペーストして、ファイル名を「Step011View」に変更してください。