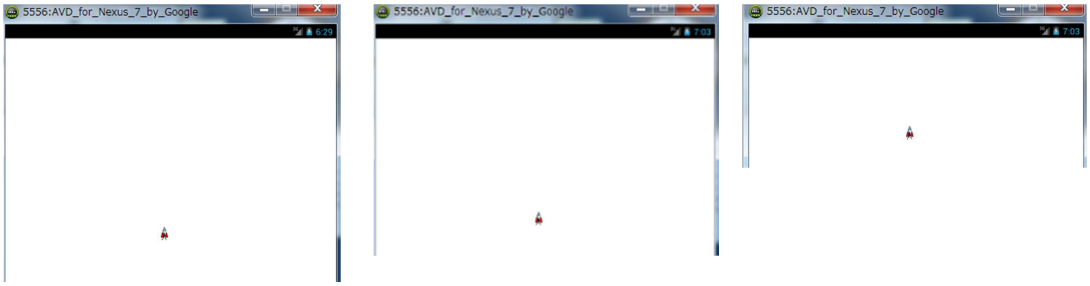
３　アニメーション

スレッド処理で、ロケットの画像を動かします。



上の画面のようにロケットが上昇します。

3.1 ソースコード

(1) テンプレート　Step の①の箇所に　Step030View　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。

Step010View をコピー＆ペーストして、ファイル名を「Step030View」に変更してください。

プロジェクト名：StepPro????(年組席)　　アプリケーション名：Step030View

※public void surfaceCreated(final SurfaceHolder holder)のソースは全部入れ替えてください。

```
/*    年    組    席  名前
 * Step030View    ScheduledExecutorService 仕様
 *    アニメーション
 */
package jp.edu.mie;
 ・・・
import android.view.SurfaceView;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class Step030View extends SurfaceView
                                        implements SurfaceHolder.Callback
{
    ・・・
    private float rotateY; //回転の中心の y
    private int px;//X 座標
    private int py;//Y 座標
    private ScheduledExecutorService executor;//マルチスレッド処理
    //コンストラクタ
    public Step030View(Context context)
    {
        ・・・
    }
    //サーフェイスの生成
```

```
    public void surfaceCreated(SurfaceHolder holder)
    {
        //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
        px=getWidth()/2-16;//X 座標
        py=getHeight()/2-16;//Y 座標
        //マルチスレッド処理      ScheduledExecutorService の取得
        executor=Executors.newSingleThreadScheduledExecutor();
        //Runnable の実行   executor.scheduleAtFixedRate
        //                        (Runnable,間隔,ディレイ （遅延時間）,時間単位 );
        executor.scheduleAtFixedRate(
            new Runnable()
            {
                public void run()
                {
                    draw(canvas);
                    py--;//py=py-1
                }
            },0,1,TimeUnit.MILLISECONDS);
    }
    public void draw(Canvas canvas)
    {
        //Canvas オブジェクトをロックして取得
        canvas=holder.lockCanvas();
        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, px, py, null);//画像の表示
        holder.unlockCanvasAndPost(canvas);//実画面に反映
    }
    //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){executor.shutdown();}
}
```

3.2　スレッド処理

最初に（384,624）の座標位置に画像を表示して、その位置より高い位置に画像を表示させるためには、(384,624)より上の位置だから、画像の y 座標を 624 より小さくすれば上の位置に表示されます。

さらにその上に表示するためには、もっと y 座標を小さくすれば上に表示されます。だんだん y 座標を小さくしていけば、画像が上に動くことになります。

プログラムは同時に一つの処理しかしません。Java のプログラムは通常、同時に一つの処理しかできませんが、今回のように同時にいろいろな処理をしたいときもあります。そこで使うのが**スレッド処理**です。

> スレッドとはプログラムの中に独立で実行される 1 つの処理を表すクラスです。

※Thread：糸

通常、プログラムが実行されると一つだけスレッドが作成されてプログラムを処理します。複数の処理を並行して実行したい場合はスレッドをいくつか使えばいいということです。

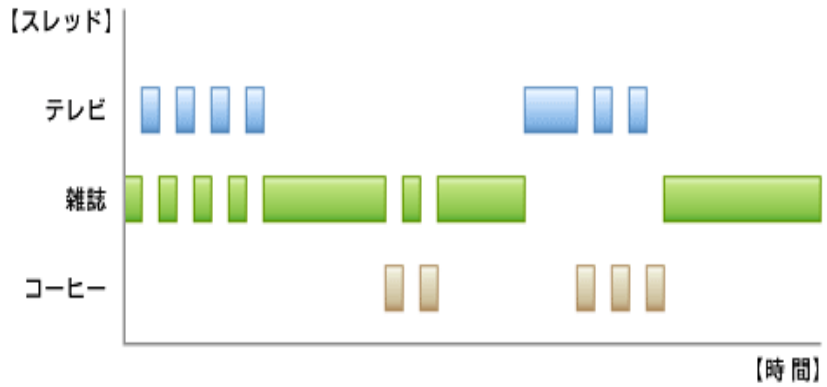
同じプログラムに属するスレッドはメモリなどのリソースを共有します。複数のスレッドは CPU を交互に占有することによって見かけ上同時実行が可能です。高度な処理を行なうアプリケーションソフトなどでは、スレッドを複数走らせることにより、同時に複数の処理を実行できます。時間のかかる演算処理中にユーザからの入力を受け付けるといった工夫が可能になります。

人は無意識のうちにいろいろなことを同時に行っています。同時に行ってはいるものの、完全に同時ではないのも理解できると思います。

テレビを見ながら雑誌を読みつつコーヒーを飲む人

テレビと雑誌はまったく同時には見られませんし、雑誌を見ながらコーヒーを飲もうとして手を伸ばすとこぼしてしまうかもしれませんから、そのときはコーヒーの方を見て確認するはずです。雑誌を読むことに 100%没頭してしまうと、テレビを見ることもコーヒーを飲むこともできません。

下の図はスレッドスケジューリングを模式図にしたものです。基本的に雑誌を読んでいますが、時々テレビを見たりコーヒーを飲んだりしているようです。ここで重要なのは、雑誌とテレビを同時に見ているようでも、細かく分解すると、短い周期で交互に見ているということです。



あなたのスレッドスケジューリング

3.3　スレッドを使う方法には

①Thread クラスを拡張する（サブクラスとする）

②Runnable インターフェースを実装する

③ScheduledExecutorService クラスを実装する

という３つの方法があります。

3.3.1　Thread クラスを拡張する（サブクラスとする）

この方法は、Thread を拡張し、run()メソッドを定義してスレッドを扱う準備をした後、start()メソッドの呼び出しでスレッドを起動します。具体的には、

```
class Car extends Thread
{
    public void run()
    {
        別スレッドで行いたい処理;
        ・・・
    }
}
class Sample
{
    public static void main(String args[])
    {
        Car car1=new Car("１号車");
        car1.start();
        ・・・
    }
}
```

3.3.2　Runnable インタフェースを実装する　（Runnable：運転できる状態の）

Java では２つ以上のクラスから多重継承することができません。つまり、Thread クラスとそのほかのクラスの２つを、スーパークラスにすることはできません。

Thread クラスを拡張しようとしたクラスが、ほかのクラスを継承しなければならない場合には、クラスライブラリの Runnable インタフェースを使います。（Runnable インタフェースを実装するともいいます。）Runnable インタフェースは、java.lang パッケージにあります。

具体的にはまず、スレッドを使う Step030View クラスの宣言を次のようにします。

```
public class Step030View extends SurfaceView
                  implements SurfaceHolder.Callback, Runnable
{
```

【インプリメント（implement)】　　道具と訳される。
ハードウエアやソフトウェアに新しい機能や仕様、部品などを組み込むこと。また、実際にその機能を組み込む際の手法も意味する。日本語でいうと「実装」。

「implements Runnable」をクラス宣言に付け加えると、「このクラスにスレッドが行う処理を記述する。」という意味になります。

これからスレッドが行う処理を、この View クラスの中で定義します。次のメソッドでスレッドが行う処理を定義する事ができます。まず、スレッドが実行する処理をこのように記述します。

```
 //スレッドが実行する処理
public void run() {
    // 以下に処理を記述
     ・・・
}
```

次に、実際にこの処理を実行させる為には以下の処理を行う必要があります。

```
//スレッドの開始
thread=new Thread(this);//Thread クラスのオブジェクトを作成します
```

```
thread.start();
```

この文は、「処理を行うスレッドを新しく作って、そのスレッドを実行する。」という意味です。

3.3.3　ScheduledExecutorService クラスを実装する

```
public class Step030View extends SurfaceView
                                        implements SurfaceHolder.Callback
{
    ・・・
    private ScheduledExecutorService executor;//マルチスレッド処理
    ・・・
    //サーフェイスの生成
    public void surfaceCreated(final SurfaceHolder holder)
    {
        ・・・
        //①マルチスレッド処理      ScheduledExecutorService の取得
        executor=Executors.newSingleThreadScheduledExecutor();
        //②Runnable の実行　executor.scheduleAtFixedRate
        //                      (Runnable,間隔,ディレイ（遅延時間）,時間単位 );
        executor.scheduleAtFixedRate(
            new Runnable()
            {
                public void run()
                {
                    draw(canvas);
                     py--;//py=py-1
                 }
            },0,1,TimeUnit.MILLISECONDS);
        }
```

①executor=Executors.newSingleThreadScheduledExecutor();

ScheduledExecutorService クラスは、Java SE 1.5 から導入された、マルチスレッドを安全・効率的に制御することができるクラスです。このクラスは、定期的もしくは一定間隔で Runnable インタフェースの run メソッドを実行することができます。従来は Thread クラスと Sleep メソッドを組み合わせたり、Timer クラスを用いることで実装していましたが、それらの欠点を補ったものです。

スレッドは、生成時に OS や Java 仮想マシンに負荷がかかります。また、生成していくたびに、CPU やメモリなどの資源を消費しますが、スレッドの数が CPU の能力を超えてしまうと、スレッドの実行が停滞してしまいます。

これらの問題点を簡単に解決できるのが **ExecutorService** クラスです。このクラスは、スレッドの生成数に制限をもたせることができます。CPU にあったスレッド数に制限することで、資源を無理なく最大限に利用できます。また、スレッドの終了を安全にできる機能もあります。

ScheduledExecutorService クラスは、**ExecutorService** クラスを継承し、定期的もしくは一定間隔にスレッドを実行する機能を追加したクラスです。ゲームは一定間隔で画面を更新する必要があります。また、処理落ちした場合は、描画処理などを省略し、本来の時間まで追いかけるアルゴリズムがありますが、そういった機能を簡単に実装できる機能が備わっています。

Executors クラスに用意されている、スケジューリングのための Executor は、次の二種類です。

・newSingleThreadScheduledExecutor()

一つのスレッドでタスク処理を行う。

・ newScheduledThreadPool()

タスクを処理するスレッドの数を指定出来ます。

ScheduledExecutorService で実装されているメソッドは 3 つあります。

(1)schedule

(2)scheduleWithFixedDelay

(3)scheduleAtFixedRate

(1)schedule

これは更に 2 つあって、第一引数が Runnable か Callable かをとるものです。

(a)Runnable バージョン。

```
ScheduledFuture<?> future = service.schedule(new Runnable()
{
   public void run() {
           System.out.println("hello, world!");
   }
}, 1, TimeUnit.SECONDS);
```

これを実行すると、1 秒後に hello, world を出力します。

(b)Callable バージョン

```
ScheduledFuture<Integer> future = service.schedule(new
Callable<Integer>() {
    public Integer call() throws Exception {
        return 5;
    }
}, 1, TimeUnit.SECONDS);

int result = future.get();
System.out.println(result);
```

戻り値を取得する事ができます。注意点は、これを実行するとプログラムが終了しない点です。

```
service.shutdown();
```

こう書くことで、呼び出し側はスケジュールされた処理が全て終わった時点で終了となります。

(2)scheduleWithFixedDelay

定期実行させる時に使用するメソッドです。

```
ScheduledFuture<?> future =
    service.scheduleWithFixedDelay(new Runnable() {
            public void run() {
                    System.out.println("あ");
            }
    }, 1, 2, TimeUnit.SECONDS);
```

この場合、1 秒後に処理が開始され、処理が終わってから 2 秒後にまた処理が開始され、という風に実行します。

次の周期までに処理が終わっていなかった場合は、処理終了を待ってからすぐ実行されます。

止めるには、

```
service.shutdown();
```

とするか、次のようにするかのどちらかです。

```
future.cancel(true);
service.shutdown();
```

(3)scheduleAtFixedRate

さきほどの scheduleWithFixedDelay と非常に似ています。

こちらは、再実行のタイミングが異なります。処理終了からどれだけ待つかという意味になります。

```
ScheduledFuture<?> future
                = service.scheduleAtFixedRate(new
              Runnable() {
   public void run() {
           System.out.println("あ");
   }
}, 1, 2, TimeUnit.SECONDS);
```

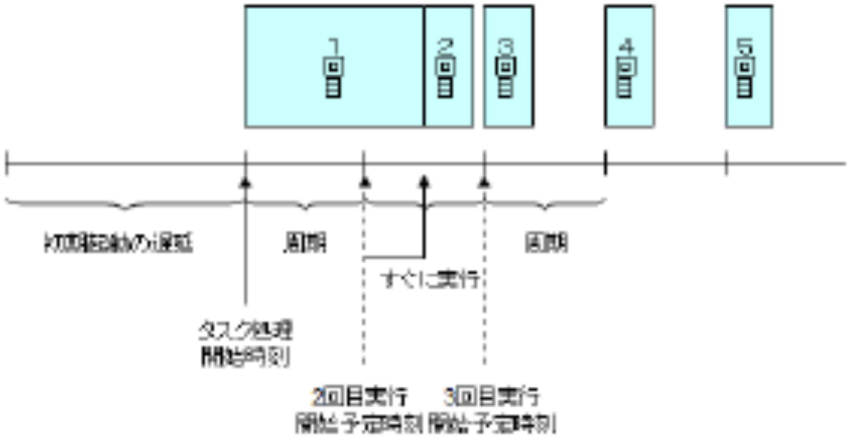
②executor.scheduleAtFixedRate(

ScheduledExecutorService クラスのスケジュール機能

①で説明した内容をさらに詳細に説明します。

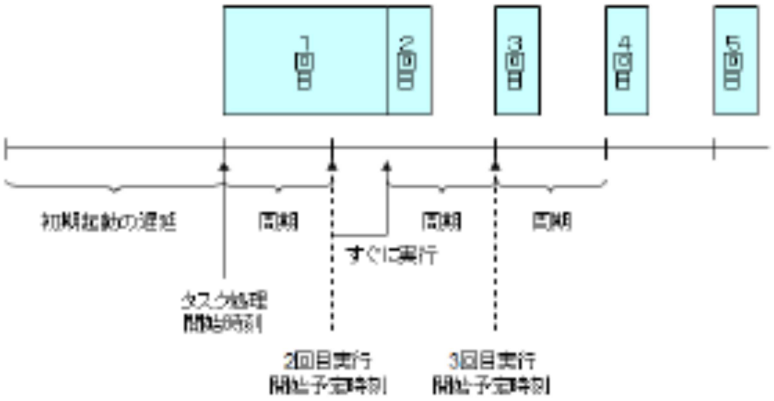
ScheduledExecutorService クラスのスケジュール方法は２つあり、「処理を開始する間隔」もしくは「処理の終了から次の処理の開始までの間隔」を指定することができます。

「処理を開始する間隔」とは、処理が開始される時期が一定であるということです。たとえば「毎週月曜日に処理が実行される」とすると、処理が火曜日に終わったとしても、日曜日までかかったとしても、次の実行は月曜日になるというものです。処理に時間がかかりすぎて、次の週の火曜日までかかった場合は、さらに次の月曜日まで待つのではなく、処理が終わり次第、すぐに次の処理にとりかかります。月曜日に実行できるようになるまで、待つことなく「次の処理」を実行し続けます。このようなスケジュール方法を行っているのが scheduleAtFixedRate メソッドです。



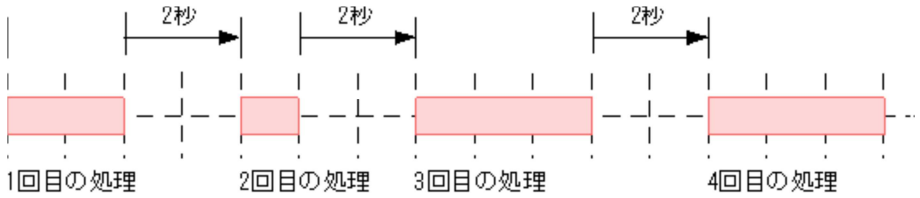
▲scheduleAtFixedRate メソッドのスケジュール方法

同じようなスケジュールを行っているのが schedule メソッドです。これは「毎週同じ曜日を目指す」のではなく「処理の開始」と「次の処理の開始」の間隔が一定というものです。間隔が７日間だとすると、最初の処理の開始が月曜日であれば、次の処理の開始は基本的には月曜日になります。しかし、ある処理で滞り、次の週の火曜日までかかってしまった場合は、その次の処理はすぐに開始されますが、さらにその次の処理の開始は次の火曜日になる、というものです。第４回で実装したメインループの間隔も、このスケジュール方法と同じです。



▲schedule メソッドのスケジュール方法

これらのメソッドのほかに、間隔そのもの(処理と処理の間の休止時間)が一定の scheduleWithFixedDelay メソッドもあります。



▲scheduleWithFixedDelay メソッドのスケジュール方法

3.4　座標を動かす処理

座標を動かす処理は、run メソッドの中に書きます。その前に画像の表示座標を変数にして値を変化できるようにしなくてはいけません。

オブジェクト（インスタンス）変数を使って座標を定義します。

```
px=getWidth()/2-16;  //X 座標
py=getHeight()/2-16; //Y 座標
```

run メソッドの中身は、次のとおりです。ロケットがすっと上昇していくように、y 座標を減少させています。

```
public void run() {
    draw(canvas);
    py--;//py=py-1 の式と同じ、y 座標の減少
}
```

```
public void draw(Canvas canvas)
{
    //Canvas オブジェクトをロックして取得
    canvas=holder.lockCanvas();
    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, px, py, null);//画像の表示
    holder.unlockCanvasAndPost(canvas);//実画面に反映
}
```

3.5 演習

【演習３１】

「宇宙船」の画像を上昇させるプログラムを作成してください。なお、宇宙船の上昇に合った背景イメージを作成して表示してください。

プロジェクト名：StepPro????(年組席)　　アプリケーション名：Step031View

ロケット画像「r0.png」

背景イメージは、ＣＧ又は画像を使用してください。

※ゆっくり移動させる処理は実習４で行います。