情報メディア基盤ユニット用資料（2016 年度第 1 章）

Processing 言語による情報メディア入門

プログラムを使って絵を描く

神奈川工科大学情報メディア学科　佐藤尚

Processing とは？

Processing とは、アメリカのマサチューセッツ工科大学の BenFry さんと CaseyReas さんによって作られた視覚デザインのためのプログラミング言語と開発環境のことです。Processing の公式サイトは http://www.processing.org です。ここから Processing のプログラムなどをダウンロード出来ます。情報メディア基盤ユニットでは、Processing 言語を利用して、情報メディア系でのプログラミングに必要とされる基本的な考え方を修得することを目指します。

Processing は以下のような特徴を持っています。

- C 言語や Java 言語を利用するよりも簡単にインタラクティブかつビジュアルなプログラムを作成することができる。
- OpenGL などの機能も利用できるので、3 次元表現を伴うようなプログラムを作成できる。
- Java の機能を利用して機能を拡張することができる。
- Windows, MacOSX, Linux で実行できる。
- Android 用のプログラムも作ることが出来る。iPhone でも、Processing 言語のプログラムを作ることが出来る。

Examples の中に色々なサンプルプログラムが入っているので、実行してみると Processing でどんなことが出来るかがわかります。

Processing を使ってみる

プログラム作成するためのテキストエディタエリア、ツールーバー、コンソールエリア、メッセージエリアから出来ています。実行ボタン (Run) を押すと、プログラムが実行されます。

Run ボタン : プログラムを実行する際に利用します。

Stop ボタン : プログラムを停止させる際に利します。

New ボタン : 新しいファイル（スケッチ）を作成する。Processing では、プログラムを書いたファイルなどをまとめてスケッチ (sketch) と呼んでいます。

Open ボタン : 既存のスケッチを読み込む。 このボタンをクリックすると、別なウィンドウが開き、そこから保存されているスケッチを読み込みます。

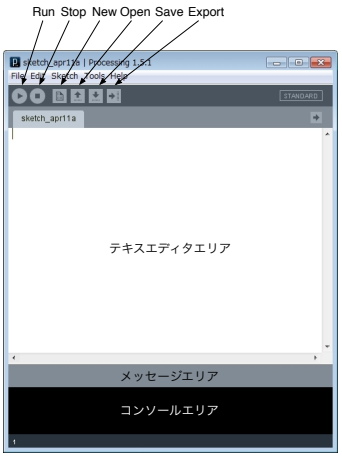
Save ボタン : 表示されているスケッチに名前をつけて保存する際に利用します。

Export ボタン : 表示されているスケッチ を Java アプレットとして保存します。その 際には、Java アプレットを表示するため に最低限必要な HTML ファイルも作成さ れます。

Processing を用いて作られるプログラムは、スケッチ (sketch)

テキストエディタ (Text Editor) とは？

基本的に文字情報のみからなるファイルを作成するために利用するソフトウェアのこと。



Processing の起動画面

と呼ばれています。保存をすると、ドキュメントフォルダの中の Processing というフォルダ内に新しくフォルダを作り、その中にスケッチを構成するプログラムやデータを保存します。

Processing プログラムの基本形その 1

プログラミングの基本にあるのは命令文です。これは、私たちの使っている言語に対応させれば、文に相当するものです。命令文は処理内容を表現したものです。普通の文の終わりに句読点を置くのと同じように、命令文の終わりには ;(セミコロン) を置きます。普通の文にも色々な文が存在するように、Processing の命令文にも様々な種類のものが存在しています。興味のある人は、http://www.processing.org/reference/index.html を見ると、どのような命令文があるのかがわかります。

Processing のプログラムでは、大文字と小文字を区別します。例えば、Size と size は異なったものとして扱われます。命令文と命令文の間の半角スペースは無視されます。ただし、全角のスペースを使うと、エラーとなるので気をつけてください。また、半角の ” と全角の ” も別なものとして扱われますので、気をつけてください。簡単に言うと、「全角文字を使うときには気をつけましょう！！」です。

Processing を起動して、テキストエディタエリアに以下の命令文を打ち込んで下さい。

最初のプログラム 1-1

```
ellipse(50,40,80,70);
```

この命令文の意味は、「(50,40) を中心に、横方向 80、縦方向 70 の楕円を描け」です。この命令文を入力し終わったら、“Run” ボタンをクリックして下さい。

この次はもう少し長い例です。同じようにエディタに入力し、入力が終わったら、“Run” ボタンをクリックして下さい。

2 番目のプログラム 1-2

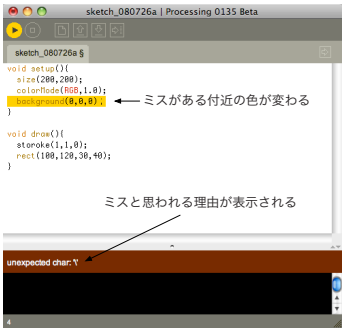
```
size(400,400);
ellipse(200,200,80,80);
ellipse(50,50,50,50);
ellipse(300,350,80,80);
```

Processing プログラムの基本形その 2

New ボタンを押して、新しいスケッチを作り、テキストエディタエリアに以下の命令を打ち込んで下さい。打ち込み終わったら、“Run” ボタをクリックして下さい。

命令文：プログラミング言語の種類によっては別な言い方をするかもしれません。

英語は苦手という人は、少し古いバージョンの物ですが、http://www.technotype.net/processing/reference/index.html に日本語訳があります。



エラー発生時の画面

基本形その 2 のプログラム 1-3

```
void setup(){
  size(640,480);
  smooth();
}

void draw(){
  if(mousePressed){
    fill(0);
  }else{
    fill(255);
  }
  ellipse(mouseX,mouseY,80,80);
}
```

適当な場所に空白や空行を入れることで読みやすいプログラムを作ることが出来ます。特に、行の開始位置を下げることで、命令文の塊を明らかにすることをインデント（indent）または字下げと呼びます。

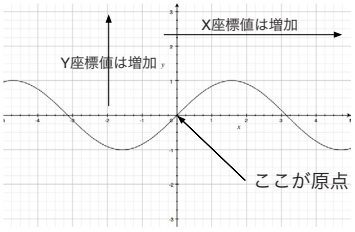
Processing のプログラムは、単純に命令文を一列に並べたものと、いくつかの塊に構造化して並べたものの 2 種類に分けることが出来ます。後者の場合には、基本的に setup と draw という 2 つの塊から成り立っています。setup には、最初だけ実行する命令文を書き、draw にはそうでない部分（プログラムの本体とでも言うべき部分）を書きます。setup の部分は実行開始時に 1 回だけ実行されますが、draw の部分は定期的に呼び出されて、何度も実行されます。Processing 言語では、この塊のことを関数と呼んでいます。少し複雑なプログラムになると、setup と draw 以外の塊（関数）を利用します。

正確には、Processing 言語では、setup や draw などを関数と呼びます。

今日の授業では、基本形 1 のような単純に命令文が一列並んだタイプのプログラムを作って行きます。

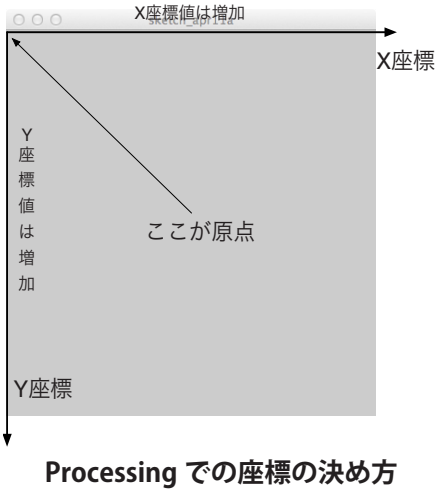
図形の描画

Processing 言語のプログラムを作る上で、おそらく最もよく使われる命令（関数）は size です。これは、横 x 画素、縦 y 画素の大きさのウィンドウを表示する命令です。



数学での座標の決め方

Processing 言語で図形を描く場合には、座標を利用して位置の指定を行います。つまり、X 座標値と Y 座標値があれば、平面上の点の位置を決めることか出来ます。そこて、2 つの値を利用して点の位置を決めます。数学では左下に原点を置きますが、Processing 言語では基本的に左上か原点となります。このため、X 軸方向は、左から右に移動するにつれて、座標値は大きくなりますが、Y 軸法では、上から下に移動するにつれて、



Processing での座標の決め方

座標値が大きくなります。Processing 言語での座標の決め方に気をつけてください。

ウィンドウの表示

命令名（関数名）	意味
size(x,y);	横 x 画素、縦 y 画素の大きさのウィンドウを表示する。

基本的な図形の描画に関連する命令（関数）

命令名（関数名）	意味
line(x1,y1,x2,y2);	点 (x1,y1) と点 (x2,y2) の間に線分を描く。
ellipse(x,y,w,h);	基本的には、(x,y) を中心として、幅 w、高さ h の楕円を描く。
triangle(x1,y1,x2,y2,x3,y3);	3 点 (x1,y1)、(x2,y2)、(x3,y3) を頂点とする三角形を描く。
rect(x,y,w,h);	基本的には、(x,y) を左上の頂点とする幅 w、高さ h の長方形を描く。
quad(x1,y1,x2,y2,x3,y3,x4,y4);	4 点 (x1,y1)、(x2,y2)、(x3,y3)、(x4,y4) を頂点とする四角形を描く。
arc(x,y,w,h,s0,s1);	基本的には、(x,y) を中心として、幅 w、高さ h で角度 s0 から角度 s1 までの半円を描く。
point(x,y);	位置 (x,y) に点を描く。
radians(theta);	度で表された角度を弧度法（ラジアン）に変換する。

複数の点を描くプログラム 1-4

```
size(400,400);

point(100,200);
point(100,100);
point(399,399);
```

線の描画するプログラム 1-5

```
size(480,120);
line(20,10,460,110);
```

両端の点の位置を指定すると線分が決まることを思い出して下さい。

円の描画プログラム 1-6

```
size(400,200);
ellipse(280,-100,400,400);
ellipse(120,100,110,110);
ellipse(360,100,18,18);
ellipse(250,180,200,60);
```

英語では、楕円のことを ellipse と言います。円は楕円の特別な場合なので、楕円を描くことが出来れば、円も描くことが出来ます。

長方形の描画プログラム 1-7

```
size(480,120);
rect(20,10,450,100);
```

長方形のことを矩形と呼ぶことあります。英語では、rectangle と言います。
座標軸に平行な辺を持つ長方形は左隅の頂点の位置と幅と高さを指定すれば決まることを思い出して下さい。

三角形と四角形の描画プログラム 1-8

```
size(400,400);
triangle(250,30,380,100,300,300);
triangle(140,30,220,380,110,350);
quad(100,100,200,80,240,300,150,200);
```

円弧の描画（その 1）プログラム 1-9

```
size(400,400);
arc(200,200,300,300,radians(30),radians(330));
```

Processing 言語では角度の大きさの指定には弧度法（ラジアン、radian）を利用します。

円弧の描画（その 2）プログラム 1-10

```
size(480,120);

arc(90,60,80,80,0,HALF_PI);
arc(190,60,80,80,0,PI+HALF_PI);
arc(290,60,80,80,PI,TWO_PI+HALF_PI);
arc(390,60,80,80,QUARTER_PI,PI+QUARTER_PI);
```

角度の大きさを指定するの弧度法を利用する場合には、円周率の値が使えると便利です。そのため、円周率πに関連する値を表す特別な名前が用意されています。

弧度法を扱うのに便利な名前（定数）

PI のように特別な値を表す名前のことを定数 (constant) と呼びます。

名前	意味	値
PI	円周率πの値を表す。	3.14159265358979323846
TWO_PI	2 πの値を表す。	6.28318530717958647693
HALF_PI	円周率の半分の値を表す。	1.57079632679489661923
QUARTER_PI	円周率の 4 分の 1 の値を表す。	0.78539816339744830961

描画の順番による結果の違い

Processing 言語では、図形の描画命令を実行する順番を変えると、描かれる画像が変化することがあります。次のサンプルプログラムを実行して、結果の違いを見て下さい。

描画命令を並び替えると（円→長方形）プログラム 1-11

```
size(400,200);
ellipse(140,0,190,190);
// The rectangle draws on top of the ellipse
// because it comes after in the code
rect(100,30,260,20);
```

描画命令を並び替えると（長方形→円）プログラム 1-12

```
size(400,200);
rect(100,30,260,20);
/*
 The ellipse draws on top of the rectangle
 because it comes after in the code
*/
ellipse(140,0,190,190);
```

順々に上書きされて描かれていくので、後から描いた図形が優先されます。一般的に、コンピュータのプログラムでは命令文を並べる順番を変更すると、実行結果が変わります。

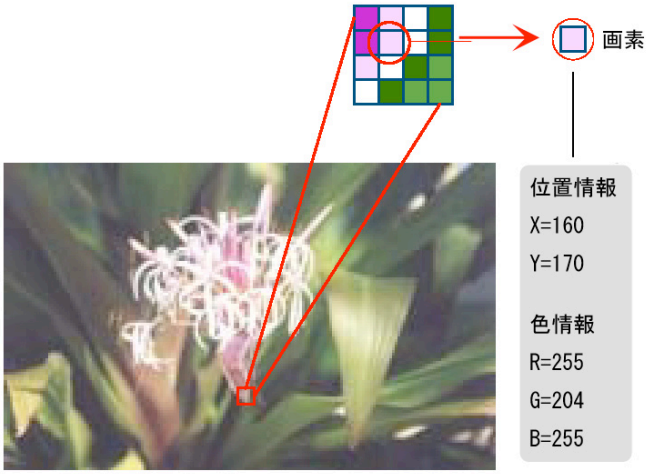
コメントとは？

プログラム内に書いた、プログラムの説明などをコメント (comment) と呼びます。Processing では、「//」をと書くと、これ以降行末まではコメントして扱われます。コメントは単なる説明なので、プログラムの動作には影響を与えます。複数行にわたるコメントを書く場合には、/* ～ */ という形式のコメントを使用することもあります。

図形の属性を変更する

基本的に、デジタル画像は画素と呼ばれる色の付いた小さい板の集まりとして記憶されています。そのため、デジタル画像では、画素の色とそれをどこの場所に置くかの情報を決める必要があります。

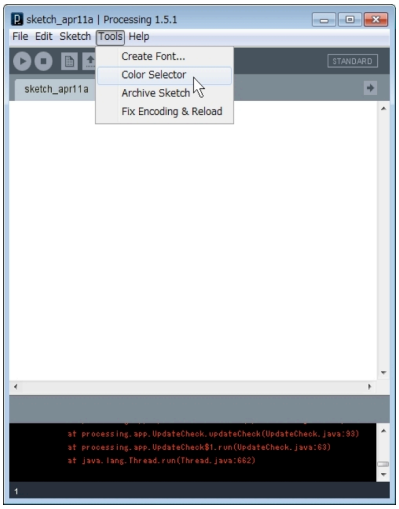
画素の色は、赤 (Red)、緑 (Green)、青 (Blue) の割合によって決めることが一般的です。Processing 言語では、色を指定する際には、特に指定をしない限り、この 3 つの値 (RGB) を 0 から 255 までの数字を用いて色を表現します。この RGB 以外にも、不透明度 (アルファ値) の情報を加えて 4 つの値 (RGBA) を用いることもあります。



デジタル画像のイメージ

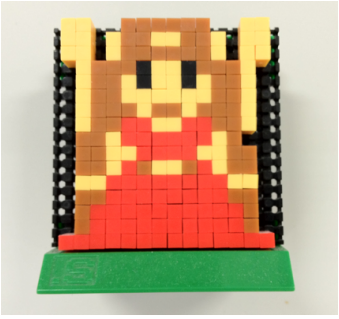
RGB 以外にも HSB と呼ばれる、画素の色指定の方法があります。これは、色味を表す色相（Hue)、色の明るさを表す彩度 (Saturation)、色の鮮やかさを表す輝度 (Brightness) の 3 つの数値で色を指定するものです。Processing 言語のデフォルトでは、色相の情報は 0 ～ 360、色相と彩度の情報は 0 ～ 100 の数値で指定します。

自分の欲しい色を RGB の数値データとして表すことは、少し難しい作業です。そこで、Processing では Tools > Color Selector と呼ばれる機能が用意されています。これを利用することで，視覚的に自分の欲しい色の数値データを確認することが出来ます。Tools>Color Selector を選択すると、次のような Color Selector ウィンドウが開きます。



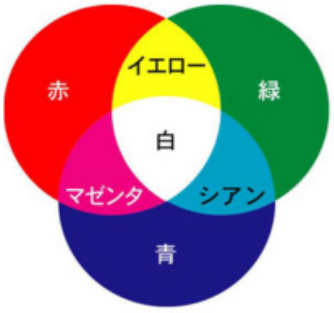
Color Selector の呼び出し方

右上の細長い四角形の色を数値データとして表したものが、HSB や RGB の部分の数字となって表示されています。細長い四角形をクリックすると色味（色相）を選択することが



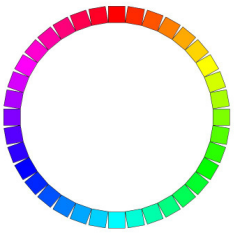
リアルなデジタル画像？

画素のことをピクセル (pixel) と呼ぶこともあります。



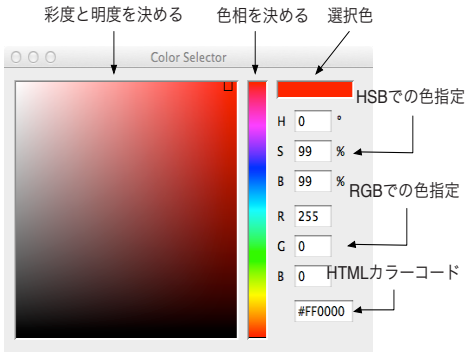
色の三原色

画素のことをピクセル (pixel) と呼ぶこともあります。色相の情報は 0 ～ 360 で表されているので、色相の異なる色を円周上に並べることが出来ます。これを色相環と呼ぶことがあります。



コンピュータ関連業界（?）では、ユーザが特に指定しない場合に、あらかじめ設定されている値また動作条件のことをデフォルト (default) と呼んでいます。

出来ます。左側の大きな四角形をクリックすると、色の明るさや鮮やかさやを変更することが出来ます。



Color Selector の機能

今までのプログラムで描かれた円や楕円を見ると、少しガタガタしているように見えます。もっと綺麗な円や楕円を描きたい時には、smooth 命令を使います。

楕円の描画滑らかにする

命令名（関数名）	意味
smooth():	楕円の描画を滑らかにする。
noSmooth();	楕円を滑らかに描画しないようにする。

楕円をもっと綺麗に描きたい（smooth と noSmooth）プログラム 1-12

```
size(400,400);
smooth(); // Turns on smoothing
ellipse(100,100,180,180);
noSmooth(); // Turn off smoothing
ellipse(300,300,180,180);
```

ellipse などで描画される図形は、外側の枠と内側の塗りつぶされる領域に分かれています。この外側の枠を示す線分の太さを変更することが出来ます。このために利用される命令が strokeWeight です。

枠線の太さを変更する

命令名（関数名）	意味
strokeWeight(width);	枠線の太さを width にする。

枠線の太さを変えたい（strokeWeight）プログラム 1-13

```
size(400,120);
smooth();
ellipse(60,60,90,90);
strokeWeight(8); // Stroke weight to 8 pixels
ellipse(180,60,90,90);
strokeWeight(20); // Stroke weight to 20 pixels
ellipse(300,60,90,90);
```

枠線は太さを変えるだけなく、表示をしないようにすることも出来ます。

この目的のためには、noStroke 命令を使用します。

枠線を表示しないようにする

命令名（関数名）	意味
noStroke();	枠線を表示しないようにする。

枠線を描かない（noStroke）プログラム 1-14

```
size(400,120);

smooth();
ellipse(60,60,90,90);
noStroke();  // without stroke
ellipse(180,60,90,90);
ellipse(300,60,90,90);
```

枠線の太さだけはなく、色を変更することも出来ます。枠線の色を変更するためには、stroke 命令を利用します。

枠線の色を変更する

命令名（関数名）	意味
stroke(x,y,z);	値 x,y,z で指定される色で枠線を描画するようになる。

Processing では枠線だけなく、図形内部の塗りつぶされる色などの変更をすることが出来ます。この目的のためには、fill,noFill 命令が使用されます。

塗り色などを変更する

命令名（関数名）	意味
fill(x,y,z);	値 x,y,z で指定される色で図形の内部を塗りつぶすようになる。
noFill();	図形の内部を塗りつぶさないようになる。

塗りつぶし色を変更したい（fill）プログラム 1-15

```
size(400,400);
smooth();
fill(255,0,0); // Red
ellipse(140,140,200,200); //  Red circle
fill(0,255,0); // Green
ellipse(200,40,200,200); // Green circle
fill(0,0,255); // Blue
ellipse(280,280,200,200); // Blue circle
```

図形を塗りつぶしたくない（noFill）プログラム 1-16

```
size(400,400);
background(255,255,255); // White
smooth();
noFill(); // Turn off filling
ellipse(140,140,200,200); //  Outline circle
fill(0,255,0); // Green
ellipse(200,40,200,200); // Green circle
fill(0,0,255); // Blue
ellipse(280,280,200,200); // Blue circle
```

描画色を変更する命令を実行すると、新たに描画色を変更する命令を実行しない限り、描画色の指定は変更されません。このような挙動をするプログラムなどは状態機械 (state machine) と呼ばれることがあります。

noFill 命令と noStroke 命令を一緒に実行すると、何も描画されなくなります。注意して下さい。

色の指定はデフォルトの RGB による方法が使用されています。

表示する図形の色だけではなく、background 命令を利用することで、背景の色を変更することが出来ます。

background という英語の単語の意味を知っていますか？

背景を指定した色で塗りつぶす

命令名（関数名）	意味
background(x,y,z);	値 x,y,z で指定される色で背景を塗りつすぶ。

背景色の変更（background）プログラム 1-17

```
size(400,400);
smooth();
background(100,100,100); // Gray
fill(255,0,0); // Red
ellipse(140,140,200,200); //  Red circle
fill(0,255,0); // Green
ellipse(200,40,200,200); // Green circle
fill(0,0,255); // Blue
ellipse(280,280,200,200); // Blue circle
```

複数しての組み合わせ プログラム 1-18

```
size(400,400);
background(255,255,255); // White
smooth();
noFill(); // Turn off filling
ellipse(140,140,200,200); //  Outline circle
fill(0,255,0); // Green
ellipse(200,40,200,200); // Green circle
fill(0,0,255); // Blue
ellipse(280,280,200,200); // Blue circle
```

複数指定の組み合わせ プログラム 1-19

```
size(400,400);
smooth();
background(100,100,100); // Gray
fill(255,0,0); // Red
ellipse(140,140,200,200); //  Red circle
noStroke();
fill(0,255,0); // Green
ellipse(200,40,200,200); // Green circle
noFill();
ellipse(280,280,200,200); // Doesn't droaw!!
```

複数指定の組み合わせ プログラム 1-20

```
size(400,400);
smooth();
background(100,100,100); // Gray
fill(255,0,0); // Red
ellipse(140,140,200,200); //  Red circle
noStroke();
fill(0,255,0); // Green
ellipse(200,40,200,200); // Green circle
noFill();
ellipse(280,280,200,200); // Doesn't droaw!!
```

描画命令を組みあせた例その 1

描画命令を組み合わせた例 1-21

```
// Learning Processing by Daniel Shifffman のサンプルを改変
size(400,400);
background(255,255,255);

// body
stroke(0,0,0);
fill(150,150,150);
rect(180,100,40,200);

// head
fill(255,255,255);
ellipse(200,140,120,120);

// eyes
fill(0,0,0);
ellipse(162,140,32,64);
ellipse(238,140,32,64);

// legs
stroke(0,0,0);
line(180,300,160,320);
line(220,300,240,320);
```

少し複雑な図形を描く

三角形 (triangle) や四角形 (rect,quad) を描く命令以外にも、多角形を描く方法が用意されています。これは描きたい多角形の頂点を vertex 命令で順番に指定していきます。どこからが描きたい多角形の頂点指定が始まっているか示すために beginShape 命令を、描きたい多角形の頂点指定の終了を示すために endShape 命令を利用します。2 つのサンプルを実行して違いを見て下さい。

vertex という英語の単語の意味を知っていますか？

便宜的に、vertex 命令、beginShape 命令、endShape 命令などと呼んでいますが、本来は、vertex 関数、beginShape 関数、endShape 関数です。

塗り色などを変更する

命令名（関数名）	意味
beginShape();	多角形の描き描きはじめを指定する。
endShape();	多角形の描き終わりを指定する。引数を CLOSE とすると、枠線を閉じて描く。
vertex(x,y);	頂点の位置を (x,y) にする。

endShape() の場合　例 1-22

```
size(400,200);
beginShape();
vertex(350,100);
vertex(290,50);
vertex(290,80);
vertex(50,80);
vertex(50,120);
vertex(290,120);
vertex(290,150);
endShape();
```

endShape(CLOSE) の場合　例 1-23

```
size(400,200);
beginShape();
vertex(350,100);
vertex(290,50);
vertex(290,80);
vertex(50,80);
vertex(50,120);
vertex(290,120);
vertex(290,150);
endShape(CLOSE);
```

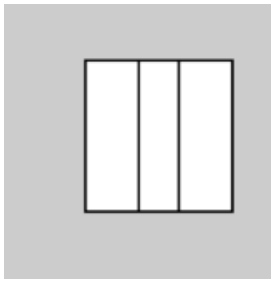
close という英語の単語の意味を知っていますか？

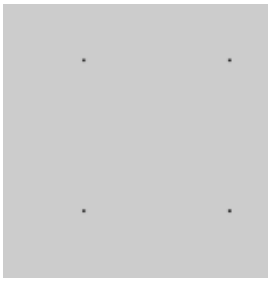
実は beginShape にも引数を指定することが出来ます。指定する引数により色々な多角形を描くことが出来ます。ここでは、Processing のマニュアルに出ている例を載せておきます。

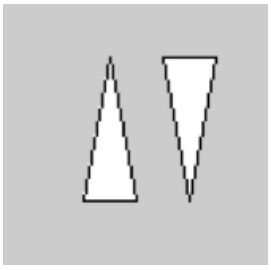
beginShape に引数指定した場合　例 1-24

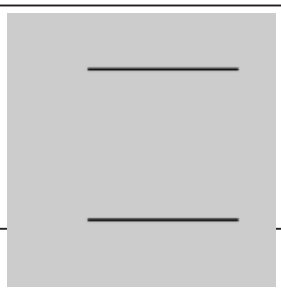
(Processging のマニュアルより)

プログラム例	描画結果

`beginShape(QUAD_STRIP);`
`vertex(30, 20);`
`vertex(30, 75);`
`vertex(50, 20);`
`vertex(50, 75);`
`vertex(65, 20);`
`vertex(65, 75);`
`vertex(85, 20);`
`vertex(85, 75);`
`endShape();`
	

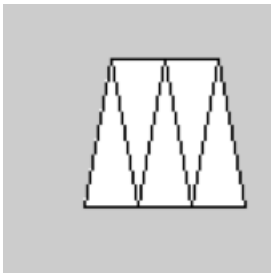
`beginShape(POINTS);`
`vertex(30, 20);`
`vertex(85, 20);`
`vertex(85, 75);`
`vertex(30, 75);`
`endShape();`
	

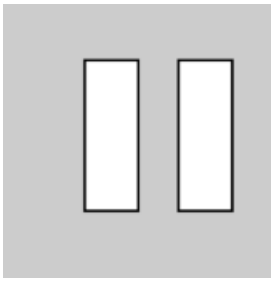
`beginShape(TRIANGLES);`
`vertex(30, 75);`
`vertex(40, 20);`
`vertex(50, 75);`
`vertex(60, 20);`
`vertex(70, 75);`
`vertex(80, 20);`
`endShape();`
	

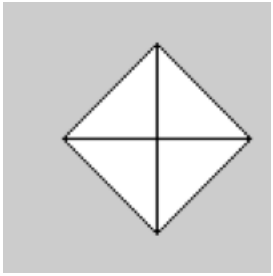
`beginShape(LINES);`
`vertex(30, 20);`
`vertex(85, 20);`
`vertex(85, 75);`
`vertex(30, 75);`
`endShape();`
	

実は、size 命令がない場合には、小さなウィンドウが開いて、描画が行われます。

プログラム例	描画結果

beginShape(TRIANGLE_STRIP);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
vertex(90, 75);
endShape();
	

beginShape(QUADS);
vertex(30, 20);
vertex(30, 75);
vertex(50, 75);
vertex(50, 20);
vertex(65, 20);
vertex(65, 75);
vertex(85, 75);
vertex(85, 20);
endShape();
	

beginShape(TRIANGLE_FAN);
vertex(57.5, 50);
vertex(57.5, 15);
vertex(92, 50);
vertex(57.5, 85);
vertex(22, 50);
vertex(57.5, 15);
endShape();
	

灰色系の色の指定

関数 fill などで色を指定する際に、白色、灰色、黒色の場合には、RGB の 3 つの値が同じ値となります。そこで、同じ数字を 3 つ並べて書くか代わりに 1 つで代用することが出来ます。

つまり、fill(255,255,255) と fill(255) は同じ意味になります。次のサンプルでは、このことを利用して色指定を行っています。

RGB を利用して色を表したときに、3 つの値が同じになるような色を無彩色と呼びます。そうでない色は有彩色と呼びます。

灰色系色の簡易指定　例 1-25

```
size(480,120);
smooth();
background(255);  // background(255,255,255);

// Left creature
fill(200);  // fill(200,200,200);
beginShape();
vertex(50,120);
vertex(100,90);
vertex(110,60);
vertex(80,20);
vertex(210,60);
vertex(160,80);
vertex(200,90);
vertex(140,100);
vertex(130,120);
endShape();
fill(0);   // fill(0,0,0);
ellipse(155,60,8,8);

// Right creature
fill(128);   // fill(128,128,128);
beginShape();
vertex(480-50,120);
vertex(480-100,90);
vertex(480-110,60);
vertex(480-80,20);
vertex(480-210,60);
vertex(480-160,80);
vertex(480-200,90);
vertex(480-140,100);
vertex(480-130,120);
endShape();
fill(0);   // fill(0,0,0);
ellipse(480-155,60,8,8);
```

曲線を描く

Processing 言語では曲線を描くことも出来ます。下に曲線を描くための関数 bezier と curve を用いた例を載せておきます。

bezier と curve のサンプル (Processging のマニュアルより)
プログラム例

```
size(100,100);
noFill();
stroke(255, 102, 0);
line(85, 20, 10, 10);
line(90, 90, 15, 80);
stroke(0, 0, 0);
bezier(85, 20, 10, 10, 90, 90, 15, 80);
size(100,100);
noFill();
stroke(255, 102, 0);
line(30, 20, 80, 5);
line(80, 75, 30, 75);
stroke(0, 0, 0);
bezier(30, 20,  80, 5,  80, 75,  30, 75);
size(100,100);
noFill();
stroke(255, 102, 0);
curve(5, 26, 5, 26, 73, 24, 73, 61);
stroke(0);
curve(5, 26, 73, 24, 73, 61, 15, 65);
stroke(255, 102, 0);
curve(73, 24, 73, 61, 15, 65, 15, 65);
```

インデント

インデント（字下げ）は、プログラミングにおいてプログラムの構造を明らかにするために、命令文の一群（コードのブロック）を字下げすることでです。

大半のプログラミング言語では、字下げは必須の事項ではありません。字下げは、自分を含むプログラマにプログラムの構造を見やすく伝えるために行います。特に、条件分岐や繰り返しといった制御構造を明確にするために利用されます。短いプログラムでは、まだ理解しづらいとおもいますが、例 1-26 と例 1-27 は同じ内容のプログラムとなっていますが、例 1-26 の方がわかりやすいプログラムになっていると思います。何文字くらいを文をずらすかにもいつかの流儀があります。通常は 4 ～ 8 文字程度をずらすようです。

どのように字下げをするかに関しては、いくつかのスタイルがあります。例えば、命令文の塊（ブロック）の開始の中括弧を制御文と同じ行に置き、ブロック内の文を字下げして表し、ブロックを閉じる中括弧を制御文と同じ字下げ位置に戻して書く（つまり、その行

Python などのプログラミング言語では、括弧やキーワードではなく字下げで構造を記述するようになっている。これをオフサイドルールと呼ぶ。これらの言語ではインデントを行うことは必須となります。

K&R スタイルと呼ばれることがあります。

は中括弧が先頭になる）というものです。このスタイルで例 1-26 を書いたものが、例 1-28 となります。

インデントあり　例 1-26

```
void setup(){
  size(640,480);
  smooth();
}

void draw(){
  if(mousePressed){
    fill(0);
  }else{
    fill(255);
  }
  ellipse(mouseX,mouseY,80,80);
}
```

インデントなし　例 1-27

```
void setup(){
size(640,480);
smooth();
}

void draw(){
if(mousePressed){
fill(0);
}else{
fill(255);
}
ellipse(mouseX,mouseY,80,80);
}
```

インデントあり（K&R スタイル）　例 1-28

```
void setup()
{
  size(640,480);
  smooth();
}

void draw()
{
  if(mousePressed){
    fill(0);
  }else{
    fill(255);
  }
  ellipse(mouseX,mouseY,80,80);
}
```

K&R スタイルに似たものでろオールマンスタイルというものがあります。このスタイルでは、例 1-29 のようになります。制御文の後の中括弧を次の行に置き、制御文と同じ字下げ位置とするもので、ブ

ロック内の文はもう一段字下げをされます。

インデントあり（オールマンスタイル） 例 1-29

```
void setup()
{
  size(640,480);
  smooth();
}

void draw()
{
  if(mousePressed)
  {
    fill(0);
  }
  else
  {
    fill(255);
  }
  ellipse(mouseX,mouseY,80,80);
}
```

インデントは自分の好きなものをスタイルで書けば良いと思います。ただ、首尾一貫して同じスタイルで書くことが重要です。

プログラムのソースコードは、自分のやりたいことをコンピュータを含めた別の人に伝えるためのコミュニケーションの手段です。しばらくすると自分で書いたプログラムでさえも、どのような動作をするものかを理解することが難しくなります。なるべくわかりやすくプログラムを書くことは、未来（明日）の自分のためです。

情報メディア基盤ユニット用資料（2016 年度第 2 章）

Processing 言語による情報メディア入門

変数、setup と draw

神奈川工科大学情報メディア学科　佐藤尚

プログラムとは？

一般には学芸会などの各種行事の進行表をプログラムと呼ぶことがあります。また、テレビ番組のことは、英語で ”TV program” と呼んでいます。つまり、プログラム（program）とは、1) あらかじめ決められている、2) ある順番やタイミングで行う処理（操作、作業）のことと考えることができます。つまり、プログラムを作るとは、自分以外のものに対して、自分が意図した動作を行うようにすることです。この ” 自分の意図した動作 ” のことがプログラムです。特に、コンピュータに対してプログラムを作成することをプログラミング（programming）と呼んでいます。

"What Most Schools Don't Teach" というビデオがあるので、是非見て下さい。URI は https://www.youtube.com/watch?v=nKIu9yen5nc です。

コンピュータは人間（プログラマ）の意図した動作を馬鹿正直に実行しようとします。ですので、プログラミングを学ぶことで、

1. 論理的な考え方
2. 論理的に情報を分析し、利用する方法
3. 正確に、なるべく早く処理をこなす工夫を考える力

などを身につけることが出来ます。また、グループで作業をする機会も多いので、他人と知識を共有する方法や議論を行い結果をまとめる力などもに見つけることが出来ます。

コンピュータのプログラミングとは？

コンピュータは、機械語と呼ばれるコンピュータに固有の命令のみを実行することが出来ます。しかし、機械語は整数値で表されている命令のため、機械語でプログラムを作成することは、非常に困難です。そこで、機械語の命令に人間がわかりやすい名前を有り当てたアセンブラを作成することが行われています。しかし、アセンブラでもプログラムを作成することはかなり面倒です。さらに、コンピュータ毎に機械語の命令は異なっているので、一つのプログラムで、別な種類のコンピュータでも動作させることの出来るプログラムを、機械語やアセンブラで記述することは絶望的に難しい（不可能な）作業です。このため、機械語やアセンブラは低水準言語 (low level language) と呼ばれることがあります。

この辺りの話は、IT 基礎で扱われる筈です。

この「わかりやすい」名前のことをニーモニック (mnemonic) と呼んでいます。

低水準言語という言葉あるということは、高水準言語と呼ばれるものも存在しています。高水準言語は、人間にとってわかりやすく命令を記述できるようになっています。また、一つのプログラムで、別な種類のコンピュータでも動作させることの出来るプログラムを

例えば、マイクロソフトが販売している Surface RT は見た目は、Window 8 です。しかし、Tegra3 というプロセッサを使っているので、一般的な Window8 用のプログラムを実行させることは出来ません。

作ることが容易となっています。高水準言語で作成した命令列のことをソースコード (source code) と呼ぶことがあります。

ソースコードを用いてコンピュータに処理を行わせるためには、通常、コンパイル（compile）またはインタープリット（interpret）という処理を行います。コンパイルは、作成したプログラムを機械語に変換し、その結果をファイルに保存します。こうして作成されたファイルは、オブジェクトコード (object code) や実行形式（executable）と呼ばれます。実行形式のファイルをコンピュータのメモリに読み込み、実行することで処理を行って行きます。なお、ソースコードを機械語に変換するプログラムのことをコンパイラ (compiler) と呼んでいます。これに対してインタープリットでは、ソースコードの命令を一つずつ読み込み、その命令に対応した処理をインタープリンタ (interpreter) というプログラムに実行させることが処理を進めて行きます。こうして出来たプログラムをコンピュータで実行させることで処理を行っています。

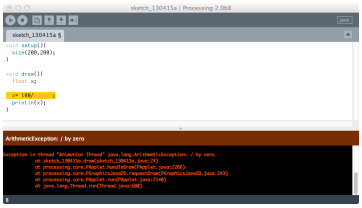
この授業で使っている Processing 言語はちょっと代わった方法を使って、ソースコードで書かれた処理を実行しています。まず、ソースコードをバイトコード (byte code) と呼ばれる仮想的な機械語に変換します。そのバイトコードを仮想機械 (virtual machine) と呼ばれるインタープリンタが読み込み、実行をしていきます。バイトコード呼ばれる命令はシンプルな命令なので、インタープリットが高速に行うことができます。この仮想機械は一般的なインタープリンタよりも単純な構造になっているので、作成が容易です。

高水準言語の例とは、C 言語、C++ 言語、Java 言語、Perl、Ruby,PHP, javascript,Common Lisp、Haskell, Prolog など色々な種類のものがあります。

バグとデバッグ

作成したプログラムが意図した通りに実行されない場合が多々あります。このようなプログラムの誤りのことをバグ (bug) と呼びます。バグを修正する作業のことをデバッグ（debug）と呼びます。バグには次の 3 つのものがあります。

1 つ目は、構文エラー (syntax error) と呼ばれるものです。ソースコードに記述される命令には、厳密な文法が決められています。構文エラーが見つかると、コンパイラやインタープリンタは処理を停止してしまいます。日常的に使っている日本語などは文法のミスや書き間違い対して、非常に寛容です。少しぐらい誤りがあっても、文の意味を取ることができます。しかし、コンパイラやインタープリンタは、この種のミスに対して、非常に不寛容です。一般的に言って、3 種類のバグの中では一番ミスを発見しやすいエラーです。
Processing 言語では、構文エラーが発生した場合には、前回のプリントで説明した場所にメッセージが表示されます。



2 つ目は、実行時エラー（run-time error）です。これは、プログラムの実行時に何らかの不都合が発生したことを意味するエラーです。例えば、0 で割り算をしたなどの場合に発生します。Processing

言語では、実行時エラーが発生した場合には、ウインドウの下部に実行時エラーの種類などに関するメッセーが表示され、実行時エラーが発生した場所がハイライトで示されます。

３つ目は、論理エラーです。論理エラーを持ったプログラムでは、構文エラーも実行時エラーも発生しません。しかし、作成したプログラムが意図通りに動作しないというものです。論理エラーの原因を突き止めることは、かなり面倒な作業です。自分の意図した動作を行わせるための手順（アルゴリズム）を考え直したり、プログラムの動作を見直したりなどの作業が必要となる場合もあります。一般的に言って、3 種類のバグの中では一番やっかいなものです。

変数

Processing 言語に限らず、コンピュータでプログラムを作るときは、変数 (variable) という考え方が出てきます。変数はコンピュータのメモリに名前をつけて、その場所に値を保存したり、読み出したりして利用します。変数には名前をつけて区別します。変数につけた名前のことを変数名と呼びます。

variable とは、どんな意味かわかりますか？

Processing 言語では、変数を使う際にどのような種類のデータを保存するのかを指定する必要があります。コンピュータの中では、全ての情報が数値（整数値）に置き換えて、記憶されています。従って、どんな種類のデータかの情報がないと、うまく情報を読み出すことや保存することが出来ません。変数にしまうデータの種類をデータ型 (data type) と呼びます。

連続量（アナログ量）を、整数値のような飛び飛びの量（デジタル量）に変換することを量子化やデジタル化と呼びます。連続的に変化している量を一定の間隔をおいて測定することを標本化やサンプリングと呼びます。この辺の詳しい話は、「情報理論とデジタル信号処理」で学習します。

Processing 言語では以下のような種類のデータ (Primitive data types) を扱うことが出来ます（一部ですが)。

Processing 言語で使用できる代表的なデータ型

データ型名	説明
boolean	true と false という 2 つの状態を表すのに使用します。
char	‘a’ や ’b’ などのような一文字（CHARacter）を表します。
String	“riho” などような文字列を表す。文字列のはじめと終わりを " で囲って表します。
int	-2147483648 から 2147483647 の範囲の整数 (INTeger) を表すときに使用します。
float	3.14159 のような実数を表すときに使用します。
PImage	jpeg や png 形式の画像情報を Processing 言語の中で利用するときに使用します。
PFont	ウィンドウに表示する文字の形状情報を保存するときに利用します。

float 型のような実数（小数点付きの数や int 型では表せない範囲の数値など）は、内部では浮動小数点形式と呼ばれる方法で数値データを記憶しています。詳しくは、IT 基礎の教科書を見て下さい。

プログラムを作る人が、自分なりのデータ型を新たに作り出すことも出来ます。

クラスと呼ばれている仕組みです。

変数の使用例その 1：同じ値の再利用

プログラムの中に同じ数字が出てくることがあります。これは偶然同じ値が出てくることもありますが、何らかの理由があって同じに値になっていることがあります。それをハッキリさせたときには、変数を使うと便利です。

Processing 言語で変数を利用する際には、どのようなデータ型のどんな名前の変数（変数名）にするかを決めて、Processing に伝える必要があります。これを変数宣言と呼んでいます。変数宣言は、次のような形になります。

変数宣言の形式
データ型　変数名 ;

変数名は、アルファベットまたは _（アンダースコアー）または $ で始まり、アルファベット、数字、_、$ を組み合わせて作られる単語です。ただし、_ で始まる変数名は特別な用途で使用されることがあります。そのため、_ で始まる _test などのような変数名は使用しないことをおすすめします。

高校で物理を勉強した人は、より

$$4.9t^2$$

の方がわかり易いですね。これ

$$\frac{1}{2}gt^2$$

も、変数 (g) の使用例です。

変数には名前とデータ型が絶対に必要です。

命令文の時と同じで、最後には ; を置きます。

単語と言っても、英単語である必要ありません。ただし、予約語と呼ばれている単語は、変数名として使用することは出来ません。例えば、int,float,string,draw,setup などの単語は予約語なので、変数名として、使用できません。簡単に言うと、予約語とは Processing 言語が使うことになっている単語です。

当然、大文字と小文字は区別しますので、test と Test は異なる変数名です。

declare の意味はわかりますか？

変数を使用プログラム例その 1 サンプル 2-01

```
size(480,120);
smooth();

int y; // Declare y as an int value
y = 60; // Assign a value to y
int d; // Declare d as an int value
d = 80; // Assign a value to d

ellipse(75,y,d,d); // Left
ellipse(175,y,d,d); // Middle
ellipse(275,y,d,d); // Right
```

一般的に、変数に値を保存するためには、= を利用して、

変数名 = 値 ;

のように書きます。変数に値を保存することを代入するということもあります。　変数の代入には、= 記号を使用しますが、数学での = とは少し役割が異なることに注意して下さい。変数の中に保存されている値を読み出す（取り出す）ためには、変数名を書けば OK です。

「代入するを」英語で言うと、assign です。

変数を使用プログラム例その 2 サンプル 2-02

```
size(480,120);
smooth();

int y; // Declare y as an int value
y = 100; // Assign a value to y
```

ちょっとわかりづらいですが、サンプル 2-01 とは、変数 y と変数 d に代入している値が異なっています。

```
int d; // Declare d as an int value
d = 130; // Assign a value to d

ellipse(75,y,d,d); // Left
ellipse(175,y,d,d); // Middle
ellipse(275,y,d,d); // Right
```

プログラムの中で、変数の値を変更することも出来ます。

変数を使用プログラム例その 3 サンプル 2-03

```
int d; // Declare d as an int value
d = 80; // Assign a value to d

ellipse(75,y,d,d); // Left
ellipse(175,y,d,d); // Middle
ellipse(275,y,d,d); // Right
y = 180;
ellipse(75,y,d,d); // Left
ellipse(175,y,d,d); // Middle
ellipse(275,y,d,d); // Right
```

ここで、変数 y の値を変えています。

サンプル 2-01 では、変数 y のデータ型は int 型として宣言をしています。従って、サンプル 2-04 のように、実数（小数点付きの数）を代入しようとすると、エラーとなります。

変数を使用プログラム例その 4 サンプル 2-04

```
size(480,120);
smooth();
int y; // Declare y as an int value
y = 60.5; // Assign a value to y
int d; // Declare d as an int value
d = 80; // Assign a value to d
ellipse(75,y,d,d); // Left
ellipse(175,y,d,d); // Middle
ellipse(275,y,d,d); // Right
```

ここで、変数 y に 60.5 という数値を代入していますが、変数 y のデータ型は int 型(整数）なので、エラーとなります。

数値の値としては、60 と 60.0 は同じ値ですが、60 は整数 (int 型)、60.0 は小数点付きの数（float 型）なので、次のような場合にもエラーとなります。

変数を使用プログラム例その 5 サンプル 2-05

```
size(480,120);
smooth();
```

```
int y; // Declare y as an int value
y = 60.0; // Assign a value to y
int d; // Declare d as an int value
d = 80; // Assign a value to d
ellipse(75,y,d,d); // Left
ellipse(175,y,d,d); // Middle
ellipse(275,y,d,d); // Right
```

ここで、変数 y に 60.0 という数値を代入していますが、数値の値としては 60 と同じなのですが、小数点がついているため、実数と判断されて、エラーとなります。

変数の使用例その 2：簡単な計算を利用

数値を保存している変数の場合には、簡単な計算式を利用することが出来ます。例えば、サンプル 2-01 を下のように変更するとプログラムの意図がよりハッキリします。

変数を使用したプログラム例その 6 サンプル 2-06

```
size(480,120);
smooth();

int y; // Declare y as an int value
y = 60; // Assign a value to y
int d; // Declare d as an int value
d = 80; // Assign a value to d
int x; // Declare x as an int value
x = 75;

ellipse(x,y,d,d); // Left
x = x+100;
ellipse(x,y,d,d); // Middle
x = x+100;
ellipse(x,y,d,d); // Right
```

真ん中の円の中心は、左の円の中心より X 軸方向に 100 だけ移動した場所に表示します。右の円の中心は、真ん中の円の中心より X 軸方向に 100 だけ移動した場所に表示します。

計算式を作る際には、次の表のような演算子が使用できます。

演算子のことを、英語では operator と呼びます。

Processing での演算子

演算子	意味	演算子	意味
+	足し算	*	かけ算
-	引き算	/	割り算
=	代入	%	剰余

割り算をしたときの、余りを求める計算のことを剰余を求める呼びます。剰余のことを英語では、modulo と呼びます。

変数を使用プログラム例その 7 サンプル 2-07

```
size(480,120);
smooth();
int y; // Declare y as an int value
y = 60; // Assign a value to y
```

```
int d; // Declare d as an int value
d = 80; // Assign a value to d
int x; // Declare x as an int value
x = 75;

ellipse(x,y,d,d);
x = x+100;
ellipse(x,y,d,d);
x = x+100;
ellipse(x,y,d,d);
x = x+100;
ellipse(x,y,d,d);
```

サンプル 2-6 より、描く円の数が増えただけです。

このサンプル 2-07 は、次のように書き換えることが出来ます。

変数を使用プログラム例その 8 サンプル 2-08

```
size(480,120);
smooth();

int y = 60; // Declare y as an int value and assign a value to y
int d = 80; // Declare d as an int value and assign a value to d
int x = 75; // Declare x as an int value and assign a value to d

ellipse(x,y,d,d);
x = x+100;
ellipse(x,y,d,d);
x = x+100;
ellipse(x,y,d,d);
x = x+100;
ellipse(x,y,d,d);
```

このサンプルのように、変数の宣言と、最初の値の代入は同時に行うことが出来ます。最初の値を代入することを「初期化する (initialize)」と呼びます。

ただし、次の使い方はエラーとなります。

```
int x; // Declare x as an int variable
int x = 12; // ERROR!! Can't two variables called x here
```

同じ名前を同時に使おうとすると、混乱しますよね。AKB の大島さんでは、大島優子さんか大島涼花さんかの区別が出来ないですよね。

もう一つ変数を利用したサンプルを示します。

変数を使用プログラム例その 9 サンプル 2-09

```
size(480,120);
int x = 25;
int h = 20;
int y = 25;
rect(x,y,300,h); // Top
x = x + 100;
rect(x,y+h,300,h); // Middle
x = x -250;
rect(x,y+2*h,300,h); // Bottom
```

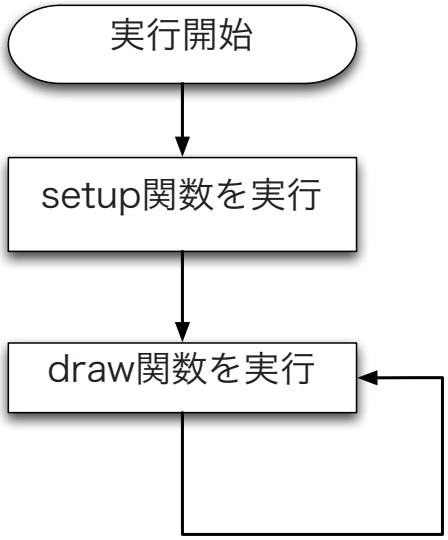
setup と draw

今までは、静止画の表示のみを行ってきましたが、動きのある画像を作り出すことも出来ます。そのために、必要となるが setup 関数と draw 関数です。

静止画を表示するだけであれば、単純に上から順番に Processing 言語の命令を実行すれば、表示が行えます。ところが、動きのある画像を表示するためには、短い時間に少しずつ異なった画像を表示する必要があります。映画では毎秒 24 枚の画像を表示していますし、テレビゲームなどでは毎秒 60 枚程度の画像を表示しています。人間は短い時間に少しずつ動いている画像を見ると、なめらかに動いているように感じます。

仮現現象と呼ばれています。

Processing 言語で動きのある画像を表示する際には、はじめに 1 度だけ実行すればよい命令と、画像を表示するために何度も繰り返し実行する必要のある命令に分けることができます。例えば、size 関数ははじめに 1 度だけ実行すれば OK です。そこで、この区別を Processing 言語に知らせる役割を担っているのが、setup 関数と draw 関数です。setup 関数と draw 関数を含んだ Processing 言語で書かれたプログラムが実行される際には、右図のような形で命令の実行が進んでいきます。



変数の初期化と同じに様に、最初に 1 回だけ実行される処理のことを初期化処理と呼びます。「初期化処理を、setup 関数で行う」というような使い方をします。

システム変数

Processing 言語にはシステム変数と呼ばれる、宣言をすることで使用できる変数が用意されています。代表的なものを表にまとめておきます。このシステム変数は値を読み出すことは出来ますが、プログラムの作成者が値を変更することは出来ません。

システム変数は、プログラムの作成者が値を変更することが出来ないので、変数と呼ぶことには、少し違和感があるかも知れません。

代表的なシステム変数

変数名	変数が保持している値の意味
width	表示しているウインドウの横幅。
height	表示しているウインドウの縦幅。
mouseX	現在のカーソル位置の X 座標の値。
mouseY	現在のカーソル位置の Y 座標の値。
pmouseX	直前のカーソル位置の X 座標の値。
pmouseY	直前のカーソル位置の Y 座標の値。
frameCount	何回目かの描画かを記録している変数。
key	どのキーが押されたかを記憶している変数。

pmouseX とか pmouseY の p は previous の頭文字の p だと思います。所で、previous の意味は大丈夫ですか？

変数名	変数が保持している値の意味
keyPressed	キーが押されているかどうかを示す変数、true か false。
mousePressed	マウスボタンが押されているかどうかを示す変数、true か false。
mouseButton	どのマウスボタンが押されているかを示す変数。RIGHT,CENTER,LEFT のどれかの値となります。

つまり、システム変数 keyPressed, mousePressed, mouseButton は boolean 型変数です。

このシステム変数を利用したプログラムのサンプルを示します。このプログラムは、ウィンドウの中央を中心とする直径がウィンドウの横幅の 4 分の 1 の円を描くプログラムです。

システム変数を使用プログラム例その 1 サンプル 2-10

```
int diameter;  // 直径

void setup(){
  size(400,400);
  smooth();
  diameter = width/4;
}
void draw(){
  background(255);
  fill(150);
  ellipse(width/2,height/2,diameter,diameter);
}
```

diameter の意味がわかりますか？

{ と } で囲まれている部分が一つの塊（ブロック）を作っています。

このプログラムはサンプル 2-11 のように書いても同じ実行結果となります。

サンプル 2-11

```
int x = 200;          // 円の中心の X 座標値
int y = 200;          // 円の中心の Y 座標値
int diameter = 100;
void setup(){
  size(400,400);
  smooth();
}
void draw(){
  background(255);
  fill(150);
  ellipse(x,y,diameter,diameter);
}
```

サンプル 2-10 とサンプル 2-11 の「size(400,400);」の数字を色々変更して、プログラムを実行してみて下さい。違いがわかりますか？

しかし、サンプル 2-10 の方が変更に強い（？）プログラムとなっています。

システム変数と setup&draw を組み合わせると、簡単な対話的（？）なプログラムを作成することが出来ます。下のサンプルは点

(mouseX,mouseY) を中心に、直径 80(=diameter) の円を描画するプログラムです。draw 関数の部分は、定期的に呼び出され、draw 関数内部に書かれている命令が実行されます。実行される度に、mouseX や mouseY の値は異なる（マウスカーソルが動いていれば）ので、その度に円が描かれる位置が変わります。そのため、動いているように見えます。

システム変数を使用プログラム例その 2 サンプル 2-12
mouseX と mouseY

ウィンドウからはみ出してしまう部分は描画されません。

```
int diameter = 80;

void setup(){
 size(400,400);
  smooth();
}

void draw(){
  background(255);
  fill(150);
  ellipse(mouseX,mouseY,diameter,diameter);
}
```

ところで、サンプル 2-12 を次の様に書き換えるとどのような動作になるでしょうか？また、なぜこのような動作になるかわかりますか？

サンプル 2-13

サンプル 2-12 とサンプル 2-13 では、「background(255);」の場所が異なっています。background はどのような動作をするかを思い出して下さい。

所で、「background(255);」は「background(255, 255, 255);」と同じ意味です。大丈夫ですね？

```
int diameter = 80;
void setup(){
  size(400,400);
  smooth();
  background(255);
}
void draw(){
  fill(150);
  ellipse(mouseX,mouseY,diameter,diameter);
}
```

もう一つ別なサンプルを示します。今度は、mouseX と mouseY だけでなく、pmouseX と pmouseY というシステム変数を利用しています。

システム変数を使用プログラム例その 3 サンプル 2-14
mouseX,mouseY,pmouseX,pmouseY

```
void setup(){
  size(400,400);
  stroke(255,0,0);
}
```

```
void draw(){
  background(255);
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

このプログラムを下のように書き換えると、どのような動作になるかわかりますね。

サンプル 2-15

```
void setup(){
  size(400,400);
  stroke(255,0,0);
  background(255);
}
void draw(){
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

物体が移動する簡単なプログラム

少しずつ表示する位置を変えながら、描画を行うことで、アニメーションを表示することが出来ます。サンプル 2-12 では、マウスカーソルを動かすことで、アニメーションのような表示を実現していました。「マウスカーソルを動かす」と同じようなことをプログラムで実現できれば良いわけです。ここでは、とても簡単なサンプルを示します。

次のサンプルでは、幾つかの円を表示するものです。

サンプル 2-16

```
int d = 20;// 円の直径
int y = 0;// 円の中心の Y 座標
int dy = 40;// 一度の移動量
size(100,200);
background(255);
fill(0,0,255);
smooth();
int x = width/2;/ 円の中心の X 座標
ellipse(x,y,d,d);
y = y+dy;
ellipse(x,y,d,d);
y = y+dy;
ellipse(x,y,d,d);
y = y+dy;
ellipse(x,y,d,d);
y = y+dy;
ellipse(x,y,d,d);
y = y+dy;
ellipse(x,y,d,d);
```

この円の移動量を小さくして、一度の一個だけ円を表示するよう

本当は、変数の有効範囲という話をしないといけないのですが、今は触れないことにしています。このことは、もう少し後で、説明します。

本当は、「int dy=40;」の直ぐ後に、「int x = width/2;」という行を持ってきたいのですが、予期した動作をしません。その理由が予想できますか？

にすれば、円が移動するアニメーションとなるはずです。そこで、setup 関数と draw 関数の組み合わせが登場します。

動く円を表示するその 1 サンプル 2-17

```
int d = 20; // 円の直径
int y = 0;  // 円の中心の Y 座標
int dy = 1; // 一度の移動量
int x;      // 円の中心の X 座標
void setup(){
  size(180,200);
  smooth();
  fill(0,0,255);
  x = width/2;
}
void draw(){
  background(255);
  ellipse(x,y,d,d);
  y = y+dy;
}
```

サンプル 2-17 を変更して、もっとゆっくり移動するようにしたものを次に示します。このサンプルでは、変数 dy に 1 より小さな正の数を指定したいので、変数 y や変数 dy のデータ型を変更しています。それ以外は、同じになっています。

動く円を表示するその 2 サンプル 2-18

```
int d = 20; // 円の直径
float y = 0;  // 円の中心の Y 座標
float dy = 0.5; // 一度の移動量
int x;      // 円の中心の X 座標

void setup(){
  size(180,200);
  smooth();
  fill(0,0,255);
  x = width/2;
}

void draw(){
  background(255);
  ellipse(x,y,d,d);
  y = y+dy;
}
```

変数値の表示

プログラムを作っていると、変数の値を調べたくなります。このような目的のために、Processing では、println という命令文

println は「print line」の略だと思います。

（関数）が用意されています。プログラム中で「println(変数名);」や「println(式);」という文を加えると、変数の値や式を計算した結果がメッセージエリアに表示されます。

「式を計算した結果」のことを、式の値を呼ぶことがあります。

println では文字を表示することも出来ます。

println の使用例 サンプル 3-13

```
void setup(){
  size(300,300);
  smooth();
  fill(51);
}
void draw(){
  background(255);
  ellipse(mouseX,mouseY,20,20);
  println(mouseX); // システム変数 mouseX の値を表示
}
```

論理エラーを持っているプログラムのデバッグを行うさいに、変数の値を表示することで、意図していない動作を起こしている場所を探すことがあります。途中の計算結果を変数に代入しておくことで、デバッグがやりやすくなる場合があります。

情報メディア基盤ユニット用資料（2016 年度第 3 章）

Processing 言語による情報メディア入門

条件分岐（if 文）

神奈川工科大学情報メディア学科　佐藤尚

条件分岐

今までのプログラムでは、並んでいる順番に命令を実行していました。このようは命令の実行の仕方を逐次処理と読んでいます。コンピュータのプログラムの実行の仕方には、この逐次処理を含めて、下の 3 つのものがあります。

1. 逐次処理
2. 条件分岐処理
3. 繰り返し処理

条件分岐処理と繰り返し処理が、コンピュータのプログラムを強く特徴づける処理となっています。これにより、大量のデータ処理や一見すると複雑な処理が実行できます。ドラクエや Watson のような複雑なシステムも、原理的にはこの 3 つの処理を組み合わせて出来ています。

日常でも、「もしお腹がすいたら何かを食べる、そうでなくもしのどが渇いていたら水を飲む、そうでなければ昼寝をする」(If I am hungry then eat some food, otherwise if I am thirsty, drink some water, otherwise, take a nap) などのような使い方をすることがあります。これに類似したものが条件分岐処理です。つまり、ある条件が満たされているときに実行する処理を指定するのが条件分岐処理の基本的な考え方です。

ここで述べる、条件分岐処理の表し方は、Processing 言語だけでなく、C 言語系の言語ではほぼ共通の書き方（構文）になっています。

C 言語系の例

- C++
- Java
- C#
- JavaScript

Watson って、知っていますか？ IBM という会社が作ったシステムなのです。どのようなものか調べてみて下さい。

Processing 言語などのプログラミング言語では、英語が命令文やその組み合わせ方に大きな影響を与えています。英語ネイティブな人の方が有利？

条件式（論理式）

Processing 言語をはじめとして、多くのプログラミング言語では、条件分岐において条件式や論理式と呼ばれる考え方が出てきます。つまり、ある条件の時に、ある処理を行ったり、行わなかったりするので、その条件を指定する必要が出てきます。この条件を指定するに利用されるものが条件式です。条件式や論理式と言うと難しく感じるかも知れませんが、簡単にいうと、数学で出ている不等式のようなものです。

条件式の一番単純なものは、次の表に挙げるようなものです。

条件式に出てくる演算子

記号	意味	使用例	使用例の意味
>	大きい	mouseX > 100	mouseX の値が 100 より大きい
<	小さい	mouseY < 200	mouseY の値が 200 より小さい

この辺は、数学の不等式と同じです。

= は代入として利用しているので、== を代わりに使っています。

記号	意味	使用例	使用例の意味
==	等しい 同じ	mouseButton == LEFT	mouseButton の値が LEFT と等しい
>=	以上	mouseY >= 100	mouseY の値が 100 以上
<=	以下	ｍｏｕｓｅＸ < width/2	mouseX の値が width/2 以下
!=	等しくない 異なっている 同じではない	mousePressed != true	mousePressed の値が true と異なっている

≧、≦や≠などの記号が使えないので、複数の記号を組み合わせて使っています。「以上」は「大きい」または「等しい」のどちらかですよね。

この表に示した条件式は、変数の値によって、正しい (true) か間違っている (false) かが決まります。そのため、条件式のことを論理式と呼ぶこともあります。

日本語では、true は真（正しい）、false は偽（正しくない）という意味です。true や false という値を利用したことが多いので、Processing 限では、Boolean 型というデータが用意されています。

これだけでは、複雑な条件を表すことが出来ないので、単純な条件式を組み合わせて複雑な条件式や論理式を作り出すことで、複雑な条件判定を行います。単純な条件式（論理式）を組み合わせる糊のような役目をするものが論理演算子と呼ばれるものです。ちょっと堅い言い方も知れませんが、日常でのも、「または」とか「かつ」とか言って、少し複雑な条件を表現することがありますよね。この「または」や「かつに」にあたるものが、論理演算子です。

単純なものから、順々に複雑なものを作り出すことを構成的方法と呼びます。

英語では、「または」は or、「かつ」は and です。

この辺の堅い表現は、履修要項などに出ています。

複雑な条件判定を作る演算子

記号	式	意味	使用例
&&	P && Q	かつ and	(mousePressed == true) && (mouseButton == RIGHT)
||	P || Q	または or	(mouseX < width/2) || (mouseY < height/2)
!	! P	否定 ではない not	!((mouseX < width/2) || (mouseY < height/2))

なれるまでは、複雑な論理式を構成する一つ一つの式を括弧で括った方がわかりやすいと思います。
(mouseX < width/2) || (mouseY < height/2)
と
mouseX < width/2 || mouseY < height/2
を比べてみて下さい。

論理式の例

x == 6	x != y
x < 7	x > 8
x <= 10	y >= x
(7 < x) && (x < 20)	(x == 6) || (y == 8)
!(x < 6)	!((x == y) && (y > 8))

Processing 言語では、7 < x <20 のような不等式を利用することが出来ないので、&& 演算子を使用して、
(7<x) && (x < 20)
のような書き方をします。

論理式の便利な性質

数式と言うと難しく感じるかもしれませんが、数式にすると機械的な計算で、色々なことをわかることがあります。論理式にも知っていると便利な性質があります。

機械的な計算で色々なことをがわかるのが、数式（数学）を利用する、大きなメリットです。

例えば、数学的には否定の否定はもとの値に戻ります。つまり、P を論理式とすると、「!(!P)」は P と同じ値になります。このような性

質は次の表のようにまとめることが出来ます。表の中では、P や Q は論理式とします。

論理式の持っている色々な性質は数理論理学で取り扱われます。

論理式の持っている性質

元の論理式	値の等しい論理式
P && true	P
P && false	false
P || true	true
P || false	P
!(!P)	P
P == true	P
P == false	!P
!(P || Q)	(!P) && (!Q)
!(P && Q)	(!P) || (!Q)

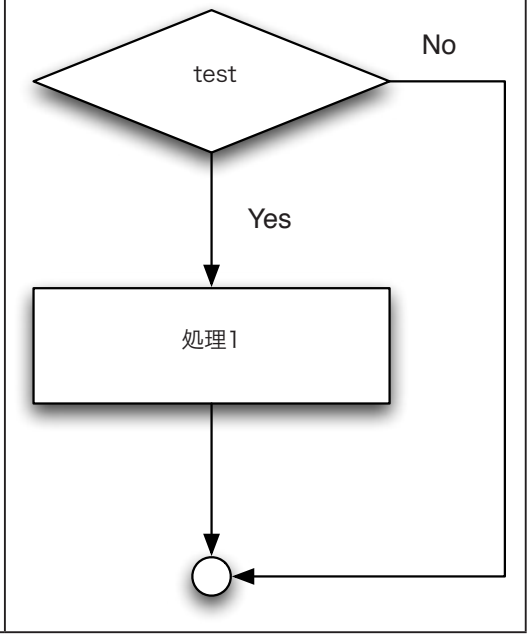
表の最後の 2 つのことをドモルガンの法則と呼びます。

一番簡単な条件分岐処理

条件分岐に if という英単語を利用することは、英語圏の人に取っては、自然ですよね。

一番簡単な条件処理は、指定された条件が正しいときに実行する命令を指定するものです。Processing 言語では、条件式を test としたときに、以下のような if という単語を利用して表現されます。

Processing 言語での表現	処理の流れ

if(test){
処理 1
}
	

右の図では、ひし形の部分で条件式部分を表し、長方形の部分で処理する命令文があることを表しています。このような図のことをフローチャート (flow char) や流れ図と呼んでいます。
処理の内容を理解するために、使用されることがあります。

{ と } で囲まれている部分が一つの塊（ブロック）を作っています。

サンプル 3-1 は、マウスボタンが押されているときに、黒色（fill(0)）の円を描画する（ellipse(mouseX,mouseY,20,20)）ものです。

条件分岐処理 (if のみ) サンプル 3-1

```
void setup(){
  size(400,400);
  smooth();
}
void draw(){
  background(255);
  if(mousePressed == true){
    fill(0);
    ellipse(mouseX,mouseY,20,20);
  }
}
```

サンプル 3-1 の中では、“if(test){” の test の部分には、条件式と呼ばれるものが書かれます。サンプルプログラム 1 では、” mousePressed == true” という条件式になっています。この条件式の意味は、mousePressed という変数の値が true と等しい（==）というものです。

サンプル 3-2 もプログラム 3-1 と同じような構造のプログラムです。プログラムを見て、どのような動作のプログラムかわかりますか？打ち込んで、実行して見ましょう。

条件分岐処理 (if のみ) サンプル 3-2

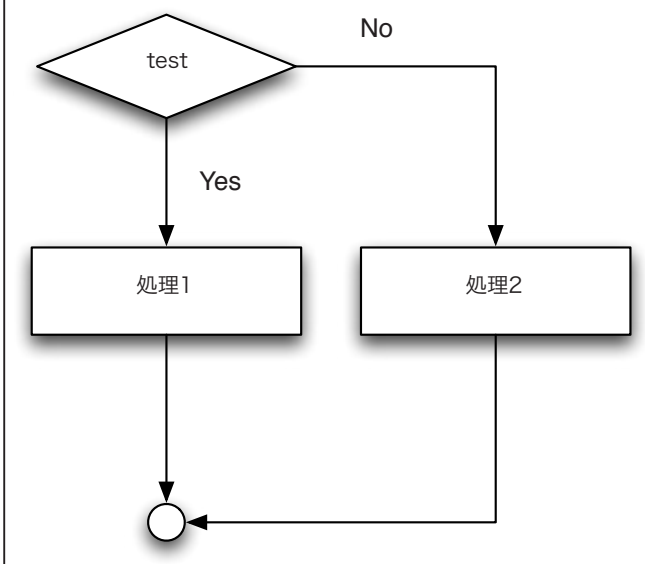
```
void setup(){
  size(400,400);
}

void draw(){
  background(255);
  if(mouseX < width/2){
    fill(0);
    rect(0,0,width/2,height);
  }
}
```

もう少し複雑な条件分岐処理 (A or B)

もう少し複雑な条件分岐処理は、指定された条件が正しいときに実行する命令と、その条件が正しくないときに実行する命令を指定するものです。Processing 言語では、このような処理は if と else という英単語を利用した文として表現されます。

システム変数 mousePressed は、マウスボタンが押されている時には値が true となり、そうでないときには値が false となります。つまり、データ型が Boolean となっている変数です。このような Boolean 型の変数を論理（型）変数と呼ぶことがあります。

Processing 言語での表現	処理の流れ

if(test){
処理 1
}else{
処理 2
}
	

右の図では、条件式 test が true の時を Yes、false の時を No で表しています。

サンプル 3-3 は、マウスボタンが押されているときには黒色（fill(0)）を、そうでない場合には灰色 (fill(170)) の円を描画する（ellipse(mouseX,mouseY,20,20)）ものです。

条件分岐処理 (if と else) サンプル 3-3

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  if(mousePressed == true){
    fill(0);
  }else{
    fill(170);
  }
  ellipse(mouseX,mouseY,20,20);
}
```

サンプル 3-4 もサンプル 3-3 と同じような構造のプログラムです。プログラムを見て、どのような動作のプログラムかわかりますか？打ち込んで、実行して見ましょう。このサンプルでは、条件分岐のための条件が「mouseX<width/2」となっています。この条件はどのような条件になっているか、わかりますか？これが OK なら、このプログラムの動作の理解も簡単だと思います。

条件分岐処理 (if と else) サンプル 3-4

```
void setup(){
  size(400,400);
}

void draw(){
  background(255);
  fill(0);
  if(mouseX < width/2){
    rect(0,0,width/2,height);
  }else{
    rect(width/2,0,width/2,height);
  }
}
```

サンプル 3-1 やサンプル 3-3 での条件式は、「mousePressed == true」となっています。システム変数 mousePressed は true か false のどちらかの値をとる論理変数です。従って、「論理式の持っている性質」を使うと、mousePressed だけで良いことなります。従って、サンプル 3-3 は次の様に書き換えることが出来ます。

論理式は true か false のどちらかに値が決まればよいので、mousePressed のように論理変数単体でも、論理式となります。

条件分岐処理 (if と else) サンプル 3-3'

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  if(mousePressed){
    fill(0);
  }else{
    fill(170);
  }
  ellipse(mouseX,mouseY,20,20);
}
```

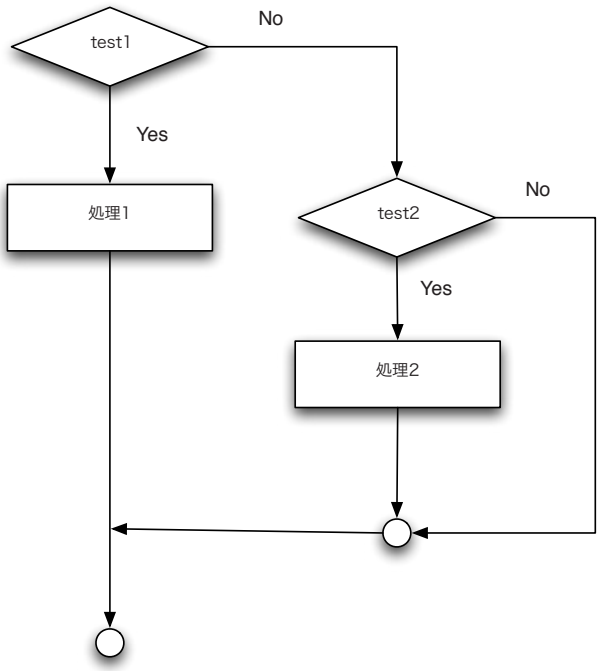
「mousePressed == true」 と書くか、単に「mousePressed」と書くかは、どちらでも良いと思います。個人的は、シンプルな後者の方が好きです。

条件分岐処理の重ね (その 1)

条件分岐処理は、重ねて処理をすることが出来ます。つまり、条件を順番に試していき、実行すべき命令を決めることが出来ます。

このような処理は、次に示す、if と else if という英単語の組み合わせで表現します。

Processin 言語の表現	処理の流れ

if(test1){
処理 1
}else if(test2){
処理 2
}
	

サンプル 3-5 は、マウスボタンが押されていない時 (mousePressed == false) には、中を塗りつぶさない（noFill()）で円を描画 (ellipse(width/2,height/2,100,100)) します。そうでない時（マウスボタンが押されていない時、つまりつまりマウスボタンが押されている時）に、押されているボタンが右ボタン（mouseButton == RIGHT）であれば、赤色 (fill(250,20,20)) の円を描画 (ellipse(width/2,height/2,100,100)) します。ところで、サンプル 3-5 を実行しているときに、マウスの左ボタンを押したら、どのような表示になるかわかりますか？

条件分岐処理 (if と else if) サンプル 3-5

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  if(mousePressed == false){
    noFill();
    ellipse(width/2,height/2,100,100);
  }else if(mouseButton == RIGHT){
    fill(250,20,20);
    ellipse(width/2,height/2,100,100);
  }
}
```

if の中に再び if が出てきます。ロシア人形のマトリョーシカのように、あるものの中に同じようなものがはいっていることを「入れ子構造」と呼んでいます。

ところで、落語に頭山という演目があります。どんなものか知っていますか？この頭山は、アニメーション作家山村浩二氏によって、アニメーション化されています。ニコニコ動画 (http://www.nicovideo.jp/watch/sm2144630) や youtube (http://www.youtube.com/watch?v=UuM8xHQSUEM) で、見ることが出来るようです。

マトリョーシカ人形

システム変数 mouseButton は、LEFT、CENTER、RIGHT のどれかの値を取ります。どのマウスボタンが押されているかによって、値が変わります。

サンプル 3-6 もサンプル 3-5 と同じような構造のプログラムです。プログラムを見て、どのような動作のプログラムかわかりますか？打ち込んで、実行して見ましょう。

条件分岐処理 (if と else if) サンプル 3-6

```
void setup(){
  size(300,300);
}

void draw(){
  background(255);
  line(width/3,0,width/3,height);
  line(2*width/3,0,2*width/3,height);
  fill(0);
  if(mouseX < width/3){
    rect(0,0,width/3,height);
  }else if(mouseX > 2*width/3){
    rect(2*width/3,0,width/3,height);
  }
}
```

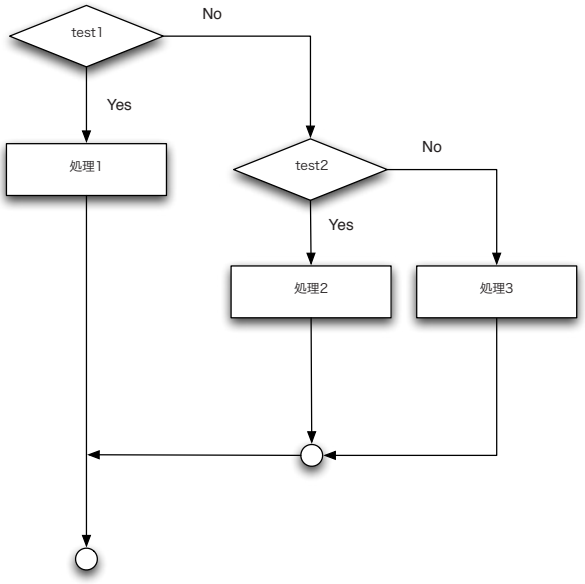
このサンプルのように、マウスカーソルの位置に応じて、表示を行うことをロールオーバー処理と呼びます。

条件分岐処理の重ね (その 2)

条件分岐処理は、もっと重ねて処理をすることが出来ます。基本的には好きなだけ if 〜 else という構造をつなげていき、沢山の条件を試していくことが出来ます。ここでは、2 つの条件式を持つ場合を示します。始めに条件式 test1 の条件が成立しているどうかを調べ

条件式の値を true となる場合は、条件が成立していると言うことがあります。false となる場合には、条件が不成立と呼ぶこともあります。

Processing 言語での表現	処理の流れ

if(test1){
処理 1
}else if(test2){
処理 2
}else{
処理 3
}
	

サンプル 3-7 は、この条件分岐処理の重ねの例題です。

条件分岐処理 (if と else if の多段の重ね) サンプル 3-7

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  if(mousePressed == false){
    noFill();
  }else if(mouseButton == LEFT){
    fill(250,20,20);
  }else{
    fill(20,250,20);
  }
  ellipse(width/2,height/2,100,100);
}
```

サンプル 3-7 は、マウスボタンが押されていない時 (mousePressed == false) には、中を塗りつぶさない（noFill()）で、そうでない時（マウスボタンが押されていない時、つまりつまりマウスボタンが押されている時）に、押されているボタンが左ボタン（mouseButton == LEFT）であれば赤色 (fill(250,20,20))、そうでないときには緑色 (fill(20,250,20)) で、円を描画 (ellipse(width/2,height/2,100,100)) します。

サンプル 3-8 もサンプル 3-7 と同じような構造のプログラムです。プログラムを見て、どのような動作のプログラムかわかりますか？打ち込んで、実行して見ましょう。

条件が複雑になってくると、日本語で説明することが、段々面倒になってきます。そのために、Processing 言語などのプログラミング言語やフローチャートのような処理の流れを図式化（可視化）を利用して表現（記述）することが重要になってきます。

条件分岐処理 (if と else if の多段の重ね) サンプル 3-8

```
int xLeft;
void setup(){
  size(300,300);
}

void draw(){
  background(255);
  line(width/3,0,width/3,height);
  line(2*width/3,0,2*width/3,height);
  fill(0);
  if(mouseX < width/3){
    xLeft = 0;
  }else if(mouseX < 2*width/3){
    xLeft = width/3;
  }else{
    xLeft = 2*width/3;
  }
  rect(xLeft,0,width/3,height);
}
```

サンプル 3-7 とサンプル 3-8 は、2 つの条件を重ねたものです。これは 2 つの条件に限らず、もっと増やすことが出来ます。ここでは、もう 1 つ条件を加えた､同じような動作をするプサンプルを示します。

条件分岐処理 (if と else if の多段の重ね) サンプル 3-9

```
int xLeft;
int w4; // width の 4 分の 1 の値を保存する

void setup(){
  size(300,300);
  w4 = width/4;
}

void draw(){
  background(255);
  line(w4,0,w4,height);
  line(2*w4,0,2*w4,height);
  line(3*w4,0,3*w4,height);
  fill(0);
  if(mouseX < w4){
    xLeft = 0;
  }else if(mouseX < 2*w4){
    xLeft = w4;
  }else if(mouseX < 3*w4){
    xLeft = 2*w4;
  }else{
    xLeft = 3*w4;
  }
  rect(xLeft,0,w4,height);
}
```

width/4 という式が沢山出てくるので、int 型変数 w4 に width/4 の値を保存しています。

今までに説明したことを利用して作ったサンプルを示します。このプログラムでやっていることは、

setup での処理

1. ウインドウを表示する

draw での処理

1. 背景を白にする
2. 真ん中に十字上に線を表示する
3. もしマウスカーソルが左上のコーナーにいれば、そのコーナーに黒色の矩形を表示する。
4. もしマウスカーソルが右上のコーナーにいれば、そのコーナーに黒色の矩形を表示する。
5. もしマウスカーソルが左下のコーナーにいれば、そのコーナーに黒色の矩形を表示する。
6. もしマウスカーソルが右下のコーナーにいれば、そのコーナーに黒色の矩形を表示する。

条件分岐処理 (if と else if の多段の重ね) サンプル 3-10

```
void setup(){
  size(400,400);
}

void draw(){
  background(255);
  line(width/2,0,width/2,height);
  line(0,height/2,width,height/2);
  fill(0);
  if(mouseX < width/2 && mouseY < height/2){
    rect(0,0,width/2,height/2);
  }else if(mouseX >= width/2 && mouseY < height/2){
    rect(width/2,0,width/2,height/2);
  }else if(mouseX < width/2 && mouseY >= height/2){
    rect(0,height/2,width/2,height/2);
  }else{
    rect(width/2,height/2,width/2,height/2);
  }
}
```

少し意味のあるサンプルその 1

FF などのゲームで世界地図上を移動する際に、一番下まで移動すると一番真上に現れます。次のサンプル 3-11 は、これと同じように、移動する直線が一番下まで移動したら、その次は一番上から出てくるようにするプログラムです。X 軸に平行な移動する直線が一番下に到達したら、この直線を一番上に移動させています。int 型変数 y が、直線を描く高さ（Y 座標の値）を表しています。

条件分岐処理を利用した直線の移動 サンプル 3-11

```
int y;

void setup(){
  size(100,300);
  y = 0;
}

void draw(){
  background(255);
  stroke(0);
  line(0,y,width,y);
  y = y+1;
  if(y >= height){
    y = 0;
  }
}
```

少し意味のあるサンプルその 2

条件分岐処理を利用すると、壁に円がぶつかったときに跳ね返るような処理を作り出すことが出来ます。これを実現したプログラムが次のサンプルプログラムです。

条件分岐処理を利用した壁での反射サンプル 3-12

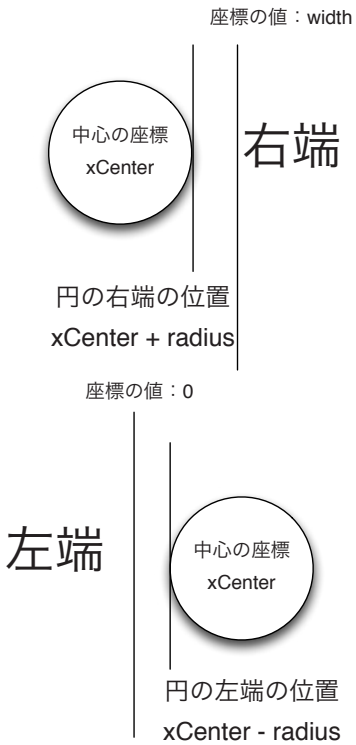
```
int xCenter; // 円の中心の X 座標
int radius;  // 円の半径
int speed;   // 円の移動速度。正の値の時は、左から右に移動する。
             // 負の時は、右から左に移動する。

void setup(){
  size(400,200);
  smooth();
  xCenter = width/2;
  radius = 20;
  speed = 1;
}

void draw(){
  background(255);
  fill(170);
  ellipse(xCenter,height/2,2*radius,2*radius);
  xCenter = xCenter + speed;
  if(((xCenter + radius) >= width) || ((xCenter-radius) < 0)){
    speed = -speed;
  }
}
```

移動する円は、左端と右端に到達したときに移動方向を変えれば、両端の壁にぶつかって移動する動作を再現出来ます。円が左端に到達した場合には、円の一番左側の位置が 0 よりも小さくなっているはずです。また、円が右端に到達した場合には、円の一番右側の位置が width 以上になっています。この 2 つの条件のどちらかが成立している時に、移動方向を変更します。移動方向を正反対にするためには、移動速度に -1 をかければ OK です。

円が両端に到達した時の移動方向の変更の処理が難しいと感じる人は、サンプル 3-12' のように明示的に変数 speed の値を変更した方が理解しやすいかも知れません。

条件分岐処理を利用した壁での反射サンプル 3-12'

このプログラムを複雑にしていくと、Pong のようなゲームを作ることが出来ます。上手くゲームをデザインすれば、今までの知識でもゲームを作ることが出来ます。



円板が壁にあたる時

速度がベクトルで表されている場合は、移動方向を正反対にするためには、-1 をかければ求めることが出来ます。

このサンプル 3-12' では、左端でぶつかるか、右端でぶつかるかで処理を変えています。

```
int xCenter; // 円の中心の X 座標
int radius;  // 円の半径
int speed;   // 円の移動速度。正の値の時は、左から右に移動する。
             // 負の時は、右から左に移動する。

void setup(){
  size(400,200);
  smooth();
  xCenter = width/2;
  radius = 20;
  speed = 1;
}

void draw(){
  background(255);
  fill(170);
  ellipse(xCenter,height/2,2*radius,2*radius);
  xCenter = xCenter + speed;
  if((xCenter + radius) >= width){   // 左端で衝突
    speed = -1;
  }else if((xCenter-radius) < 0){    // 右端で衝突
    speed = 1;
  }
}
```

変数の値の変更

サンプル 3-11 やサンプル 3-12 などでは、「y=y+1;」や「xCenter = xCenter + speed;」などのように、右辺で計算した値を左辺の変数に代入する形の式が出てきます。これらの式でやりたいことをよく考えると、「変数の y の値を 1 増やす」や「変数 xCenter の値を speed だけ増やす」などになります。このような、「変数の値を増やす（減らす）」などの処理は、プログラム中で良く使用される式となっています。このような、意図をハッキリさせるために、Processing 言語では特別な式の書き方が用意されています。これを、次の表にまとめておきます。

「意図」は、人間にとっても、コンピュータ側にとっても、有効な情報になっています。

C 言語系のプログラミング言語でも同じ機能が提供されています。

複合代入演算子、増分演算子、減分演算子

普通の記法	意図を発揮させた記法	記法の使用例	
変数 = 変数 + 値 ;	変数 += 値 ;	x = x+3;	x += 3;
変数 = 変数 * 値 ;	変数 *= 値 ;	x = x*5;	x *= 5;
変数 = 変数 - 値 ;	変数 -= 値 ;	x = x-2;	x -= 2;
変数 = 変数 / 値 ;	変数 /= 値 ;	x = x / 7;	x /= 7;
変数 = 変数 % 値 ;	変数 %= 値 ;	x = x % 5;	x %= 5;
変数 = 変数 + 1;	変数 ++;	x = x+1;	x++;
変数 = 変数 + 1;	++ 変数 ;	x = x+1;	++x;
変数 = 変数 -1;	変数 --;	x = x-1;	x--;
変数 = 変数 -1;	-- 変数 ;	x = x-1;	--x;

この表の上の 5 つは複合代入演算子と呼ばれています。

++ は増分演算子、-- は減分演算子と呼ばれています。正確には、++ 変数を前置増分演算子、変数 ++ を後置増分演算子と呼んで区別をしています。この 2 つは良く似ていますが、少しだけ挙動が異なります。-- も同じです。

変数値の表示

プログラムを作っていると、変数の値を調べたくなります。このような目的のために、Processing では、println という命令文（関数）が用意されています。プログラム中で「println(変数名);」や「println(式);」や「println(文字列);」という文を加えると、変数の値は式を計算した結果や引数内の文字列がメッセージエリアに表示されます。

println は「print line」 の 略だと思います。

「式を計算した結果」のことを、式の値を呼ぶことがあります。

println の使用例 サンプル 3-13

```
void setup(){
  size(300,300);
  smooth();
  fill(51);
}

void draw(){
  background(255);
  ellipse(mouseX,mouseY,20,20);
  println(mouseX); // システム変数 mouseX の値を表示
}
```

プログラミングにおいて重要な考え方

プログラムを作るという立場から見ると、主に、この授業では次の 4 つのことが重要となります。

変数：数値や文字列などのデータの記憶とそれを取り出す仕組み

条件分岐：条件により処理内容を変更する仕組み

繰り返し：指定された回数または指定された状態の間処理を繰り返す仕組み

関数：一塊の処理をひとまとめにして、管理するための仕組み

前回の授業では、変数に関して学びました。今回の授業では条件分岐の仕組みを学びました。次回の講義では、繰り返しの仕組みについて学びます。

情報メディア基盤ユニット用資料（2016 年度第 4 章）

Processing 言語による情報メディア入門

繰り返し処理その 1（for 文）

神奈川工科大学情報メディア学科　佐藤尚

乱数

ゲームなどを作成する際には、敵キャラの出現位置をデタラメに決めたいことがあります。これを実現するためには、敵キャラの出現位置を決める座標値をデタラメに設定すれば可能です。ゲームなどを作る際には、デタラメな値が必要となることがあります。これを実現するために、乱数という仕組みが Processing 言語などのプログラミング言語では用意されています。Processing 言語の場合には、random 関数を使用します。この関数は呼び出される度に、デタラメな数値を返します。

真面目な科学技術計算や金融関係の計算でも、乱数は利用されます。そのため、乱数を作り出す方法について、沢山の方法が知られています。

random の意味はわかりますか？

乱数を返す random 関数

使い方	意味
random(high)	0 以上 high 未満の乱数を返す
random(low,high)	low 以上 high 未満の乱数を返す

サンプル 4-1 は random 関数を使って、円を描く場所を決めているので、実行する度に異なった場所に円が描かれます。

乱数を使ったサンプル 4-1

```
size(400,200);
smooth();
// デタラメな場所に円を描く
ellipse(random(width),random(height),20,20);
```

random(1) とすると、0 以上 1 未満のデタラメな数が返されます。このため、random 関数では float 型の値が返されます。つまり、次のプログラムはエラーとなってしまいます。

このような値を戻り値 (return value) と呼びます。

乱数を使ったサンプル 4-2

```
size(400,200);
smooth();
// デタラメな場所に円を描く
int x = random(width);
int y = random(height);
ellipse(x,y,20,20);
```

エラーメッセージは「cannot convert float to int」となっているはずです。これは、float 型の値を int 型の値には変更出来ない (cannot convert) という意味です。

そこで、整数の乱数値 (int 型の乱数値) が必要となる場合には、「int(random(10))」などとします。この「int(…)」は、強制的に整数値 (int 型) の値に変更する関数です。

乱数を使ったサンプル 4-3

```
size(400,200);
smooth();
// デタラメな場所に円を描く
int x = int(random(width)); //random(width) の値を int 型に変換
int y = int(random(height));//random(height) の値を int 型に変換
ellipse(x,y,20,20);
```

このように強制的にデータ型を変える関数としては、以下のようなものがよく使われます。

データ型を変える関数

使い方	意味
int(value)	値 value を強制的に int 型の値に変更する
float(value)	値 value を強制的に float 型の値に変更する
str(value)	値 value を強制的に String 型に変更する

強制的にデータの型を変更することを明示的型変換（キャスト変換）と呼びます。一方、データ格納領域がより広い型への変換は自動的に行われます。これを、暗黙的な型変換と呼んでいます。char < int <float の順にデータ格納領域が広くなっているので、char 型の値を int 型や float 型の変数に代入するや int 型の値を float 型の変数に代入するなどの処理においては、暗黙の型変換が行われます。

最後に、もう一つ乱数を利用したサンプルを挙げておきます。

乱数を使ったサンプル 4-4

```
void setup(){
  size(400,200); // 400X200 のウインドウを表示
  smooth();      // アンチエリアシングをして図形を描画
  background(0); // 背景を黒で塗りつぶす
}

void draw(){
  fill(random(100,256));  // 塗りつぶし色を乱数で設定
   ellipse(random(width),random(height),20,20);  // 直径 20 の円を乱数で決めた場所に表示
}
```

関数

今までの説明の中では、random や ellipse などを命令文や関数と呼んできました。ellipse 命令を使って楕円を描くためには、コンピュータ内部では沢山の処理が行われています。多くのプログラミング言語では、まとまった処理に名前をつける機能が用意されています。Processing 言語では、このような仕組みのことを関数 (function) もしくはメソッド (method) と呼んでいます。random や ellipse のように Processing 言語がどのような処理を行うべきかを最初から知っているものは、組み込み関数 (built-in function) や組み込みメソッド (built-in method) と呼ばれます。Processing には、組み込み関数が用意されています。

関数やメソッドには、関数名やメソッド名と呼ばれる名前がついています。関数名やメソッド名として使える名前の付け方は、変数名の付け方と同じです。

関数とメソッドの違いは別な機会に紹介します。今まで出てきた「命令文」は、関数と呼ばれるものです。

ですから正確には、random 関数や elipse 関数と呼ぶべきものです。

関数名や変数名のことを識別子 (identifier) と呼びます。

正確には、この引数を実引数と呼びます。ということは、仮引数 (parameter) と呼ばれるものもあります。別の機会に説明します。

256 などのように数値を表した表現をリテラル (literal) または数値リテラルと呼びます。

関数が知っている処理内容を実行させることを、関数を呼び出す (call) と言うことがあります。関数を呼び出す場合には、何らかの付加的な情報をつけて呼び出す場合と、付加的な情報をつけることなく呼び出す場合の 2 通りがあります。この付加的な情報を引数 (argument) と呼びます。関数を呼び出す場合には、次のように記述します。

通常、引数の部分には値が置かれます。値 (value) とは、数値、文字列、true や false などのことです。簡単に言ってしまうと、Processing のプログラム中で扱う全てのデータが値です。また、式 (expression) は計算することで値を得ることの出来るものです。簡単に言うと、引数には数値、式、変数名などが使われます。「random(10)」なども値となります。

数学で出てくるような式と if 文などで使用される条件式も式です。

関数呼び出し

引数の数	関数の呼び出し方	例
引数が無い場合	関数名 ()	smooth()
引数が 1 つの場合	関数名 (引数 1)	random(width/2)
引数が 2 つの場合	関数名 (引数 1, 引数 2)	size(400,400)
引数が 3 つの場合	関数名 (引数 1, 引数 2, 引数 3)	fill(10,20,30)
全てまとめて書くと	関数名 ([引数 1[, 引数 2…[, 引数 3…]])	

「全てのまとめて書くと」の [と] は実際に書くわけではありません。カギ括弧 [ ] は省略可能を示す、コンピュータ業界では、一般的な書き方です。

関数を利用しなくても、原理的にはプログラムを作成することが出来ます。それでは、なぜ関数を利用するのでしょうか？大雑把に言うと、2 つの理由（可読性の向上、再利用）があります。

長い行数のプログラムを作ると、全体の処理の流れを理解することが難しくなっていきます。しかし、処理の塊が一定の意味をもった処理を行っていることがあります。そこで、この意味をもった処理の塊の部分を取り出して、名前をつけます。これが関数です。

関数を使ってプログラムを作ると言うことは、部品を組み合わせて、ものを作ることに似ています。料理などでも、以前は自分で野菜を切ったりして下準備をするのが当たり前でしたが、最近ではカット野菜を利用することが多くなってきています。また、プログラムの中で、同じような処理が繰り返し行われていることがあります。他の人が作成した関数を利用したり、自分で処理の塊に名前をつけて関数を作ったりして、その関数を呼び出すことで、プログラムを作って行くことができます。また、ある処理の塊に名前をつけているので、ある不都合が起きたときには、まずはその不具合を起こしそうな関数を調べるというデバッグ作業を開始することが出来ます。CD プレイやで再生が上手く行かないときには、CD が汚れていないか、ピックアップが汚れていないかなど、チェックをして行くことができます。もし、CD と CD プレイーが一体のものとして作られていたら、こんなことは出来ません。

ガルパンで、大洗女子学園のチームのメンバの名前を全て覚えることは難しいです。でも、乗っている戦車のチーム毎に把握すれば、何とかなりますよね。指揮をしているときに、いちいちメンバの名前を呼んでいられません。ウサギさんチームのメンバ全員の名前を呼んでいられません。車長の澤さんに命令を出せが、彼女が他のウサギさんチームのメンバに必要な指示を出してくれます。

回数指定型繰り返し処理（その 1）

Processing 言語をはじめとして、多くのプログラミング言語では、命令の実行に関しては、以下の 3 つものがあります。

1. 逐次処理
2. 条件分岐処理
3. 繰り返し処理

前回の授業では条件分岐処理を紹介しました。今回は繰り返し処理を紹介します。繰り返し処理は少ないプログラムの記述量で沢山の命令を実行させようとするものです。

次のサンプルプログラムは乱数を使ってデタラメな場所に 10 個の円を表示するものです。ちょっと長いので 3 列に分けてプログラムを書いてあります。

円を 10 個描く サンプル 4-5

```
float x,y;
size(400,200);
smooth();
background(150);
fill(255);

x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);
// 隣列上に続く
```

```
x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);
// 隣列上に続く
```

```
x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);

x = random(width);
y = random(height);
ellipse(x,y,20,20);
```

random 関数を使っているので、実行する度に描かれる画像が変化します。

このサンプル 4-5 では 10 個の円を表示するために、表示する円の中心を決める命令（x = random(width); と y=random(height);）と円を表示する命令 (ellipse(x,y,20,20);) の組を 10 回実行しています。このように同じ命令を何回も実行したい場合を考えます。繰り返し回数が少なければ、単純に命令を書いていけば、プログラムを作ることが出来ます。しかし、その回数が多ければ、この方法でプログラムを作ることは困難になります。例えば、このプログラムの円の表示個数を 100 個や 1000 個にする場合を考えれば、想像できると思います。

繰り返し処理のことをループ (loop) 処理と呼ぶこともあります。

そこで、同じ命令を何度も繰り返し実行した場合に使われるのが、繰り返し処理です。多くのプログラミング言語では、繰り返し処理を行うために、2 つの方法が用意されています。それは、

1. 繰り返し回数指定型
2. 繰り返し条件指定型

「繰り返し回数」が「繰り返し条件」と見なせば、、繰り返し回数指定型と繰り返し条件指定型は同じだと考えることも出来ます。

です。一般的は、回数指定型繰り返し処理には for 命令、条件指定型繰り返し処理には while 命令を使用します。

サンプル 4-6 はサンプル 4-5 を for 命令を利用するように書き換えた者です。

for 命令を利用した円を 10 個描く サンプル 4-6

```
float x,y; // 変数 x,y は円の中心座標を表す float 型の変数

size(400,200); // 400X200 のウインドウを表示
smooth();      // アンチエリアシングをして図形を描画
background(150); // 背景を灰色で塗りつぶす
fill(255);     // 図形の塗りつぶし色を白色に設定

for(int i=0;i<10;i++){// カウンタ変数 i を 0 ～ 9 で変化させながら
  x = random(width); // 円を表示する x 座標を乱数で決定
  y = random(height);// 円を表示する y 座標を乱数で決定
  ellipse(x,y,20,20);// (x,y) を中心として直径 20 の円を描画
}
```

サンプル 4-6 の先頭部分が、「float x,y;」となっています。これは、同時に複数の変数を宣言する方法です。同じデータ型の変数を複数宣言する場合に便利な方法です。つまり、変数宣言の形式は以下のようになります。

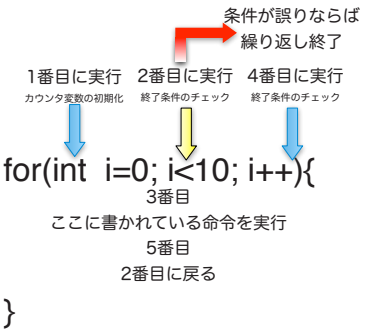
変数宣言の形式
データ型　変数名 ;
データ型　変数名 1, 変数名 2,…;
データ型　変数名 1[, 変数名 2[,…]]

ここからが、本題です。サンプル 4-6 の「for(int i=0;i<10;i++){」から次の「}」の部分が繰り返し処理を行う部分になっています。「for(int i=0;i<10;i++){」部分の 10 によって繰り返し回数を指定してます。繰り返し処理を行う場合には、何回目の繰り返しかを知りたい場合があるので、繰り返し回数を記憶させるための変数を用意します。この変数のことをカウンタ変数と呼ぶことがあります。サンプル 4-6 では、カウンタ変数として i を使っています。「int i=0」の部分でカウンタ変数を指定します。カウンタ変数の変数名には特に制約はありません。

for 命令の括弧 () 内は、”;” で区切られた 3 つ部分から構成されています。1 つ目は、「int i=0」の部分で、カウンタ変数の宣言とカウンタ変数の値を 0 に設定しています。2 つ目は、「i<10」の部分で、繰り返し回数のチェックをしている部分です。この場合には、10 回繰り返し処理を行いたいので、「i<10」となっています。3 つ目は、「i++」の部分で、カウンタ変数の値を 1 増やしています。

for 命令の使い方は、まとめると次の様になります。

ここで述べる、繰り返し処理の表し方は、Processing 言語だけでなく、C 言語系の言語ではほぼ共通の書き方（構文）になっています。
for 命令が非常に強力なので、条件指定型繰り返し処理も for 命令で書かれる場合が多くあります。



「for(int i=0;…){ ～ }」となってい部分の、中括弧 { と } で作っているブロックの部分が繰り返し実行されます。

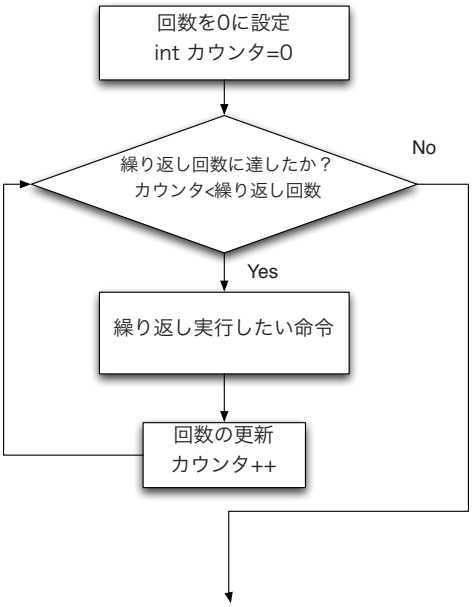
カウンタ変数も普通の変数と代わりないので、カウンタ変数名としては、普通の変数名として利用できるものであれば、何でも OK です。
通常は、i,j,k などのシンプルな名称を利用することが多いようです。この習慣は、FORTRAN 言語と呼ばれる、非常に古いプログラミング言語からの影響だと思います。初期の FORTRAN では、I ～ N で始まる変数名の変数には整数型変数と見なすルールがありました。

for 命令のシンプルな使い方

Processing 言語での記法

```
for(int カウンタ名 =0; カウンタ名 < 繰り返し回数 ; カウンタ名 ++){
  繰り返し実行したい命令
}
```

処理の流れ



カウンタ変数の初期値を 0 としているので、繰り返し回数を指定している部分では、比較に「<」を使用しています。

サンプル 4-6 の繰り返し回数を 100 回に変更したものがサンプル 4-7 です。このように簡単に繰り返し回数を変更することができます。

カウンタ変数の値を更新 (1 増やす) するために、増分演算子 (++) を使用しています。増分演算子に関しては、前回授業資料を参考にして下さい。

for 命令を利用した円を 100 個描く サンプル 4-7

```
float x,y; // 変数 x,y は円の中心座標を表す float 型の変数

size(400,200); // 400X200 のウインドウを表示
smooth();       // アンチエリアシングをして図形を描画
background(150); // 背景を灰色で塗りつぶす
fill(255);     // 図形の塗りつぶし色を白色に設定

for(int i=0;i<100;i++){// カウンタ変数 i を 0 ～ 99 で変化させながら
  x = random(width); // 円を表示する x 座標を乱数で決定
  y = random(height);// 円を表示する y 座標を乱数で決定
  ellipse(x,y,20,20);// (x,y) を中心として直径 20 の円を描画
}
```

変更部分は赤色になっています。

サンプル 4-8 は、カウンタ変数の値を表示するものです。

カウンタ変数の値を表示する サンプル 4-8

```
// カウンタ変数は loop
for(int loop=0;loop < 20;loop++){//loop を 0 ～ 19 まで変化させながら
  println(loop); // 変数 loop の値を表示
}
```

19 は「繰り返し回数 -1」です。0 から繰り返し回数を数えているので、「繰り返し回数 -1」回まで繰り返せば、ちょうど繰り返し回数だけ「{ ～ }」の部分を実行できます。

このサンプル 4-8 を実行すればわかるように、カウンタ変数の値は、0 から始まって 19 まで 1 ずつ増えながら、「{ ～ }」の部分を実行していきます。

繰り返し処理でのカウンタ変数の利用

単純に同じ処理を繰り返すだけの繰り返し処理では、あまり利用する機会がありません。例えば、サンプル 4-9 では line 関数を 11 回実行しています。しかし、引数の値が異なっているため、全く同じではありません。従って、単純な繰り返し処理では扱うことが出来ません。

line 関数を 11 回実行する サンプル 4-9

```
size(300,200);          //300X200 のウインドウを表示
background(255);        // 背景を白色で塗りつぶす
stroke(0);              // 線分の描画色を黒色に設定

line(25,20,25,180);     // 線分を描画
line(50,20,50,180);     // 線分を描画
line(75,20,75,180);     // 線分を描画
line(100,20,100,180);   // 線分を描画
line(125,20,125,180);   // 線分を描画
line(150,20,150,180);   // 線分を描画
line(175,20,175,180);   // 線分を描画
line(200,20,200,180);   // 線分を描画
line(225,20,225,180);   // 線分を描画
line(250,20,250,180);   // 線分を描画
line(275,20,275,180);   // 線分を描画
```

このサンプル 4-9 では、同じ命令を繰り返し実行している訳でないので、for を利用した繰り返し処理に書き換えることが出来ないように見えます。そこで、このプログラムにおける line による線分を描画する位置を、次のプログラムのように書き換えてみます。一見するとトリッキーな書き換えのように見えます。しかし、描画する線分の両端の x 座標の値を指定している引数の値は、25、50、75…のように、25 からから始まり、ちょうど 25 ずつ増加しています。これは、中学生の時に学習した一次関数となっています。そこで、一次関数の式y=ax+bを思い出してもらえば、この書き換えがトリッキーな書き換えでないことが理解できると思います。

line 関数を 11 回実行する サンプル 4-10

```
size(300,200);     //300X200 のウインドウを表示
background(255);   // 背景を白色で塗りつぶす
stroke(0);         // 線分の描画色を黒色に設定
```

```
line( 0*25+25,20,  0*25+25,180);  // 線分を描画
line( 1*25+25,20,  1*25+25,180);  // 線分を描画
line( 2*25+25,20,  2*25+25,180);  // 線分を描画
line( 3*25+25,20,  3*25+25,180);  // 線分を描画
line( 4*25+25,20,  4*25+25,180);  // 線分を描画
line( 5*25+25,20,  5*25+25,180);  // 線分を描画
line( 6*25+25,20,  6*25+25,180);  // 線分を描画
line( 7*25+25,20,  7*25+25,180);  // 線分を描画
line( 8*25+25,20,  8*25+25,180);  // 線分を描画
line( 9*25+25,20,  9*25+25,180);  // 線分を描画
line(10*25+25,20, 10*25+25,180);  // 線分を描画
```

このように変更すると、共通部分の構造が見えてくると思います。赤字の部分は異なっていますが、それ以外の部分は共通になっています。赤字となっている数字の部分は、0 から 1 ずつ増加しています。カウンタ変数の値が 0 から 1,2... と 1 ずつ増加していくのと同じように変化しています。つまり、繰り返し部分を書く場所で、カウンタ変数に記録されている値を利用することで、for を利用した繰り返し処理のプログラムに書き換えることが出来ます。それを行ったものがサンプル 4-11 です。

カウンタ変数の値を利用その 1 サンプル 4-11

```
size(300,200);     //300X200 のウインドウを表示
background(255);   // 背景を白色で塗りつぶす
stroke(0);         // 線分の描画色を黒色に設定
for(int i=0;i<11;i++){ // カウンタ変数 i の値を 0 ～ 10 まで変えながら
  // 2 点 (i*25+25,20),(i*25+25,180) の間に線分を描画する
  line(i*25+25,20, i*25+25,180);
}
```

「i*25+25」の部分は、中学生の時に学習した一次関数の計算となっています。

サンプル 4-11 と同じように、カウンタ変数の値を利用した繰り返し処理のサンプルを示します。このサンプルでは、カウンタ変数の値を利用して、長方形を描く位置を決定しています。

カウンタ変数の値を利用その 2 サンプル 4-12

```
size(200,405);     //200X405 のウインドウを表示
background(255);   // 背景を白色で塗りつぶす
fill(170);         // 背景を白色で塗りつぶす
for(int j=0;j<10;j++){ // カウンタ変数 j の値を 0 ～ 9 まで変えながら
  // 点 (30,10+40*j) を頂点とする横 140、縦 20 の矩形を表示する
  rect(30,10+40*j,140,20);
}
```

「10+40*j」で y 座標の値を決めているので、10,50,90 のように、10 から 40 ずつ増加しながら値が変化しています。

サンプル 4-12 では、for 命令の実行が始まり、

1. 最初にカウンタ変数 j の値が 0 になり (int j=0)、
2. カウンタ変数 j の値（今は 0）は繰り返し回数よりも少ないので、

矩形を描く命令が実行されます、

3. 次にカウンタ変数 j の値が 1 増やされます (j++)、
4. カウンタ変数 j の値（今は 1）は繰り返し回数よりも少ないので、矩形を描く命令が実行されます、
5. 次にカウンタ変数 j の値が 1 増やされます (j++)、
6. **中略**
7. カウンタ変数 j の値（今は 9）は繰り返し回数よりも少ないので、矩形を描く命令が実行されます、
8. 次にカウンタ変数の値が 1 増やされます (j++)、
9. カウンタ変数 j の値（今は 10）は繰り返し回数よりも小さくないので、for 命令による繰り返し処理は終了します。

「繰り返し回数よりも小さくない」とは、「j<10」が false になることを意味しています。

矩形を描く際には、カウンタ変数の値を利用して、矩形の描画位置 (30,10+40*j) を決めています。

サンプル 4-13 では、カウンタ変数を利用して長方形の描画位置を決めるだけでなく、塗りつぶし色の変更 (fill(10+20*j);) もカウンタ変数の値を利用して行っています。

塗りつぶし色は (10,10,10), (30,30,30), (50,50,50)…のように、変化してきます。最後の色は RGB で表すとどんな色になるでしょうか？

カウンタ変数の値を利用その 3 サンプル 4-13

```
size(200,405);    //200X405 のウインドウを表示
background(255);  // 背景を白色で塗りつぶす
for(int j=0;j<10;j++){
  fill(10+20*j);  // 塗りつぶし色を (10+20*j,10+20*j,10+20*j) に変更
  // 点 (30,10+40*j) を頂点とする横 140、縦 20 の矩形を表示する
  rect(30,10+40*j,140,20);
}
```

サンプル 4-14 と 4-15 は、for 命令の繰り返し処理を行う部分で、新たな変数を利用する例（int x=10+8*i;; と int x=20+20*i;）となっています。

カウンタ変数の値を利用その 4 サンプル 4-14

```
size(400,200);    //400X200 のウインドウを表示
smooth();         // アンチエリアシングをかけながら図形を描画
strokeWeight(2); // 線分の太さを 2 に変更
for(int i=0;i<40;i++){//カウンタ変数の値を 0 ～ 39 まで変化させながら、
  int x=10+8*i;       // int 型の変数 x を宣言し、値 10+8*i を代入
  line(x,40,x+60,160);// 2 点 (x,40)、(x+60,160) を結ぶ線分を描画
}
```

変数宣言の仕方は同じです。プログラムの先頭で変数を宣言するとの違いは、変数の有効範囲の違いです。これに関しては、別の機会に説明します。

カウンタ変数の値を利用その 5 サンプル 4-15

```
size(400,200);    //400X200 のウインドウを表示
smooth();         // アンチエリアシングをかけながら図形を描画
strokeWeight(2);  // 線分の太さを 2 に変更
```

ついでながら、プログラム中（除く、コメント文中）に全角の空白があると、「unexpected char :''」のようなエラーとなります。気をつけて下さい。

```
for(int i=0;i<20;i++){//カウンタ変数の値を0～19まで変化させながら、
  int x=20+20*i;    // int型の変数xを宣言、値20+20*iを代入
  // 3点(x,0)、(x+x/2,120)、(1.2*x,height)を順番に結ぶ線分を描画
  line(x,0,x+x/2,120);
  line(x+x/2,120,1.2*x,height);
}
```

プログラム中には、複数個の繰り返し処理を書くことができます。サンプル 4-16 は複数個の繰り返し処理を記述したものです。

複数個の繰り返し処理その 1 サンプル 4-16

```
size(400,200);
background(0);
smooth();
fill(255);
for(int x=0;x<11;x++){ // この繰り返しでは、変数xがカウンタ変数
  ellipse(40*x,0,40,40);
}
for(int y=0;y<6;y++){ // この繰り返しでは、変数yがカウンタ変数
  ellipse(0,40*y,40,40);
}
```

このサンプルでは、2 箇所の for 命令では異なった変数名のカウンタ変数を使用しています。このサンプル 4-16 のように「独立」した繰り返し処理では、同じ変数名のカウンタ変数を利用することが出来ます。同じカウンタ変数名を使用して、書き換えたものがサンプル 4-17 です。

複数個の繰り返し処理その 2 サンプル 4-17

```
size(400,200);
background(0);
smooth();
fill(255);
for(int x=0;x<11;x++){ // この繰り返しでは、変数xがカウンタ変数
  ellipse(40*x,0,40,40);
}
for(int x=0;x<6;x++){ // この繰り返しでも、変数xがカウンタ変数
  ellipse(0,40*x,40,40);
}
```

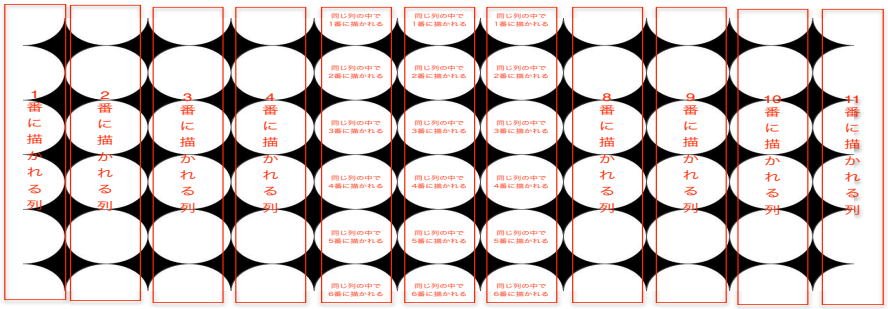
繰り返し処理の入れ子

for命令の繰り返し処理の中に、また繰り返し処理を入れることが出来ます。for による繰り返し処理を入れ子にする場合には、カウンタ変数名は異なる変数名にする必要があります。

サンプル 4-18 では繰り返し処理を入れ子にしています。このサン

多くの言語学者は、この for 命令の入れ子のように、入れ子になった文が扱えることが人間の言語能力の大きな特徴だと考えています。

プルは円を格子状に配置して表示するプログラムです。カウンタ変数 x の値を利用して表示する円の中心の x 座標の値を決め、カウンタ変数 y の値を利用して表示する円の中心の y 座標の値を決めています。「for(int x=0;x<11;x++)」による繰り返し処理では、カウンタ変数 x の値を 1 ずつ増やしながら、「for(int y=0;y<6;y++)」による繰り返し処理を行っています。「for(int y=0;y<6;y++)」による繰り返し処理では、カウンタ変数の y の値を 1 ずつ増やしながら、円を描く命令(ellipse(40*x,40*y,40,40);) を実行しています。これにより、ellipse 関数は 66 回実行されています。



サンプル 4-18 の実行例

複数個の繰り返し処理その 3 サンプル 4-18

```
size(400,200);
background(0);
smooth();
fill(255);
for(int x=0;x<11;x++){  // 一番外側の繰り返し処理のカウンタ変数は x
  for(int y=0;y<6;y++){ // この繰り返し処理のカウンタ変数 y
    ellipse(40*x,40*y,40,40);
  }
}
```

サンプル 4-19 も繰り返し処理を入れ子にしたものです。何回でも繰り返し処理の入れ子にすることが出来ますが、実際の使用では、サンプル 4-18 や 4-19 のように、2 重の入れ子や 3 重の入れ子が多いように思います。

複数個の繰り返し処理その 4 サンプル 4-19

```
size(400,200);
background(0);
smooth();
fill(255);
stroke(100);
```

for 命令や if 命令などで、処理ブロックをハッキリさせるために、字下げやインデント(indent) と呼ばれることを行います。これは、処理ブロック毎に一律に右方向に移動して、命令を書くことです。インデントを行うことで、プログラムの構造を理解しやすくします。Processing 言語ではインデントをしなくても、エラーとはなりません。

```
for(int j = 0;j < 17;j++){
  int y = 20 + 10*j;
  for(int i = 0;i < 37;i++){
    int x = 20+10*i;
    ellipse(x,y,4,4);
    line(x,y,width/2,height/2);
  }
}
```

Python 言語などでは、インデントを行うことで、処理ブロックを指定します。つまり、適切にインデントを行わないと、エラーとなります。

インデント無しバージョン サンプル 4-19'

```
size(400,200);
background(0);
smooth();
fill(255);
stroke(100);
for(int j = 0;j < 17;j++){
int y = 20 + 10*j;
for(int i = 0;i < 37;i++){
int x = 20+10*i;
ellipse(x,y,4,4);
line(x,y,width/2,height/2);
}
}
```

サンプル 4-19' はサンプル 4-19 をインデントを行わないで書いたものです。プログラムの構造が把握しにくいと思います。

インデントがないと、for による繰り返し処理の範囲がわかりづらくなっていると思います。

繰り返し処理を利用したサンプルをいくつかのせておきます。

複数個の繰り返し処理その 5 サンプル 4-20

```
size(400,400);
background(255);
noStroke();
for(int y = 0;y < 10;y++){
  for(int x = 0; x < 10;x++){
    fill(25*x,25*y,20);
    rect(40*x,40*y,30,30);
  }
}
```

規則的な配置をもった画像を作り出すことが出来ます。また、乱数を使用することで揺らぎをもった画像を作り出すことも出来ます。この辺りが、コンピュータのプログラムを利用して作り出す画像の面白さだと思います。
この辺りの、コンピュータ使った作品に関することは、メディアアートやディジタルデザインで詳しく扱われると思います。

乱数と繰り返し処理の組み合わせ サンプル 4-21

```
size(400,400);
background(255);
for(int k=0;k<100;k++){
  stroke(random(256),random(256),random(256));
  line(random(width),random(height),random(width),random(height));
}
```

カウンタ変数の値を利用その 6 サンプル 4-22

```
size(400,400);
background(255);
stroke(0);
```

サンプル 4-22 を実行すると、曲線が見えてきませんか？このような曲線のことを、包絡線と呼びます。

```
for(int y=0;y < 40;y++){
  int v =10*y; // 何回も同じ値を使うので変数に計算結果を保存
  stroke(255,10,10);
  line(0,v,v,height);
  stroke(10,10,255);
  line(v,0,width,v);
}
```

for 命令の繰り返し処理では、繰り返し処理を行う部分に置いては、カウンタ変数と同じ名前の変数を宣言して、使うことは出来ません。従って、サンプル 4-22 では、int 型の変数名 v という変数を宣言して使っています。

再び四角形の描画

rectMode 関数を使うと、rect 関数を利用して長方形を描くときに、色々な長方形の描画位置指定の方法を選ぶことが出来ます。長方形を描く際の位置指定方法を変更すると、次に rectMode 関数を呼び出して明示的に変更しない限り、位置指定方法は変更されません。

rectMode：rect の座標指定方法を変更する命令

rectMode 関数の呼び出し	rect 命令の引数に与える値の役割	備考
rectMode(CORNER)	rect(左上隅 x 座標 , 左上隅 y 座標 , 幅 , 高さ)	デフォルトの指定方法
rectMode(CENTER)	rect(中心の x 座標 , 中心の y 座標 , 幅 , 高さ)	長方形の中心位置を指定
rectMode(CORNERS)	rect(左上隅 x 座標 , 左上隅 y 座標 , 右下隅 x, 右下隅 y)	長方形の左上と右下の位置を指定

正確に言うと、rectMode(CORNERS) では、長方形の対角線の両端の座標を指定しています。

rectMode(CENTER) での長方形描画　サンプル 4-23

```
void setup(){
  size(400,400);
  rectMode(CENTER);
}
void draw(){
  background(255);
  fill(175);
  rect(mouseX,mouseY,100,60);
  fill(0);
  rect(mouseX-30,mouseY-35,20,10);
  rect(mouseX+30,mouseY-35,20,10);
  rect(mouseX-30,mouseY+35,20,10);
  rect(mouseX+30,mouseY+35,20,10);
}
```

車のつもりなのですが。

rectMode(CENTER) での長方形描画　サンプル 4-24

```
size(400,400);
background(255);
stroke(0);
rectMode(CENTER);
```

```
for(int i=0;i<12;i++){
  rect(width/2,height/2,370-30*i,370-30*i);
}
```

傾きが負の一次関数を利用して、正方形の辺の長さを決めています。

rectMode(CORNERS) での長方形描画　サンプル 4-25

```
void setup(){
  size(400,300);
  rectMode(CORNERS);
  noStroke();
  fill(175);
}
void draw(){
  background(255);
  rect(mouseX,mouseY,width-mouseX,height-mouseY);
}
```

落ち葉拾い：折れ線の描画

4 月 12 日配布した資料の「少し複雑な図形を描く」の補足説明です。

多角形を描くために使用される beginShape 関数、endShape 関数、vertex 関数は、noFill 関数と組み合わせて使用することで、折れ線を描くためにも使用できます。endShape 関数の引数に何も値を指定しないで呼び出すと、最初に指定頂点を最後に指定した頂点を結ぶ辺を描画しないで、多角形が描かれます。そこで、noFill 関数を利用して塗りつぶしを行わないような設定にすると、多角形の辺だけを描くようになります。つまり、最初に指定頂点を最後に指定した頂点を結ぶ辺を描画しないで多角形の辺だけを描画するので、折れ線が描くことが出来るようになります。

折れ線の描画方法

1. noFill 関数を呼び出し、塗りつぶしを行わない設定にする
2. beginShape() を実行する
3. vertex 関数で折れ線の始点の位置を指定する
4. vertex 関数で折れ線の途中の点の位置を指定する
5. vertex 関数で折れ線の終点の位置を指定する
6. endShape() を実行する

この方法で折れ線を描画するサンプルを示します。

当然、line 関数を利用して、折れ線を描くことも出来ます。でも、ちょって面倒になります。なぜかわかりますか？

beginShape と endShape による折れ線描画　サンプル 4-26

```
size(400,400);
noFill();  // 塗りつぶしを行わないようにする
stroke(10,10,255);
beginShape(); // 折れ線の頂点位置指定を開始
vertex(0,height/2);  // 始点位置を指定
vertex(width/4,random(height));  // 途中の頂点位置を指定
vertex(width/2,random(height));
vertex(3*width/4,random(height));
vertex(width,height/2);  // 終点位置を指定
endShape(); // 折れ線の頂点位置指定の終了
```

乱数を利用した折れ線描画　サンプル 4-27

```
size(400,400);
noFill();
stroke(255,10,10);
beginShape();         // 折れ線の頂点位置指定の開始
float y = height/2;  // 始点の Y 座標の値は height/2
vertex(0,y);          // 始点位置を指定
for(int x=0;x<40;x++){
  y = y + random(-10,10); // 乱数を使って頂点の Y 座標を変更
  vertex(10*x+10,y);      // 頂点位置を指定
}
endShape();           // 折れ線の頂点位置指定の終了
```

サンプル 4-26 を line 関数を使って折れ線を描画するようにしたものをサンプル 4-28 としてのせておきます。

line 関数による折れ線描画　サンプル 4-28

```
size(400,400);
stroke(10,10,255);
float y0 = height/2;
float y1 = random(height);
line(0,y0,width/4,y1);
y0 = y1;
y1 = random(height);
line(width/4,y0,width/2,y1);
y0 = y1;
y1 = random(height);
line(width/2,y0,3*width/4,y1);
line(3*width/4,y1,width,height/2);
```

一つ前の線分の終点位置を憶えている必要があります。これが、ちょっとプログラムが複雑になる理由だと思います。

情報メディア基盤ユニット用資料（2016 年度第 5 章）

Processing 言語による情報メディア入門

繰り返し処理その 2（while 文）

神奈川工科大学情報メディア学科　佐藤尚

条件指定型繰り返し処理

通常、繰り返し処理には、次の 2 つのパターンがあります。一つは前回説明した回数指定型繰り返し処理です。もう一つは、条件指定型繰り返し処理です。

回数指定型繰り返し処理

条件指定型繰り返し処理

今回は、後者の条件指定型繰り返し処理の説明をします。前回の授業では、サンプル 4-9 の line 関数で X 座標の値を指定している部分を、サンプル 4-10 のように解釈して、for 命令による繰り返し処理を使ったサンプル 4-11 のプログラムを作成しました。

「繰り返し回数」を条件だと思うと、回数指定型繰り返し処理は条件指定型繰り返し処理の特殊な場合と見なすことができます。

line 関数を 11 回実行する サンプル 4-9

```
size(300,200);
background(255);
stroke(0);
line(25,20,25,180);
line(50,20,50,180);
line(75,20,75,180);
line(100,20,100,180);
// 右隣上に続く
```

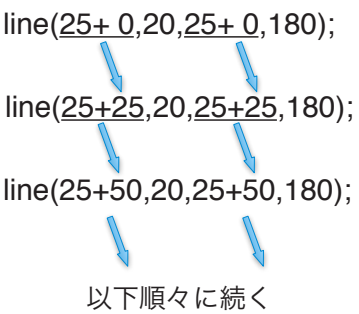
```
line(125,20,125,180);
line(150,20,150,180);
line(175,20,175,180);
line(200,20,200,180);
line(225,20,225,180);
line(250,20,250,180);
line(275,20,275,180);
```

このサンプル 4-9 は、サンプル 5-1 のように書き換えることができます。このままだと、単純な繰り返し処理に書き換えることができません。

line 関数を 11 回実行する サンプル 5-1

```
size(300,200);
background(255);
stroke(0);
line(25+  0,20,25+  0,180);
line(25+ 25,20,25+ 25,180);
line(25+ 50,20,25+ 50,180);
line(25+ 75,20,25+ 75,180);
// 右隣上に続く
```

```
line(25+100,20,25+100,180);
line(25+125,20,25+125,180);
line(25+150,20,25+150,180);
line(25+175,20,25+175,180);
line(25+200,20,25+200,180);
line(25+225,20,25+225,180);
line(25+250,20,25+250,180);
```



このサンプル 5-1 を変数を使うと、次のように書き換えることができます。

変数を使用した line 関数を 11 回実行する サンプル 5-2

```
size(300,200);
background(255);
stroke(0);
int x = 25;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
// 右隣上に続く
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x;
line(x,20,x,180);
x = 25+x
```

このように書き換えると、繰り返しの共通部分がハッキリしてきます。つまり、「line(x,20,x,180);」と「x=x+25;」の 2 行が共通となっています。従って、サンプル 5-3 のように、単純な繰り返し処理を使って、書き換えることが出来ます。

変数と単純な繰り返し命令を利用した書き換え サンプル 5-3

```
size(300,200);
background(255);
stroke(0);
int x = 25;
for(int i=0;i<11;i++){
  line(x,20,x,180);
  x = x+25;
}
```

サンプル 4-9 のプログラムは、単に line 関数を 11 回繰り返し実行したいと言う風にも考えられますが、X 座標の値が 25 から始めて、25 ずつ離れた場所に、X 座標の値が 275 以下の間、繰り返し線分を描くために line 関数を実行するという風にも考えられます。

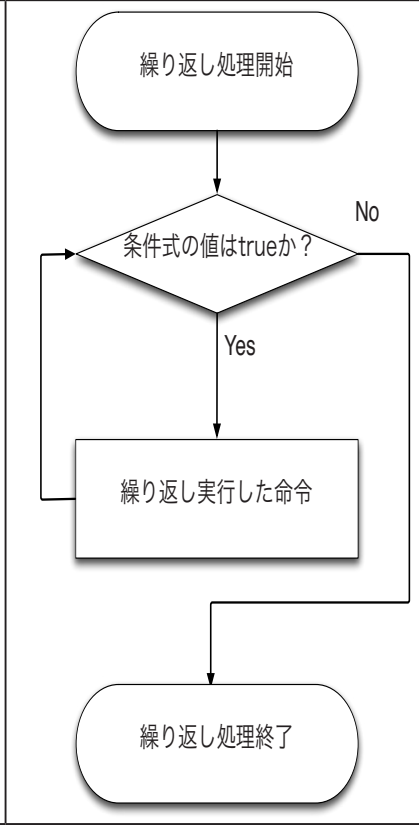
折角、用意した int 型変数 x の出番が少ないのもかわいそうな気がします。

考え方 1	考え方 2
11 本線分を描画するために、line 関数を 11 回実行する。	X 座標の値が 25 から始めて、25 ずつ離れた場所に、X 座標の値が 275 以下の間、線分を描くために line 関数を実行する。

そこで、「X 座標の値が 275 以下の間、繰り返す」のように実行することができる命令があれば、**考え方 2** のようなプログラムを作ることができます。Processing では、この目的のために、while 命令が用意されいます。while 命令は、次に示すように非常に単純な形式になっています。

while 命令の使い方

Processing 言語での記法	処理の流れ

while(条件式){
繰り返し実行したい命令
}
	

while の意味はわかりますよね。

HP が 0 より大きい間、戦闘を繰り返すなどの、「ある条件の間繰り返す」といプログラムの実行の仕方は良く現れるパターンです。

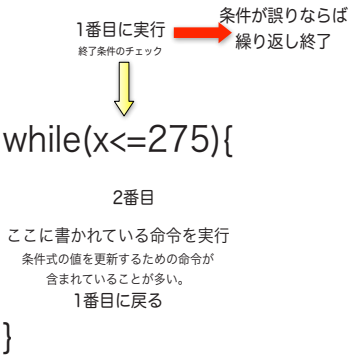
プログラミング言語によっては、ある条件が満たされるようになるまで、繰り返し処理をするという、「〜になるまで」型 (until 型) の繰り返し命令を持っているものを持っていることがあります。

サンプル 5-2 の考え方をベースに、while 命令を使って、サンプル 4-9 を書き換えると次の様になります。

while 命令を利用した繰り返し処理その 1 サンプル 5-4

```
size(300,200);
background(255);
stroke(0);
int x = 25;       // 繰り返し条件の初期値
while(x <= 275){ // 繰り返し条件のチェック
  line(x,20,x,180);  // 繰り返し実行したい命令
  x = x+25;          // 繰り返し条件の更新
}
```

繰り返し条件は、「変数 x の値が 275 以下」なので、while 命令の条件式の部分には、「x <= 275」となっています。



サンプル 5-4 のように、while 命令を使った繰り返し処理では、while 命令を実行する前に、繰り返し条件の初期化と、繰り返し実行したい命令の中に繰り返し条件の更新に関するものが含まれていることが一般的です。

別のサンプルとして、幅 160、高さ 20 の長方形を並べて描くことを考えます。このサンプルでは、次の様な条件となっています。

1. 一番上の長方形の左上の頂点の Y 座標は 10。
2. 長方形の間隔は 5。
3. 長方形の下の部分がウインドウからはみ出ないようにする。

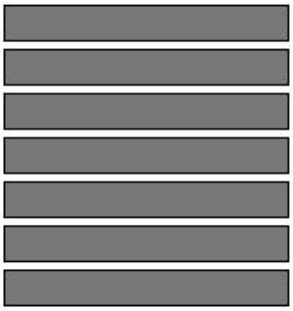
長方形の下の部分がウインドウからはみ出ないようにするためには、長方形の左上のＹ座標と長方形の高さを加えた値がウインドウの高さより小さければ、ウインドウからはみ出ません。つまり、繰り返し条件は、「長方形の左上のＹ座標と長方形の高さを加えた値がウインドウの高さより小さい」間となります。この方針で作成したものがサンプル 5-5 です。

「長方形の左上のＹ座標と長方形の高さを加えた値」とは、長方形の左下のＹ座標の値のことです。

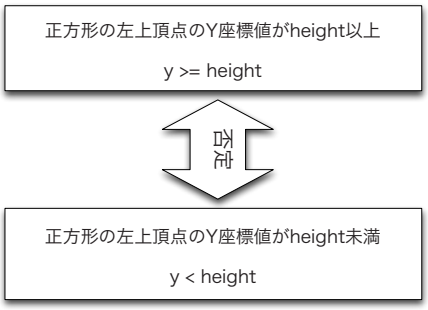
while 命令を利用した繰り返し処理その 2 サンプル 5-5

```
int y;   // 長方形の左上のＹ座標値
int w = 160;  // 長方形の幅
int h = 20;   // 長方形の高さ
int interval = 5;  // 長方形の間隔

size(200,200);
background(255);
fill(120);

y = 10;   // 最初に描く長方形の位置
while((y+h) < height){  // 繰り返し条件のチェック
  rect(20,y,w,h);       // 長方形の描画
  y = y+h+interval;     // 次に長方形を描く場所を求める
}
```

サンプル 5-5 の実行結果



条件指定型繰り返し処理を利用すると、回数指定型繰り返し処理では書くことの難しいプログラムを簡単に書くことができます。次のサンプル 5-6 では、マウスのカーソルの位置から下方向に向けて正方形を書いていきます。正方形の左上頂点のＹ座標値が height 以上の値になったら、正方形の描画を終了します。逆に言うと、正方形の左上頂点のＹ座標値が height 未満の間、正方形の描画を繰り返します。



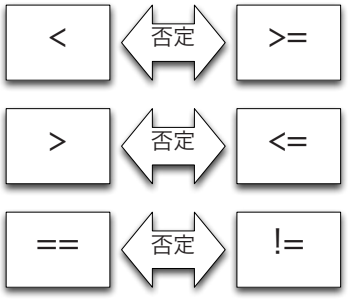
長方形の高さや間隔から何個の長方形を描くことが出来るかは計算で求めることが出来ます。でも、面倒なことはプログラムにさせるのが楽ですよね。

繰り返し処理の停止条件と繰り返し処理の繰り返し条件は、それぞれ否定の関係にあります。「否定」が難しいと「反対」のほうがわかり易いですか？

while 命令を利用した繰り返し処理その 3 サンプル 5-6

```
int y; // 正方形の左上頂点のＹ座標値
int sideLength = 20; // 正方形の一辺の長さ

void setup(){
  size(400,400);
  noFill();
  stroke(0);
  strokeWeight(2);
}
```

良くある否定の関係



```
void draw(){
  background(255);
  y = mouseY;  // マウスカーソルの位置から書き始める
  while(y < height){ // 正方形の左上頂点の Y 座標値が height 未満
    rect(mouseX,y,sideLength,sideLength);
    y = y + sideLength; // 次に正方形を描く位置を計算
  }
}
```

「y = y + sideLength;」ですが、「y += sideLength;」と書かれることのほうが一般的かも知れません。「+=」などの複合代入演算子に関しては、2 回前の授業プリントを参照。

サンプル 5-6 では、Y 座標値だけを変更しました。今度は、X 座標値も変えてみることにします。今回の正方形の左上頂点がウインドウにある間、正方形を描き続けることにします。どんな繰り返し条件式を書けば良いかわかりますか？

while 命令を利用した繰り返し処理その 4 サンプル 5-7

```
int x,y; // 正方形の左上頂点の X 座標値と Y 座標値
int sideLength = 20; // 正方形の一辺の長さ

void setup(){
  size(400,400);
  noFill();
  stroke(0);
  strokeWeight(2);
}

void draw(){
  background(255);
  x = mouseX;// マウスカーソルの位置から書き始める
  y = mouseY;
  while(x < width && y < height){
    rect(x,y,sideLength,sideLength);
    x += sideLength; // 次に正方形を描く位置を計算
    y += sideLength;
  }
}
```

同じデータ型の変数は、「,」で繋ぐことで、複数の変数を一度に宣言することが出来ます。どこかで、書いたかと思いますが。

複合代入演算子を使って書いてみました。

正方形の中を塗りつぶさないのもちょっと寂しいので、塗りつぶす色を変えながら、描画するサンプルをのせておきます。

while 命令を利用した繰り返し処理その 5 サンプル 5-8

```
int x,y; // 正方形の左上頂点の X 座標値と Y 座標値
int sideLength = 20; // 正方形の一辺の長さ
int c;   // 塗りつぶし色を決める変数

void setup(){
  size(400,400);
  noStroke();
}
```

変数 c も int 型変数なので、「int x,y,c;」と書いてもかまいません。通常は、まとまりのあるものを同時に宣言します。変数 x と変数 y は正方形の頂点位置ですが、変数 c は色を表しています。ちょっと、違いますよね。

```
void draw(){
  background(255);
  c = 0;
  x = mouseX;// マウスカーソルの位置から書き始める
  y = mouseY;
  while(x < width && y < height){
    fill(c);
    rect(x,y,sideLength,sideLength);
    c = c+10;
    x += sideLength; // 次に正方形を描く位置を計算
    y += sideLength;
  }
}
```

変数 c の値を draw の中で変えているので、毎回初期化（c=0;）する必要があります。

サンプル 5-5 ～ 5-8 のように、一定の大きさの図形を描くだけであれば、割と簡単に描く必要のある図形の個数を計算できることがあります。そこで、正方形の大きさを 1 割ずつ大きくしながら描画するサンプルを示します。

サンプル 5-6 とは、正方形の描画条件を変えています。

while 命令を利用した繰り返し処理その 6 サンプル 5-9

```
int x,y; // 正方形の左上頂点の X 座標値と Y 座標値
float sideLength; // 正方形の一辺の長さ

void setup(){
  size(400,400);
  stroke(0);
  noFill();
}

void draw(){
  background(255);
  sideLength = 20;
  x = mouseX;// マウスカーソルの位置から書き始める
  y = mouseY;
  while(x < width && y < height){
    rect(x,y,sideLength,sideLength);
    x += sideLength; // 次に正方形を描く位置を計算
    y += sideLength;
    sideLength = 1.1*sideLength;
  }
}
```

辺の長さを 1 割ずつ大きくしながら描画をするので、sideLength を float 型で宣言してます。辺の長さを 1 割長くするために、1.1 倍しているからです。

等比級数の和を求めれば、何とかなる気がします。やっぱり、面倒なことはコンピュータにやらせましょう。

実は、1 割増しなどの単純な変更方法では、がんばると描く必要のある図形の個数を計算できることがあります。そこで､乱数を使って、図形を描く間隔を変えることで、事前に描画する図形の個数を計算できないようなサンプルをのせておきます。このサンプルでは、正方形ではなく、円を描き、色も乱数で変えています。

while 命令を利用した繰り返し処理その 7 サンプル 5-10

```
float diam; // 円の直径
float x,y;  // 円の中心座標
int c;      // 描画色

void setup(){
  size(400,400);
  smooth();
  noStroke();
}

void draw(){
  background(255);
  x = mouseX;
  y = mouseY;
  c = 0;
  diam = 10;
  while(x < width && y < height){
    fill(c);
    ellipse(x,y,diam,diam);
    x = x+random(1,diam);  // X 軸方向の移動は乱数で決定
    y = y+diam/2;          // Y 軸方向には半径分だけ移動
    c = c+int(random(5,10)); // 描画色の決定
    diam = 1.1*diam;         // 直径を 1 割増やす
  }
}
```

乱数を使って、円の位置を変更しているので、変数 x,y は float 型になっています。円の直径も 1 割ずつ増加しているので、変数 diam も float 型になっています。塗りつぶし色も乱数で変えているのです が、int(random(5,10)) で乱数を作っているので、5 以上 10 未満の整数の乱数となっているので、描画色を決めている変数 c は int 型となっています。

もう一つ別な while を使ったサンプルをのせておきます。このサンプルでは、2 段の線分を描いています。ただし、上の段と下の段では、線分を描く間隔が異なっています。この間隔はマウスカーソルの位置 (mouseX の値) で決めています。サンプル 5-11 には、if 命令の部分があります。この部分がないと不都合が起きることがあります。

ここでの不都合とは、描画が終わらないことを指しています。

今まで、キチンと説明をしていませんでしたが、int 型と int 型の割り算は、割り算の結果も int 型になります。例えば、4/2 は 2 となりますが、5/2 は 2.5 ではなく、2 となります。また、1/10 は 0.1 ではなく、0 となります。つまり、mouseX の値が 0 や 1 の時には、mouseX/2 は 0 となります。すると、interval の値は 0 となりますので、常に「x < width」が正しくなるので、描画が終了しないことになります。そこで、interval が 0 の時には、強引に 1 に変更しています。

5.0/2 や 5/2.0 とすれば、float 型と int 型の計算になるので、計算結果は 2.5 となります。

while 命令を利用した繰り返し処理その 8 サンプル 5-11

```
int x,y;  // 線分の描画位置を示す変数
int interval;  // 描画する線分の間隔を表す変数
int len;  // 描画する線分の長さを表す変数
```

```
void setup(){
  size(200,200);
  y = height/2;
  len = height/5;
}

void draw(){
  background(255);
  x = 0;
  interval = mouseX/2;
  if(interval == 0){
    interval = 1;
  }
  while (x < width){    // 上の段の描画
    line(x,y-len,x,y-2*len);
    x = x + interval;
  }
  x = 0;
  while(x < width){    // 下の段の描画
   line(x,y+len,x,y+2*len);
    x = x + 2*interval;
  }
}
```

強力な for 命令

Processing 言語の先祖のようなプログラミング言語に、C 言語があります。C 言語はプログラミング業界に大きな影響を及ぼしています。C 言語以前の多くの言語では、回数指定型の繰り返し処理と条件指定型の繰り返し処理は、明確に異なった命令文を使って、表されていました。C 言語をデザインした人は、回数指定型の繰り返し処理を行う際に指定する必要のあることを、次の 3 つだと考えました。

1. カウンタ変数の**初期化**
2. 繰り返し回数に関する**条件チェック**
3. カウンタ変数の値の**更新**

この 3 つの条件が指定できるように C 言語での for 命令をデザインしました。これの考え方を受け継いで作られたのが Processing 言語の for 命令です。

これをベースに前回説明した for 命令の表記方法を見直してみます。前回は、for 命令の使い方は下のようになっていると説明しました。

C 言語は、1972 年に AT&T ベル研究所のデニス・リッチー (Dennis M. Ritchie,1941-2011) が中心となってデザインしたプログラミング言語です。C 言語と類似の文法が多くのプログラミング言語に取り入れられています。

キチンと調べた訳でないのですが、このように回数指定型の繰り返し処理を考えて、for 命令をデザインしたことが、C 言語の大きな特徴の一つではないかと思います。

for 命令の表記方法（ちょっと退化型？）

Processing 言語での記法

for(int カウンタ名 =0; カウンタ名 < 繰り返し回数 ; カウンタ名 ++){
　繰り返し実行したい命令
}

これは、カウンタ変数を 0 から数え始め、カウンタ変数を 1 ずつ増やしながら、カウンタ変数が「繰り返し回数 -1」になるまで、繰り返し実行したい命令を実行するというものです。これには、次のような対応関係があります。

カウンタ変数を 0 から数え始め	＝	int カウンタ名 = 0
カウンタ変数を 1 ずつ増やす	＝	カウンタ名 ++
カウンタ変数が「繰り返し回数 -1」になるまで	＝	カウンタ名 < 繰り返し回数

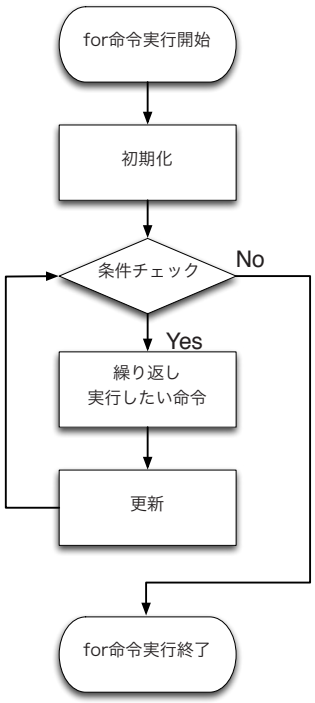
Processing 言語での for 命令は、本来、次の様にデザインされています。

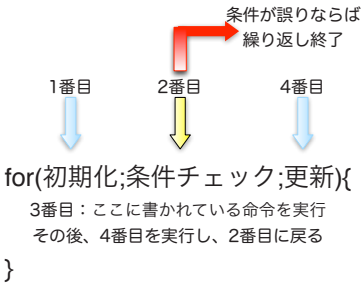
for 命令の本来の使い方（一般型）

Processing 言語での記法

for(初期化 ; 条件チェック ; 更新){
　繰り返し実行したい命令
}

処理の流れ





サンプル 4-9 を次のように書き換えてみます。このように書き換えると、カウンタ変数を 1 で初期化し、カウンタ変数を 1 ずつ増やしながら、カウンタ変数が 11 以下の間、繰り返すということが実現

できれば、良いことがわかります。この方針で、for 命令を使用した作成したプログラムがサンプル 5-13 です。

line 関数を 11 回実行する サンプル 5-12

```
size(300,200);
background(255);
stroke(0);
line( 1*25,20,  1*25,180);
line( 2*25,20,  2*25,180);
line( 3*25,20,  3*25,180);
line( 4*25,20,  4*25,180);
line( 5*25,20,  5*25,180);
line( 6*25,20,  6*25,180);
line( 7*25,20,  7*25,180);
line( 8*25,20,  8*25,180);
line( 9*25,20,  9*25,180);
line(10*25,20, 10*25,180);
line(11*25,20, 11*25,180);
```

カウンタ変数を 1 から始める サンプル 5-13

```
size(300,200);
background(255);
stroke(0);
for(int i=1;i<=11;i++){
  line(i*25,20,i*25,180);
}
```

カウンタ変数 i を 1 から数え始め	=	int i = 1
カウンタ変数 i を 1 ずつ増やす	=	i++
カウンタ変数 i が 11 以下の間	=	i <=11 または i<12

カウンタ変数を 1 から始める サンプル 5-13'

```
size(300,200);
background(255);
stroke(0);
for(int i=1;i<12;i++){
  line(i*25,20,i*25,180);
}
```

サンプル 5-13 では、カウンタ変数を 1 からに変更し、カウンタ変数が 1 ずつ増えるサンプルでした。次は、カウンタ変数を 1 ずつではなく、別な値で変更するサンプルを示します。

前回の講義プリントのサンプル 4-16 では、中心が (0,0)、(40,0)、(80,0)、…、(400,0) と (0,0)、(0,40)、(0,80)…、(200,0) の位置に直径 40 の円を描いています。

for 命令や if 命令などで、処理ブロックをハッキリさせるために、字下げやインデント (indent) と呼ばれることを行います。これは、処理ブロック毎に一律に右方向に移動して、命令を書くことです。インデントを行うことで、プログラムの構造を理解しやすくします。Processing 言語ではインデントをしなくても、エラーとはなりません。

Python 言語などでは、インデントを行うことで、処理ブロックを指定します。つまり、適切にインデントを行わないと、エラーとなります。

整数値の場合には、「11 以下」ということと「12 未満」は同じことなので、左のように 2 通りのやり方があります。

上の注意の意味がわかると、サンプル 5-13 とサンプル 5-13' が同じ動作となることがわかります。

複数個の繰り返し処理その 1 サンプル 4-16

```
size(400,200);
background(0);
smooth();
fill(255);
for(int x=0;x<11;x++){ // この繰り返しでは、変数 x がカウンタ変数
  ellipse(40*x,0,40,40);
}
for(int y=0;y<6;y++){ // この繰り返しでは、変数 y がカウンタ変数
  ellipse(0,40*y,40,40);
}
```

規則的な配置をもった画像を作り出すことが出来ます。また、乱数を使用することで揺らぎをもった画像を作り出すことも出来ます。この辺りが、コンピュータのプログラムを利用して作り出す画像の面白さだと思います。
この辺りの、コンピュータ使った作品に関することは、メディアアートやディジタルデザインで詳しく扱われると思います。

つまり、X 座標に関しては、カウンタ変数を 0 からはじめて、40 ずつ増加させ 400 以下の間、Y 座標に関しては、カウンタ変数を 0 からはじめて、40 ずつ増加させ、200 以下の間、円を描いています。そこで、サンプル 5-14 のようにすれば、サンプル 4-16 と同じ動作になります。

カウンタ変数を 40 ずつ増やすサンプル 5-14

```
size(400,200);
background(0);
smooth();
fill(255);
for(int x=0;x<=400;x+=40){// この繰り返しでは、変数 x がカウンタ変数
  ellipse(x,0,40,40);
}
for(int y=0;y<=200;y+=40){// この繰り返しでは、変数 y がカウンタ変数
  ellipse(0,y,40,40);
}
```

カウンタ変数 x と y を 40 ずつ増やすと言うことを、それぞれ「x+=40」、「y+=40」という複合代入演算子で実現しています。

カウンタ変数の更新は、定数値で決めなくても良いので、while 命令を使って作ったサンプル 5-11 は次のサンプル 5-15 のように書き換えることが出来ます。

変数を利用したカウンタ変数の更新　サンプル 5-15

```
int y;  // 線分の縦位置を決めるための変数
int interval;  // 描画する線分の間隔を表す変数
int len;  // 描画する線分の長さを表す変数

void setup(){
  size(200,200);
  y = height/2;
  len = height/5;
}
```

```
void draw(){
  background(255);
  interval = mouseX/2;
  if(interval == 0){
    interval = 1;
  }
  for(int x=0;x < width; x += interval){// 上の段の描画
    line(x,y-len,x,y-2*len);
  }
  for(int x=0;x < width; x += 2*interval){// 下の段の描画
   line(x,y+len,x,y+2*len);
  }
}
```

また、for 命令の「初期化」、「条件チェック」、「更新」の部分を全て記入する必要はありません。例えば、サンプル 5-10 を次のように for 命令を使って書き換えることが出来ます。この例では、「初期化」と「更新」の部分が省略されています。このサンプル 5-16 のように、「while(条件){ 〜 }」は「for(; 条件 ;){ 〜 }」と全く同じことになります。

for 命令による while 命令の置き換えサンプル 5-16

```
float diam; // 円の直径
float x,y;  // 円の中心座標
int c;      // 描画色
void setup(){
  size(400,400);
  smooth();
  noStroke();
}
void draw(){
  background(255);
  x = mouseX;
  y = mouseY;
  c = 0;
  diam = 10;
  for(;x < width && y < height;){
    fill(c);
    ellipse(x,y,diam,diam);
    x = x+random(1,diam);  // X 軸方向の移動は乱数で決定
    y = y+diam/2;          // Y 軸方向には半径分だけ移動
    c = c+int(random(5,10)); // 描画色の決定
    diam = 1.1*diam;         // 直径を 1 割増やす
  }
}
```

つまり、Processing 言語のような C 言語系のプログラミング言語にとっては、while 命令は盲腸のような存在です。

カウンタ変数を条件チェックでは、どのような条件チェックを行っても良いので、カウンタ変数の値が条件チェックの中に入っていなくてもかまいません。すると、サンプル 5-16 は次のように書き換え

ることも出来ます。ちょっとやり過ぎかも知れませんが。

for 命令による while 命令の置き換えサンプル 5-17

```
float x,y;  // 円の中心座標
int c;      // 描画色
void setup(){
  size(400,400);
  smooth();
  noStroke();
}
void draw(){
  background(255);
  x = mouseX;
  y = mouseY;
  c = 0;
  for(float diam=10;x < width && y < height;diam = 1.1*diam){
    fill(c);
    ellipse(x,y,diam,diam);
    x = x+random(1,diam);  // X 軸方向の移動は乱数で決定
    y = y+diam/2;          // Y 軸方向には半径分だけ移動
    c = c+int(random(5,10)); // 描画色の決定
  }
}
```

キチンと説明はしませんが、サンプル 5-17 はさらに次のように書き換えることが出来ます。ちょっとやり過ぎな気もしますが。

for 命令による while 命令の置き換えサンプル 5-16

```
int c;
void setup(){
  size(400,400);
  smooth();
  noStroke();
}

void draw(){
  background(255);

  c = 0;
  for(float diam=10,x=mouseX,y=mouseY;
      x < width && y < height;
      x += random(1,diam),
      y += diam/2,
      diam *= 1.1,
      c += int(random(5,10))){
    fill(c);
    ellipse(x,y,diam,diam);
  }
}
```

複合代入演算子を使用して「diam *= 1.1」と書いている部分は、「diam = 1.1*diam」書くことも出来ます。

色相、彩度、明度による色指定

色の指定の方法に、色の三原色の組み合わせ (RGB) による方法と色相・彩度・明度 (HSB) による方法があることを説明しました。今まで作ってきたプログラムは RGB により色を指定していました。しかし、HSB による色指定を行うと、赤っぽい色を乱数で出したいなどの場合に便利です。また、色を指定する際には、不透明度（α値）という情報を付加して使用することもあります。Processing では、HSB による色指定を行うことも出来ます。色指定方法を変更するために、colorMode 関数を利用します。colorMode 関数には、沢山の呼び出し方があります。それを以下の表にまとめました。なお、不透明度の値を指定する場合には、明示的に変更しない限り、0 以上 255 以下となっています。

色相：Hue
彩度：Saturation
明度：Brightness

α：ギリシャ文字の小文字の a です。アルファと呼びます。

この不透明度の使い方は、少し後で説明します。

colorMode 関数の使い方

色指定の方法	colorMode 関数の呼び出し方	意味
RGB による色指定	colorMode(RGB)	RGB の各値は 0 以上 255 以下の数値で表します。
	colorMode(RGB, colorMax)	RGB の各値は 0 以上 colorMax 以下の数値で表します。
	colorMode(RGB, rMax, gMax, bMax, alphaMax)	それぞれ、R の値は 0 以上 rMax 以下、G の値は 0 以上 gMax 以下、B の値は 0 以上 bMax 以下の数値で指定します。この場合には、不透明度の値の範囲を明示的に変更していますので、不透明度の値は 0 ～ alphaMax の数値で指定します。
HSB による色指定	colorMode(HSB)	HSB の各値は、0 ～ 255 の数値で指定します。
	colorMode(HSB, huMax, saturationMax, brightnessMax)	それぞれ、hue の値は 0 以上 hueMax 以下、saturation の値は 0 以上 saturationMax 以下 , briightness の値は 0 以上～ brightnessMax 以下の数値で指定します。
	colorMode(HSB, hueMax, saturationMax, brightnessMax, alphaMax)	それぞれ、hue の値は 0 以上 hueMax 以下、saturation の値は 0 以上 saturationMax 以下 , briightness の値は 0 以上～ brightnessMax 以下の数値で指定します。この場合には、不透明度の値の範囲を明示的に変更していますので、不透明度の値は 0 ～ alphaMax の数値で指定します。

基本的に、0 より小さい値を指定した場合には 0、上限値よりも大きな値を指定した場合には上限値となります。

Processing の ColorSelector では、HSB での色指定の際には、hue に関しては 0 以上 359 以下、saturation と brightness に関しては 0

以上 99 以下で表しています。基本的に HSB での色指定を行う場合には、colorMode(HSB,359,99,99) での指定を利用したいと思います。

以下に、HSB により色指定のサンプルを示します。サンプル 5-19 では、色相 (Hue) の値を一定にして、彩度と明度を少し変更したものです。

HSB による色指定その 1 サンプル 5-19

```
size(400,200);
colorMode(HSB,359,99,99);
smooth();

background(190,60,99);
// 上段の円、色相の値は 0
fill(0,99,99);
ellipse(100,50,80,80);
fill(0,60,99);
ellipse(200,50,80,80);
fill(0,30,99);
ellipse(300,50,80,80);
// 下段の円、色相の値は 100
fill(100,99,99);
ellipse(100,150,80,80);
fill(100,60,99);
ellipse(200,150,80,80);
fill(100,30,99);
ellipse(300,150,80,80);
```

サンプル 5-20 では、色相、彩度、明度の値の範囲を、ウインドウの大きさので決めています。このように値の範囲を設定すると、マウスカーソルの位置情報 mouseX と mouseY の値を、直接色指定の情報として利用することが出来ます。

HSB による色指定その 2 サンプル 5-20

```
void setup(){
  size(400,400);
  colorMode(HSB,width-1,height-1,height-1);
  noStroke();
}

void draw(){
  fill(mouseX,mouseY,mouseY);
  rect(mouseX,0,5,height);
}
```

サンプル 5-21 は、HSB による色指定と繰り返し処理を組みあわせたものです。X 軸方向に移動すると色相が変化し、Y 軸方向に移動すると彩度が変化するようになっています。

HSB による色指定その 2 サンプル 5-21

```
size(400,400);
colorMode(HSB,359,99,99);
noStroke();

for(int y = 0;y < 10;y++){
  for(int x = 0;x < 10;x++){
    fill(36*x,9+10*y,99);
    rect(40*x+5,40*y+5,30,30);
  }
}
```

これらのサンプルでもわかるように、HSB による色指定を行うと、何とか色っぽいという色の指定が簡単に行えます。

サンプル 5-22 は、HSB による色指定と乱数を組みあわせたものです。

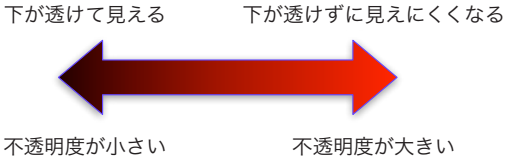
HSB による色指定その 2 サンプル 5-22

```
size(400,400);
colorMode(HSB,359,99,99);
rectMode(CENTER);
background(0,0,99); // 背景を白にする、彩度の値は 0
stroke(0,0,0);      // 枠線を黒にする、明度の値は 0
for(int i=0;i<400;i++){
  float r1 = random(30);
  float r2 = random(50,100);
  fill(r1,r2,r2);
  rect(random(width),random(height),30,30);
}
```

重要：HSB による色指定において、白は彩度の値を 0、明度を 99、黒は彩度と明度の値を 0 とします。彩度の値を 0 にして、明度の値を 0 から 99 に変化させると、黒から白に変化していきます。

不透明度の指定

今までの Processing 言語のサンプルで見たようにように、図形の上に別な図形を描画すると、最初に描かれていた図形は完全に塗りつぶされてしまい、消えてしまいます。このように、最初に描かれた図形を完全に消えてしまうことを防ぐために、不透明度付きの色を利用します。不透明度の値を小さくすると、その下に描かれている図形が透けて見えるようになります。



不透明度のことをαチャンネルと呼ぶこともあります。

不透明度の付きの色の場合には、次のような式で描画色を求めます。簡単のために、不透明度は 0 ～ 255 の数値で表されているものとします。c が

$$c = \frac{\alpha}{255} fgColor + (1 - \frac{\alpha}{255}) bgColor$$

実際に描画される色の値、bgColor が背景色、fgColor が指定描画色です。

このような計算方法を線形補間と呼びます。CG やゲームなどでよく使われる計算方法です。

サンプル 5-2 3 は、不透明度付き色指定を行ったサンプルです。

不透明度付き色指定その 1 サンプル 5-23

```
size(400,200);
smooth();
colorMode(HSB,359,99,99); // 不透明度は 0 ～ 255
background(200,60,99);

fill(0,100,91,255); // 不透明度が 255 なので、下にある図形は隠れる
ellipse(50,height/2,150,150);

fill(0,60,91,127);// 不透明度が 127 なので、下にある図形は少し隠れる
ellipse(150,height/2,150,150);

fill(0,60,91,63);// 不透明度が 127 なので、下にある図形は透ける
ellipse(250,height/2,150,150);

noFill();// 塗りつぶしなし
ellipse(350,height/2,150,150);
```

サンプル 5-24 は、サンプル 5-22 の色指定に不透明度の情報を追加したものです。

不透明度付き色指定その 2 サンプル 5-24

```
size(400,400);
colorMode(HSB,359,99,99);
rectMode(CENTER);
background(0,0,99); // 背景を白にする、彩度の値は 0
stroke(0,0,0);      // 枠線を黒にする、明度の値は 0
for(int i=0;i<400;i++){
  float r1 = random(30);
  float r2 = random(50,100);
  fill(r1,r2,r2,random(100,200));
  rect(random(width),random(height),30,30); // 不透明度を追加
}
```

不透明度の情報を使うと、フェードアウトするような処理を実現出来ます。つまり、下に描かれている図形を完全に消去しないで、不透明度の付きの色で塗りつぶすことで、徐々に消えていくような処理を再現できます。このような方針で作成したものがサンプル 5-25 です。

このサンプルでは、draw 関数の先頭で background 関数を利用した塗りつぶしを行わずに、不透明度の付きの色指定 (不透明度 64 の白) でウインドウ全体を塗りつぶしています。不透明度付きの色でで塗りつぶしているので、始めに描かれている画像が完全に消えずに、少し残ります。従って、昔に書かれた円は色が徐々に薄くなっていき、最終的には消えてしまます。これが繰り返し行われるので、残像が残ったような効果が再現出来ます。不透明度の値を大きくすると、残像は直ぐに消えるようになります。逆に、不透明度の値を小さくすると、

残像が長く残るようになります。

不透明度付き色指定その 3 サンプル 5-25

```
void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
  noStroke();
}

void draw(){
  fill(0,0,99,64); // 不透明度付きの色で塗りつぶす
  rect(0,0,width,height);
  fill(0,99,99);
  ellipse(mouseX,mouseY,30,30);
}
```

不透明度を使うと、色々な面白い効果を得ることが出来ます。

不透明度付き色指定その 4 サンプル 5-26

```
size(400,400);
colorMode(RGB);
smooth();
background(255);
fill(255,10,10,50);
stroke(10,100,255,80);
strokeWeight(100);
ellipse(100,100,400,400);
ellipse(300,100,400,400);
ellipse(100,300,400,400);
ellipse(300,300,400,400);
```

Processing では、色の情報を保存するために、color 型というデータが用意されています。この color 型を利用すると、色のデータを保存しておくことが出来ます。ただし、色の情報は複数の値を指定しないと確定しないので、color 関数を使用して color 型のデータを作り出します。color 関数の引数は、fill 関数や stroke 関数などと同じです。この color 型を利用したサンプルを次に示します。

color 型による色指定サンプル 2 サンプル 5-27

```
size(400,400);
smooth();

color c1 = color(10,100,255);
color c2 = color(255,200,10,128); // 不透明度の情報を利用できる
color c3 = color(255);  // これは白
color c4 = color(0);    // これは黒
```

```
background(c3);  // background でも利用可能
stroke(c4);      // stroke でも利用可能
fill(c1);        // fill でも利用可能
ellipse(150,150,250,250);
fill(c2);
ellipse(250,250,250,250);
```

color 型による色指定サンプル 2 サンプル 5-28

```
color c,c1,c2;

void setup(){
  size(400,400);
  rectMode(CENTER);
  c1 = color(255,10,10);
  c2 = color(10,255,10);
}

void draw(){
  background(255);
  if(mousePressed){
    c = c1;  // 当然、color 型変数への代入も出来ます
  }else{
    c =c2;
  }
  fill(c);
  rect(width/2,height/2,300,300);
}
```

情報メディア基盤ユニット用資料（2016 年度第 6 章）

Processing 言語による情報メディア入門

文字列と画像の表示と座標変換

神奈川工科大学情報メディア学科　佐藤尚

今までのプログラムでは、図形の表示だけを扱ってきました。色々なプログラムを作っていく際には、図形の表示だけではなく、文字や画像の表示を行いたいことがあります。今回は、文字列や画像の表示を取り扱います。

文字列の表示

Processing 言語で、様々な種類のフォントを表示することが出来ます。そのためには、いくつかの手順が必要となります。大雑把に言うと、文字列を表示するためには、1) 使用したいフォントを指定してから、2) 文字列の表示位置と表示文字列を指定するという手順になります。文字列表示の手順を詳しく述べると、以下のようになります。

手順 1：フォントの指定

文字列を表示するためは、まず Processing 内部にフォントの情報を取り込む必要があります。そのために、コンピュータで使えるフォント情報を Processing で扱うことの出来る vlw フォーマットに変換する必要があります。この変換処理は、Tools メニューの「Create Font...」を選ぶと表示されるダイアログボックスで行うことが出来ます。このダイアログボックスで、変換したいフォントを指定し、その大きさ (size) を指定します。OK ボタンをクリックすると、Filename 欄に表示されている名前がファイル名となっている vlw フォーマットのファイルが作成されます。作成されたファイルは、Sketch メニューの「Show Sketch Folder」を選ぶと表示されるフォルダー内にある data フォルダに保存されています。

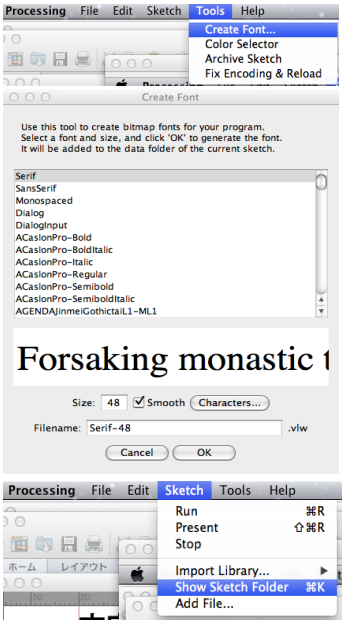


図 6-1 vlw ファイルの作成

次に、生成したフォント情報を PFont 型変数の変数に読み込みます。この読み込みのためには、loadFont 関数を使用します。さらに、どのフォント情報を使用して文字列の表示を行うかを決めるために、textFont 関数を使用します。textFont 関数では、表示するフォントの大きさを指定することも出来ます。

vlw フォーマットでは、文字の形の情報をイメージとして保持しています。

作成するフォントのサイズを大きくすると、変換後に作られるファイルのサイズが大きくなります。同様に、漢字などを含むフォントを変換すると変換後に作られるファイルのサイズがかなり大きくなりますので、注意して下さい。

後で、作成したフォント名が必要となるので、Filename 欄のファイル名をコピーしておくと便利です。

Processing のスケッチ（プログラム）で使用されるデータファイルは、この data フォルダ内に保存することが一般的です。

複数の vlw 形式のファイルを読み込むことで、複数のフォントで表示を行うことが出来ます。

表 6-1 文字列表示関連のデータ型と関数その 1

関数名など	説明
PFont	フォント情報を格納するデータ型
String	文字列を格納するデータ型
loadFont(file)	引数 file で指定された vlw ファイルを読み込む関数
textFont(f)	PFont 型の引数 f で指定したフォントを表示に利用する
textFont(f,size)	PFont 型の引数 f で指定したフォントを大きさ size で表示に利用する
text(str,x,y)	引数 str で指定された文字列を位置 (x,y) に表示する関数
text(str,x,y,w,h)	引数 str で指定された文字列を位置 (x,y)、幅 w、高さ h の長方形の内部に表示する関数
textSize(size)	表示に利用するフォントの大きさを size に設定する関数
createFont(fname,size)	大きさ size で引数 fname で指定したフォント情報を作成する。

正方形の指定方法は、実行時の rectMode により変化します。

createFont 関数を使用すると、「Tools > Create Font」で vlw ファイルを作成しなくても大丈夫です。

手順 2：フォントの表示

文字列を表示する際には、text 関数を使用します。text 関数では、表示する文字列と表示位置を指定します。文字列の表示色は、fill 関数で指定した色が使用されます。stroke の指定は無視されます。3 つの引数をとる text 関数での表示位置の指定は、基本的に図の赤丸の場所を指定します。

赤丸を通る X 軸に平行な線をベースラインと呼んでいます。

図 6-2 フォント表示の際の位置指定

文字列を表示する単純なサンプル 6-1

```
PFont font;  // フォント情報を保存する変数
String msg = "Riho";
size(400,200);
font = loadFont("Geneva-48.vlw");  // vlw ファイルの読み込み
textFont(font);  // 表示するフォントの指定
background(255);
fill(0);
text("Kanagawa Institute Of Technology",10,50); // 文字列を表示
textSize(16);    // 表示するフォントの大きさの変更
text("Kanagawa Institute Of Technology",10,100);
textSize(32);    // 表示するフォントの大きさの変更
text(msg,10,150);  // String 変数の保存されている文字列を表示
```

このサンプルでは、Geneva フォントを使用して、大きさ 48 の vlw ファイルを作成して使用しています。

作成した vlw ファイルのフォントサイズより大きなサイズでは、フォントを表示しない方が望ましと思います。

次に複数のフォントを表示するサンプルを示します。

複数のフォントで表示する単純なサンプル 6-2

```
PFont f1;// フォント情報を保存する変数
PFont f2;
String msg = "The quick brown fox jumps over the lazy dog";
size(400,300);
f1 = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
f2 = loadFont("SansSerif-48.vlw");// vlw ファイルの読み込み
background(255);
fill(50);
textFont(f1,20);// 表示するフォントと大きさの指定
text("Riho",50,100);  // 文字列の表示
textFont(f2,18);// 表示するフォントと大きさの指定
text(msg,50,200); // String 変数の保存されている文字列を表示
```

英語の文「The quick brown fox jumps over the lazy dog」の特徴がわかりますか？割と有名なフレーズなのですが。

このサンプルでは、サイズ 48 の Serif フォントと SanSerifu フォントを利用して、vlw ファイルを作成しています。

文字列は、長方形の領域を指定して、その内部に表示することが出来ます。この目的のためには、5 つの引数をとる text 関数を使用します。この関数は最初の引数で表示する文字列を指定し、残りの引数を利用して、表示を行う長方形を指定します。この長方形領域の指定方法は rect 関数の場合と同じです。従って、現在の rectMode で長方形が指定されます。

長方形領域での文字列表示の単純なサンプル 6-3

```
PFont font;// フォント情報を保存する変数
String msg = "The quick brown fox jumps over the lazy dog";
size(400,300);
font = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
background(255);
fill(0);
textFont(font);// 表示するフォントの指定
text(msg,50,50,350,200);//
```

サンプル 6-4 では、rectMode を CENTER に変更したものを示します。表示の違いを確認して下さい。

CENTER 指定での長方形領域での文字列表示のサンプル 6-4

```
PFont font;// フォント情報を保存する変数
String msg = "The quick brown fox jumps over the lazy dog";
size(400,300);
font = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
rectMode(CENTER); // rectMode を CENTER に変更
background(255);
fill(0);
textFont(font);// 表示するフォントの指定
text(msg,50,50,350,200);//
```

サンプル 6-5 で、長方形領域の指定方法をマウスボタンが押され

ているかどうかで変更するものを示します。

長方形領域指定方法切り替えでの文字列表示のサンプル 6-5

```
PFont font;// フォント情報を保存する変数
String msg = "The quick brown fox jumps over the lazy dog";
void setup(){
  size(400,300);
  font = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
  rectMode(CENTER);
}
void draw(){
  background(255);
  if(mousePressed ){
    rectMode(CENTER); // マウスボタンが押されていたら、CENTER
  }else{
    rectMode(CORNER);// マウスボタンが押されていなければ、CORNER
  }
  fill(0);
  textFont(font);// 表示するフォントの指定
  // 後ろの 4 つの引数で表示領域の長方形の指定を行っている
  text(msg,mouseX,mouseY,350,200);
  noFill();
  stroke(255,10,10);
  rect(mouseX,mouseY,350,200);  // 表示領域の表示
}
```

rectMode が CORNER が デフォルトの指定です。

サンプル 6-3 ～ 6-5 のように文字列を表示する長方形領域を指定できるとすると、文字揃えなども行いたくなります。これを実行するのが、textAlign 関数です。サンプル 6-6 で、この textAlign 関数を利用して、文字揃えを変更したものを示します。

文字揃えの変更を行う 6-6

```
PFont f;
String msg = "The quick brown fox jumps over the lazy dog";
size(600,300);
f = loadFont("Serif-48.vlw");
background(255);
stroke(200);
line(width/2,0,width/2,height);
fill(50);
textFont(f,16);
text(msg,width/2,60);
textAlign(CENTER);
text(msg,width/2,120);
textAlign(LEFT);
text(msg,width/2,180);
textAlign(RIGHT);
text(msg,width/2,240);
```

サンプル 6-6 の実行例

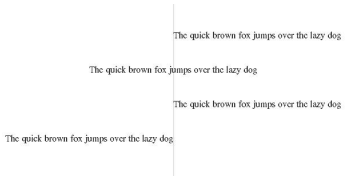


表 6-2 文字列表示関連のデータ型と関数その 2

関数名など	説明
textAlign(CENTER)	指定した長方形領域に、文字列を中央揃えで表示するようにする。
textAlign(LEFT)	指定した長方形領域に、文字列を左揃えで表示するようにする。
textAlign(RIGHT)	指定した長方形領域に、文字列を右揃えで表示するようにする。
textWidth(str)	現在の表示文字設定で、文字列 str を表示した時の幅を求める関数
textDescent()	現在の表示文字設定で、文字列を表示した時のベースラインからどれだけ下に表示されるかを求める関数。
textAscent()	現在の表示文字設定で、文字列を表示した時のベースラインからどれだけ上に表示されるかを求める関数。

図 6-2 に示すように、文字 g はベースラインの下の方にも文字の一部が出ています。また、文字によって文字の高さが異なっていますし、文字によってその幅も異なっています。これらの情報がわかれば、文字列が描かれる仮想的な長方形を決めることができます。このようなことが出来れば、ウインドウの端で跳ね返るようなプログラムを作成することができます。

サンプル 6-7 は、図 6-1 を描く際に利用したプログラムです。

テキストの表示位置を求めるサンプル 6-7

```
PFont font;
String msg = "Anegasaki";
void setup(){
  size(1024,512);
  font = loadFont("Serif-128.vlw");
  textFont(font);
}
void draw(){
  background(255);
  fill(0);
  text(msg,mouseX,mouseY);
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(mouseX,mouseY,10,10);
  line(-128+mouseX,mouseY,width,mouseY);
  line(-128+mouseX,mouseY+textDescent(),
       width,mouseY+textDescent());
  line(-128+mouseX,mouseY-textAscent(),width,mouseY-textAscent());
  line(mouseX,mouseY+128,mouseX,0);
  line(mouseX+textWidth(msg),mouseY+textDescent(),
       mouseX+textWidth(msg),0);
}
```

次のサンプル 6-8 は textWidth 関数を使用した例です。左端で表示文字列が消えると、右側から現れてきます。長方形や円の場合と同じように表示位置を少しずつ変化させると、文字列の移動が実現出来ます。このサンプルでは、文字列の最後がウインドウの左端に到達したら、再び右端から文字列を表示するようにしています。

文字列の移動サンプル 6-8

```
PFont f;
String msg = "The quick brown fox jumps over the lazy dog";
float x; // 文字列の表示開始位置の x 座標

void setup(){
  size(400,200);
  colorMode(RGB);
  f = loadFont("Serif-48.vlw");
  x = width;
}
void draw(){
  background(255);
  fill(50);
  textFont(f,16);
  float widthOfMsg = textWidth(msg);
  text(msg,x,height/2);
  x -= 2;  // 2 ずつ左に移動
  if(x < -widthOfMsg){ // 文字列の最後がウインドウから消えたら
    x = width;
  }
}
```

文字列の表示位置の X 座標の値は x です。文字列を表示する際に必要となる幅は widthOfMsg なので、文字列を表示した際の右端の X 座標の値は x+widthOfMsg となります。この場所がウインドウの左端から出てしまうのは、x+widthOfMsg < 0 の時です。widthOfMsg を右辺に移項すると、if 命令の条件部分になります。

この後に学習する知識などを使用すると、Star Wars のオープニングのようなプログラムを簡単に作ることができます。

おまけのサンプル 6-9

```
PFont f;
String msg = "A New Hope¥n¥nA long long time ago¥nIn a galaxy far far away....";
float y=0;
void setup(){
  size(400,400,P3D); // 3D 表示を行う
  f = loadFont("Serif-48.vlw");
  textFont(f);
  textAlign(CENTER);
}
void draw(){
  background(0);
  fill(5,200,255);
  translate(width/2,height/2);
  rotateX(PI/4);
  text(msg,0,y);
  y--;
}
```

文字列中の ¥n は表示されることがありません。この ¥n があると、そこで改行が行われます。C 言語系のプログラミング言語では、文字列中に ¥n があると改行を意味しています。このような ¥ と組みになって特別な意味を表すものをエスケースシーケンスと呼んでいます。
本によっては、¥ の代わりに \ を使用している場合があります。これは文字コードの問題が関連しています。IT 基礎で関連する話題が出てくると思います。

画像ファイルとしての保存

今までのプログラムでは、作成した画像はプログラムの実行を終了すると消えていました。プログラムを実行せずに作成した画像を見るためには、作成した画像を保存することが必要となります。Processingでは、ウインドウの内容を画像ファイルとして保存する save 関数と saveFrame 関数が用意されています。save 関数は、ウインドウの内容を 1 つの画像ファイルとして保存します。一方、saveFrame 関数は連番の番号付きファイル名で画像を保存します。保存される画像ファイルは、Tools メニューの「Show Sketch Folder」を選んだ時に出てくるフォルダ内にある data フォルダの中に保存されます。

表 6-3 画像ファイルとしての保存関連の関数

関数	説明
save(filename)	filename で指定したファイル名でウインドウの内容を画像ファイルとして保存します。ファイル名には、保存する画像ファイルの形式を指定する拡張子が必要です。指定できる拡張子は tif,tga,jpg,png です。
saveFrame()	ウインドウの内容を連番の画像ファイルとして保存します。保存されるファイル名は screen- 連番 .tif です。この関数を呼び出す度に、連番部分の数字が増えていきます。
saveFrame(filename)	ウインドウの内容を連番の画像ファイルとして保存します。引数の filename は ”filename-####. 拡張子 ” の形となります。#### の部分が連番の数字となります。# が 4 つあれば、4 桁の数字で連番の部分の数字が決まります。指定できる拡張子は tif,tga,jpg,png です。

つまり、次の 4 つのファイル形式で画像をファイルに保存出来ます。
tif：TIFF 形式
tga：TARGA 形式
jpg：JPEG 形式
png：PNG 形式

サンプル 6-10 はサンプル 6-1 の最後に save 関数の呼び出しを追加して、ウインドウの内容を画像ファイルとして保存するものです。

save 関数を利用した サンプル 6-10

```
PFont font;  // フォント情報を保存する変数
String msg = "Riho";
size(400,200);
font = loadFont("Geneva-48.vlw");  // vlw ファイルの読み込み
textFont(font);  // 表示するフォントの指定
background(255);
fill(0);
text("Kanagawa Institute Of Technology",10,50); // 文字列を表示
textSize(16);    // 表示するフォントの大きさの変更
text("Kanagawa Institute Of Technology",10,100);
textSize(32);    // 表示するフォントの大きさの変更
text(msg,10,150);  // String 変数の保存されている文字列を表示
save("test.jpg"); // ウインドウの内容を test.jpg ファイルに保存
```

draw 関数内で save 関数を呼び出し画像ファイルとして保存する場合には、プログラムの実行を終了するタイミングによっては、正しく保存されない場合があります。

画像ファイルがどこに保存されるか、ちゃんと確認しておいて下さい。

画像ファイルの読み込み

プログラムを作成していると、外部で作成した画像をファイルの表示や利用などを行いたいことがあります。Processing 言語では、画像データを保存するために PImage 型が用意されています。PImage 型の変数には画像データを記憶させておくことが出来ます。

Processing でファイルに保存されている画像を表示するためには、

1. 画像ファイルに保存されている画像データをコンピュータ内に読み込む、
2. 読み込んだ画像データを表示する、

という手順をとります。

文字列の表示と同じような手順となっています。

手順 1：画像ファイルの読み込み

このプログラムで読み込む画像ファイルはどこに置かれているのでしょうか？ Processing では、プログラム中で読み込む画像ファイルなどの保存場所が決まっています。それは、“Show Sketch Folder” で表示されるフォルダ内にある data という名称のフォルダです。もし、そこの data フォルダが無い場合には、data フォルダを作成して、そこに利用する画像ファイルなどをコピーして下さい。これが面倒な場合には、Processing のプログラムを書いている部分にドラッグ & ドロップすると、自動的に data フォルダにコピーされます。

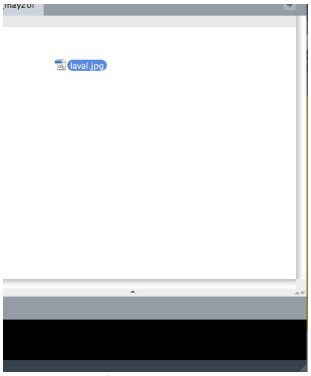


図 6-3 データファイルのドラッグ & ドロップ

画像ファイルを読み込むためには、loadImage 関数を使用します。読み込んだ情報を PImage 型の変数に代入します。

実は、loadImage 関数はかなり強力な機能を持っています。

手順 2：画像ファイルの表示

読み込んで PImage 型変数に保存されている画像を表示するためいには、image 関数を使用します。

表 6-4 画像データ表示関連の関数など

関数	説明
PImage	画像情報を保存するためのデータ型
loadImage(filename)	filename で指定した画像ファイルを読み込む。TIFF 形式、TARGA 形式、JPEG 形式、PNG 形式の画像ファイルを読み込むことが出来ます。
loadImage(name,ext)	引数 ext では読み込む画像データの種類 (png,jpg,gif など) を指定する。
image(img,x,y)	PImage 型引数 img で指定した画像の内容を、引数 x,y で指定された場所に表示する。位置の指定方法は、imageMode 関数で指定します。
image(img,x,y,w,h)	PImage 型引数 img で指定した画像の内容を、引数 x,y で指定された場所に、横方向の大きさを w、縦方向の大きさを h に変更して表示する。

画像は長方形になっているので、rect 関数で長方形を描画するのと同じ方法で、画像の描画位置を指定します。

image 関数のように、引数の数やデータ型によって処理の内容によって異なる関数定義を行うことを多重定義（オーバーロード）と呼びます。以

関数	説明
imageMode(mode)	引数 mode で指定された方法で、表示する画像位置を指定できるようになります。引数 mode には、値 CORNER,CENTER,CORNERS を指定することが出来ます。それぞれの値の意味は、rectMode の場合と同じです。

前から使用していた fill 関数や stroke 関数も多重定義されている関数です。

サンプル 6-11 は loadImage 関数を imae 関数を利用した単純なプログラムです。

image 関数を利用した サンプル 6-11

```
PImage src; // 画像データを保存するための変数
void setup(){
 size(400,400);
 src = loadImage("laval.jpg");// ファイル名は適当なものに変えること
}
void draw(){
 image(src,random(width),random(height)); // デタラメな位置に表示
}
```

Processing では、異なる変数に画像データを記録させておけば、複数の画像を扱うことが出来ます。サンプル 6-12 は複数の画像を取り扱うサンプルプログラムです。このプログラムは、マウスボタンを押した状態と離した状態で表示する画像を切り替えています。また、src2.width と src2.height のようにすると、src2 に保存されている画像の横方向の画素数と高さ方向の画素数を取り出すことが出来ます。

複数の画像を利用した サンプル 6-12

```
PImage src1,src2;
void setup(){
  size(600,600);
  src1 = loadImage("2cv.jpg");// ファイル名は適当なものに変えること
  src2 = loadImage("laval.jpg");
}
void draw(){
  background(255);
  if(mousePressed==true){
    image(src1,mouseX,mouseY);
  }else{
    // 画像を半分の大きさにして表示
    image(src2,mouseX,mouseY,src2.width/2,src2.height/2);
  }
}
```

表 6-5 画像データの取得

フィールド名	意味
PImage 型変数 .width	記録されている画像の横方向の画素数を記憶している。
PImage 型変数 .height	記録されている画像の縦方向の画素数を記憶している。

QR コードの作成（おまけ）

画像ファイルを表示するために使用した loadImage 関数は単にパソコン内の画像を読み込むだけではなく、もう少し高度なことも出来ます。その一例として、loadImage 関数を利用して、QR コードを作ることに挑戦します。Google では Chart API と呼ばれる web を利用してグラフを描く機能を提供しています。この機能の中に QR コードを描くものがあるので、これを利用します。サンプル 6-13 は QR コードを表示するものです。

この方法は、クラウドと呼ばれている情報処理を利用していると見ることもできます。

QR コードの表示 サンプル 6-13

```
PImage qrcode_img; // QR コード画像を保存する変数
String uri; // 文字列を保存する変数
String qrcode_google_api = "http://chart.apis.google.com/chart?";
int qrcode_size = 300; // 表示する QR コードの大きさを決める値
String data = "www.kait.jp"; // QR コードの中に埋め込みたい情報

size(qrcode_size, qrcode_size);
// Google Chart API を呼び出すための URL を作る。
uri = qrcode_google_api + "chs=" + qrcode_size + "x" + qrcode_size
+ "&cht=qr&chl=" + data;
// Google Chart API の機能を利用して、QR コード作成。
qrcode_img = loadImage(uri,"PNG");// PNG 型で画像ファイル
image(qrcode_img,0,0); // 作成した QR コードを表示
save("myqrcode.png"); // 表示した QR コードを保存、ファイル名は変更可能
```

ここで使用している + も多重定義されています。int 型や float 型の際には加算として機能しますが、String 型の際には文字列の連結となっています。

鋭い人は気がついたかもしれませんが、Processing の loadImage 関数は data フォルダ内の画像だけでなく、ファイル名を URI で指定すると、web サイトなどに置かれている画像を読み出すことも出来ます。この機能を利用したものがサンプル 6-14 です。

Uniform Resource Identifier

URI 指定での画像の表示 サンプル 6-14

```
PImage img;

void setup(){
  size(300,300);
  // setup 内で画像を読み込まないと、ちょっと面倒なことが起きるかも
  img = loadImage("http://www.kait.jp/images/top2011/index.jpg");
}

void draw(){
  background(255);
  image(img,0,0);
}
```

ネット越しに画像データの取得を行っているので、loadImage 関数の実行が終わっても画像データの読み込みが完全に終了していない場合があります。その場合には、上手く表示が行われないことになります。
このような処理の仕方をノンブロッキング処理と呼びます。ネット系のプログラムでは良く使用される方法です。

座標軸の移動

今までの知識で、図 6-4 のような画像を作ろうとすると、ちょっと大変です。このような画像を作るために、必要となる座標軸の移動（座標変換）について説明します。

図 6-4

Processing では、座標軸は図 6-5 のように決まっています。この座標軸を基準に図形の描画が行われています。そこで、図 6-4 のような斜めになった図形を描くためには、描画の基準となる座標軸が傾いていれば可能です。このように、基準となる座標軸を傾けたり、移動させたりすることを座標変換と呼んでいます。平面の場合にの座標変換は、下の式のようなもので表すことが出来るものです。しかし、コンピュータグラフィックなどで

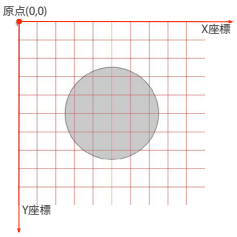
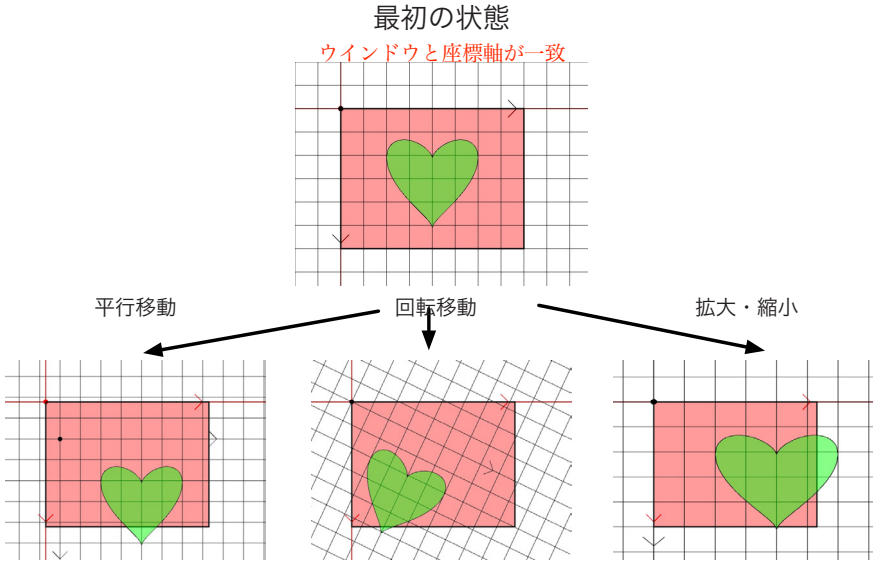


図 6-5

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} f \\ g \end{pmatrix}$$

は余り一般的な座標変換は扱わずに、次の 4 種類の特別な座標変換とそれを組み合わせたものを扱います。CG では、最初の 3 つを良く使用します。

1. 平行移動：translate
2. 回転：rotate
3. 拡大・縮小：scale
4. 剪断（傾け）：shear



画用紙を傾けて絵を描き、その画用紙をものと向きに戻すと、傾いた絵になっていますよね。

座標変換の詳しい話は、線形代数学やグラフィクス基礎論で学びます。

正確には、逆行列をもっているようなものに限定されますが。

Processing でも、applyMatrix 関数を利用すると一般的な座標変換も取り扱うとこが出来ます。

Processing では、scale は余り利用することが少ないような気が。図形の枠線の大きさも拡大・縮小されてしまうので。

実は、Processing では 3 次元物体の表示を行うことができます。そんため、3 次元空間での座標変換の関数も持っています。関数名はほとんどおなじです。

表 6-6 座標変換に関わる関数その 1

関数	意味
translate(x,y)	「現在の原点」を移動させる関数。「現在の X 軸」方向に x、「現在の Y 軸」方向に y だけ「現在の原点」を移動させる。移動した先が新たな「現在の原点」となります。座標軸の向きは変わりません。
rotate(angle)	「現在の原点」を中心に X 軸と Y 軸を回転させる関数。引数 angle は、回転角度の指定はラジアンで行います。
scale(s)	「現在の座標軸」を s 倍する。つまり、現在の 1 の長さが s となる。
scale(sx,sy)	「現在の X 軸」の長さを sx 倍、「現在の Y 軸」の長さを sy 倍する。
randians(angle)	angle 度をラジアンでの値に変換する関数。

平行移動

平行移動は一番簡単な座標変換です。平行移動では、原点の位置を移動させます。原点の位置を移動させるだけですので、X 軸や Y 軸の向きは変化しません。座標変換を行う関数を実行すると、その度に座標軸が移動していきます。サンプル 6-15 は、これを確認するプログラムです。

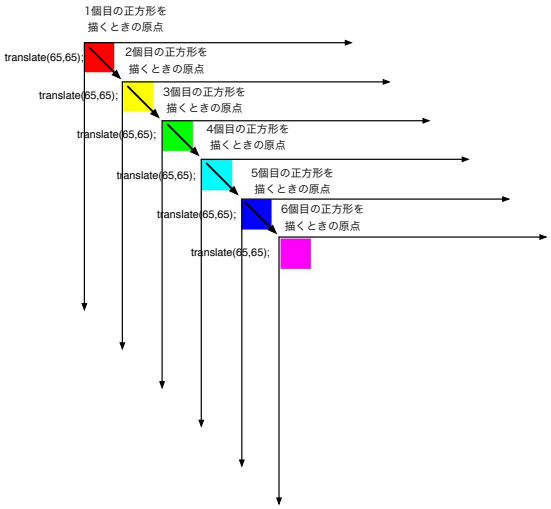


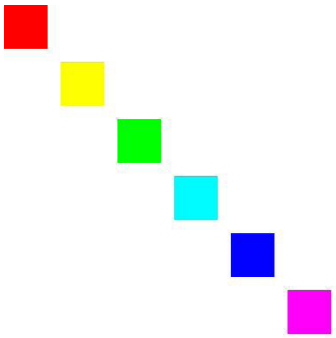
図 6-6

数学的には、変換の合成（簡単にいうと、行列のかけ算）になっています。

実は、この座標軸の移動ですが、image 関数などの実行にも有効です。

translate を利用した例その 1 サンプル 6-15

```
size(400,400);
colorMode(HSB,359,99,99);
background(0,0,99);
noStroke();
for(int x=0;x <6;x++){
  fill(60*x,99,99);
  rect(0,0,50,50);  // 原点 (0,0) で正方形を描く
  translate(65,65); // 原点を X 軸方向に 65、Y 軸方向に 65 移動させる
}
```



サンプル 6-15 は for 命令を利用して、色を変えながら 6 個の正方形を描くプログラムです。for 命令の繰り返し部分の rect 関数は、毎回同じ場所 (0,0) で正方形を描いています。ところが、このプログラムを実行してみると、異なった場所に正方形を描かれています。

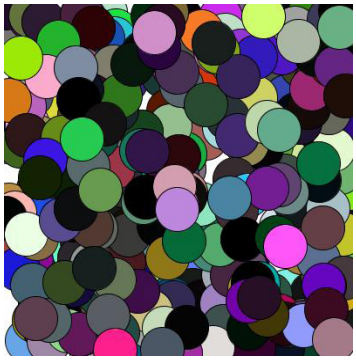
これは、なぜでしょうか？それは、rect 関数の後に実行している translate 関数に理由があります。つまり、rect 関数で正方形を描いた後に、「translate(65,65);」で原点の位置 (0,0) を動かしているからです。

つまり、一番目の赤色の正方形を描くときには、通常のウインドウの左上に原点がある状態で rect(0,0,50,50) が実行されるので、左上に正方形が描画されます。その後、transalte(65,65) が実行されるので、「現在の原点」（ウインドウの左上）が「現在の X 軸」方向に 65、「現在の Y 軸」方向に 65 だけ移動します。つまり、ウインドウの左上から横方向に 65、縦方向に 65 だけ移動した場所に「現在の原点」が移動します。ですから、2 番目の黄色の正方形を描くときには、この「現在の原点」を基準に描画を行うので、rect(0,050,50) を実行すると、赤色の正方形より少し右下の部分に描かれることになります。その後、再び translate(65,65) が実行されるので、「現在の原点」が「現在の X 軸」方向に 65、「現在の Y 軸」方向に 65 だけ移動します。つまり、ウインドウの左上から横方向に 130、縦方向に 130 だけ移動した場所に「現在の原点」が移動します。3 番目の緑色の正方形を描くときには、この「現在の原点」を基準に描画を行うので、rect(0,050,50) を実行すると、黄色の正方形より少し右下の部分に描かれることになります。この後、translate(65,65) を実行するので、「現在の原点」が「現在の X 軸」方向に 65、「現在の Y 軸」方向に 65 だけ移動します。つまり、ウインドウの左上から横方向に 195、縦方向に 195 だけ移動した場所に「現在の原点」が移動します。このような操作を繰り返すので、徐々に右下に移動しながら、正方形が描かれるようになります。

translate 関数を利用した、別のサンプルを示します。このサンプル 6-16 では、ellipse 関数を利用して、毎回「現在の原点」に直径 50 の円を描いています。ただし、ellipse 関数を実行するまえに、translate 関数を呼び出して、「現在の原点」の位置を変更しています。そのために、異なった位置に円が描画されています。なお、draw 関数の一番先頭では、「現在の原点」はウインドウに左上の位置に初期化されています。

translate を利用した例その 2 サンプル 6-16

```
void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
  background(0,0,99);
}
// 次ページに続く
```



```
void draw(){
// この場所では、「現在の原点」はウインドウに左上に初期化されている
  fill(random(360),random(100),random(100)); // 色はランダム
  transalte(random(width),random(height));//「現在の原点」を移動
  ellipse(0,0,50,50); //「現在の原点」を中心に円を描画
}
```

回転移動

平行移動だけだと、図形を描画する位置を変更するだけ対応出来るので、あまりありがたみがわきません。ここで説明をする回転移動と組み合わせると描画できる形状がより豊富になります。

rotate 関数による回転移動は、「現在の原点」を中心として座標軸の回転を行います。回転を考える際には、回転方向の向きが問題となりますが、Processing では図 6-7 のようになっています。

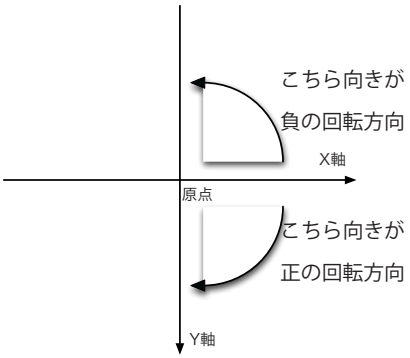


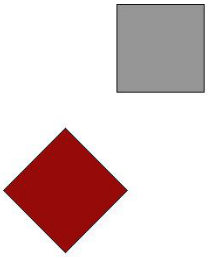
図 6-7 回転の向き

rotate 関数を実行しただけでは、「現在の原点」の位置は移動しません。座標軸の傾きだけが変わります。

まず、シンプルな rotate 関数を使用したサンプル 6-17 を示します。

rotate を利用した例 サンプル 6-17

```
size(400,400);
background(255);
stroke(0);
fill(150);
rect(200,0,100,100);
rotate(PI/4);
fill(150,10,10);
rect(200,0,100,100);
```



サンプル 6-17 では、2 回 rect(200,0,100,100) を実行することで、2 つの正方形を描いていますが、異なった場所に描かれています。1 番目の灰色の正方形が描かれたときには、座標変換の関数を一切実行していないので、ウインドウの左上に原点があり、横方向の左から右方向に X 軸が、上から下方向に Y 軸が位置しています。その座標軸の状態で rect(200,0,100,100) を実行するので、灰色の正方形が、ウインドウの上に水平の描画がされます。この後、rotate(PI/4) を実行するので、「現在の原点」の位置は変わりませんが、「現在の原点」を中心に PI/4(=45 度) だけ、X 軸と Y 軸を回転させます。その後、再び rect(200,0,100,100) を実行するので、画面の中央部分に傾いた赤い正方形が描かれることにな

前にも説明しましたが、PI は円周率を表す定数です。

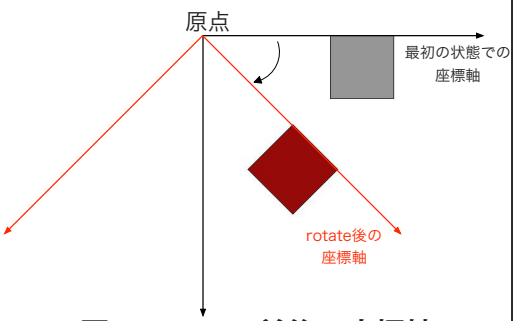


図 6-8 rotate 前後の座標軸

ります。

rotate 関数は「現在の原点」を中心に「現在の座標軸」を回転させるので、rotate 関数単体では、使い道が限られてしまいます。座標変換を行う関数を実行するたびに、「現在の原点」や「現在の座標軸」が移動していくので、translate 関数を利用して「現在の原点」を回転の中心に移動させ、その後、rotate 関数を実行するとで、任意の場所で座標軸の回転を行うことが出来ます。

サンプル 6-18 は、マウスカーソルの位置で長方形を回転させるものです。

translate 関数で、回転の中心となる「現在の原点」を適切な場所に移動させます。その後、rotate 関数で「現在の座標軸」を回転させます。これにより、希望する場所での回転を実現できます。

translate と rotate を利用した例その 1 サンプル 6-18

```
float angle = 0; //「現在の座標軸」の回転角度

void setup(){
  size(400,400);
  rectMode(CENTER);
  smooth();
  fill(128);
  stroke(0);
}
void draw(){
// この時点では、「現在の原点」と「現在の座標軸」は初期位置
  background(255);
//「現在の原点」をマウスカーソルの位置に移動
  translate(mouseX,mouseY);
  rotate(angle);  //「現在の座標軸」を angle だけ回転させる
  rect(0,0,50,100);
  angle = angle + PI/180; // 回転角度を増やす
}
```

サンプル 6-18 では、最初に translate 関数で「現在の原点」の位置をマウスカーソルの場所に移動させます。その後、rotate 関数で「現在の座標軸」を angle だけ回転させます。その後、回転角度を示す変数 angle の値を少しだけ増加 (PI/180 = 1 度) させます。draw 関数が呼び出されるたびに、回転角度が増加するので、マウスカーソルの位置で長方形が回転しているように見えます。

サンプル 6-19 では、translate(width/2, height/2) で「現在の原点」をウインドウの中心に移動させます。その後、色を変えながら ellipse(150,0,60,60) で円を描画します。円の描画後、rotate 関数を使って、24 度ずつ「現在の原点」を中心に「現在の座標軸」を回転させます。これにより、円周上に円を配置することが出来ます。図 6-9 では、3 種類の座標軸が表示されています。1 つ目は tranlsate 直後の

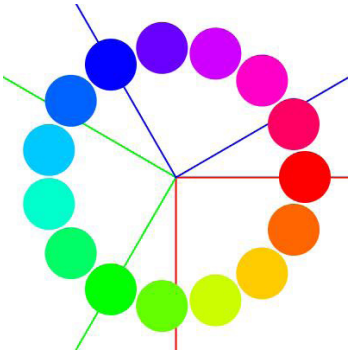
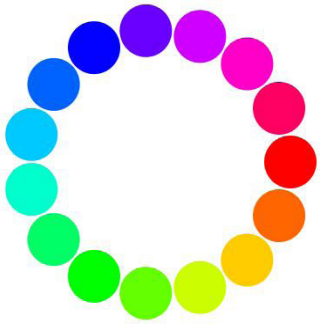


図 6-9 rotate 後の座標軸

座標軸、２つ目は rotate 関数を 5 回実行した直後の座標軸、3 つ目は rotate 関数を 10 回実行した直後の座標軸となっています。

translate と rotate を利用した例その 2 サンプル 6-19

```
size(400,400);
colorMode(HSB,359,99,99);
smooth();
background(0,0,99);
noStroke();
//「現在の原点」をウインドウの中心に移動
translate(width/2,height/2);
for(int angle = 0;angle < 360;angle += 24){
  fill(angle,99,99); // 描画色の変更
  ellipse(150,0,60,60); // 円の描画
  rotate(radians(24)); // 現在の座標軸を 24 度回転させる
}
```



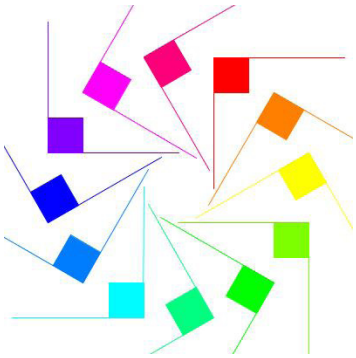
回転角度はラジアンで指定する必要があるので、radians 関数を用いて、ラジアンに変換しています。

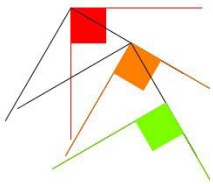
この処理は円運動の簡易的なモデルになっています。

サンプル 6-20 は、translate 関数と rotate 関数を上手く利用して、画像を作成したものです。今までのサンプルプログラムとは異なり、「rotate → translate」が組みになって、繰り返し実行しています。つまり、X 軸方向にちょっと移動して、少し向きを変えるという処理になっています。

translate と rotate を利用した例その 3 サンプル 6-20

```
size(400,400);
colorMode(HSB,359,99,99);
smooth();
noStroke();
background(0,0,99);
translate(240,60);  // スタート位置に移動
for(int i=0;i<12;i++){
  fill(i*30,99,99); // 描画色の設定
  stroke(i*30,99,99);
  line(0,0,150,0); // 座標軸を表示
  line(0,0,0,150);
  rect(0,0,40,40);  // 正方形の描画
  rotate(radians(30));  //「現在の原点」を中心に座標軸を回転
//「現在の原点」を「現在の X 軸」の正の方向に 80 移動
  translate(80,0);
}
```





途中までの描画状況

拡大・縮小

scale 関数は、translate 関数や rotate 関数に比べると、利用の機会は少ないですが、座標軸の拡大・縮小が行えます。この関数は、座標軸の目盛りを拡大・縮小します。scale 関数の実行結果は、「現在の座標軸」に対して有効です。従って、scale 関数を呼び出す前の図形は関わりません。「現在の座標軸」の目盛りの大きさを変えてしまうの

で、線の太さなども変わってしまいます。一応、サンプルをのせておきます。

scale を利用した例 サンプル 6-21

```
size(400,400);
smooth();
background(255);

for(int x = 80; x < width;x += 80){
  fill(x % 256,10,10);
  ellipse(x,height/4,50,50);
}

scale(0.5); //「現在の座標軸」の目盛りを半分の長さにする
for(int x = 80; x < width;x += 80){
  fill(x % 256,10,10);
  ellipse(x,height,50,50);
}
```

座標軸の記憶

何回も座標変換を行っていくと、どんどん「現在の座標軸」が移動していきます。プログラムによっては、「前の座標軸」の状態に戻りたいということがおきます。これを実現するために、Processing 言語では pushMatrix 関数と popMatrix 関数が用意されています。pushMatrix 関数は、「現在の座標軸」の状況を一時的に記憶します。逆に、popMatrix 関数は、「現在の座標軸」を一時的に記憶されている「過去の座標軸」に変更します。

表 6-7 座標変換に関わる関数その 2

関数名	意味
pushMatrix()	「現在の座標軸」を一時的に記憶する。
popMatrix()	「現在の座標軸」を一時的に記憶されている「過去の座標軸」に変更します。つまり、pushMatrix 関数で記憶させた座標軸に戻すということを行います。
resetMatrix()	「現在の座標軸」を初期状態に戻します。

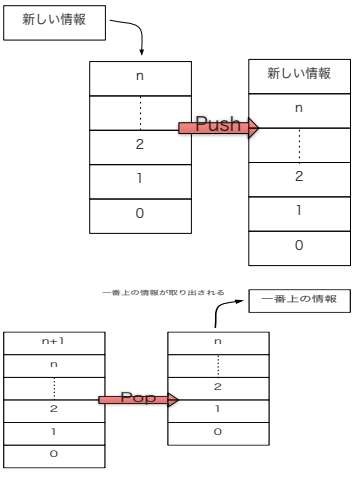
pushMatrix 関数と popMatrix 関数による座標軸の記憶は、普通の変数によるものとは、ちょっと異なっています。次のような制限があります。

1. popMatrix 関数の説明で書かれている「過去の座標軸」とは、この popMatrix 関数を呼び出す直前に呼び出された pushMatrix 関数が保存した「現在の座標軸」です。
2. popMatrix 関数を実行すると「過去の座標軸」は消えてしまいます。
3. pushMatrix 関数と popMatrix 関数が一対のものになっている。

この辺りの話をやり出すと、ちょっと面倒なので、今回はあまり

このあたりの設計は、OpenGL と呼ばれる 3D-CG 用の API の影響を強く受けています。

このような情報の記憶の仕方をスタック (stack) と呼んでいます。スタックを利用して、情報を記憶することをプッシュ (push)、記憶されている情報を取り出すことをポップ (pop) と呼びます。



深入りはしません。

pushMatrix と popMatrix 利用した例 サンプル 6-22

```
size(400,400);
smooth();
colorMode(HSB,359,99,99);
background(0,0,99);
noStroke();

//「現在の原点」をウインドウの中心に移動させる
translate(width/2,height/2);
float len = 10;
for(int angle=0;angle < 1080;angle += 24){
  pushMatrix(); //「現在の座標軸」を一時的に記憶
  rotate(radians(angle));  //「現在の座標軸」を回転させる
  translate(len,0); //「現在の原点」を X 軸方向に len だけ移動させる
  fill(angle % 360,99,99);// 塗りつぶし色の決定
  ellipse(0,0,20,20); // 円の描画
  popMatrix();// 直前に記憶した座標軸の状況を「現在の座標軸」にする
  len *= 1.1; // 移動量を 1 割増やす
}
```

resetMatrix 利用した例 サンプル 6-23

```
size(200,200);

smooth();
background(255);
translate(width/2,height/2); //「現在の原点」をウインドウ中央に移動
fill(255,10,10); //「現在の原点」を中心に赤色の円を描く
ellipse(0,0,100,100);

resetMatrix();//「現在の原点」を初期状態（ウインドウの左上）に移動
fill(10,255,10);//「現在の原点」を中心に緑色の色の円を描く
ellipse(0,0,100,100);
```

応用：時計の制作

今回学習した内容に加えて、時間を取得する方法があれば、時計を作ることが出来ます。Processing では、現在の時刻を取得できる関数が用意されています。

表 6-8 時間に関わる関数

関数名	意味
hour()	現在の時間の時（0 ～ 23 の整数）を返す関数。
minute()	現在の時間の分 (0 ～ 59 の整数) を返す関数。
second()	現在の時間の秒 (0 ～ 59 の整数) を返す関数。
year()	現在の年を返す関数。
month()	現在の月 (1 ～ 12 の整数) を返す関数。

関数名	意味
day()	現在の日 (1 ～ 31 の整数) を返す関数。
millis()	プログラムを実行してからの時間をミリ秒単位で返す。

1 秒 =1000 ミリ秒です。

サンプル 6-24 ではデジタル時計のサンプルです。text 関数では、文字列 (String) しか表示できません。そこで、str 関数を利用して、強制的に int 型を String 型に変換しています。実は、draw 関数の中は、サンプル 6-25 のように書いても同じ動作となります。これは、式「hour()+":"+minute()+":"+second()」を見ると、数値データ同士の加算ではなく、文字列の連結と判断できるので、自動的に hour() などの値を String 型に変換してくれます。

hour()+":" などのように、数字と文字列に対して、+ を計算しようとしてるので、+ は加算ではなく、文字列の連結と判断しています。

デジタル時計 サンプル 6-24

```
PFont font;

void setup(){
  size(400,64*3);
  smooth();
  font = loadFont("Helvetica-128.vlw");
  textFont(font,64);
  textAlign(CENTER);
  rectMode(CENTER);
  fill(0);
}

void draw(){
  background(255);
  String h = str(hour()); // 時間を String 型に変換
  String m = str(minute()); // 分を String 型に変換
  String s = str(second()); // 秒を String 型に変換
  String t = h + ":" + m + ":" + s; // 表示する文字列作成
  int hs = textAscent()+textDescent();// 表示する文字列の高さを取得
  text(t,width/2,height/2,width,hs);
}
```

デジタル時計 の一部サンプル 6-25

```
void draw(){
  background(255);
  // 表示する文字列作成
 String t = hour() + ":" + minute() + ":" + second();
  int hs = textAscent()+textDescent();
  text(t,width/2,height/2,width,hs);
}
```

サンプル 6-26 では、アナログ時計の秒針の部分だけを表示するサンプルです。0 秒時には鉛直上方向に秒針が来る必要があります。初期状態での座標軸は、原点を中心に -PI/2 だけ回転させた位置にあることに注意が必要です。

秒針だけのアナログ時計 サンプル 6-26

```
void setup(){
  size(400,400);
  smooth();
}
void draw(){
  background(255);
//「現在の原点」をウインドウの中心に移動
  translate(width/2,height/2);
  float angle = -90+6*second();// 現在の秒から、秒針の角度を求める
  rotate(radians(angle)); //「現在の X 軸」を秒針方向に傾ける
  fill(50);
  triangle(0,5,180,0,0,-5); // 秒針を三角形として表示
}
```

おまけのサンプル

座標変換を利用すると、複雑な図形を表示出来るようになります。やっていることは単純で、「現在の座標軸」を色々と移動させながら、line(0,0,0,-100) で直線を描いているだけです。

フラクタルと呼ばれる図形の描画の基礎となるプログラムです。フラクタルは CG における自然物を再現するにしばしば用いられる手法です。

おまけ サンプル 6-23

```
size(400,400);
smooth();
stroke(0);
background(255);
translate(width/2,height);

line(0,0,0,-100);
translate(0,-100);
pushMatrix(); // (a)
rotate(-PI/6);
line(0,0,0,-100);
translate(0,-100);
pushMatrix(); // (b)
rotate(-PI/6);
line(0,0,0,-100);
popMatrix(); // (b)
// 右上に続く
```

```
rotate(PI/6);
line(0,0,0,-100);
popMatrix(); //(a)
rotate(PI/6);
line(0,0,0,-100);
translate(0,-100);
pushMatrix(); // (c)
rotate(-PI/6);
line(0,0,0,-100);
popMatrix(); // (d)
rotate(PI/6);
line(0,0,0,-100);
```

情報メディア基盤ユニット用資料（2016 年度第 7 章）

Processing 言語による情報メディア入門

座標変換 (続き) と関数 (その 1)

神奈川工科大学情報メディア学科　佐藤尚

プログラムが動かない - Σヽ(` д ´ ;)ノうぉぉぉ！となる前に

サンプルのプログラムを入力すると、上手く実行出来ないことがあります。その時に、チェックした方がよい点を挙げておきます。これ以外にも、原因があると思いますが、とりあえず気がついた点です。基本的には、「ちゃんとプログラムは入力されていますか？」、「ちゃんと正しい場所にデータはコピーされていますか？」です。

表 7-1 プログラムが動作しないときのチェックリスト

挙動	チェックポイントなど
なんだか動かない “The function 関数名 does not exist.” と表示される。	プログラムは正しく入力されていますか？特に、関数名は正しく入力されていますか？大文字と小文字は正しく区別されていますか？ Processing では大文字と小文字を区別します。
なんだか動かない “The field Component. 変数 is not visible.” “Cannot find anything named 変数名 ” ”Cannot find class or type named 名前 ” などと表示される。	プログラムは正しく入力されていますか？特に、変数名やデータ型の名称は正しく入力されていますか？大文字と小文字は正しく区別されていますか？ Processing では大文字と小文字を区別します。
なんだか動かない “Syntax error, maybe a missing right parenthesis?” と表示さる。	どこかに ”)” を忘れていませんか？特に、黄色くなっている行の辺りです。
なんだか動かない “Unexpected token: 文字列 ” と表示される。	どこかに ”(”、"{" や "}" を忘れていませんか？特に、黄色くなっている行やその少し上の行の辺りです。
なんだか動かない “The method 関数名 (……” などと表示がされる。	引数同士の区切りが "," ではなく、"." になっていませんか？小数点(.)が "," になっていませんか？引数はあっていますか？特に、黄色くなっている行の辺りです。
なんだか動かない unexpected char: '\'” と表示される。	変数名や関数名などに全角文字を使っていませんか？プログラムの空白に全角スペースを利用していませんか？

人間は間違える生き物です。間違えなく入力したと思っても、普通は間違いがあります。本人は間違いなく入力したと思い込んでいるので、間違えを見つけることが出来ません。情報システムのデザインをするときには、人間は間違えるかもという視点を持ってデザインすることは重要です。
ところで、ファミコンとスーパーファミコンのカセットの違いを知っていますか？ 3.5 インチのフロッピーディスクの形を知っていますか？どちらも、古すぎる話でしょうか？

エディタ上で CTRL-t を（control キーと t キーを同時に）押すと、プログラムのインデントなどを自動でつけてくれます。

(と) の対応や { と } の対応はエディタ上で簡単にチェックが出来ます。この機能を上手く使って下さい。
例えば、(の辺りにカーソルを持って行くと、対応する) が、四角形で囲まれます。

```
void setup()
img = loadImage("test.jpg");
size(400,400);
}
```

ここです。

挙動	チェックポイントなど
なんだか動かない “Syntax error, maybe a missing semicolon?” などと表示されます。	セミコロン ; を忘れていませんか？黄色で表示されている部分またはその少し前にセミコロンを忘れている場所はありませんか？
なんだか動作がおかしい 指定した大きさのウインドウが開かない。	「void setup(){」の部分で、setup の綴りを間違えていませんか？
なんだか動作がおかしい ウインドウが灰色のままで、画像が表示されない。	「void draw(){」の部分で、draw の綴りを間違えていませんか？
途中でプログラムが止まる ”NullPointerExceptoin” と表示される。	loadImage 関数で指定している画像ファイル名は正しいですか？指定の場所に画像ファイルがありますか？ 通常、表示したい画像ファイルは Sketch > Show Sketch Folder で表示されるフォルダ内の data フォルダ内に保存します。
途中でプログラムが止まる ”A null PFont was passed …” などと表示される。	きちんとフォントデータが読み込まれていますか？
なんだか動作がおかしい **or** **途中でプログラムが止まる** ”Could not load font vlw ファイル名 .” と表示される。	loadFont で指定している vlw ファイルのファイル名は正しいですか？指定の場所に vlw ファイルが保存されていますか？ Tools > Create Font で vlw ファイルを作成していますか？ 通常、ｖｌｗファイルは Sketch > Show Sketch Folder で表示されるフォルダ内の data フォルダ内に置いておきます。

座標変換の続き

座標軸の移動の続きです。translate 関数や rotate 関数単体での動きはわかりやすいと思います。

translate 関数は、translate 関数の引数で指定された値分だけ、「現在の原点」を移動させます。この時に、移動方向は、「現在の座標軸」が基準となります。図 7-1 のように、黒い座標軸が「現在の座標軸」とすると、これを基準に移動を行います。「現在の座標軸」が傾いていれば、傾いた方向に移動することなります。

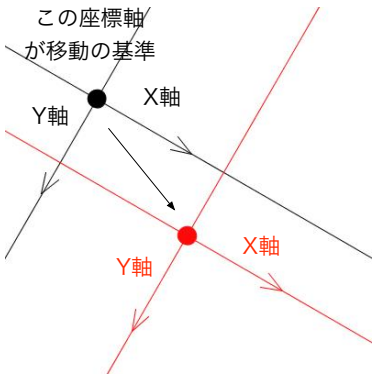


図 7-1 translate での座標軸の移動

rotate 関数も translate 関数と同じように、「現在の座標軸」を基準に回転を行います。回転の中心は、「現

在の原点」です。図 7-2 のように、黒い座標軸が「現在の座標軸」とすると、これを基準に回転を行います。

setup 関数と draw 関数を使ってプログラムを作成する際には、draw 関数の先頭では、「現在の座標軸」は初期状態（原点はウインドウの左上、水平に X 軸、垂直に Y 軸）となります。

translate 関数や rotate 関数は単体で用いるより、組み合わせて使用することが普通です。ある場所で、物体を回転させたい場合には、回転の中心としたい場所に、translate 関数を用いて「現在の原点」に移動させ、その後に、rotate 関数を使えば、好きな場所で物体を回転させることができます。

translate 関数や rotate 関数を使って「現在の座標軸」を動かしていくと、「現在の座標軸」を初期状態に移動させたいことがあります。これを行うのが、resetMatrix 関数です。この関数を実行すると、「現在の座標軸」が初期状態に戻ります。

また、「途中の座標軸」の状態を保存をしておきたいこともあります。Processsing では、どこかの変数に状態を保存するのでなく、行列スタックと呼ばれる場所に保存します。「現在の座標軸」を保存するのに使用するのが pushMatrix 関数で、「現在の座標軸」を保存されている座標軸の状態に戻す際に使用するのが popMatrix 関数です。pushMatrix 関数と popMatrix 関数はペアになって使用します。もし、ペアになって使用されていないと、直ぐにエラーが発生します。

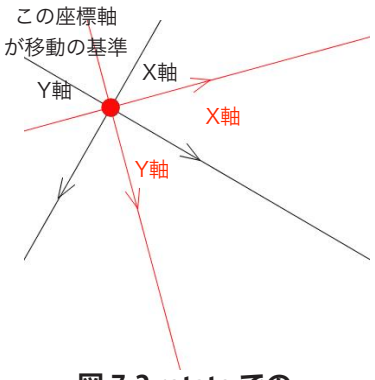


図 7-2 rotate での座標軸の移動

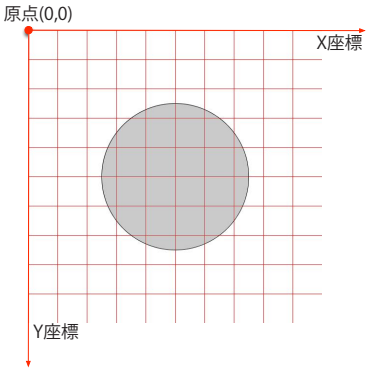


図 7-3 初期状態の座標軸

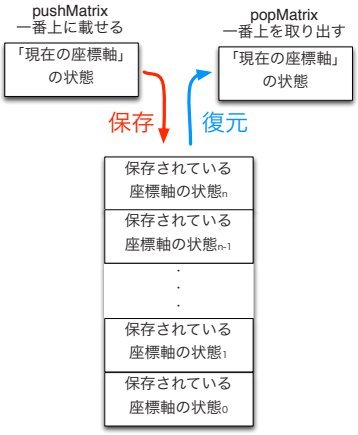


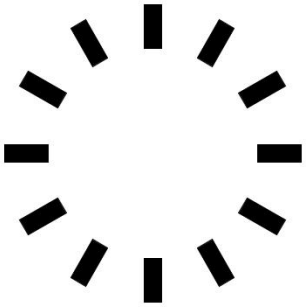
図 7-4 行列スタック

前回のサンプル 6-18 が、このことを利用して、マウスカーソルの周りで長方形を回転させています。

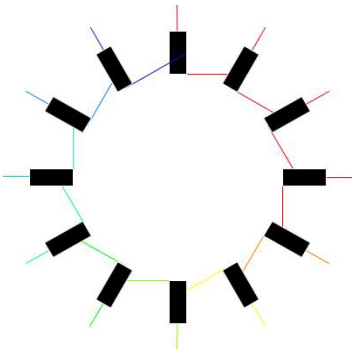
スタック (stack) は、コンピュータのプログラムでは良く出てくるデータを蓄えるための考え方です。最後に入れたデータ (push) が、最初に出てくる (pop) という動作をするようなものです。
学食などにある、トレーを沢山つんである山を想像すると良いかもしれません。push はデータを書いた紙をトレーの山の一番上に載せることに相当します。pop はトレーの山の一番上にあるトレーを取り除き、そこに書かれているデータを読み出すことに相当します。

translate と rotate を利用した例その 1 サンプル 7-1

```
size(400,400);
smooth();
stroke(0);
fill(0);
background(255);
// ウインドウの中心に「現在の座標軸」を移動
translate(width/2,height/2);
```



```
for(int i=0;i < 12;i++){
  pushMatrix(); //「現在の座標軸」をスタックの一番上に載せる
  rotate(radians(-90+30*i)); //「現在の座標軸」を回転させる
  translate(120,0); //「現在の原点」を移動させる
  rect(0,-10,50,20); // 長方形を描画する
  // 現在の座標軸」をスタックの一番上にある座標軸の状態に変更する
  popMatrix();
}
```



サンプル 7-1 は pushMatrix 関数や popMatrix 関数を使わなくても書くことが出来ます。

translate と rotate を利用した例その 1' サンプル 7-2

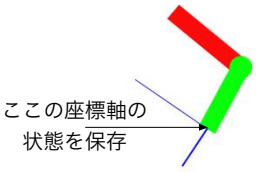
```
size(400,400);
smooth();
stroke(0);
fill(0);
background(255);
for(int i=0;i < 12;i++){
// ウインドウの中心に「現在の座標軸」を移動
  resetMatrix();
  translate(width/2,height/2);
  rotate(radians(-90+30*i)); //「現在の座標軸」を回転させる
  translate(120,0); //「現在の原点」を移動させる
  rect(0,-10,50,20); // 長方形を描画する
}
```

サンプル 7-1 は pushMatrix 関数と popMatrix 関数を使用しなくても、簡単に同じ動作をするプログラムを書くことができます。しかし、次の様なロボットアームのような動作をするプログラムでは、pushMatrix 関数と popMatrix 関数を使用することなくプログラムを作成することは困難です。

translate と rotate を利用した例その 2 サンプル 7-3

```
float angle;
color arm1,arm2,arm3;

void setup(){
  size(400,400);
  smooth();
  angle = 0;
  arm1 = color(255,10,10);
  arm2 = color(10,255,10);
  arm3 = color(10,10,255);
}
```

```
void draw(){
  background(255);
  translate(width/2,height/2);
  rotate(radians(angle));
  stroke(arm1);
  fill(arm1);
  rect(0,-10,100,20);
  translate(100,0);
  stroke(arm2);
  fill(arm2);
  ellipse(0,0,25,25);
  rotate(radians(2*angle));
  rect(0,-10,80,20);
  translate(80,0);
  stroke(arm3);
  pushMatrix();
  rotate(radians(6*minute()));
  strokeWeight(2);
  line(0,0,50,0);
  popMatrix();
  rotate(radians(6*second()));
  strokeWeight(1);
  line(0,0,100,0);
  angle += 0.1;
}
```



今週の演習問題で出題されているアナログ時計を作る際には、draw 関数の内部で

1. pushMatrix 関数を実行
2. 時間を表す針を描画
3. popMatrix 関数を実行
4. pushMatrix 関数を実行
5. 分を表す針を描画
6. popMatrix 関数を実行
7. pushMatrix 関数を実行
8. 秒を表す針を描画
9. popMatrix 関数を実行

のような順番で処理をおこなって行きます。

7 番の pushMatrix 関数と 9 番の popMatrix 関数は実行しなくても大丈夫です。

変数の有効範囲

情報メディア学科で、鈴木先生と言うと、鈴木浩先生のことを指します。情報工学科で、鈴木先生と言うと、鈴木孝幸先生のことを指します。この二人が同時にいる場所で、鈴木先生と言うと、どちらの鈴木先生を指しているのかわからなくなります。お父さんというと、家によって誰を指すのかが変わります。このように、ある名前がどこまで有効かを決めないと、混乱してしまいます。そこで、Processing などのプログラミング言語でも、変数の有効範囲の規則


どちらも鈴木先生

が決まっています。

変数の有効範囲により、次の 2 つの区別があります。

1. 大域変数（グローバル変数）
2. 局所変数（ローカル変数）

大域変数は基本的にプログラム中のどこでも利用することができます。一方、局所変数は、その変数を使える場所が限られている変数です。大域変数と局所変数の宣言の仕方に違いはなく、どの場所でその変数を宣言したかで決まります。

今までのサンプルでは、基本的にプログラムの先頭で変数の宣言を行ってきました。このようにプログラムの先頭で変数を宣言すると大域変数として扱われます。一方、関数内部の変数を使い始めたい場所で、変数宣言を行うと、局所変数として扱われます。局所変数は使える場所は、基本的に局所変数を宣言したブロック内部（{〜}）となります。

例えば、サンプル 7-4 では、プログラムの先頭で変数宣言を行っている int 型の変数 xPos は大域変数となります。また、「int x=xPos;」となっている int 型の変数 x は局所変数となります。そして、変数 x の有効範囲は、変数宣言を行ったブロックの内部の、変数宣言を行った以降の部分（赤色の文字の部分）となります。変数 xPos は大域変数なので、setup 関数の内部や draw 関数の内部の両方で利用することが出来ます。

この規則のことを、スコープ規則やスコープルールと呼ぶことがあります。

有効範囲とは、その変数が使える場所という意味です。

後で、メンバ変数というものが出てきます。

対応する { と } の間がブロックです。

Processing の変数の有効範囲に関する知識は、そのまま C++ 言語や Java 言語でも使えます。しかし、C++ 言語では少し異なる部分があります。C 言語では、もっと異なる部分が増えます。

大域変数と局所変数の例 1 サンプル 7-4

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  int x = xPos; // この変数は局所変数です。有効範囲は赤色の部分のみ。
  while(x < width){
    ellipse(x,height/2,20,20);
    x += 10;
  }
  xPos--;
}
```

このサンプルは、for 命令を使っても書くことが出来ます。これを行ったのものがサンプル 7-5 です。このサンプルでは、変数 xPos は大域変数です。「for(int x=xPos;…」の部分で宣言している変数 x は局

所変数となります。この変数 x の有効範囲は「for(int x=xPos;…){ 〜 }」の部分（赤字の部分）になります。

大域変数と局所変数の例 2 サンプル 7-5

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  for(int x=xPos;x < width;x += 20){ // 変数 x は局所変数です。
    ellipse(x,height/2,20,20);
  }
  xPos--;
}
```

for 命令の場合には、for 命令全体でブロックを作っているように動作となっています。ちょっと注意が必要かも知れません。

この 2 つのサンプルは同じ動作をするものですが、局所変数 x を宣言する場所が少し異なっているので、サンプル 7-4 は次の様に while 命令終了後に変数 x の値を表示させることが出来ますが、サンプル 7-5 では for 命令終了後に変数 x の値を表示させる命令を追加するとエラーとなります。

大域変数と局所変数の例 3 サンプル 7-4'

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  int x = xPos; // この変数は局所変数です。有効範囲は赤色の部分のみ。
  while(x < width){
    ellipse(x,height/2,20,20);
    x += 10;
  }
  println(x);
  xPos--;
}
```

大域変数と局所変数の例 4 サンプル 7-5'

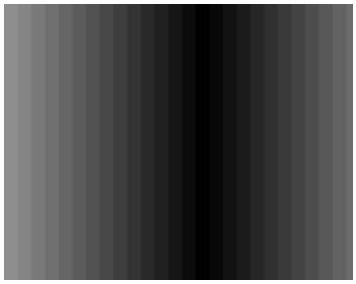
```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  for(int x=xPos;x < width;x += 20){ // 変数 x は局所変数です。
    ellipse(x,height/2,20,20);
  }
  println(x);
  xPos--;
}
```

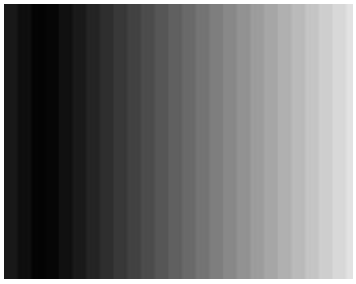
赤色の文字の部分が局所変数 x の有効範囲です。この場所は変数 x の有効範囲を出ているので、エラーメッセージ" The filed Componet.x is not visible." 表示されます。

サンプル 7-6 では、大域変数は使用していませんが、局所変数 x と gray を使用しています。局所変数 gray は 2 箇所で宣言していますが、異なるブロックで宣言しています。従って、名前の混乱を引き起こすことがないので、このような使い方が可能です。

大域変数と局所変数の例 5 サンプル 7-6

```
void setup(){
  size(255,200);
  smooth();
}

void draw(){
  background(0);
  noStroke();
  int x= 0; // 局所変数、赤字の部分で有効
  while(x <= mouseX){// (1)
    int gray = mouseX-x; // この gray は、(1) の while 命令内で有効
    fill(gray);
    rect(x,0,10,height);
    x += 10;
  }
  while(x < width){  // (2)
    int gray = x-mouseX; // この gray は、(2) の while 命令内で有効
    fill(gray);
    rect(x,0,10,height);
    x += 10;
  }
}
```





局所変数 x と同じように、サンプル 7-6 の局所変数 gray をサンプル 7-7 のように使用するとエラーとなります。これは、局所変数 gray を宣言した場所が、while 命令のブロックの中なので、局所変数 gray の有効範囲は、このブロック（赤字の部分）に限られるためです。

大域変数と局所変数の例 5' サンプル 7-7

```
void setup(){
  size(255,200);
  smooth();
}
void draw(){
  background(0);
  noStroke();
  int x= 0;
  while(x <= mouseX){// (1)
    int gray = mouseX-x; // この gray は、(1) の while 命令内で有効
    fill(gray);
    rect(x,0,10,height);
    x += 10;
  }
  while(x < width){ // (2)
    fill(gray);   // 異なるブロックで宣言された変数なので使えない
    rect(x,0,10,height);
    x += 10;
  }
}
```

変数の有効範囲外になると、変数に記憶されていた情報は消えてしまいます。

この場所は変数 gray の有効範囲を出ているので、"Cannot find anything named "gray"" というエラーメッセージが表示されます。

サンプル 7-7 の while 命令をサンプル 7-8 のように for 命令に書きかえるとエラーとなります。

大域変数と局所変数の例 5'" サンプル 7-8

```
void setup(){
  size(255,200);
  smooth();
}
void draw(){
  background(0);
  noStroke();
  for(int x=0;x <= mouseX;x += 10){ // 変数 x は赤色の部分で有効
    int gray = mouseX-x;
    fill(gray);
    rect(x,0,10,height);
  }
// 変数 x は異なるブロックで宣言されているの無効
  for(;x < width;x+=10){
    int gray = x - mouseX;
    fill(gray);
    rect(x,0,10,height);
  }
}
```

この場所は変数 x の有効範囲を出ているので、エラーメッセージ "The filed Componet.x is not visible." 表示されます。

この場合には、サンプル 7-9 のように局所変数 x を宣言すると、エラーとならずサンプル 7-6 と同じ動作を行います。

大域変数と局所変数の例 5"" サンプル 7-9

```
void setup(){
  size(255,200);
  smooth();
}

void draw(){
  background(0);
  noStroke();
  int x; // 変数 x は赤字の部分で有効
  for(x=0;x <= mouseX;x += 10){
    int gray = mouseX-x;
    fill(gray);
    rect(x,0,10,height);
  }
  for(;x < width;x+=10){
    int gray = x - mouseX;
    fill(gray);
    rect(x,0,10,height);
  }
}
```

{ ～ } で作られるブロックの中に新たなブロックを作ることが出来ます。サンプル 7-5 の for 命令を使用したサンプルやサンプル 7-6 の while 命令などが、その例になっています。入れ子になっているブロックで、外側のブロックで宣言した局所変数と同じ名前の局所変数を宣言することは出来ません。ですから、サンプル 7-10 はエラーとなります。

大域変数と局所変数の例 6 サンプル 7-10

```
void setup(){
  size(360,360);
  colorMode(HSB,359,99,99);
}
void draw(){
  for(int x=0;x<width;x += 10){
    color c = color(x,99,99);
    for(int y=0;y < height;y += 10){
// 外側のブロックの局所変数と同じ名前の局所変数は宣言出来ない。
      color c = color(y,99,99);
      fill(c);
      rect(x,y,10,10);
    }
  }
}
```

大域変数と同じ名前の局所変数を定義することは出来ます。ただし、同じ名前の局所変数が定義されているブロックでは、同じ名前の大域変数にアクセスするには、ちょっと工夫が必要です。
「this. 大域変数名」でアクセスすることが出来ます。詳しくは、説明しません。

"Duplicate local variable c" というエラーメッセージが表示されます。

変数の有効範囲を示す例

```
int x,y;

void setup(){
  size(360,360);
  colorMode(HSB,359,99,99);
  background(0,0,99);
  x = int(random(width));
  y = int(random(height));
  frameRate(60);
}

void draw(){
  float d = random(10,50);
  for(int i=0;i<10;i++){
    int t = 36*i;
    stroke(t,40,40);
    fill(t,99,99);
    pushMatrix();
    translate(x,y);
    rotate(radians(t));
    ellipse(3*d,0,d,d);
    popMatrix();
  }
  x = int(random(width));
  y = int(random(height));
}
```

変数xとyの有効範囲

変数dの有効範囲

変数iの有効範囲

変数tの有効範囲

今日は朝から授業に遅刻した
夢の中では間に合ったのに
どうして朝から大学に行けないんだ
どうして朝から大学に行けないんだ!!
ドォワッハッハー！
自分のせいなのね
そうなのね
ウォッチ！　今何時？
ランチタイム

関数の宣言（その 1）

コンピュータのプログラム作成では、同じような処理を行っている場合は、なるべくまとめて書くということが基本的な指針となっています。このような考え方から繰り返し処理の紹介を行ってきました。今度は、別の角度から同じような処理をまとめて書くということを行って行きます。

サンプル 7-11 は、ボディを表す長方形と 4 つのタイヤを表す長方形を描画することで、1 台の車のような形を表示するものです。

同じような処理をまとめるということの発想の裏には、モジュール化という発想があります。コンピュータの世界では、モジュール化という発想は非常に重要です。第二次世界大戦開始時には、戦車開発で遅れを取っていた米国は、このモジュール化という発想で、戦車を製作し、大量生産が可能になりました。だから勝てた？

1 台の車状の絵を表示その 1 サンプル 7-11

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  rectMode(CENTER);
  float carX = width/2; // 車の中心の X 座標
  float carY = height/2;// 車の中心の Y 座標
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;  // タイヤの横幅
  float tireH = carH/6.0;  // タイヤの縦幅
  // 4 つのタイヤの描画
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
}
```

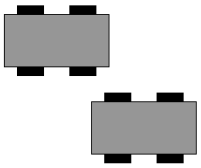
たくさんの局所変数を宣言していますが、このほうがやっていることの意味がハッキリすると思います。

サンプル 7-11 では、車の中心座標を変数で与えているので、ちょっとした変更で、複数の車を表示するサンプルに書きかえることが出来ます。サンプル 7-12 は、2 台の車を表示するものです。

2 台の車状の絵を表示その 1 サンプル 7-12

```
void setup(){
  size(400,400);
  smooth();
}
```

```
void draw(){
  background(255);
  float carX = width/2; // 車の中心の X 座標
  float carY = height/2;// 車の中心の Y 座標
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;  // タイヤの横幅
  float tireH = carH/6.0;  // タイヤの縦幅
  // 4 つのタイヤの描画
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);

  carX = 100;// 車の中心の X 座標
  carY = 100;// 車の中心の Y 座標
  carW = 120;// 車の横幅
  carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH);// ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
// 4 つのタイヤの描画
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
}
```

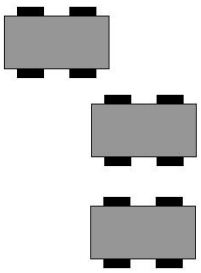


このサンプルは調子に乗ると、もっとたくさんの車を表示させることが出来ます。サンプル 7-13 では 3 台の車を表示させています。

3 台の車状の絵を表示その 1 サンプル 7-13

```
void setup(){
  size(400,400);
  smooth();
}
void draw(){
  background(255);
  float carX = width/2; // 車の中心の X 座標
  float carY = height/2;// 車の中心の Y 座標
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
```

```
  rectMode(CENTER);
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;  // タイヤの横幅
  float tireH = carH/6.0;  // タイヤの縦幅
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
  carX = 100;// 車の中心の X 座標
  carY = 100;// 車の中心の Y 座標
  carW = 120;// 車の横幅
  carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH);// ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
  carX = mouseX;// 車の中心の X 座標
  carY = mouseY;// 車の中心の Y 座標
  carW = 120;// 車の横幅
  carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH);// ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
}
```



サンプル 7-11 は、translate 関数を利用すると、サンプル 7-14 のように書き換えることが出来ます。車が固定したままだと面白くないので、マウスで移動できるようにもしてみました。

1 台の車状の絵を表示その 2 サンプル 7-14

```
void setup(){
  size(400,400);
  smooth();
}
```

```
void draw(){
  background(255);
  rectMode(CENTER);
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  translate(mouseX,mouseY);  // 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
}
```

translate(carX,carY) で 点(carX,carY) に「現在の原点」を移動させているので、原点が車の中心と考えることができるので、タイヤの描画位置の計算が簡単になっています。

translate 関数を使って、2 台の車を表示するサンプルを作ってみます、この場合には、「現在の座標軸」の状態を記録するために、pushMatrix 関数と popMatrix 関数を使っています。

2 台の車状の絵を表示その 2 サンプル 7-15

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  rectMode(CENTER);
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅

  pushMatrix(); //「現在の座標軸」の状態を保存
  translate(mouseX,mouseY);  // 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();
```

```
  pushMatrix(); //「現在の座標軸」の状態を保存
  translate(width/2,height/2);  // 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();
}
```

このようにサンプルを作ると、車の描画する部分の共通部分がハッキリしてきます。そこで、共通部分をまとめるのが関数と呼ばれる仕組みです。実は、今までも関数を使ってきました。つまり、setup や draw です。この場合には、setup 関数や draw 関数は、事前に Processing の側で使われることを知っている関数です。このようなもの以外に、プログラムを作る人が自由に関数を作ることが出来ます。自分なりの関数の作り方は、いくつかのパターンがあります。まずは、一番単純な関数の定義の仕方を紹介します。まず、関数を定義するためには、その関数の名前を決める必要があります。関数の名前のことを、関数名と呼びます。

英語では、関数のことを function と呼びます。

自分なりの関数を作ることを、関数を定義すると呼ぶことがあります。

簡単にいうと、setup や draw と同じです。

表 7-2 関数定義の仕方（その 1）

関数定義のパターン

void 関数名 (){
関数処理の内容を書きます。
変数なども使うことができます。
}

サンプル 7-15 を関数を使って書きかえてみます。車を描く部分を関数としてまとめるので、関数名は drawCar とします。定義した drawCar 関数を使いたいときには、使いたい部分で、「drawCar();」とするだけです。

2 台の車状の絵を表示その 2 サンプル 7-16

```
void setup(){
  size(400,400);
  smooth();
}
```

```
// drawCar 関数の定義
void drawCar(){
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
}

void draw(){
  background(255);
  pushMatrix();//「現在の座標軸」の状態を保存
  translate(mouseX,mouseY);// 車の中心を「現在の原点」にする
  drawCar(); // 定義した関数を呼び出す
  popMatrix(); //「現在の座標軸」を保存されている状態に戻す
  pushMatrix(); //「現在の座標軸」の状態を保存
  translate(width/2,height/2);// 車の中心を「現在の原点」にする
  drawCar(); // 定義した関数を呼び出す
  popMatrix();//「現在の座標軸」を保存されている状態に戻す
}
```

ここで定義した変数 carW、carH は、drawCar 関数内部でのみ使用できます。局所変数 carW と carH が定義されているブロックはどこでしょうか？

このサンプルをよく考えると、drawCar 関数を呼び出す際に、車を描く位置も指定できると、もっと簡潔にプログラムが書けるように考えられます。rect 関数や ellipse 関数では、図形を描く場所や大きさを引数として指定することが出来ます。これと同じことが自分で定義した関数でも出来れば、良いはずです。引数を使った関数定義の仕方は、次の様になります。

表 7-3 関数定義の仕方（その 2）

関数定義のパターン
void 関数名 (データ型名　引数名){
関数処理の内容を書きます。
変数なども使うことができます。
}
void 関数名 (データ型名 1　引数名 1,
データ型名 2　引数名 2…){
関数処理の内容を書きます。
変数なども使うことができます。
}

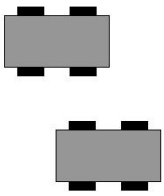
ここで出てくる引数名、引数名 1、引数名 2 などは、この関数の中だけで、有効な変数となります。
また、引数として宣言された変数は、関数内で局所変数として利用することが出来ます。

この引数付きの関数定義を利用してサンプル 7-16 を書きかえて見ます。非常にシンプルになったことがわかると思います。

2 台の車状の絵を表示その 3 サンプル 7-17

```
void setup(){
  size(400,400);
  smooth();
}

// drawCar 関数の定義
void drawCar(float x,float y){
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  pushMatrix();//「現在の座標軸」の状態を保存
  translate(x,y);// 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();//「現在の座標軸」を保存されている状態に戻す
}

void draw(){
  background(255);
  drawCar(mouseX,mouseY); // 定義した関数を呼び出す
  drawCar(width/2,height/2); // 定義した関数を呼び出す
}
```

float 型の変数 x と y は、drawCar 関数の内部だけで有効な変数となります。



引数付きの関数を呼び出すときには、少し動作が複雑になります。サンプル 7-17 で、「drawCar(mouseX,mouseY)」が実行されると、mouseX の値が drawCar 関数の引数 x に、mouseY の値が drawCar 関数の引数 y に、それぞれコピーされます。このコピーが終わった後に、drawCar 関数で指定されている処理の実行が始まります。

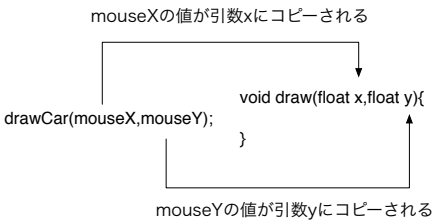


図 7-5 関数呼び出し時の引数のコピー

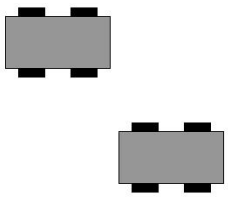
この drawCar 関数を使った、別のサンプルを載せておきます。このサンプル 7-18 では、自動車が移動していきます。また、サンプル 7-18 では、大域変数と同じ局所変数（引数）を使っています。あま

り良い習慣ではないと思いますが、大域変数名と同じ名前の局所変数を定義することが出来ます。その局所変数が定義されているブロックの中では、その局所変数が優先されますので、大域変数の値をアクセスすることは出来ません。

裏技 (this. 大域変数名) を使うとアクセスすることが出来ます。

移動する車 サンプル 7-18

```
int x;

void setup(){
  size(400,400);
  smooth();
  x = 0;
}

// drawCar 関数の定義
void drawCar(float x,float y){
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  pushMatrix();//「現在の座標軸」の状態を保存
  // この変数 x は drawCar 関数の引数 x を指します。
  translate(x,y);// 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();//「現在の座標軸」を保存されている状態に戻す
}

void draw(){
  background(255);
  drawCar(x,height/3); // 定義した関数を呼び出す
  drawCar(2*x,2*height/3);// 定義した関数を呼び出す
  x = (x+1) % width;
}
```



剰余演算 %（余りを求める）を使って、車の繰り返し移動を実現しています。
あることを行うプログラムには、色々なやり方があります。コンピュータに指示するやり方のことをアルゴリズム (algorithm) と呼んでいます。

もう一つの関数の使い方のサンプルを示します。サンプル 7-19 はウインドウの真ん中を左右にボールが移動し、壁にぶつかると反射するというものです。

移動するボール サンプルその 1 7-19

```
int xPos;
int speed;
int radius;

void setup(){
  size(400,200);
  smooth();
  xPos = width/2;
  speed = -1;
  radius = 20;
}

void draw(){
  background(255);
  // ボールを移動させる
  xPos = xPos+speed;
  // ボールの壁での反射処理を行う
  if((xPos+radius) > width){
    speed = -1;
    xPos = width-radius;
  }else if((xPos-radius) < 0){
    speed = 1;
    xPos = radius;
  }
  // ボールを描く
  stroke(0);
  fill(127);
  ellipse(xPos,height/2,2*radius,2*radius);
}
```

サンプル 7-19 は、draw 関数の中にすべての処理を書いています。このように、この程度の小さなプログラムでは、1 つの関数の中にすべての処理を書いてしまっても、大きな問題は発生しません。人間はあまり記憶力が良くないので、1 つの関数の中にたくさんの処理を詰め込んでしまうと、その関数の中で何をやっているのかを理解することが困難になります。ここでは、大域変数は、プログラム中のどこからでもアクセスできるということに着目して、プログラムを書き換えてみます。サンプル 7-19 の draw 関数の中では、

デカルトの「検討しようとする難問をよりよく理解するために、多数の小部分に分割すること」という考え方が基礎にあります。

1. 背景を白色にする
2. ボールを移動させる
3. ボールの壁での反射処理を行う
4. ボールを描く

ということを行っています。そこで、処理 2,3,4 を独立した move, bounce, display 関数として定義することにします。また、中心座標と半径を指定して円を描く関数 drawCircle を定義します。このよう

な方針で書き換えを行ったものがサンプル 7-20 です。

移動するボールその 2（関数化版）サンプル 7-19

```
int xPos;
int speed;
int radius;
oid drawCircle(float x, float y, float r) {
  ellipse(x, y, 2*r, 2*r);
}
void display() {
  stroke(0);
  fill(127);
  drawCircle(xPos, height/2, radius);
}
void move() {
  xPos += speed;
}

void bounce() {
  if ((xPos+radius) > width) {
    speed = -1;
    xPos = width-radius;
  }
  else if ((xPos-radius) < 0) {
    speed = 1;
    xPos = radius;
  }
}

void setup() {
  size(400, 200);
  smooth();
  xPos = width/2;
  speed = -1;
  radius = 20;
}

void draw() {
  background(255);
  display();
  move();
  bounce();
}
```

このように書き換えると、ここの処理が独立して書かれることになるので、1 つ 1 つの処理がやっている内容が明確になると思います。

自分で定義した関数は、自由に使うことが出来ます。つまり、自分で定義した関数の中で、自分の定義した関数を利用することが出来ます。サンプル 7-20 は、自分で定義した関数を自分で定義した関

モジュール化（関数の利用）の特徴として、「複雑な機能を単純な独立した機能に分割して管理する」があります。

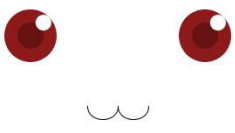
数の中で使うものです。ここまで来ると、かなり複雑なプログラムを作れるようになっている筈です。

某アニメキャラもどきを表示 サンプル 7-20

```
// 目を描く
void drawEye(float x, float y, float r) {
  pushMatrix();
  translate(x, y);
  noStroke();
  fill(0, 80, 55);
  ellipse(0, 0, r*2, r*2);
  fill(0, 80, 40);
  ellipse(0, 0, r*2*0.5, r*2*0.5);
  rotate(-PI/4);
  translate(r*0.7, 0);
  fill(0, 0, 99);
  ellipse(0, 0, r*2.0*0.3, r*2.0*0.3);
  popMatrix();
}
//  口を描く
void drawMouth(float x, float y, float w, float h) {
  pushMatrix();
  translate(x, y);
  noFill();
  stroke(0, 0, 0);
  bezier(-w, 0, -w, h, 0, h, 0, 0);
  bezier(w, 0, w, h, 0, h, 0, 0);
  popMatrix();
}
// 顔全体を描く
void drawQB(float x,float y,float w,float h){
  drawEye(x-w/2,y,30);
  drawEye(x+w/2,y,30);
  drawMouth(x,y+0.4*h,35,20);
}
void setup() {
  size(400, 400);
  colorMode(HSB, 359, 99, 99);
  smooth();
}
void draw() {
  background(0, 0, 99);
  drawQB(mouseX,mouseY,width/2,height/2);
}
```

情報メディア基盤ユニットの単位は必ず取得してよ。必修科目の単位を落としていると卒研につけないんだ。卒研をクリアできないと卒業できないんだ。これは契約だよ。

顔の輪郭部分なども欲しい気がするのですが。そうすると耳とかもいるのかな？
でも、シンプルな方が良いかな？



情報メディア基盤ユニット用資料（2016 年度第 8 章）

Processing 言語による情報メディア入門

関数 (その 2)

神奈川工科大学情報メディア学科　佐藤尚

組み込み関数

今までにも、いくつか使ってきましたが、Processing では沢山の関数が用意されています。その中でよく使いそうなものを以下に挙げておきます。ここで紹介する関数は、呼び出すと何らかの値を求めて、その値を返すものです。この返される値のことを戻り値と呼んでいます。また、値を返す関数を呼び出すと、呼び出された関数がその戻り値に置き換わるような動作となります。

プログラミング言語において、事前に定義されている関数を**組み込み関数** (built-in function) と呼ぶことがあります。


```
int m = 60*hour() + minute();
```

現在の時刻が18時24分なら、hour()は18に、minute()は24に置き換わり、変数mには60*18+24=1104が代入される。

図 8-1 関数を呼び出すと

関数を実行する目的で、プログラム中に関数を置くことを**関数を呼び出す**と呼ぶことがあります。

これ以外にも沢山の組み込み関数が用意されています。気になるひとは、リファレンスマニュアルを見て下さい。

時間に関連した関数には以下のようなものがあります。これらは、パソコンの時計に連動して、情報を求めています。

表 8-1 時間関連の関数

関数名	関数が返す値の意味
year()	現在の年を返す。
month()	現在の月（1 ～ 12）を返す。
day()	現在の日（1 ～ 31）を返す。
hour()	現在の時刻の時間を返す。
minute()	現在の時刻の分を返す。
second()	現在の時刻の秒を返す。
millis()	プログラムを実行してから経過時間をミリ秒単位で返す。

関数と言うと数学で出てくるものを思い浮かべると思います。Processing では、数学で出てくるような関数が用意されています。まずは、数の大きさに係わる関数です。

表 8-2 最小、最大関連の関数

関数名	関数が返す値の意味
min(x1,x2)	x1 と x2 の中で小さい方の値（最小値）を求める。
min(x1,x2,x3)	x1,x2,x3 の中で最小値を求める。
max(x1,x2)	x1 と x2 の中で大きい方の値（最大値）を求める。
max(x1,x2,x3)	x1,x2,x3 の中で最大値を求める。

これらの関数には、別な使い方もあります。それは次回に紹介します。min は minimum、max は maximum を省略したものです。

もう少し数学っぽい関数もあります。

表 8-3 ちょっと数学っぽい関数

関数名	関数が返す値の意味
abs(x)	引数 x の絶対値を求めます。例えば、abs(-1.1) は 1.1、aba(3) は 3 になります。
sqrt(x)	引数 x の平方根の値を求めます。る。例えば、sqrt(4) なら 2.0 になります
sq(x)	引数 x の二乗を求めます。
pow(x,n)	x の n 乗を求めます。例えば、pow(2,4) は 16 になります。
exp(x)	指数関数の値を求めます。ネイピア数 e の x 乗を求めます。
log(x)	自然対数の値を求めます。
dist(x1, y1, x2, y2)	2 点 (x1,y1) と (x2,y2) の間の距離を求めます。
constrain(v, m0, m1)	引数 v の値が m0 以上 m1 以下なら v を返し、v の値が m0 よりも小さければ m0 を返し、v の値が m1 よりも大きければ m1 を返すような関数です。
lerp(v0,v1,t)	(1-t)*v0+t*v1 という値を求めます。線形補間と呼ばれる計算方法です。
map(v, low1, high1, low2, high2)	2 点 (low1,low2) と (high1,high2) を通る直線において、X 座標の値が v の時の Y 座標の値を求める関数です。別な言い方をすると low1 以上 high1 以下の値 v を low2 以上 high2 以下の値に変換するとどんな値になるかを求めるものです。要するに一次関数の値を計算しています。 $y = \frac{high2 - low2}{high1 - low1}(v - low1) + low2$

この 2 つの関数は数 III をやっていないと出てこないですね。

このように、言葉で説明するようも、式で説明する方が簡単になる場合もあります。この map 関数は意外に使い機会の多い関数です。

でもやっぱり、関数と言うと三角関数のような気がします。Processing でも三角関数が用意されています。

表 8-4 三角関数関連

関数名	関数が返す値の意味
sin(x)	正弦関数 sin の値を求めます。
cos(x)	余弦関数 cos の値を求めます。
tan(x)	正接関数 tan の値を求めます。
degrees(x)	ラジアンから度に値を変換します。
radians(x)	度からラジアンに値を変換します。
asin(x)	sin の逆関数の値を求めます。つまり、sin y = x となる y の値を求めます。ただし、y の値は -PI/2 から PI/2 となります。

sin 関数の逆関数のことを arcsin と呼ぶことがあります。そこで、asin、acos、atan という名称になっています。

関数名	関数が返す値の意味
acos(x)	cos の逆関数の値を求めます。つまり、cos y = x となる y の値を求めます。ただし、y の値は -PI/2 から PI/2 となります。
atan(x)	tan の逆関数の値を求めます。つまり、tan y = x となる y の値を求めます。ただし、y の値は -PI/2 から PI/2 となります。
atan2(y,x)	原点と点 (x,y) を通る直線と X 軸のなす角度をもとめます。ただし、角度の値は -PI から PI の範囲の値となります。

引数の順番が直感的なものと逆になっているので、注意して下さい。良く使う機会のある関数です。

全然サンプルがないのも何なので、少し載せておきます。まずは、map 関数を使ったものです。これは、マウスカーソルの動きに合わせて、真ん中にある円を動かすものです。円はある範囲 (x0 ～ x1) の間しか動きません。このような動作を map 関数を使って作りだしています。つまり、mouseX の値を x0 から x1 の値に変換し、その値を円の中心の X 座標値として使っています。

map 関数の使用例 サンプル 8-1

```
int x0,x1;

void setup(){
  size(400,200);
  smooth();
  x0 = 80;
  x1 = width-x0;
}

void draw(){
  background(255);
  strokeWeight(3);
  stroke(0);
  line(x0,height/2,x1,height/2);
  fill(100);
 // mouseX の値を x0 ～ x1 の間の値に変換
  float x = map(mouseX,0,width-1,x0,x1);
  strokeWeight(1);
  ellipse(x,height/2,20,20);
}
```

次は、atan2 を使ったサンプルです。原点とマウスカーソルの位置を結ぶ直線と X 軸のなす角を弧で示すようなサンプルです。ついでに、その角度の値を degree 関数を使って、度単位で表示しています。なの、弧の部分は arc 関数を使って描画しています。arc 関数では、弧がスタートする時の角度と終了する時の角度を指定する必要があります。また、原点との距離を計算して、3 分の 1 の位置に弧を表

示するようにしています。

atan2,dist などの関数の使用例 サンプル 8-2

```
PFont font;

void setup(){
  size(400,400);
  smooth();
  font = loadFont("Serif-48.vlw");
  textFont(font);
}

void draw(){
  background(255);
  stroke(0);
  line(0,0,mouseX,mouseY);
  noFill();
   float theta = atan2(mouseY,mouseX);// 線分と X 軸のなす角度を求める
  float l = dist(0,0,mouseX,mouseY);// 原点との距離を求める
  arc(0,0,2*l/3,2*l/3,0,theta);
  line(0,0,mouseX,mouseY);
  String deg = str(degrees(theta));
  fill(255,10,10);
  text(deg,width/2-textWidth(deg)/2,height/2);
}
```

これらのサンプルのように、色々な関数を組み合わせることでどんどん複雑なプログラムを作ることが出来るようになります。

関数の宣言（その 2）

Processing が用意している関数について説明してきました。今回説明した関数は、何らかの値（戻り値）を返すような関数でした。前回の講義では、処理をまとめるという観点から関数の説明をしました。そのため、値を返すという話はありませんでした。今回説明したような値を返すような関数を定義することも出来ます。

前回説明した値を返さない関数も void という特別なデータ型の値を返していると見なすことも出来ます。

そのためには、表 8-5 のような形でプログラムを書きます。値を返す必要があるために、戻り値のデータ型を指定する必要があります。関数定義の中で、戻り値を決定する（どんな値を返すのか）必要があります。そのために、関数名の前に戻り値のデータ型を置きます。戻り値を指定するために、return 命令を使います。「return　式;」とすると、この式の値が関数の戻り値となります。また、return 命令を実行すると、その場所で関数の実行が終わります。関数の定義中に、複数の return 命令があっても、問題はありません。逆にどこにも return 命令がないと、Processing はどんな値を戻り値とすればよいのか、わからないので、エラーとなります。関数の定義は、プログラム中のどこからでも始めることが出来ます。ただし、他の関数の定義中などでは出来ません。

1 つの関数内に複数の return 命令を置くことは、良くないと考える人たちもいます（いた？）。会社によっては、複数の return 命令を置くことを禁止してるところもあります。このような、プログラム作成上で決めた制限（規則）を、コーディング規約と呼ぶことがあります。

表 8-5 関数定義の仕方（その 3）

関数定義のパターン

戻り値のデータ型 関数名 (){
関数処理の内容を書きます。
どこかに、return 命令が必要です。
変数なども使うことができます。
}

戻り値のデータ型 関数名 (データ型名　引数名){
関数処理の内容を書きます。
どこかに、return 命令が必要です。
変数なども使うことができます。
}

戻り値のデータ型 関数名 (データ型名 1　引数名 1,
データ型名 2　引数名 2…){
関数処理の内容を書きます。
どこかに、return 命令が必要です。
変数なども使うことができます。
}

サンプル 8-3 では、” 年 / 月 / 日 ” の形式で、今日の日付を返す関数 today を定義しています。

戻り値を持った関数定義の例その 1 サンプル 8-3

```
PFont font;

// 今日の日付を返す関数 today を定義、戻り値は String 型
String today(){
  String msg = year()+"/"+month()+"/"+day();
  return msg; // 戻り値は msg
}

void setup(){
  size(300,200);
  smooth();
  font = loadFont("Serif-48.vlw");
  textFont(font);
}

void draw(){
  background(255);
  fill(0);
  String msg = today(); // 自分で定義した関数は自由に使うことが出来る。
  text(msg,width/2-textWidth(msg)/2,height/2);
}
```

サンプル 8-4 に関数定義の部分だけの部分の例を示します。

return 命令がないと、「This method must return a result of type データ型名」というエラーメッセージが表示されます。

関数の定義は、どこかのブロックに属しているところでは出来ません。

戻り値を持った関数定義の例その 2 サンプル 8-4

```
float myConstraint1(float v,float m0,float m1){
  flaot ans;
  ans = v;
  if(v > m1){
    ans = m1;
  }else if(v < m0){
    ans = m0;
  }
  return ans;
}

float myConstraint2(float v,float m0,float m1){
  if(v > m1){
    return m1;
  }else if(v < m0){
    return m0;
  }else{
    return v;
  }
}

float myDist(float x0,float y0,float x1,float y1){
  return sqrt(sq(x0-x1)+sq(y0-y1));
}
```

この場合は日本語の説明より、プログラムの方がわかり易いでよね。

dist 関数は三平方の定理を使うと、自分で作ることも出来ます。自分のプログラムの定義中に他の関数を利用することも出来ます。

動作しないサンプルでは、面白くないので、サンプル 8-3 を改良して、今日の日付を " 月 / 日 / 年 " の形で表示することにします。この際に、月は英語表記の略称とします。今回のサンプルでは if 命令の山になるので、ちょっとプログラムは長くなります。

戻り値を持った関数定義の例その 3 サンプル 8-5

```
PFont font;

void setup(){
  size(300,200);
  smooth();
  font = loadFont("Serif-48.vlw");
  textFont(font);
}

void draw(){
  background(255);
  fill(0);
   String msg = today();// 自分で定義した関数は自由に使うことが出来る。
  text(msg,width/2-textWidth(msg)/2,height/2);
}
```

```
// 今日の日付を返す関数 today を定義、戻り値は String 型
String today(){
  int m = month();
  String result = "/"+day()+"/"+year(); // 後ろの部分は簡単に作れる
  // 月の値で分岐する
  if(m == 1){
    result = "Jan"+result;
  }else if(m == 2){
    result = "Feb"+result;
  }else if(m == 3){
    result = "Mar"+result;
  }else if(m == 4){
    result = "Apr"+result;
  }else if(m == 5){
    result = "May"+result;
  }else if(m == 6){
    result = "Jun"+result;
  }else if(m == 7){
    result = "Jul"+result;
  }else if(m == 8){
    result = "Aug"+result;
  }else if(m == 9){
    result = "Sep"+result;
  }else if(m == 10){
    result = "Oct"+result;
  }else if(m == 11){
    result = "Nov"+result;
  }else if(m == 12){
    result = "Dec"+result;
  }else{
    result = "Unknow"+result;
  }
  return result; // 戻り値は result
}
```

一般的に、人為的に決めた規則に合うようにデータを変換することはちょっと面倒です。

10 月なのに October とか、12 月なのに December とかちょっと変に思いませんか？

関数を利用してプログラムを書くことにより、修正部分を一部にとどめることが出来ます。サンプル 8-5 でも、関数 today の定義部分を変更しただけですよね。

定義を書いている場所が少し移動していますが。

このように関数を利用してプログラムを作成すると、わかりやすく、変更しやすいプログラムを作成することが出来ます。単に関数を使えばわかりやすいプログラムが作れるわけではありません。上手い関数名や変数名をつけたり、複雑な処理をわかりやすい関数の組み合わせに分解するなど､色々なことが重要になります。ですから、ゲームのような複雑なプログラムを作るためには、様々な力をもった人が必要となります。

変更しやすいプログラムを作成することはとても重要です。ゲームの仕様が少し変わっただけで、プログラムを全て作り直していたら、ゲームは完成しませんよね。

サンプル 8-6 では、ウインドウ中心に表示されている円にマウスカーソルが来ると、円の色を変えるものです。ある点が円の中に入っているかどうかを、inDisk 関数を定義して、判定しています。入っ

ているかどうかをあらわすので、戻り値は boolean 型とするのが自然です。inDisk 関数は、ある点が円の中に入っているかどうを、円の中心とその点の距離を調べることで判定しています。つまり、点と円の中心の距離が半径以下なら円の中に入っています。dist 関数は 2 点の距離を求める組み込み関数です。これを使って、円の中心と点との距離を求めることができます。そして、この値と半径の値 r と比較することで、判定を行っています。

円の内外判定

戻り値を持った関数定義の例その 4 サンプル 8-6

```
int radius = 150;

void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  noStroke();
  if(inDisk(mouseX,mouseY,width/2,height/2,radius)){
    fill(255,10,10);
  }else{
    fill(10,10,255);
  }
  ellipse(width/2,height/2,2*radius,2*radius);
}

/*
inDisk 関数は点 (x,y) が中心座標が (cx,cy) で半径の r の円の中に入っているかどうかを判定します。
*/
boolean inDisk(float x,float y,float cx,float cy,float r){
  float d = dist(x,y,cx,cy);
  if(d <= r){
    return true;
  }else{
    return false;
  }
}
```

サンプル 8-6 の inDisk 関数は、次の様に書くことも出来ます。

戻り値を持った関数定義の例その 5 サンプル 8-6'

```
boolean inDisk(float x,float y,float cx,float cy,float r){
  float d = dist(x,y,cx,cy);
  return (d <= r);
}
```

コールバック関数

今までのように、mousePressed 変数などだけを使って、少し複雑なマウス操作を伴ったプログラムを作成することは、困難です。そこで、コールバック関数と言う仕組みが用意されています。

つまり、マウスなどが指定された動作（イベントと呼びます）が行われた時に、呼び出す関数を決めておき、その関数内でイベントに対応する処理を定義します。Processing 言語では、以下のようなコールバック関数が用意されています。当然、処理の中身はユーザが定義します。

イベントの処理を行うということで、コールバック関数のことをイベントハンドラと呼ぶこともあります。今まで使ってきた、setup 関数や draw 関数もコールバック関数です。setup は起動時というイベントにより呼び出される関数、draw は一定時間が経過したいうイベントで呼び出される関数です。

表 8-6 コールバック関数

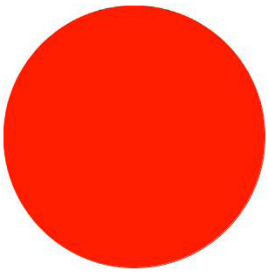
呼び出すイベント	コールバック関数名	補足
マウスボタンが押された	mousePressed()	この関数内で、mouseButton 変数の値が、LEFT なら左、CENTER なら真ん中、RIGHT なら右ボタンが押されています。
マウスボタンが離れた	mouseReleased()	
マウスボタンを押さない状態でマウスが動かされた	mouseMoved()	
マウスがドラッグされた	mouseDragged()	マウスボタンを押した状態で、マウスを移動させる動作です。この関数内で、mouseButton 変数の値が、LEFT なら左、CENTER なら真ん中、RIGHT なら右ボタンが押されています。
マウスがクリックされた	mouseClicked()	マウスをクリックするためには、マウスボタンを押して、離すという動作が必要なので、この関数が動作する前に、コールバック関数 mousePressed と mouseReleased が実行されます。
キーボードが押された	keyPressed()	システム変数 key にどのキーが押されたかの情報が保存されています。なお、矢印キーなどを押した場合には、システム変数 key には CODED という特別な値が保存され、どのキーが押されたかの情報はシステム変数 keyCode に保存されます。
キーボードが離された	keyReleased()	
キーボードが押された	keyTyped()	keyPressed 関数と異なり、1 回だけ呼び出されます。

システム変数 key には、押したキーの ASCII コードの値が保存されています。この方法では、日本語の入力が出来ません。また、矢印キーの処理には、2 段階の処理が必要となります。

これらのコールバック関数を利用したサンプルを示します。サンプル 8-7 は mouseClicked 関数を利用したものです。円の内部でマウスをクリックすると、円の描画色をランダムに変更するものです。描画色を color 型の fColor 変数に保存しておきます。マウスがクリックされた際のマウスカーソルの位置をしらべ、それが円の中であれば、fColor 変数の値を変更しています。サンプル 8-6 で作成した関数 inDisk を利用しています。

コールバック関数の利用例その 1 サンプル 8-7

```
color fColor;
void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
  fColor = color(random(360),99,99);
}
// サンプル 8-6' のものをそのまま利用
boolean inDisk(float x,float y,float cx,float cy,float r){
  float d = dist(x,y,cx,cy);
  return (d <= r);
}
void draw(){
  background(0,0,99);
  fill(fColor);
  stroke(fColor);
  ellipse(width/2,height/2,2*150,2*150);
}
// マウスがクリックされた際のコールバック関数
void mouseClicked(){
  if(inDisk(mouseX,mouseY,width/2,height/2,150)){
    fColor = color(random(360),99,99);
  }
}
```



サンプル 8-8 は、マウスボタンを押すとウインドウが黒くなり、マウスボタンを離すと徐々に色が白になるようなものです。描画色は変数 gray を使用して決めています。マウスボタンが押されると gray の値を 0 とし、マウスを動かすことにより、徐々に gray の値を大きくしていきます。ただし、255 より大きな値とすることができないので、constrain 関数を使って、255 よりも大きな値とならないようにしています。

コールバック関数の利用例その 2 サンプル 8-8

```
float gray=128;
void setup() {
  size(200, 200);
  smooth();
}
```

```
void draw() {
  stroke(gray);
  fill(gray);
  rect(0, 0, width, height);
}
// マウスを押したときの処理
void mousePressed() {
  gray = 0;
}
// マウスを移動させたときの処理
void mouseMoved() {
  gray = constrain(gray+1, 0, 255);
}
```

サンプル 8-9 は、マウスの移動とドラッグを組み合わせたサンプルです。

コールバック関数の利用例その 3 サンプル 8-9

```
boolean mustDraw = false;
color WHITE;
float diam=10;
void setup() {
  size(400, 400);
  smooth();
  colorMode(HSB,359,99,99);
  WHITE = color(0,0,99);
}
void draw() {
  fadeTo(WHITE);
  if(mustDraw){
    fill(random(360),99,99,150);
    float x = mouseX+random(-diam,diam);
    float y = mouseY+random(-diam,diam);
    ellipse(x,y,diam,diam);
    mustDraw = false;
  }
}
void fadeTo(color c){
  stroke(c,20);
  fill(c,20);
  rectMode(CORNER);
  rect(0,0,width,height);
}
void mouseMoved(){
  mustDraw = true;
  diam = random(10,20);
}
void mouseDragged(){
  mustDraw = true;
  diam = random(40,80);
}
```

プログラム中の fadeTo 関数は、ウインドウ全体を指定した色にフェードさせる関数です。色に不透明度の情報を付加して、実現しています。マウスをドラッグしているときには、少し大きな円を描画し、単にマウスを動かしている時には、小さな円を描画しています。boolean 型変数 mustDraw によって、円を描画する必要があるかどうかを判定しています。

サンプル 8-10 は、マウスのドラッグによる、物体の移動の例です。mouseDragged 関数は、マウスボタンを押しながらマウスを移動させると呼び出される関数です。一つ前のマウスの位置は pmouseX と pmouseY 変数に保存されています。つまり、mouseX-pmouseX の値は X 軸方向の移動距離を表してます。同様に、mouseY-pmouseY の値は Y 軸方向の移動距離を表しています。そこで、この 2 つの値を物体の位置に加えることにより、ドラッグ時の物体移動を再現できます。物体をつまんで動かしているような動作とするために、物体上でクリックした場合のみ移動するようになっています。

このサンプルでは、物体は円となっています。ですので、以前に作った inDisk 関数を利用しています。

コールバック関数の利用例その 4 サンプル 8-10

```
color fColor;
float diam=40;
float xBall,yBall;
void setup() {
  size(400, 400);
  smooth();
  colorMode(HSB,359,99,99);
  fColor = color(random(360),99,99);
  xBall = random(width);
  yBall = random(height);
}
void draw() {
  background(0,0,99);
  stroke(fColor);
  fill(fColor);
  ellipse(xBall,yBall,diam,diam);
}
void mouseClicked(){
  fColor = color(random(360),99,99);
  xBall = random(width);
  yBall = random(height);
}
void mouseDragged(){
  if(inDisk(mouseX,mouseY,xBall,yBall,diam/2)){
    xBall += (mouseX-pmouseX);
    yBall += (mouseY-pmouseY);
  }
}
boolean inDisk(float x,float y,float cx,float cy,float r){
  return (dist(x,y,cx,cy) <= r);
}
```

このプログラムには、一つ欠点があります。それは、マウスを早く動かすと、物体がついてこないことです。つまり、これを解決したものがサンプル 8-11 です。このサンプルでは、物体が移動中かどうかを示す boolean 型変数 moving を使っています。

1 つ前の状態からのマウスの移動量が円の半径より大きくなると、この現象が発生します。

コールバック関数の利用例その 4 サンプル 8-11

```
color fColor;
float diam=40;
float xBall,yBall;
boolean moving = false;
void setup() {
  size(400, 400);
  smooth();
  colorMode(HSB,359,99,99);
  fColor = color(random(360),99,99);
  xBall = random(width);
  yBall = random(height);
}
void draw() {
  background(0,0,99);
  stroke(fColor);
  fill(fColor);
  ellipse(xBall,yBall,diam,diam);
}
void mouseClicked(){
  fColor = color(random(360),99,99);
  xBall = random(width);
  yBall = random(height);
}
void mousePressed(){
   if(inDisk(mouseX,mouseY,xBall,yBall,diam/2)){
    moving = true;
  }
}
void mouseReleased(){
  moving = false;
}
void mouseDragged(){
  if(moving){
    xBall += (mouseX-pmouseX);
    yBall += (mouseY-pmouseY);
  }
}
boolean inDisk(float x,float y,float cx,float cy,float r){
  return (dist(x,y,cx,cy) <= r);
}
```

次にキーボードを使用したサンプルを示します。

コールバック関数の利用例その 5 サンプル 8-12

```
float xBall;

void setup(){
  size(400,200);
  smooth();
  xBall = width/2;
}

void draw(){
  background(255);
  stroke(0);
  fill(128);
  ellipse(xBall,height/2,30,30);
}

void keyPressed(){
  if(key == 'r'){
    xBall = constrain(xBall+random(2,4),0,width-1);
  }else if(key == 'l'){
    xBall = constrain(xBall-random(2,4),0,width-1);
  }else if(key == 'c'){
    xBall = width/2;
  }else if(key == CODED){
    if(keyCode == LEFT){
      xBall = constrain(xBall-1,0,width-1);
    }else if(keyCode == RIGHT){
      xBall = constrain(xBall+1,0,width-1);
    }
  }
}
```

r キーが押されたかどうかは、key == 'r' でわかります。通常は、key == ' 調べたいキー ' とすれば、指定したキーが押されたかどうかがわかります。直接 ASCII コードの値を書いてもかまいません。

システム変数 keyCoded は、以下のようなキーに対応しています。これ以外の BACKSPACE, TAB, ENTER, RETURN, ESC, DELETE キーは、通常のキーのように処理されます。つまり、ASCII コードで表されています。Processing では、この 6 つのキーの値は、BACKSPACE, TAB, ENTER, RETURN, ESC, DELETE という定数で定義されています。

Windows では ENTER キーが利用されますが、Mac では RETERN キーが利用されます。状況によっては、プログラムする際に注意が必要です。

表 8-7 keyCode が対応指定しているキー

キーの名称	keyCode の値	キーの名称	keyCode の値
上カーソルキー	UP	左カーソルキー	LEFT
下カーソルキー	DOWN	右カーソルキー	RIGHT
ALT キー	ALT	シフトキー	SHIFT
コントロールキー	CONTROL		

単純に keyPressed 関数を使った場合には、複数のキーが押されたかの処理ができません。しかし、少しプログラムを書くことで実現することができます。

最後にカーソルキーを利用して、円形の物体を動かすようなサンプルを示します。このサンプルにおける変数 x,y は円の中心座標を表しており、変数 vx,vy は物体の移動を表す速度ベクトルです。vx=vx=0 の時には、物体は移動しません。カーソルキーが押されたときに、コー

ルバック関数 keyPressed 内において適切な値を変数 vx,vy に設定します。カーソルキーが離されたときには、変数 vx,vy の値を 0 にすることで物体の移動を止めます。なお、キーボード上の C のキーが押されたときには、物体の位置をウインドウの中心に移動させます。

コールバック関数の利用例その 6 サンプル 8-13

```
float xBall,yBall; // 円の中心位置の座標
float vx,vy;       // 円の速度ベクトル
void setup(){
  size(600,600);
  smooth();
  xBall = width/2;
  yBall = height/2;
  vx = vy = 0; // この時には円は動かない
}
void draw(){
  background(255);
  fill(255,10,10);
  ellipse(xBall,yBall,20,20);
  xBall += vx;   // 速度ベクトルを加えることで、円を移動させる。
  yBall += vy;
}
void keyPressed(){
  if(key == 'c' || key == 'C'){
    xBall = width/2;
    yBall = height/2;
  }else if(key == CODED){
    if(keyCode == LEFT){
      vx = -1;
      vy = 0;
    }else if(keyCode == RIGHT){
      vx = 1;
      vy = 0;
    }else if(keyCode == UP){
      vy = -1;
      vx = 0;
    }else if(keyCode == DOWN){
      vy = 1;
      vx = 0;
    }
  }
}
```

```
void keyReleased(){
  if(key == CODED){
    if(keyCode == LEFT || keyCode == RIGHT ||
       keyCode == UP || keyCode == DOWN){
      vx = vy = 0;
    }
  }
}
```

情報メディア基盤ユニット用資料（2016 年度第 9 章）

Processing 言語による情報メディア入門

配列（その 1）

神奈川工科大学情報メディア学科　佐藤尚

配列

次のような上から下へ円が移動するようなプログラムを考えます。このサンプルでは、1 つの円を動かしています。変数 y に円の中心の Y 座標値を保存し、縦方向の移動量を表す変数 v を使って、1) 円の描画位置を計算、2) 下まで到達するとしたら、円を上に移動させる、ついでに中心の X 座標の値も変更、3) 円を描画する、というアルゴリズムでプログラムを作っています。

プログラムの処理内容をアルゴリズム (algorithm) と呼びます。この名称は、現在のイラクのバグダードにおける 9 世紀の数学者アル・フワーリズミー の名前から来ていると言われています。日本語では算法と呼ぶこともあります。現在では、この呼び方をする人は、超少数派だと思いますが。

1 個の円の移動 サンプル 9-1

```
float y; // 円の中心の Y 座標
float x; // 円の中心の X 座標
float v; // 円の縦方向の移動速度
int radius;

void setup(){
  size(300,400);
  smooth();
  radius = 10;
  v = random(1,2); // 移動速度を乱数で決める
  x = random(radius,width-radius); // 出現位置をずらす
  y = -random(radius,2*radius);  // 出現タイミングをずらすため
}

void draw(){
  background(255);

  y = y+v;
  if(y -radius> height){
    x = random(radius,width-radius); // 出現位置をずらす
    y = -random(radius,2*radius);    // 出現タイミングをずらすため
  }
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(width/2,y,2*radius,2*radius);
}
```

ゲームなどでは、沢山の敵キャラ（敵機）が移動してきます。例えば、2 つの円が移動するようなプログラムは、次の様に書くことが出来ます。

2 個の円の移動 サンプル 9-2

```
float y0,y1; // 円の中心の Y 座標
float x0,x1; // 円の中心の X 座標
float v0,v1; // 円の縦方向の移動速度
int radius;

void setup(){
  size(300,400);
  smooth();
  radius = 10;
  v0 = random(1,2); // 移動速度を乱数で決める
  y0 = -random(radius,2*radius);  // 出現タイミングをずらすため
  x0 = random(radius,width-radius); // 出現位置をずらす
  v1 = random(1,2); // 移動速度を乱数で決める
  y1 = -random(radius,2*radius);  // 出現タイミングをずらすため
  x1 = random(radius,width-radius); // 出現位置をずらす
}

void draw(){
  background(255);
  // 中心 (x0,y0) の円の処理
  y0 = y0+v0;
  if(y0 -radius> height){
    x0 =random(radius,width-radius);// 出現位置をずらす
    y0 = -random(radius,2*radius);  // 出現タイミングをずらすため
  }
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(x0,y0,2*radius,2*radius);

  // 中心 (x1,y1) の円の処理
  y1 = y1+v1;
  if(y1 -radius> height){
    x1 = random(radius,width-radius);// 出現位置をずらす
    y1 = -random(radius,2*radius);   // 出現タイミングをずらすため
  }
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(x1,y1,2*radius,2*radius);
}
```

変数 x0 は x1 に、変数 y 0 は y1 に変数 v0 は v1 に変わっているだけです。

サンプル 9-1 とサンプル 9-2 では大きな違いはありません。サンプル 9-2 では、2 つの円を扱う必要があるので、中心が (x0,y0) の円と中心が (x1,y1) の円の 2 つの円を扱ってるので、同じ内容の処理で、変数名の部分が異なっているものが書かれています。

サンプル 9-2 の方針で、もっと沢山の円が移動するプログラムを作るとすると、沢山の変数を用意する必要があります。例えば、10 個の円を表示するように拡張する場合には、変数 x0 ～ x9、y0 ～ y9、v0 ～ v9 が必要になるようの思います。このように沢山の変数

0 から数え始めているので、終わりは 10 ではなく、9 になります。大丈夫ですか？

を扱うプログラムを書くことは可能ですが、変数名の数字の部分だけが異なった処理を 10 回書く必要があります。これはかなり面倒です。

この面倒を避けるためには、変数を変数名と数字のペアで指定できれば、解決できそうです。サンプル 9-2 でも、変数名の x,y,v の部分は共通で、変数名の最後の数字の部分が異なっているだけです。Processing では、このような変数名と数字のペアで変数を指定する方法として、配列と呼ばれるものが準備されています。

配列も変数の一種なので、使う前に宣言が必要となります。また、どんな種類のデータを変数に保存するかというデータ型の情報も必要となります。そのため、次のような方法で配列型変数の宣言を行います。

プログラムを作成するときの、大きな方針の 1 つは、「同じような処理はまとめて書く」です。優秀なプログラマはものぐさです。それに、コンピュータは同じ繰り返しを飽きずに処理することが得意です。

数学などでも、x_0,x_1 のような添え字を使うことがあります。

英語では、配列のことを array と呼びます。
配列に保存されている一つ一つのデータのことを要素 (elment) と呼ぶことがあります。

表 9-1 配列型変数の宣言

宣言方法	宣言例
データ型　[] 配列変数名	float [] x; int [] radius; String [] msg;

表 9-1 の宣言だけでは、何個の分のデータを保存するかということがわからないので、この宣言以外に何個のデータを保存する場所を用意するために、次のような処理を行う必要があります。

この処理を行わないとエラーになります。また、確保した個数以上のデータを使う場合にもエラーとなります。

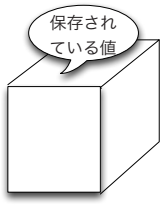
表 9-2 配列型変数のための場所の確保

宣言方法
配列変数名 = new データ型 [確保するデータ数];
x = new float[10];
radius = new int[10];
msg = new String[20];

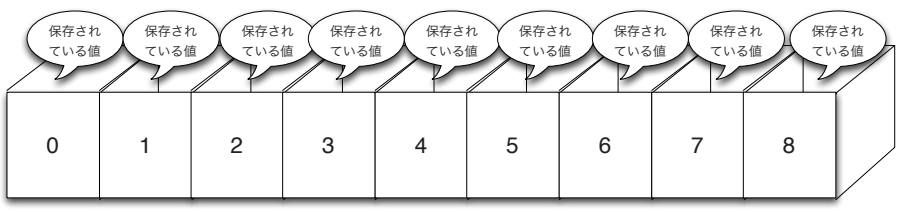
変数の宣言と場所の確保を同時に行うことが出来ます。

表 9-1 通常の変数と配列型変数のイメージ

変数：一つの名前に、一つのデータを保存



配列型変数：一つの名前に、複数のデータを保存



配列に保存されたデータは、添え字の番号順に一列に並んでいるイメージとなっています。恐らくメモリ内でも一列に並んでいると思います。

表 9-3 配列型変数の宣言と場所の確保

宣言方法
データ型 [] 配列変数名 = new データ型 [確保するデータ数];
float[] x = new float[10];
int[] radius = new int[10];
String [] msg = new String[20];

同じデータ型の名前を 2 箇所に書かないといけないのが、面倒ですよね。もう少し上手く設計して欲しい気がします。

配列の中にデータを保存したり、読み出したするためには、以下のような方法をとります。

表 9-4 配列の要素へのアクセス

配列のアクセス	例
配列変数名 [番号]	x[0] = random(10); y[1] = y[1]+v[1];

これらのサンプルのように、色々な関数を組み合わせることでどんどん複雑なプログラムを作ることが出来るようになります。

つまり、プログラム中で配列の中に保存されているデータを読み出したり、配列の中にデータを保存したりする場合には、必ず [と] の間に数字を指定して指示します。この数字を添え字またはインデックスと呼びます。添え字は配列の中のどのデータを使うのかを指定したり、配列のどの場所にデータを保存するのかを指定します。プログラム中では複数の配列型変数を使用することができます。そのために、どの配列かを区別するために配列変数名を利用します。Processing の配列では添え字の番号は 0 からスタートします。そのため、一番最後の要素の添え字番号は「配列の要素数 -1」となります。例えば、10 個の要素を持つ配列の要素は 0 から 9 までの添え字番号で指定することが出来ます。

配列は数多くのデータを添え字番号で指定することが出来るので、繰り返し処理と相性の良い方法となっています。

以上の説明をもとに、サンプル 9-2 を配列を使ったものに書きかえてみます。

2 個の円の移動（配列版）サンプル 9-3

```
float[] x = new float[2];// 円の中心の X 座標
float[] y = new float[2];;// 円の中心の Y 座標
float[] v = new float[2];;// 円の縦方向の移動速度
int radius;

void setup(){
  size(300,400);
  smooth();

  radius = 10;
  v[0] = random(1,2); // 移動速度を乱数で決める
  y[0] = -random(radius,2*radius);  // 出現タイミングをずらすため
  x[0] = random(radius,width-radius);
  v[1] = random(1,2); // 移動速度を乱数で決める
  y[1] = -random(radius,2*radius);  // 出現タイミングをずらすため
  x[1] = random(radius,width-radius);
}
```

このサンプルでは、配列型変数の宣言と保存場所の確保を同時に行っています。

```
void draw(){
  background(255);

  y[0] = y[0]+v[0];
  if(y[0]-radius > height){
    x[0] = random(radius,width-radius);// 出現位置をずらす
    y[0] = -random(radius,2*radius);; // 出現タイミングをずらすため
  }
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(x[0],y[0],2*radius,2*radius);

  y[1] = y[1]+v[1];
  if(y[1]-radius > height){
    x[1] = random(radius,width-radius);// 出現位置をずらす
    y[1] = -random(radius,2*radius);; // 出現タイミングをずらすため
  }
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(x[1],y[1],2*radius,2*radius);
}
```

サンプル 9-2 とサンプル 9-3 では、大きく変わっていません。サンプル 9-2 で x0 などなっている部分が x[0] などになっているだけです。例えば、「y[0]=y[0]+v[0];」は、配列変数 y の添え字番号 0 番の値と配列変数 v の添え字番号 0 番の値を加えて、その結果を配列変数 y の添え字番号 0 番に保存するという意味です。

サンプル 9-3 の赤字の部分に注目すると、サンプル 9-4 のように for 命令を使った繰り返し処理で書けるように思えます。そこで、for 命令を使って「書きかえたものがサンプル 9-4 です。

2 個の円の移動（for 命令 + 配列版）サンプル 9-4

```
float[] x;// 円の中心の X 座標
float[] y;// 円の中心の Y 座標
float[] v;// 円の縦方向の移動速度
int radius;
void setup(){
  size(300,400);
  smooth();
  radius = 10;
  x = new float[2];
  y = new float[2];
  v = new float[2];
  for(int i=0;i<2;i++){
    v[i] = random(1,2); // 移動速度を乱数で決める
    y[i] = -random(radius,2*radius);  // 出現タイミングをずらすため
    x[i] = random(radius,width-radius);
  }
}
```

配列変数の宣言と、保存する場所の確保を別な位置で行うように変更してみました。

繰り返し処理のカウンタ変数 i を使って、配列の要素に値を保存しています。

```
void draw(){
  background(255);

  for(int i=0;i<2;i++){
    y[i] = y[i]+v[i];
    if(y[i]-radius > height){
      x[i] =random(radius,width-radius);// 出現位置をずらす
      y[i] = -random(radius,2*radius);  // 出現タイミングをずらす
    }
    stroke(255,10,10);
    fill(255,10,10);
    ellipse(x[i],y[i],2*radius,2*radius);
  }
}
```

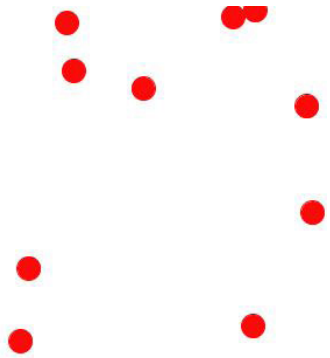
繰り返し処理のカウンタ変数 i を使って、配列の要素にアクセスしています。

サンプル 9-4 のようになると、配列と繰り返し処理の組み合わせが強力なことが見えてきます。例えば、円の数を 10 個に増やしたい場合には、サンプル 9-5 のようになります。

10 個の円の移動（for 命令 + 配列版）サンプル 9-5

```
float[] x;// 円の中心の X 座標
float[] y;// 円の中心の Y 座標
float[] v;// 円の縦方向の移動速度
int radius;
void setup(){
  size(300,400);
  smooth();
  radius = 10;
  x = new float[10];
  y = new float[10];
  v = new float[10];
  for(int i=0;i<10;i++){
    v[i] = random(1,2); // 移動速度を乱数で決める
    y[i] = -random(radius,2*radius);  // 出現タイミングをずらすため
    x[i] = random(radius,width-radius);
  }
}
void draw(){
  background(255);

  for(int i=0;i<10;i++){
    y[i] = y[i]+v[i];
    if(y[i]-radius > height){
      x[i] =random(radius,width-radius);// 出現位置をずらす
      y[i] = -random(radius,2*radius);  // 出現タイミングをずらす
    }
    stroke(255,10,10);
    fill(255,10,10);
    ellipse(x[i],y[i],2*radius,2*radius);
  }
}
```

配列に関わる 2 を 10 に置き換えただけです。



繰り返し回数を配列の要素数より大きな値を指定すると、「y[i]=y[i]+v[i];」の部分で、カウンタ変数 i の値が添え字番号の範囲を超えてしまいます。そうすると、ArrayIndexOutOfBoundsException というエラーが出て、実行が停止します。

サンプル 9-5 を書きかえて、100 個の円を表示するようにするためには、サンプル 9-5 中の赤字の 10 の部分を 100 に書きかえるだけで実現出来ます。配列と繰り返し処理を組み合わせると沢山の物体を表示したりするような処理を簡単に書くことができます。サンプル 9-4 や 9-5 で、for 命令の繰り返し回数を指定している数字は、配列にいくつのデータを保存することが出来るのか（要素数）を指定しています。配列を使ったプログラムでは、要素数を知りたいことが多いので、length というプロパティが用意されています。

length の意味はわかりますか？

表 9-5 配列の要素数の取得

要素数の取得方法	例
配列変数名 .length	for(int i=0;i<x.length;i++){ radius.length; msg.length;

この length を使うと、サンプル 9-5 は次のように書きかえることが出来ます。

10 個の円の移動（length 使用版）サンプル 9-5'

書きかえたのは、赤字の部分のみです。

```
float[] x;// 円の中心の X 座標
float[] y;// 円の中心の Y 座標
float[] v;// 円の縦方向の移動速度
int radius;
void setup(){
  size(300,400);
  smooth();
  radius = 10;
  x = new float[10];
  y = new float[10];
  v = new float[10];
  for(int i=0;i<x.length;i++){
    v[i] = random(1,2); // 移動速度を乱数で決める
    y[i] = -random(radius,2*radius);  // 出現タイミングをずらすため
    x[i] = random(radius,width-radius);
  }
}
void draw(){
  background(255);

  for(int i=0;i<y.length;i++){
    y[i] = y[i]+v[i];
    if(y[i]-radius > height){
      x[i] =random(radius,width-radius);// 出現位置をずらす
      y[i] = -random(radius,2*radius);  // 出現タイミングをずらす
    }
    stroke(255,10,10);
    fill(255,10,10);
    ellipse(x[i],y[i],2*radius,2*radius);
  }
}
```

for 命令で繰り返し回数を指定している部分は、配列変数 x,y,v の要素数は全て同じのなので、x.length は、y.length でも v.length でも同じ結果となります。draw 関数の内の for 命令での繰り返しも同じです。

当然、配列は float 型以外でも利用することが出来ます。サンプル 9-5' を書きかえて、color 型の配列変数を使って円の色を変え、int 型の配列変数を使って円の半径を変えてみます。また、表示する円の個数を 50 個に増やしてみます。これを行ったのが、サンプル 9-6 です。色を乱数で変化させるために、colorMode を HSB に変更しています。また、半径の情報を保存している配列 radius には int 型のデータを保存するので、「int(random(5,15);」のように、乱数の値を int 型の値に変更して保存しています。配列名は、変数名と同じように使えますので、x や y のように短い名前ではなく、長い名前を使うことも出来ます。また、円の y 座標の値を決めるために、円の半径の情報を利用しているので、最初に半径の大きさを決めています。

10 個の円の移動（length 使用版）サンプル 9-6

```
float[] x = new float[50];// 円の中心の X 座標
float[] y = new float[50];// 円の中心の Y 座標
float[] v = new float[50];// 円の縦方向の移動速度
color[] cols = new color[50]; // 円の色の情報
int[] radius = new int[50]; // 円の半径の情報

void setup(){
  size(300,400);
  smooth();
  colorMode(HSB,359,99,99);
  for(int i=0;i<x.length;i++){
    radius[i] = int(random(5,15));
    v[i] = random(1,2);
    y[i] = -random(radius[i],2*radius[i]);
    x[i] = random(radius[i],width-radius[i]);
    cols[i] = color(random(360),random(50,100),random(50,100));
  }
}

void draw(){
  background(0,0,99);

  for(int i=0;i<x.length;i++){
    y[i] = y[i]+v[i];
    if(y[i] > height){
      x[i] =random(radius[i],width-radius[i]);
      y[i] = -random(radius[i],2*radius[i]);
    }
    stroke(cols[i]);
    fill(cols[i]);
    ellipse(x[i],y[i],2*radius[i],2*radius[i]);
  }
}
```

配列には、色々な使い方があります。少し複雑な配列の使い方を紹介します。

サンプル 9-7 は、ウインドウ上に表示された円をドラッグするというサンプルです。このサンプルでは、円の中心座標を xBall と yBall という配列に保存しています。同じ添え字番号のデータが同じ円の情報を表しています。つまり、添え字番号で、円を区別していることになります。マウスボタンが押されたときに、マウスの座標と円の中心座標との距離を計算し、その距離が円の半径以下ならば、その円を摑んだと判定しています。円を摑んだと判定したら、その円の添え字番号を int 型変数 pickedID に保存し、boolean 変数 picking に true を代入します。そして、マウスがドラッグされる度に、直前のマウスの位置 (pmouseX と pmouseY) と現在のマウスの位置（mouseX と mouseY）の差がマウスの移動量となるので、摑まれている円の中心座標にマウスの移動量を加えています。マウスボタンが離されたら、円を摑む動作を終了したと判断し、picking の値を false に、pickedID には、あり得ない数字である -1 を代入しています。

円を摑んで移動させる サンプル 9-7

```
int pickedID = -1;
boolean picking=false;
float[] xBall = new float[10];
float[] yBall = new float[10];
color[] cBall = new color[10];
int radius = 10;

void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
  for(int i=0;i<xBall.length;i++){
    xBall[i] = random(radius,width-radius);
    yBall[i] = random(radius,height-radius);
    cBall[i] = color(random(360),99,99);
  }
}

void mouseDragged(){
  if(picking){
    xBall[pickedID] += (mouseX-pmouseX);
    yBall[pickedID] += (mouseY-pmouseY);
  }
}

void mouseReleased(){
  picking = false;
  pickedID = -1;
}
```

picking が true の時には、どれかの円を摑んでおり、どの円かを表す情報は pickedID に保存されています。

この処理は、mousePressed 関数の中に書かれています。

この処理は、mouseDragged 関数の中に書かれています。

この処理は、mouseReleased 関数の中に書かれています。

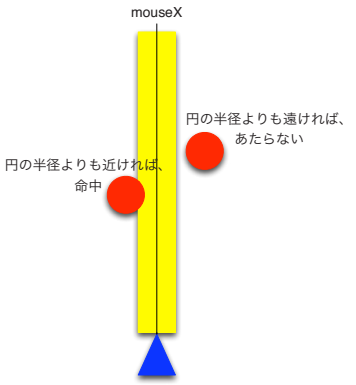
円を摑んでいる場合、pickedID には、添え字番号が記録さているので、-1 は「あり得ない」値となっています。

```
void mousePressed(){
  for(int i=0;i<xBall.length;i++){
    if(dist(mouseX,mouseY,xBall[i],yBall[i]) <= radius){
      picking = true;
      pickedID = i;
    }
  }
}
void draw(){
  background(0,0,99);
  for(int i=0;i < xBall.length;i++){
    stroke(cBall[i]);
    fill(cBall[i]);
    ellipse(xBall[i],yBall[i],2*radius,2*radius);
  }
}
```

サンプル 9-8 は、赤い玉を打ち落とすようなサンプルです。mousePressed 関数の中で、ビーム（？）の円への命中判定を行っています。ここでは、簡易的な命中判定を行っています。

ビーム攻撃 サンプル 9-8

```
float[] x;// 円の中心の X 座標
float[] y;// 円の中心の Y 座標
float[] v;// 円の縦方向の移動速度
int radius;
int hit; // 命中回数
PFont font = loadFont("Serif-48.vlw");
// 引数 i で指定された円の位置を初期状態に設定する
void setRandomPosition(int i){
  v[i] = random(1,2); // 移動速度を乱数で決める
  y[i] = -random(radius,2*radius);  // 出現タイミングをずらすため
  x[i] = random(radius,width-radius);
}
// 宇宙船（？）を表示する
void drawShip(){
  stroke(10,10,255);
  fill(10,10,255);
  triangle(mouseX-14,mouseY+20,
           mouseX,mouseY-14,mouseX+14,mouseY+20);
  if(mousePressed){ // マウスボタンが押されていたらビームを描画
    stroke(255,255,10);
    line(mouseX,mouseY-14,mouseX,0);
  }
}
// 得点（今回は単に命中回数）を表示
void displayScore(){
  fill(255);
  textAlign(RIGHT);
  text(hit,width-10,60);
}
```



ビームの命中判定方法

どんなときに、命中判定を誤るかわかりますか？

```
void setup(){
  size(400,400);
  smooth();
  hit = 0;
  radius = 10;
  x = new float[10];
  y = new float[10];
  v = new float[10];
  for(int i=0;i<x.length;i++){
    setRandomPosition(i);
  }
  textFont(font,48);
}

void draw(){
  background(0);

  for(int i=0;i<y.length;i++){
    y[i] = y[i]+v[i];
    if(y[i]-radius > height){
      setRandomPosition(i);
    }
    stroke(255,10,10);
    fill(255,10,10);
    ellipse(x[i],y[i],2*radius,2*radius);
  }
  drawShip();
  displayScore();
}

void mousePressed(){
  for(int i=0;i<x.length;i++){
    // ビームと円との命中判定を行う。
    if(abs(x[i]-mouseX) <= radius && mouseY >= y[i]){
      hit++;
      setRandomPosition(i);
    }
  }
}
```

2 つの例では、配列の扱い方に関しては、単純なものを紹介しました。次は、もう少し複雑な配列の操作を伴ったサンプルです。

このサンプル 9-9 では、frameCount という Processing 変数を使用しています。この frameCount 変数は、何回画面を描画したかを保存しています。そのため、描画回数に依存して何かの状況を変化させたいときに、便利な変数です。サンプル 9-9 では、frameCount の値を 360 で割ったときの余りを色相の情報として利用しています。このサンプルでは、100 回分のマウスの位置とその時の色の情報を配列に保存しています。そして、描画が行われる度に、配列に入っている情報

を 1 ずつ移動させています。つまり、

x[99] と y[99] に現在のマウスの位置、cols[99] にその時の色情報
x[98] と y[98] に一つ前のマウスの位置、cols[98] にその時の色情報
x[97] と y[97] に 2 つ前のマウスの位置、cols[97] にその時の色情報
x[96] と y[96] に 3 つ前のマウスの位置、cols[96] にその時の色情報
:
:
:
x[2] と y[2] に 97 つ前のマウスの位置、cols[2] にその時の色情報
x[1] と y[1] に 98 つ前のマウスの位置、cols[1] にその時の色情報
x[0] と y[0] に 99 つ前のマウスの位置、cols[0] にその時の色情報

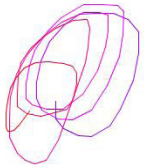
というようにマウスの位置と色の情報を配列に保存しています。そして、過去の情報を一つ前に移動させます。つまり、x[i+1] の値を x[i] に、y[i+1] の値を y[i] に、cols[i+1] の値を cols[i] に代入しています。この処理を i の値を変えながら繰り返し行う必要があるので、for 命令による繰り返し処理で、この処理を実現しています。

過去の情報を利用する サンプル 9-9

```
int[] x = new int[100];// マウスの X 座標の値を保存
int[] y = new int[100];// マウスの Y 座標の値を保存
color[] cols = new color[100];// 色の情報を保存

void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
}

void draw(){
  background(0,0,99);
  stroke(255,10,10);
  for(int i=0;i<x.length -1;i++){ // 一つずつ前に移動させる
    x[i] = x[i+1];
    y[i] = y[i+1];
    cols[i] = cols[i+1];
  }
  x[x.length-1] = mouseX; // 最後 (99) に現在の情報を代入する
  y[y.length-1] = mouseY;
  cols[cols.length-1] = color(frameCount % 360,99,99);
  // 配列に保存されている情報を利用して、折れ線を描画する
  for(int i=1;i<x.length;i++){
    stroke(cols[i]);
    line(x[i-1],y[i-1],x[i],y[i]);
  }
}
```

new を使って、float 型や int 型の配列変数のための場所を確保した場合には、0 が代入されています。

x.length-1 を x.length にすると、エラーになります。なぜだか、わかりますか？



配列の使い方には、1) 配列変数の宣言、2)new を利用して情報を保存する場所を確保、という手順を取ることが一般的です。しかし、配列の要素の記憶する情報が簡単に作れる場合で、その数が少ない場合には、次のよう方法を取ることが出来ます。この場合には、new を利用した場所の確保は必要ありません。

表 9-6 配列の宣言と初期化

宣言と初期化	例
データ型 [] 変数名 = {0 番に保存するデータ , 1 番に保存するデータ , … };	String [] msg = {"Riho", "Tomoyo", "Nene"};

これを利用したサンプル 9-10 を示します。

配列の宣言と初期化 サンプル 9-10

```
String [] names = {"Riho",
                   "Tomoyo",
                   "Nene",
                   "Manaka",
                   "Rinko",
                   "Narumi"};
PFont font = loadFont("Serif-48.vlw");

void fadeToWhite(){
  stroke(0,0,99,20);
  fill(0,0,99,20);
  rectMode(CORNER);
  rect(0,0,width,height);
}

void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
  textFont(font,48);
}

void draw(){
  fadeToWhite();
  // 表示する文字列を選択する
  int idx = int(random(names.length));
  fill(color(random(360),99,99));
  text(names[idx],random(width),random(height));
}
```

ちょっと、高度な話題です。

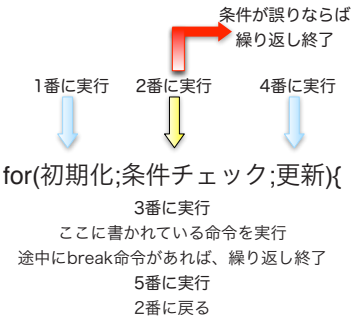
ところで、円を摑んで移動させるというサンプル 9-7 では、mousePressed 関数の for 命令の利用した繰り返し部分では、どの円

を摑むかが決まれば、最後まで繰り返し処理を実行する必要はありません。繰り返し処理では、途中で繰り返し処理を終了しても良い場合があります。このような機能を実現するために、Processing では break 命令が用意されています。繰り返しの処理の中で break 命令が来ると、一番内側の処理から抜け出します。

break 命令を利用して、サンプル 9-7 を書きかえたものが、サンプル 9-11 です。

円を摑んで移動させる (break 版) サンプル 9-11

```
int pickedID = -1;
boolean picking=false;
float[] xBall = new float[10];
float[] yBall = new float[10];
color[] cBall = new color[10];
int radius = 10;

void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
  for(int i=0;i<xBall.length;i++){
    xBall[i] = random(radius,width-radius);
    yBall[i] = random(radius,height-radius);
    cBall[i] = color(random(360),99,99);
  }
}

void mouseDragged(){
  if(picking){
    xBall[pickedID] += (mouseX-pmouseX);
    yBall[pickedID] += (mouseY-pmouseY);
  }
}

void mouseReleased(){
  picking = false;
  pickedID = -1;
}

void mousePressed(){
  for(int i=0;i<xBall.length;i++){
    if(dist(mouseX,mouseY,xBall[i],yBall[i]) <= radius){
      picking = true;
      pickedID = i;
      break; // もう探す必要がないので、繰り返し処理を終了する
    }
  }
}
```



break 命令は、Processing をはじめとする C 言語系列のプログラミング言語で利用することが出来ます。

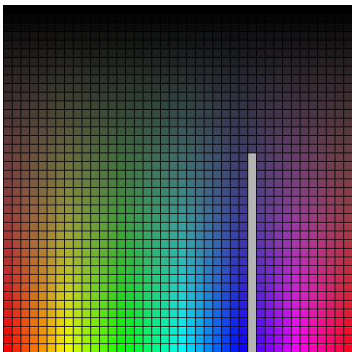
一番内側の繰り返し処理から抜け出します。とはいえ、この例では、「一番内側」というのがピンと来ないですよね。

```
void draw(){
  background(0,0,99);
  for(int i=0;i < xBall.length;i++){
    stroke(cBall[i]);
    fill(cBall[i]);
    ellipse(xBall[i],yBall[i],2*radius,2*radius);
  }
}
```

サンプル 9-12 を見ると break 命令で抜け出す範囲の状況がわかるかも知れません。実行結果とプログラムを見比べて下さい。break 命令が含まれている一番内側の繰り返しを抜けるので、for(int y…){…} の部分から抜け出すだけなので、外側の繰り返し処理 for(int x…){…} の部分は引き続き実行されます。

break の例その 2 サンプル 9-12

```
void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
}

void draw(){
  background(255);
  stroke(0);
  for(int x=0;x<width;x+=10){
    for(int y=0;y<height;y+=10){
      if((x <= mouseX && mouseX < x+10) &&
         (y <= mouseY && mouseY < y+10)){
        break;
      }
      fill(map(x,0,width-1,0,359),
           map(y,0,height-1,0,99),
           map(y,0,height-1,0,99));
      rect(x,y,10,10);
    }
    // break 命令が実行されると、ここに来る
  }
}
```



この if 命令の条件式はどんな条件を表しているでしょうか？

break 命令は、for 命令による繰り返しだけでなく、while 命令の繰り返し処理からも抜け出ることができます。

ここで定義した変数 carW、carH は、drawCar 関数内部でのみ使用できます。局所変数 carW と carH が定義されているブロックはどこでしょうか？

時間に関連した関数には以下のようなものがあります。これらは、パソコンの時計に連動して、情報を求めています。

表 8-1 時間関連の関数

関数名	関数が返す値の意味
year()	現在の年を返す。
month()	現在の月（1 ～ 12）を返す。
day()	現在の日（1 ～ 31）を返す。
hour()	現在の時刻の時間を返す。
minute()	現在の時刻の分を返す。
second()	現在の時刻の秒を返す。
millis()	プログラムを実行してから経過時間をミリ秒単位で返す。

関数と言うと数学で出てくるものを思い浮かべると思います。Processing では、数学で出てくるような関数が用意されています。まずは、数の大きさに係わる関数です。

表 8-2 最小、最大関連の関数

関数名	関数が返す値の意味
min(x1,x2)	x1 と x2 の中で小さい方の値（最小値）を求める。
min(x1,x2,x3)	x1,x2,x3 の中で最小値を求める。
max(x1,x2)	x1 と x2 の中で大きい方の値（最大値）を求める。
max(x1,x2,x3)	x1,x2,x3 の中で最大値を求める。

もう少し数学っぽい関数もあります。

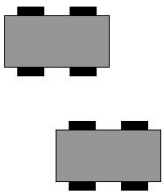
表 8-3 ちょっ数学っぽい関数

関数名	関数が返す値の意味
abs(x)	引数 x の絶対値を求めます。例えば、abs(-1.1) は 1.1、aba(3) は 3 になります。
sqrt(x)	引数 x の平方根の値を求めます。る。例えば、sqrt(4) なら 2.0 になります
sq(x)	引数 x の二乗を求めます。
pow(x,n)	x の n 乗を求めます。例えば、pow(2,4) は 16 になります。
exp(x)	指数関数の値を求めます。ネイピア数 e の x 乗を求めます。
log(x)	自然対数の値を求めます。
dist(x1, y1, x2, y2)	2 点 (x1,y1) と (x2,y2) の間の距離を求めます。
constrain(v, m0, m1)	引数 v の値が m0 以上 m1 以下なら v を返し、v の値が m0 よりも小さければ m0 を返し、v の値が m1 よりも大きければ m1 を返すような関数です。

ここで出てくる引数名、引数名 1、引数名 2 などは、この関数の中だけで、有効な変数となります。
また、引数として宣言された変数は、関数内で局所変数として利用することが出来ます。

関数名	関数が返す値の意味
lerp(v0,v1,t)	(1-t)*v0+t*v1 という値を求めます。線形補間と呼ばれる計算方法です。
map(v, low1, high1, low2, high2)	2 点 (low1,low2) と (high1,high2) を通る直線おいてい、X 座標の値が v の時の Y 座標の値を求める関数です。別な言い方をすると low1 以上 high1 以下の値 v を low2 以上 high2 以下の値に変換するとどんな値になるかを求めるものです。要するに一次関数の値を計算しています。 $y = \frac{high2 - low2}{high1 - low1}(v - low1) + low2$

float 型の変数 x と y は、drawCar 関数の内部だけで有効な変数となります。



でもやっぱり、関数と言うと三角関数のような気がします。Processing でも三角関数が用意されています。

表 8-4 三角関数関連

関数名	関数が返す値の意味
sin(x)	正弦関数 sin の値を求めます。
cos(x)	余弦関数 cos の値を求めます。
tan(x)	正接関数 tan の値を求めます。
degrees(x)	ラジアンから度に値を変換します。
radians(x)	度からラジアンに値を変換します。
asin(x)	sin の逆関数の値を求めます。つまり、sin y = x となる y の値を求めます。ただし、y の値は -PI/2 から PI/2 となります。
acos(x)	cos の逆関数の値を求めます。つまり、cos y = x となる y の値を求めます。ただし、y の値は -PI/2 から PI/2 となります。
atan(x)	tan の逆関数の値を求めます。つまり、tan y = x となる y の値を求めます。ただし、y の値は -PI/2 から PI/2 となります。
atan2(y,x)	原点と点 (x,y) を通る直線と X 軸のなす角度をもとめます。ただし、角度の値は -PI から PI の範囲の値となります。

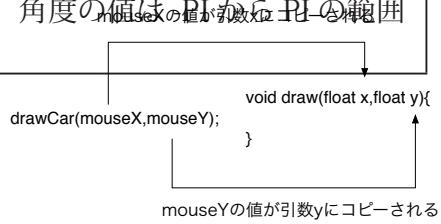


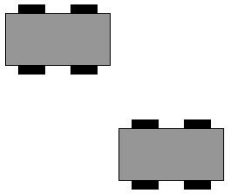
図 7-5 関数呼び出し時の引数のコピー

全然サンプルがないのも何なので、少し載せておきます。まずは、map 関数を使ったものです。これは、マウスカーソルの動きに合わせて、真ん中にある円を動かすものです。円はある範囲 (x0 ～ x1) の間しか動きません。このような動作を map 関数を使って作りだしています。つまり、mouseX の値を x0 から x1 の値に変換し、その値を円の中心の X 座標値として使っています。

map 関数の使用例 サンプル 8-1

```
int x0,x1;

void setup(){
  size(400,200);
  smooth();
  x0 = 80;
  x1 = width-x0;
}

void draw(){
  background(255);
  strokeWeight(3);
  stroke(0);
  line(x0,height/2,x1,height/2);
  fill(100);
 // mouseX の値を x0 ～ x1 の間の値に変換
  float x = map(mouseX,0,width-1,x0,x1);
  strokeWeight(1);
  ellipse(x,height/2,20,20);
}
```

裏技 (this. 大域変数名) を使うとアクセスすることが出来ます。



次は、atan2 を使ったサンプルです。原点とマウスカーソルの位置を結ぶ直線と X 軸のなす角を弧で示すようなサンプルです。ついでに、その角度の値を degree 関数を使って、度単位で表示しています。なの、弧の部分は arc 関数を使って描画しています。arc 関数では、弧がスタートする時の角度と終了する時の角度を指定する必要があります。また、原点との距離を計算して、3 分の 1 の位置に弧を表示するようにしています。

atan2,dist などの関数の使用例 サンプル 8-2

剰余演算 %（余りを求める）を使って、車の繰り返し移動を実現しています。
あることを行うプログラムには、色々なやり方があります。コンピュータに指示するやり方のことをアルゴリズム (algorithm) と呼んでいます。

```
PFont font;

void setup(){
  size(400,400);
  smooth();
  font = loadFont("Serif-48.vlw");
  textFont(font);
}

void draw(){
  background(255);
  stroke(0);
  line(0,0,mouseX,mouseY);
  noFill();
   float theta = atan2(mouseY,mouseX);// 線分と X 軸のなす角度を求め
る
  float l = dist(0,0,mouseX,mouseY);// 原点との距離を求める
  arc(0,0,2*l/3,2*l/3,0,theta);
  line(0,0,mouseX,mouseY);
  String deg = str(degrees(theta));
  fill(255,10,10);
  text(deg,width/2-textWidth(deg)/2,height/2);
}
```

関数の宣言（その 2）

Processing が用意している関数について説明してきました。今回説明した関数は、何らかの値（戻り値）を返すような関数でした。前回の講義では、処理をまとめるという観点から関数の説明をしました。そのため、値を返すという話はありませんでした。今回説明したような値を返すような関数を定義することも出来ます。

そのためには、表 8-5 のような形でプログラムを書きます。値を返す必要があるために、戻り値のデータ型を指定する必要があります。関数定義の中で、戻り値を決定する（どんな値を返すのか）必要があります。そのために、関数名の前に戻り値のデータ型を置きます。戻り値を指定するために、return 命令を使います。「return　式;」とすると、この式の値が関数の戻り値となります。また、return 命令を実行すると、その場所で関数の実行が終わります。関数の定義中に、複数の return 命令があっても、問題はありません。逆にどのにも return 命令がないと、Processing はどんな値を戻り値とすればよいのか、わからないので、エラーとなります。関数の定義は、プログラム中のどこからでも始めることが出来ます。ただし、他の関数の定義中などでは出来ません。

デカルトの「検討しようとする難問をよりよく理解するために、多数の小部分に分割すること」という考え方が基礎にあります。

表 8-5 関数定義の仕方（その 3）

関数定義のパターン

戻り値のデータ型 関数名 (){
関数処理の内容を書きます。
どこかに、return 命令が必要です。
変数なども使うことができます。
}

戻り値のデータ型 関数名 (データ型名　引数名){
関数処理の内容を書きます。
どこかに、return 命令が必要です。
変数なども使うことができます。
}

戻り値のデータ型 関数名 (データ型名 1　引数名 1,
データ型名 2　引数名 2…){
関数処理の内容を書きます。
どこかに、return 命令が必要です。
変数なども使うことができます。
}

サンプル 8-3 では、” 年 / 月 / 日 ” の形式で、今日の日付を返す関数 today を定義しています。

戻り値を持った関数定義の例その 1 サンプル 8-3

```
PFont font;

// 今日の日付を返す関数 today を定義、戻り値は String 型
String today(){
  String msg = year()+"/"+month()+"/"+day();
  return msg; // 戻り値は msg
}

void setup(){
  size(300,200);
  smooth();
  font = loadFont("Serif-48.vlw");
  textFont(font);
}

void draw(){
  background(255);
  fill(0);
  String msg = today(); // 自分で定義した関数は自由に使うことが出来る。
  text(msg,width/2-textWidth(msg)/2,height/2);
}
```

サンプル 8-4 に関数定義の部分だけの部分の例を示します。

戻り値を持った関数定義の例その 2 サンプル 8-4

モジュール化（関数の利用）の特徴して、「複雑な機能を単純な独立した機能に分割して管理する」があります。

```
float myConstraint1(float v,float m0,float m1){
  flaot ans;
  ans = v;
  if(v > m1){
    ans = m1;
  }else if(v < m0){
    ans = m0;
  }
  return ans;
}

float myConstraint2(float v,float m0,float m1){
  if(v > m1){
    return m1;
  }else if(v < m0){
    return m0;
  }else{
    return v;
  }
}

float myDist(float x0,float y0,float x1,float y1){
  return sqrt(sq(x0-x1)+sq(y0-y1));
}
```

情報メディア基盤ユニットの単位は必ず取得してよ。これは契約だよ。

動作しないサンプルでは、面白くないので、サンプル 8-3 を改良して、今日の日付を " 月 / 日 / 年 " の形で表示するにします。この際に、月は英語表記の略称とします。今回のサンプルでは if 命令の山になるので、ちょっとプログラムは長くなります。

戻り値を持った関数定義の例その 3 サンプル 8-5

```
PFont font;

void setup(){
  size(300,200);
  smooth();
  font = loadFont("Serif-48.vlw");
  textFont(font);
}

void draw(){
  background(255);
  fill(0);
   String msg = today();// 自分で定義した関数は自由に使うことが出来る。
  text(msg,width/2-textWidth(msg)/2,height/2);
}
```

顔の輪郭部分なども欲しい気がするのですが。そうすると耳とかもいるのかな？
でも、シンプルな方が良いかな？

情報メディア基盤ユニット用資料（2016 年度第 10 章）

Processing 言語による情報メディア入門

配列（その 2）

神奈川工科大学情報メディア学科　佐藤尚

最大値を見つける

A,B,C,D の 4 つの金の塊があります。この中で一番重たいものを見つけるという問題を考えてみます。ここで使えるのは、天秤ばかりだとします。つまり、2 つのうち、重い方（もしくは軽い方）を見つけることだけが出来るものとします。

Processing での大きさの比較は、基本的に、大きいか、同じか、小さいかどうかしかわかりません。つもり、天秤ばかりを使って重さを比較しているのと同じ状況になっています。

この問題は次のようにすると、解くことができます。まず、” 一番重いかも ” という名前をつけた箱を用意します。

1. この箱の中に、A をしまいます。
2. そして、” 一番重いかも ” という箱に入っている金塊と B の重さを比較します。そこで、” 一番重いかも ” の方が、B より重ければ、そのままにします。もし、B の方が重ければ、" 一番重いかも " に入っている金塊を取り出し、B をその中に入れます。
3. 次に、” 一番重いかも ” という箱に入っている金塊と C の重さを比較します。そこで、” 一番重いかも ” の方が、C より重ければ、そのままにします。もし、C の方が重ければ、" 一番重いかも " に入っている金塊を取り出し、C をその中に入れます。
4. 次に、” 一番重いかも ” という箱に入っている金塊と D の重さを比較します。そこで、” 一番重いかも ” の方が、D より重ければ、そのままにします。もし、D の方が重ければ、" 一番重いかも " に入っている金塊を取り出し、D をその中に入れます。

” 一番重いかも ” の箱の重さは、0 とします。それが嘘くさければ、金塊も同じ箱にないっているとして、比較すれば、箱の重さは無関係となります。

この時に、” 一番重いかも ” という箱に入っている金塊が一番重い金塊となります。これを Proceesing の命令風に書くと次のようになります。

4 個の金塊のなから一番重いもものを見つける 疑似コード 11-1

```
一番重いかも = A;
if( 一番重いかも < B){// 一番重いかも > B の時には、何もしなくて良い
  一番重いかも = B;
}
if( 一番重いかも < C){// 一番重いかも > C の時には、何もしなくて良い
  一番重いかも = C;
}
if( 一番重いかも < D){// 一番重いかも > D の時には、何もしなくて良い
  一番重いかも = D;
}
```

処理の内容を、プログラミング言語風に書いたものを、疑似コードと呼ぶことがあります。

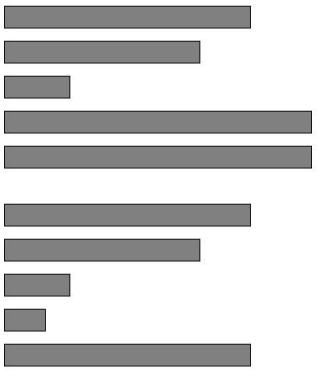
この方法を利用して作ったものがサンプル 11-1 です。このサンプ

ルでは、変数 a,b,c の値は乱数で決定し、変数 d の値は mouseX の値を指定します。上から下に横幅がそれぞれ、a、b、c、d の長方形を描き、一番下には、a,b,c,d の中で一番大きな値が横幅になるように長方形を描いています。上の疑似コードの中で、" 一番重いかも " といものは、tentativeMax という変数で表しています。

tentative の意味はわかりますか？

4 つの中から最大値を求める サンプル 11-1

```
float a,b,c,d;

void setup(){
  size(400,200);
  a = random(width);
  b = random(width);
  c = random(width);
}
void draw(){
  background(255);
  stroke(0);
  fill(128);
  d = mouseX;
  float tentativeMax = a;
  if(tentativeMax < b){
    tentativeMax = b;
  }
  if(tentativeMax < c){
    tentativeMax = c;
  }
  if(tentativeMax < d){
    tentativeMax = d;
  }
  rect(0,10,a,25);
  rect(0,50,b,25);
  rect(0,90,c,25);
  rect(0,130,d,25);
  rect(0,170,tentativeMax,25);
}
```



この 4 つ最大値を見つけるという処理を関数として書きかえたものがサンプル 11-2 です。関数名は myMax4 として、float 型の引数 x0,x1,x2,x3 をとる関数としました。

4 つの中から最大値を求める（関数版その 1）サンプル 11-2

```
float a,b,c,d;
void setup(){
  size(400,200);
  a = random(width);
  b = random(width);
  c = random(width);
}
```

0 から数え始めているので、終わりは 10 ではなく、9 になります。大丈夫ですか？

```
void draw(){
  background(255);
  stroke(0);
  fill(128);
  d = mouseX;
  float m4 = myMax4(a,b,c,d);
  rect(0,10,a,25);
  rect(0,50,b,25);
  rect(0,90,c,25);
  rect(0,130,d,25);
  rect(0,170,m4,25);
}
float myMax4(float x0,float x1,float x2,float x3){
  float tentativeMax = x0;
  if(tentativeMax < x1){
    tentativeMax = x1;
  }
  if(tentativeMax < x2){
    tentativeMax = x2;
  }
  if(tentativeMax < x3){
    tentativeMax = x3;
  }
// 一番大きな値が tentativeMax に入っているので、この値を返します。
  return tentativeMax;
}
```

関数の引数として与えらえれているので、変数 a ～ d が変数 x0 ～ x3 に変わっています。

この処理を行わないとエラーになります。また、確保した個数以上のデータを使う場合にもエラーとなります。

ところで、変数を x0 ～ x3 などと表してみると、配列を使ってサンプルが書きかえられそうな気がしてきます。サンプル 11-2 の myMax4 関数の中をジッと見てみると、

```
if(tentativeMax < x 数字 ){
  tentativeMax = x 数字 ;
}
```

ということを繰り返していることに気がつきます。この観察に基づいてサンプル 11-1 を書きかえたものがサンプル 11-3 です。

4 つの中から最大値を求める（配列版その 1）サンプル 11-3

```
float[] x = new float[4];

void setup(){
  size(400,200);
  x[0] = random(width);
  x[1] = random(width);
  x[2] = random(width);
}
```

配列に保存されたデータは、添え字の番号順に一列に並んでいるイメージとなっています。恐らくメモリ内でも一列に並んでいると思います。

```
void draw(){
  background(255);
  stroke(0);
  fill(128);
  x[3] = mouseX;
  float tentativeMax = x[0];
  for(int i=1;i<4;i++){
    if(tentativeMax < x[i]){
      tentativeMax = x[i];
    }
  }
  for(int i=0;i<4;i++){
    rect(0,40*i+10,x[i],25);
  }
  rect(0,170,tentativeMax,25);//170 = 40*4+10
}
```

x[1] と tentativeMax の比較から始めたいので、「i=1」となっています。後は、いつもの for 命令を利用した繰り返し処理です。

配列を使うことで、長方形を描く部分も、繰り返し処理を使って書いてみました。この方が、シンプルですよね。

配列を関数の引数にする

配列型も Processing にとっては、単に一つのデータ型です。つまり、Processing にとっては、単純な int 型や String 型などと配列型の扱い方を変える必要はありません。つまり、関数の引数などにも配列型の変数を使用することが出来ます。このことを利用して作成したものがサンプル 11-5 です。

配列変数を引数として渡す場合（仮引数）には、単に変数名だけを書けば OK です。関数を定義する側の引数（実引数）では、配列型であることを明示する必要があります。

つまり、データ型の部分が、「float[] 変数名」などようにします。

4 つの中から最大値を求める（配列版その 2）サンプル 11-5

```
float[] x = new float[4];

void setup(){
  size(400,200);
  x[0] = random(width);
  x[1] = random(width);
  x[2] = random(width);
}

void draw(){
  background(255);
  stroke(0);
  fill(128);
  x[3] = mouseX;
  float m4 = myMax4(x);
  for(int i=0;i<4;i++){
    rect(0,40*i+10,x[i],25);
  }
  rect(0,170,m4,25);
}
```

```
float myMax4(float[] x){
  float tentativeMax = x[0];
  for(int i=1;i<4;i++){
    if(tentativeMax < x[i]){
      tentativeMax = x[i];
    }
  }
  return tentativeMax;
}
```

ところで、サンプル 11-5 の 4 という数字は、配列の要素数を表しています。そこで、サンプル 11-6 は次の様に書きかえることが出来ます。つまり、4 の部分は x.length に書きかえることができます。この書き換えは、些細な書きかえに見えるかもしれませんが、実は非常に大きな書き換えとなっています。このように書きかえるとmyMax4 関数は 4 つの中から最大値を求めるだけでなく、引数としてわたされた配列 x に含まれいている全ての要素の中の最大値を求める関数として、機能するようになります。

4 つの中から最大値を求める（配列版その 3）サンプル 11-6

```
float[] x = new float[4];

void setup(){
  size(400,200);
  x[0] = random(width);
  x[1] = random(width);
  x[2] = random(width);
}

void draw(){
  background(255);
  stroke(0);
  fill(128);
  x[3] = mouseX;
  float m4 = myMax4(x);
  for(int i=0;i<x.length;i++){
    rect(0,40*i+10,x[i],25);
  }
  rect(0,170,m4,25);
}
float myMax4(float[] x){
  float tentativeMax = x[0];
  for(int i=1;i<x.length;i++){
    if(tentativeMax < x[i]){
      tentativeMax = x[i];
    }
  }
  return tentativeMax;
}
```

配列変数の宣言と、保存する場所の確保を別な位置で行うように変更してみました。

繰り返し処理のカウンタ変数 i を使って、配列の要素に値を保存しています。

プログラムの中では、配列の中の最大値や最小値を求めるという処理を行うことがよくあります。そのため、Processing では、配列の中の最大値や最小値を求める関数 max と min が用意されています。

表 10-1 max 関数と min 関数

関数名	意味
max(x)	配列変数 x の要素の最大値を返す関数。 x は int 型や float 型の配列です。
min(x)	配列変数 x の要素の最小値を返す関数。 x は int 型や float 型の配列です。

引数に配列を取ることが出来るのと同様に、関数の戻り値として配列を利用することができます。サンプル 11-6 の配列を確保し、乱数の値を入れている部分を関数として書きかえたものがサンプル 11-7 です。このサンプルでは、関数 initX の中で、配列を確保しているので、配列変数 x を宣言している場所で、確保する必要がなくなります。

4 つの中から最大値を求める（配列版その 4）サンプル 11-7

```
float[] x;

float[] initX(){
  float[] y = new float[4];
  y[0] = random(width);
  y[1] = random(width);
  y[2] = random(width);
  return y;
}

void setup(){
  size(400,200);
  x = initX();
}

void draw(){
  background(255);
  stroke(0);
  fill(128);
  x[3] = mouseX;
  float m4 = myMax4(x);
  for(int i=0;i<x.length;i++){
    rect(0,40*i+10,x[i],25);
  }
  rect(0,170,m4,25);
}
```

配列を関数の戻り値とする場合は、単に戻り値としたい配列変数を return 命令に書けば OK です。

関数 initX 内で確保 (new) された配列変数 y が戻り値として渡され、それが配列変数 x にコピーされるので、配列変数 x に対して、明示的に new をする必要はありません。

```
float myMax4(float[] x){
  float tentativeMax = x[0];
  for(int i=1;i<x.length;i++){
    if(tentativeMax < x[i]){
      tentativeMax = x[i];
    }
  }
  return tentativeMax;
}
```

このサンプルはサンプル 11-8 のように書きかえることもできます。このサンプル中の関数 initX は配列型の引数 xx をとっています。関数の引数として配列を利用するには、注意をする点があります。それは、関数内で配列変数の要素の値を変更すると、関数を呼び出す時点で仮引数として指定している配列の値も変わってしまう点です。つまり、関数内で配列型引数の値を変更する場合には、注意が必要です。

元の配列の要素の値も変更されることを利用して、プログラムを作る場合もあります。

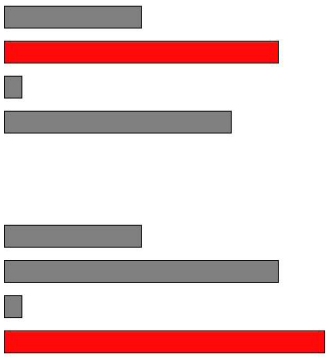
4 つの中から最大値を求める（配列版その 5）サンプル 11-8

```
float[] x = new float[4];
void initX(float[] xx){
  xx[0] = random(width);
  xx[1] = random(width);
  xx[2] = random(width);
}
void setup(){
  size(400,200);
  initX(x);
}
void draw(){
  background(255);
  stroke(0);
  fill(128);
  x[3] = mouseX;
  float m4 = myMax4(x);
  for(int i=0;i<x.length;i++){
    rect(0,40*i+10,x[i],25);
  }
  rect(0,170,m4,25);
}
float myMax4(float[] x){
  float tentativeMax = x[0];
  for(int i=1;i<x.length;i++){
    if(tentativeMax < x[i]){
      tentativeMax = x[i];
    }
  }
  return tentativeMax;
}
```

配列引数 xx の値を書きかえると、元の配列 x の値も書きかえられてしまいます。このサンプルでは、このことを利用して作成しています。

少し別な例題を考えてみます。単に最大値や最小値の値を求めるだけでなく、どの添え字の番号で最大値や最小値になっているかを知りたい場合があります。そこで、このようなことを利用したものがサンプル 11-9 です。このサンプルでは、最大値となっている値の長方形を赤色で塗りつぶします。このサンプル内の findMaxPos 関数は、引数として渡された配列の中で、最大値となっている値の添え字の番号を返す関数です。この関数の中では、直接最大値の候補となる値を保存するのではなく、添え字の値がわかれば、その配列の要素の値がわかることを利用して、最大値の候補となる値が入っている添え字の番号を変数 tentativePos に保存しながら、最大値を求めています。そして、この関数は見つけた最大値の値を返すのではなく、最大値が入っている添え字の番号を関数の戻り値としています。

複数の場所に最大値となる値が入っている場合には、添え字の番号が最も小さいものが戻り値となります。



4 つの中から最大値の入っている添え字を求める サンプル 11-9

```
float[] x = new float[4];
void setup(){
  size(400,200);
  x[0] = random(width);
  x[1] = random(width);
  x[2] = random(width);
}
void draw(){
  background(255);
  stroke(0);
  x[3] = mouseX;
  int maxPos = findMaxPos(x);
  for(int i=0;i<x.length;i++){
    fill(128);
    if(i == maxPos){
      fill(255,10,10);
    }
    rect(0,40*i+10,x[i],25);
  }
}
int findMaxPos(float[] x){
  int tentativePos = 0;
  for(int i=1;i<x.length;i++){
    if(x[tentativePos] < x[i]){
      tentativePos = i;
    }
  }
  return tentativePos;
}
```

多次元配列

今までの配列は、1 つの添え字の番号で要素にアクセスすることが出来ました。用途によっては、複数の添え字の番号で配列の要素にアクセス出来ると便利なこともあります。Processing では、このような用途のために、多次元配列という機能をもっています。2 次元配列の場合には 2 つの添え字で要素へのアクセスができますし、5 次元配列の場合には 5 つの添え字で要素へのアクセスができます。通常は多次元配列としては、2 次元配列を利用する場合が多いように思います。ここでは、2 次元配列について、説明を行います。

たとえば、オセロやマインスイーパーのように盤面の形で情報を保存する必要がある場合などです。

3 次元以上の多次元配列を同じような形で利用できます。

[] の数が配列の次元数を表しています。つまり、[] のペアが 2 つあるので、2 次元配列となっています。

表 10-2 2 次元配列型変数の宣言

宣言方法	宣言例
データ型　[][] 配列変数名	float [][] x; int [][] radius; String [][] msgs;

また、場所の確保は次の様になります。

表 10-3 配列型変数の宣言と場所の確保

宣言方法
データ型 [][] 配列変数名 = new データ型 [確保するデータ数 1][確保するデータ数 2];
float[][] x = new float[10][10];
int[][] radius = new int[10][3];
String [] msg = new String[20][2];

また、2 次元配列の要素へのアクセスは次のようになります。

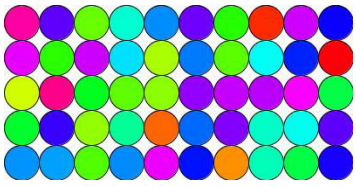
表 10-4 配列の要素へのアクセス

配列のアクセス	例
配列変数名 [番号 1][番号 2]	x[0][1] = random(10); y[1][10] = y[1][0]+v[1][2];

2 次元配列を利用したプログラムがサンプル 11-10 です。

2 次元配列のサンプル（その 1）サンプル 11-10

```
color[][] cols; // 2 次元配列の宣言
void setup(){
  size(400,200);
  colorMode(HSB,359,99,99);
  smooth();
  cols = new color[5][10];//2 次元配列ための場所の確保
  for(int y=0;y<5;y++){
    for(int x=0;x<10;x++){
      cols[y][x] = color(random(360),99,99);
    }
  }
}
```



```
void draw(){
  background(0,0,99);
  stroke(0,0,0);
  for(int y=0;y<5;y++){
    for(int x=0;x<10;x++){
      fill(cols[y][x]);
      ellipse(40*x+20,40*y+20,40,40);
    }
  }
}
```

通常の配列 (1 次元配列) の場合には、length を使って配列の要素数を取り出すことができました。2 次元配列の場合はどのようになるでしょうか？サンプル 11-10 を length を利用して書きかえたものがサンプル 11-11 です。「cols.length」のようにすると、多次元配列の最初の数字で指定できる数の個数が取り出せます。また、「cols[0].length」のようにすると、多次元配列の 2 番目の数字で指定できる数の個数が取り出せます。サンプル 11-11 の場合には、cols.length の値は 5 となり、cols[0].length や cols[1].length の値は 10 となります。

2 次元配列のサンプル（その 2）サンプル 11-11

```
color[][] cols;

void setup(){
  size(400,200);
  colorMode(HSB,359,99,99);
  smooth();
  cols = new color[5][10];
  for(int y=0;y<cols.length;y++){
    for(int x=0;x<cols[0].length;x++){
      cols[y][x] = color(random(360),99,99);
    }
  }
}

void draw(){
  background(0,0,99);
  stroke(0,0,0);
  for(int y=0;y<cols.length;y++){
    for(int x=0;x<cols[0].length;x++){
      fill(cols[y][x]);
      ellipse(40*x+20,40*y+20,40,40);
    }
  }
}
```

[] の数が配列の次元数を表しています。つまり、[] のペアが 2 つあるので、2 次元配列となっています。

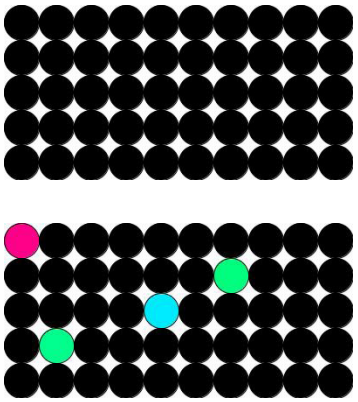
Processing で は、cols[0].length と cols[1].length の値が異なるような多次元配列を使用することが出来ます。このような多次元配列のことをジャグ配列 (jagged array) と呼びます。

多次元配列の場合も、要素の値を指定しての配列の宣言を行うことができます。これを行ったものが、サンプル 11-12 です。このサ

ンプルでは、マウスをクリックすると、その場所の円の色が cols 変数に保存されているものになるというものです。どの場所の円がクリックされたかを boolean 型 2 次元配列 flipped に保存します。最初の状態では、クリックされた円が無い状態なので、flipped 配列の要素の値は、全て false になっています。

2 次元配列のサンプル（その 3）サンプル 11-12

```
color[][] cols;
// 2 次元配列の宣言と初期化を同時に行う
boolean[][] flipped = {{false,false,false,false,false,
                        false,false,false,false,false},
                       {false,false,false,false,false,
                        false,false,false,false,false},
                       {false,false,false,false,false,
                        false,false,false,false,false},
                       {false,false,false,false,false,
                        false,false,false,false,false},
                       {false,false,false,false,false,
                        false,false,false,false,false}};
void setup(){
  size(400,200);
  colorMode(HSB,359,99,99);
  smooth();
  cols = new color[5][10];
  for(int y=0;y<cols.length;y++){
    for(int x=0;x<cols[0].length;x++){
      cols[y][x] = color(random(360),99,99);
    }
  }
}
void draw(){
  background(0,0,99);
  stroke(0,0,0);
  for(int y=0;y<cols.length;y++){
    for(int x=0;x<cols[0].length;x++){
      if(flipped[y][x]){
        fill(cols[y][x]);
      }else{
        fill(0,0,0);
      }
      ellipse(40*x+20,40*y+20,40,40);
    }
  }
}
void mouseClicked(){
  flipped[mouseY/40][mouseX/40] = true;
}
```



2 次元配列の宣言と初期化を同時に行う場合には、サンプル 11-12

の flipped 配列の宣言と初期化のように行います。

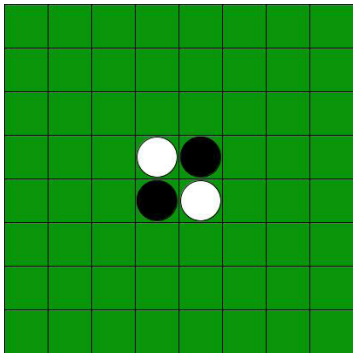
表 10-5 配列型変数の宣言と初期化

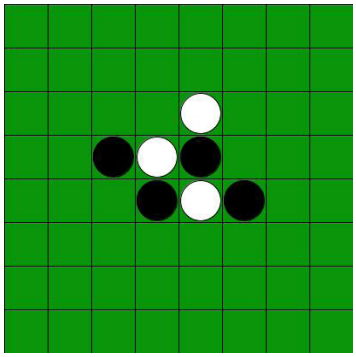
宣言方法

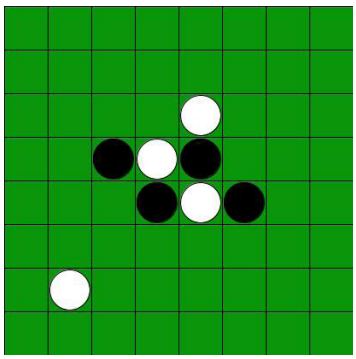
データ型 [][] 配列変数名 =
{{[0][0] に入れるデータ ,[0][1] に入れるデータ ,…},
{[1][0] に入れるデータ ,[1][1] に入れるデータ ,…},
{[2][0] に入れるデータ ,[2][1] に入れるデータ ,…},
…
};
int[][] radius = new int[10][3];
String [] msg = new String[20][2];

2 次元配列のサンプル（その 2）サンプル 11-13

```
int EMPTY = 0;
int BLACK = 1;
int WHITE = 2;
int[][] board = new int[8][8];
boolean turn = true;

void setupBoard(int[][] board){
  for(int i=0;i<board.length;i++){
    for(int j=0;j<board.length;j++){
      board[j][i] = EMPTY;
    }
  }
  board[3][3] = WHITE;
  board[4][4] = WHITE;
  board[4][3] = BLACK;
  board[3][4] = BLACK;
}
void displayBoard(int[][] board){
  background(10,150,10);
  rectMode(CENTER);
  stroke(0);
  for(int i=0;i<board.length;i++){
    for(int j=0;j<board.length;j++){
      noFill();
      rect(25+50*i,25+50*j,50,50);
      if(board[j][i] == BLACK){
        fill(0);
      }else if(board[j][i] == WHITE){
        fill(255);
      }
      if(board[j][i] != EMPTY){
        ellipse(25+50*i,25+50*j,46,46);
      }
    }
  }
}
```







```
void setup(){
  size(400,400);
  smooth();
  setupBoard(board);
}

void draw(){
  displayBoard(board);
}

void mouseClicked(){
  int x = mouseX/50;
  int y = mouseY/50;
  if(board[y][x] == EMPTY){
    if(turn){
      board[mouseY/50][mouseX/50] = BLACK;
    }else{
      board[mouseY/50][mouseX/50] = WHITE;
    }
    turn = !turn;
  }
}
```

情報メディア基盤ユニット用資料（2016 年度第 11 章）

Processing 言語による情報メディア入門

音情報の取り扱い

神奈川工科大学情報メディア学科　佐藤尚

はじめに

Special Thanks: 黒川先生

今までの授業では、Processing を用いて図形を描いたり、画像の表示を行ってきました。Processing では、音を取り扱うことができます。そのためには、外部ライブラリを利用する必要があります。外部ライブラリには、Processing 本体に同梱されているものと、ユーザがインストールする必要があるものがあります。ここでは、Processing に同梱されている Minim（ミニム）を使って、音情報を取り扱います。他にも､Processing で音を扱うライブラリがあります。また、MAX/MSP などと組み合わせて使うことも出来るようです。

Minim は、非常に高度な機能を持っています。今回の授業ではすべてを扱うことは出来ません。興味のある人は、公式のサイトを覗いてみてください。公式のサイトの URL は、
http://code.compartmental.net/tools/minim
です。

また、今回紹介する Minim の機能や説明の際に何気なく使っているいくつかの言葉の意味を理解するためには、オブジェクト指向と呼ばれる考え方を知っている必要があります。

オブジェクト指向の話は、は次回にやる予定です。

Minim は次のような機能を持っています。

- 音声データファイルの再生、wav,aiff,au,snd,mp3 などの形式のファイルに対応しています。
- MP3 ファイルの ID3 タグデータの取得。トラック名､アーティスト名などの情報を取り出すことが出来ます。
- マイクなどからの音声入力の取得
- 基本波形、ノイズの発生
- ローパスフィルタなどの適応
- FFT

などです。

以下の説明で何気なく使っているいくつかの言葉の意味を理解するためには、オブジェクト指向と呼ばれる考え方を知っている必要があります。今のところは、メソッドという言葉は関数のことだと思って下さい。また、クラスという言葉も出てきますが、これはデータ型のことだと思って下さい。

Processing で音を鳴らす

Processing の外部ライブラリである Minim を利用して音声ファイルを再生するためには、次の 3 つの方法があります。

- AudioPlayer：ファイルサイズの大きい mp3 音声ファイルをストリーミング再生します。
- AudioSnippet: ファイルサイズの大きくない音声ファイルの再生等を行います。

- AudioSample: 非常に短い音声ファイルの再生を行います。

はじめに

まずは準備です。このライブラリを使うには、メニューから [Sketch] > [Import Library] > [minim] を選びます。スケッチコードの先頭から

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;
```

が挿入されれば準備完了です。この 4 行を自分で入力してもかまいません。

最初は、AudioPlayer を利用した、ストリーミング再生の方法を紹介します。ストリーミング再生のため大きなファイルサイズの音声ファイルを取り扱うことが出来ます。ただし、再生開始が少し遅れることがあります。

音源ファイルを準備する

演奏する音楽ファイルを準備してください。ファイル形式は wav や mp3 など様々な形式が扱えます。ファイルは画像のときと同じようにまずは作成しているプログラムの pde ファイルと同じ場所 (Show Sketch Folder で表示される) に保存してください。 画像ファイルと同じ方法で、音声ファイルも Processing のスケッチ内に取り込むことが出来ます。

一番単純な音声ファイル生成のためのプログラムがサンプル 11-1 です。このサンプルはマウスをクリックすると音声ファイルの再生を行うものです。

サンプルプログラム 11-1

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer player;

void setup(){
  size(100,100);
  minim = new Minim(this);// Mimin オブジェクトの生成
  player = minim.loadFile("schoolsong.mp3");
}
void draw(){
  // やりたいことを書く
}
```

```
void mouseClicked(){
  player.play();
}

void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

このサンプルは次のような手順で音声ファイルの再生を行っています。

1.Minim オブジェクトの生成

Minim ライブラリに含まれている様々な機能（メソッド）を利用するためには、まず Minim クラスのインスタンス（Minim オブジェクト）を生成します。Minim オブジェクトはコード全体で利用するのでグローバル変数として宣言します。つまり、プログラムの先頭に以下の行を追加します。

```
Minim minim;
```

2. 音源ファイルの読み込み

minim オブジェクトの準備ができたら、メンバメソッドである loadFile を用いて音源ファイルを読み込みます。loadFile の引数は音源ファイルのファイル名です。loadFile メソッド（関数）は AudioPlayer 型のデータが戻り値となっています。そこで、その戻り値を AudioPlayer 型の変数に保存しておきます。画像ファイルを読み込んだ際に、PImage 型の変数に保存したのと似ています。この変数はコード全体で利用するのでグローバル変数として宣言します。読み込みは準備的なことですので setup 中で書いています。これで準備は完了です。

正確には、AudioPlayer 型は「AudioPlayer クラスのインスタンス（AudioPlayer オブジェクト）」です。

3. 再生

読み込んだ音声ファイルを再生するためには、loadFile 関数の結果を保存した変数（このサンプルでは player）に対して、メンバメソッド play を呼び出します。つまり、

```
player.play();
```

とすれば演奏が開始されます。戻り値を代入した変数が player でない場合には、その変数に置き換えて下さい。例えば、song 変数に代入した場合には、

```
song.play();
```

となります。サンプル 11-1 では、mouseClicked 関数の中に再生開始の命令 player.play() が入っていますので、マウスをクリックすると再生が開始されます。

4. 後始末

サンプル 11-1 の最後の方に注目して下さい。音声ファイル再生のようにプログラム本体とは別の処理が続くような処理を行っている時には、プログラム実行終了時に明示的に後始末処理を書くことが必要になる場合があります。今回の Minim ライブラリも明示的に終了処理を書く必要があります。stop 関数はプログラムの実行終了時（Stop ボタンを押す、ウインドウの閉じるボタンを押すなど）に呼び出される関数です。Minim クラス、AudioPlayer クラスのインスタンスを生成し利用した場合は、スケッチが終了するときに必ず後始末として、AudioPlayer クラスのメンバメソッド close、Minim クラスのメンバメソッド stop、および、super, stop を呼び出す必要があります。

この処理を行わないと、再度の実行の際に音声ファイルの再生などが正常に行われないなどの不都合が起きる場合があります。

繰り返し再生

サンプル 11-1 では、一度再生が終了してしまうと、再びマウスボタンをクリックしても、再生が行われません。再び再生が行われるようにするためには、どのようにしたら良いでしょうか？一番簡単な解決方法は、繰り返し再生させることです。音声ファイルの繰り返し再生を行うためには、play メソッドの代わりに loop メソッドを使用することです。これを行ったものがサンプル 11-2 です。

サンプルプログラム 11-2

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer player;

void setup(){
   size(100,100);
  minim = new Minim(this);// Mimin オブジェクトの生成
  player = minim.loadFile("schoolsong.mp3");
}

void draw(){
    // やりたいことを書く
}


void mouseClicked(){
  player.loop(); // ここを変更しました。
}
```

```
void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

もう一つの方法は、rewind メソッドと play メソッドを組み合わせて使う方法です。つまり、再生を開始する前に rewind メソッドを呼び出し、その直後に play メソッドを呼び出します。つまり、rewind メソッドで再生開始位置をファイルの先頭に戻してから、play メソッドで再生を開始します。サンプル 11-1 をこのように変更すると、マウスボタンをクリックする毎に、先頭から音声ファイルの再生が開始されます。この変更を加えたものがサンプル 11-3 です。

サンプルプログラム 11-3

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer player;

void setup(){
  size(100,100);
  minim = new Minim(this);// Mimin オブジェクトの生成
  player = minim.loadFile("schoolsong.mp3");
}

void draw(){
  // やりたいことを書く
}

void mouseClicked(){
  player.rewind(); // ここを変更しました。
  player.play(); // ここを変更しました。
}

void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

一時停止

音声ファイルの再生を一時的に停止したいことがあります。この目的のために、pause メソッドが用意されています。マウスのクリッ

クだけでは、沢山の操作を区別することができないので、keyPressed 関数と組み合わせたサンプルを作ってみます。サンプル 11-4 では、マウスをクリックすると再生開始、p ボタンを押すと一時停止、r ボタンを押すと巻き戻しとします。

サンプルプログラム 11-4

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer player;

void setup(){
  size(100,100);
  minim = new Minim(this);
  player = minim.loadFile("schoolsong.mp3");
}

void draw(){
  // Write what you do
}

void mouseClicked(){
  player.play();
}

// ここを追加しました。
void keyPressed(){
  if(key == 'p'){
    player.pause();
  }else if(key == 'r'){
    player.rewind();
  }
}

void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

ここで使用したメソッドをまとめると、以下の表のようになります。

表 11-1 AudioPlayer の再生に関連したメソッド

メソッド（関数）	機能
loadFile(“ ファイル名 ”)	音声ファイルの読み込む
play();	再生の開始

メソッド（関数）	機能
pause();	再生の一時停止
rewind();	再生開始位置を先頭に移動させる
close();	再生を中止し、ファイルストリーミングを閉じる
play(millis);	ファイルの先頭から millis 秒経過した場所から再生を開始する

ファイルサイズの余り大きくない場合

ファイルサイズの大きくないファイルを再生する場合には、AudioPlayer データ型（クラス）ではなく、AudioSnippet データ型（クラス）を利用します。AudioSnippet を利用する際には、音源ファイルを読み込むときに Minim クラスの loadSnippet メソッドを利用します。再生の方法は、AudioPlayer の場合と同じです。AudioSnippet を利用したものがサンプル 11-5 です。機能はサンプル 11-5 と同じです。

サンプルプログラム 11-5

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioSnippet player;

void setup(){
  size(100,100);
  minim = new Minim(this);
  player = minim.loadSnippet("schoolsong.mp3");
}

void draw(){
  // Write what you do
}

void mouseClicked(){
  player.play();
}

void keyPressed(){
  if(key == 'p'){
    player.pause();
  }else if(key == 'r'){
    player.rewind();
  }
}
```

```
void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

表 11-2 AudiSnippet の再生に関連したメソッド

関数（メソッド）	機能
loadSnippet(“ ファイル名 ”)	音声ファイルの読み込む
play();	再生の開始
pause();	再生の一時停止
rewind();	再生開始位置を先頭に移動させる

効果音を鳴らす

効果音など短い音を鳴らす時には AudioPlayer や AudioSnippet クラスではなく、AudioSample クラスを利用します。このクラスを利用するためには、音源ファイルを読み込むときに Minim クラスの loadSample メソッドを利用します。また、再生には trigger メソッドを使用します。この trigger メソッドは、必ず先頭から再生が始まります。また、ストリーミング再生ではないので、再生開始に遅れが発生することもありません。サンプル 11-6 は AudioSample を使ったものです。

サンプルプログラム 11-6

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioSample player;

void setup(){
  size(100,100);
  minim = new Minim(this);
// 読み込むファイルが変わっています。
  player = minim.loadSample("score.wav");
}

oid draw(){
  // Write what you do
}
void mouseClicked(){
  player.trigger();
}
```

```
void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

表 11-3 AudiSample の再生に関連したメソッド

関数（メソッド）	機能
loadSample(“ ファイル名 ”)	音声ファイルの読み込む
trigger();	再生の開始

音声ファイル情報の取得

mp3 ファイルにはアーティスト情報などのメタデータが記録されていることがあります。Minim には、このメタデータを取り出すための仕組みが用意されています。このメタデータの取り込みを行ったものがサンプル 11-7 です。

音声ファイルのメタデータを取り出すためには、getMetaData メソッドを使用します。メタデータを取り出した音声ファイルのデータが変数 player に代入されているとすると、「player.getMetaData();」を実行します。戻り値は、AudioMetaData 型となりますので、AudioMetaData 型の変数に代入しておきます。例えば、「meta = player.getMetaData();」を実行すると、メタデータが変数 meta に代入されます。そして、「meta.fileName()」とするとファイル名が取り出せます。また、「meta.length()」とすると再生時間（ミリ秒）の情報が取り出せます。これ以外にも、表 11-4 のようなメタデータを取り出すことが出来ます。なお、メタデータの情報が日本語で保存されていると、文字化けしてしまうようです。ちょっと面倒なことをすると、直ると思うのですが。

サンプルプログラム 11-7

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;
Minim minim;
AudioPlayer player;
AudioMetaData meta;
PFont font;
void setup(){
  size(400,400);
  minim = new Minim(this);
  player = minim.loadFile("schoolsong.mp3");
  meta = player.getMetaData();// メタデータの読み込み
  font = loadFont("MS-Mincho-36.vlw");
  textFont(font,24);
}
```

```
void draw(){
  background(255);
  fill(0);
  text("File Name:" + meta.fileName(), 5, 50);
  text("Length (in milliseconds):" + meta.length(), 5, 50+60);
  text("Title:" + meta.title(), 5, 50+2*60);
  text("Author:" + meta.author(), 5, 50+3*60);
}

void mouseClicked(){
  player.play();
}

void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

表 11-4 メタデータ取得に関連したメソッド

メソッド	機能
getMetaData()	メタデータの取得
fileName()	FileName
length()	演奏時間 (単位はミリ秒)
title()	タイトル
author()	演奏者
album()	アルバム名
date()	日付
comment()	コメント
track()	Track
genre()	Genre
copyright()	コピーライト
disc()	Disc
composer()	Composer
orchestra()	Orchestra
publisher()	Publisher
encoded()	Encoded

メタデータの各項目のうまい日本語訳があれば、教えて下さい。

再生中の情報取得

AudioPlayer などで再生を行っている場合には、どの場所を再生しているかの情報を取り出すことも出来ます。次の表 11-5 のようなデータを AudioPlayer 型などの変数から取り出すことが出来ます。

表 11-5 AudioPlayer 型などから直接取り出せる情報

メソッド	機能
length()	演奏時間 (単位はミリ秒)
position()	再生経過時間（単位はミリ秒）
isPlaying()	再生中かどうかを boolean 型のデータとして返す

サンプル 11-8 は、length と position を利用したものです。再生時間に応じて、バーが伸びてきます。音声ファイルの演奏時間とウインドウの幅は同じではないので、map 関数を利用して、バーの幅を計算しています。どうも、ちゃんと動いていな気が。length が返す値が少し大きいようです。

音声ファイルによっては、ちゃんと動くのですが。何か情報を持っている人がいたら教えて下さい。

サンプルプログラム 11-8

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer player;

void setup(){
  size(400,100);
  minim = new Minim(this);
  player = minim.loadFile("schoolsong.mp3");
}

void draw(){
  background(255);
  float x = map(player.position(),0,player.length(),0,width-1);
  stroke(0);
  fill(120);
  rect(0,0,x,height);
}

void mouseClicked(){
  player.play();
}

void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

サンプル 11-9 は、isPlaying を利用して、再生中にマウスをクリックすると再生が中断し（pause）、再度マウスをクリックすると演奏が開始されるものです。なお、再生はプログラムの実行時から loop で行っています。

サンプルプログラム 11-9

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;
```

```
Minim minim;
AudioPlayer player;

void setup(){
  size(400,100);
  minim = new Minim(this);
  player = minim.loadFile("schoolsong.mp3");
  player.loop();
}

void draw(){

}

void mouseClicked(){
  if(player.isPlaying()){
    player.pause();     // 再生中なら pause を実行
  }else{
    player.play();      // 再生中でなければ、play を実行
  }
}

void stop(){
  player.close(); // AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

正弦波などを鳴らす

音は空気の振動です。音の高低は波の周波数で決まります。Processing では、音楽ファイルを再生するだけでなく、周波数を指定して、音を発生させることが出来ます。ある一定の周波数の音を正弦波と呼びます。これを行っているのが、サンプル 11-10 です。正弦波で音の鳴らすためには、どのような波形かという情報とそれをどこに音を出すかの 2 つの情報が必要となります。

正弦波の発生 サンプル 11-10

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioOutput out;
SineWave sine;
```

```
void setup(){
  minim = new Minim(this);
  out = minim.getLineOut(Minim.STEREO);
  sine = new SineWave(440, 0.5, out.sampleRate());
  out.addSignal(sine);
}
void draw(){
}
void stop(){
  out.close(); // ライン出力の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

前者の情報を与えるために、SineWave メソッドを利用します。このメソッドは任意の周波数の正弦波を生成することができます。後者の情報を与えるために、getLineOut メソッドを利用しています。getLineOut メソッドは AudioOutput 型の値を返します。このサンプルでは、ステレオで音の出力を行うので、getLineOut の引数に、Minim.STEREO を渡しています。もし、モノラルでの出力を行う場合には、Minim.MONO とします。これに、addSignal メソッドで発生される波形を設定することで、正弦波の音波を出すことができます。周波数で考えると、周波数を倍にすると 1 オクターブ上，半分にすると 1 オクターブ下になります。

通常の音声や楽器の音などは、1 つの周波数の音だけでなく、複数の正弦波が混ざって出来ています。フーリエ級数という数学の理論を利用すると、様々な波形は複数の正弦波を足しあわせたものとして表現することが出来ます。Processing でも複数の正弦波を足しあわせた音を鳴らすことが出来ます。これを行ったものが、サンプル 11-11 です。

正弦波の組み合わせ　サンプル 11-11

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioOutput out;
SineWave sine1,sine2,sine3;
```

```
void setup(){
  minim = new Minim(this);
  out = minim.getLineOut(Minim.STEREO);
  sine1 = new SineWave(440, 0.5, out.sampleRate());
  sine2 = new SineWave(880, 0.2, out.sampleRate());
  sine3 = new SineWave(1760, 0.1, out.sampleRate());
  out.addSignal(sine1);
  out.addSignal(sine2);
  out.addSignal(sine3);
}
void draw(){
}
void stop(){
  out.close(); // ライン出力の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

Minim では、正弦波だけなく、矩形波やノコギリ波の発生も行うことが出来ます。矩形波やノコギリ波で音を鳴らすためには、サンプル 11-10 の正弦波を生成している部分を、矩形波やノコギリ波を発生させるものに変更すれば、大丈夫です。表 11-6 に各種の波形の発生方法をのせておきます。サンプリングレートは、どれ位の間隔で発生させるデータの値を計算するか、どの位の間隔でデータを取り込むかを表す値です。通常は、現在のサンプリングレートを使えば大丈夫です。サンプル 11-10 や 11-11 では、波形情報を出力する先である、AudioOutput 型の out が持っているメソッド sampleRate() を使って、現在のサンプルレートを取得しています。

表 11-6 波形発生のメソッド

波形の種類	メソッド名
正弦波	SineWave(周波数 , 振幅 , サンプリングレート) 振幅は 0 ～ 1 の数値
矩形波	SquareWave(周波数 , 振幅 , サンプリングレート)
ノコギリ波	SawWave(周波数 , 振幅 , サンプリングレート)

そこで、矩形波を発生させるようにしたものが、サンプル 11-12 です。

矩形波の発生 サンプル 11-12

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioOutput out;
SquareWave squ;

void setup(){
  minim = new Minim(this);
  out = minim.getLineOut(Minim.STEREO);
  squ = new SquareWave(440, 0.5, out.sampleRate());
  out.addSignal(squ);
}

void draw(){
}
void stop(){
  out.close(); // ライン出力の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

サンプル 11-13 はノコギリ波を発生させるものです。

ノコギリの発生 サンプル 11-13

```
import ddf.minim.*;
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioOutput out;
SawWave saw;

void setup(){
  minim = new Minim(this);
  out = minim.getLineOut(Minim.STEREO);
  saw = new SawWave(440, 0.5, out.sampleRate());
  out.addSignal(saw);
}
void draw(){
}
void stop(){
  out.close(); // ライン出力の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

また、SineWave 型、SquareWave 型、SawWave 型は、次のようなメソッドを持っています。

表 11-7 波形発生に関するメソッド

メソッド名	昨日
setFreq(周波数);	派生させる音の周波数を変更する
setAmp(振幅)	派生させる音の振幅を変更する (0 ～ 1)
setPan(パン位置)	-1（左チャンネルのみ）～ 0（中央）～ 1（右チャンネルのみ）の範囲の数値で、パン位置を設定する

表 11-7 に載っている setFreq と setAmp を使用したサンプルを示します。サンプル 11-14 では、マウスの X 座標の値を使ってパンの値を決めています。つまり、マウスの X 座標の値が 0 なら（一番左なら）パンの位置を -1 に、マウスの X 座標の値が width-1 なら（一番右なら）パンの位置を 1 に設定しています。また、マウスの Y 座標の値によって周波数を変更しています。マウスが一番上なら（0 なら）周波数を 400Hz、マウスが一番下なら（height-1 なら）1600Hz に設定しています。途中の値は、map 関数を使って計算し、その値を setPan メソッドや setFreq メソッドに渡しています。これらの設定は、マウスが動いたときに行えば良いので、これらの処理は mouseMoved 関数の中に書かれています。

表 11-7 のメソッドを利用 サンプル 11-14

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioOutput out;
SineWave sine;

void setup(){
  size(600,200);
  minim = new Minim(this);
  out = minim.getLineOut(Minim.STEREO);
  sine = new SineWave(440, 0.5, out.sampleRate());
  out.addSignal(sine);
}
void draw(){
  background(255);
  stroke(0);
  fill(200);
  ellipse(mouseX,mouseY,40,40);
}
```

```
void mouseMoved(){
// 周波数を計算する
  float freq = map(mouseY,0,height-1,400,1600);
// パンの値を計算する
  float pan = map(mouseX,0,width-1,-1,1);
  sine.setFreq(freq);// 周波数を変更する
  sine.setPan(pan);  // パン位置を変更する
}
void stop(){
  out.close(); // ライン出力の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

複数のファイルを扱う

ここでは、複数のファイルを扱うサンプルを紹介します。このサンプルでは、AudioSample を利用して、音声ファイルの再生を行います。1 ～ 4 までの数字キーを押すと対応するファイルの再生が行われます。このサンプルでは、AudioSample 型の配列に loadSample メソッドの実行結果を保存し、keyPressed 関数の中で Processing 変数の key の値を調べ、対応する音声ファイルの再生を行っています。変数 key には押されたキーの情報が入っているので、キー 1 が押されたかは、key=='1' で調べることができます。

複数音声ファイルの扱い例 サンプル 11-15

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioSample[] se; // 音声ファイルの情報をしまう配列
void setup(){
  size(100,100);
  minim = new Minim(this);
  se = new AudioSample[4];// 音声ファイルの情報をしまう配列の確保
// 音声ファイルの読み込み
  se[0] = minim.loadSample("appear01.wav");
  se[1] = minim.loadSample("appear02.wav");
  se[2] = minim.loadSample("appear03.wav");
  se[3] = minim.loadSample("appear04.wav");
}

void draw(){
// 何も書いてなくても、これがないと音が鳴りません。
}
```

```
void keyPressed(){
  if(key == '1'){
    se[0].trigger();
  }else if(key == '2'){
    se[1].trigger();
  }else if(key == '3'){
    se[2].trigger();
  }else if(key == '4'){
    se[3].trigger();
  }
}
void stop(){
// すべての AudioPlayer の機能を終了する必要があります。
  for(int i=0;i<se.length;i++){
    se[i].close();// AudioPlayer の機能を終了する
  }
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

サンプル 11-15 にちょっとした機能を付け加えると、少しゲームのようなプログラムを作ることが出来ます。サンプル 11-16 では、millis 関数を利用して時間を計り、1 秒 (1000 ミリ秒) 経つと、押すべきキーの表示が変わります。押すべきキーの決定には、random 関数を利用しています。

複数音声ファイルの扱い例その 2 サンプル 11-16

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;
Minim minim;
AudioSample[] se;
int startTime; //経過時間を計るための変数
int idx; // どのキーを押すべきかを決める変数
PFont font;
void setup() {
  size(300, 100);
  font = createFont("Serif", 48);
  textFont(font, 36);
  textAlign(CENTER);
  minim = new Minim(this);
  se = new AudioSample[4];
  se[0] = minim.loadSample("appear01.wav");
  se[1] = minim.loadSample("appear02.wav");
  se[2] = minim.loadSample("appear03.wav");
  se[3] = minim.loadSample("appear04.wav");
  update();
}
```

```
// 一定時間経過したので、情報を更新する
void update() {
  startTime = millis();
  idx = int(random(4))+1;
}
void draw() {
  background(255);
  fill(0);
  text("Hit "+idx+" key", width/2, height/2);
  if (millis()-startTime >= 1000) {// 1 秒経過したので情報を更新
    update();
  }
}
void keyPressed() {
  if (key == '1') {
    if (idx == 1) { // 押されたキーが指定されたキーかを調べる
      se[0].trigger();
    }
  }else if (key == '2') {
    if (idx == 2) { // 押されたキーが指定されたキーかを調べる
      se[1].trigger();
    }
  }else if (key == '3') {
    if (idx == 3) { // 押されたキーが指定されたキーかを調べる
      se[2].trigger();
    }
  }else if (key == '4') {
    if (idx == 4) { // 押されたキーが指定されたキーかを調べる
      se[3].trigger();
    }
  }
}
void stop(){
// すべての AudioPlayer の機能を終了する必要があります。
  for(int i=0;i<se.length;i++){
    se[i].close();// AudioPlayer の機能を終了する
  }
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

波形の描画

Minim は音声ファイルの再生だけはなく、色々なことができます。そのうちの一つが入力や出力される音声データの取り込みです。これを利用すると波形の描画をすることが出来ます。

読み込まれた音声データは、バッファ (buffer) と呼ばれる場所に少しずつコピーをされながら、再生されていきます。このバッファの中に保存されている値を取り出すのが get メソッドです。ステレオの場合には、左右があるので、サンプル 11-17 の赤色の行のように、左

右を指定して取り出します。取り出される情報は -1 ～ 1 までの数値データです。このバッファの大きさ(いくつのデータが入っているか)を取り出すのが bufferSize メソッドです。これを利用して波形データを描いたものが、サンプル 11-17 です。

波形データの表示その 1 サンプル 11-17

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer song;

void setup(){
  size(400,200);
  smooth();
  minim = new Minim(this);
  song = minim.loadFile("schoolsong.mp3");
  song.loop();
}
void draw(){
  background(255);
  stroke(0);
  beginShape();
  for(int i = 0; i < song.bufferSize() ; i++){
    float x=map(i,0,song.bufferSize(),0,width-1);
    vertex(x, 60 + song.left.get(i)*50);
  }
  endShape();
  beginShape();
  for(int i = 0; i < song.bufferSize() ; i++){
    float x=map(i,0,song.bufferSize(),0,width-1);
    vertex(x, 170 + song.right.get(i)*50);
  }
  endShape();
}
void stop(){
  song.close();// AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

Minim では、音声ファイルからだけはなく、パソコンに付いているマイクからの音声情報を取り出すことが出来ます。基本的にはサンプル 11-17 と同じですが、マイクからの入力になるので、AudioPlayer の代わりに AudioInput 型の変数に minim.getLineIn(Minim.STEREO) の戻り値を代入します。この変数から音声情報を取り出すことが出来ます。これを利用したものがサンプル 11-18 です。サンプル 11-17

と異なっているのは、赤字の部分です。

波形データの表示その 2 サンプル 11-18

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;
Minim minim;
AudioInput in;
int bufferSize = 1024;
float [] buffer = new float[bufferSize];
void setup(){
  size(400,230);
  smooth();
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO);
}
void draw(){
  background(255);
  stroke(0);
  beginShape();
  for(int i = 0; i < in.bufferSize() ; i++){
    float x=map(i,0,in.bufferSize(),0,width-1);
    vertex(x, 60 + in.left.get(i)*50);
  }
  endShape();
  beginShape();
  for(int i = 0; i < in.bufferSize() ; i++){
    float x=map(i,0,in.bufferSize(),0,width-1);
    vertex(x, 170 + in.right.get(i)*50);
  }
  endShape();
}
void stop(){
 in.close();// AudioPlayer の機能を終了する
 minim.stop(); // Minim の機能を停止する
 super.stop(); // 停止の際のおまじない
}
```

FFT

音声情報などを分析する際に FFT と呼ばれる方法を利用することがあります。Minim では、この FFT の機能を持っています。FFT の説明をするには、下手をすると一学期かかってしまいます。このあたりの話は、サウンド解析やサウンド情報処理で扱われます。その授業を受ける際にでも、Processing で FFT が出来たことを思い出して下さい。FFT の機能を使ったものがサンプル 11-19 です。興味のある人は、マニュアルなどを頼りに、動作を調べて見て下さい。

FFT サンプル 11-19

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.analysis.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer song;
FFT fft;

void setup(){
  size(512, 200);
// always start Minim first!
  minim = new Minim(this);
  song = minim.loadFile("schoolsong.mp3", 512);
  song.loop();
  fft = new FFT(song.bufferSize(), song.sampleRate());
}
void draw(){
  background(0);

  fft.forward(song.mix);
  stroke(255, 0, 0, 128);
  for(int i = 0; i < fft.specSize(); i++){
    line(i, height, i, height - fft.getBand(i)*4);
  }
  stroke(255);
  for(int i = 0; i < song.left.size() - 1; i++){
    line(i, 50 + song.left.get(i)*50, i+1, 50 + song.left.
get(i+1)*50);
    line(i, 150 + song.right.get(i)*50, i+1, 150 + song.right.
get(i+1)*50);
  }
}

void stop(){
  song.close();// AudioPlayer の機能を終了する
  minim.stop(); // Minim の機能を停止する
  super.stop(); // 停止の際のおまじない
}
```

情報メディア基盤ユニット用資料（2016 年度第 12 章）

Processing 言語による情報メディア入門

オブジェクト指向入門

神奈川工科大学情報メディア学科　佐藤尚

はじめに

最近のプログラミング言語では、オブジェクト指向 (object oriented) と呼ばれる機能を持っているものが多くあります。オブジェクト指向は、次の 2 つの仕組みを提供しようとするものです。

1) 複数のデータをまとめて一つに扱う仕組み。
2) 機能拡張を容易に行えるようにする仕組み。

そこで、円が上から下に移動するようなプログラムを考えてみます。これはサンプル 12-1 のようになります。どのように、操作するプログラムかはわかりますね。

サンプルプログラム 12-1

```
float xBall;// 円の中心の X 座標
float yBall; // 円の中心の Y 座標
float rBall; // 円の半径
color cBall; // 円の色
void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
// 円の初期状態の決定
  rBall = random(10,20);
  xBall = random(rBall,width-rBall);
  yBall = -rBall;
  cBall = color(random(360),random(50,100),random(50,100));
}
void draw(){
  background(0,0,99);
// 円を移動させる
  yBall += 1;
  if(yBall-rBall > height){
    yBall = -rBall;
  }
// 円を描く
  stroke(cBall);
  fill(cBall);
  ellipse(xBall,yBall,2*rBall,2*rBall);
}
```

今度は 1 つの円ではなく、沢山の円を表示するようなサンプルを考えてみます。円の中心座標の値が入っている xBall や yBall などの値を配列変数にすることで、沢山の円に対する処理を簡単に記述できるようになります。サンプル 12-2 もどのような動作をしているかわかりますね。

サンプルプログラム 12-2

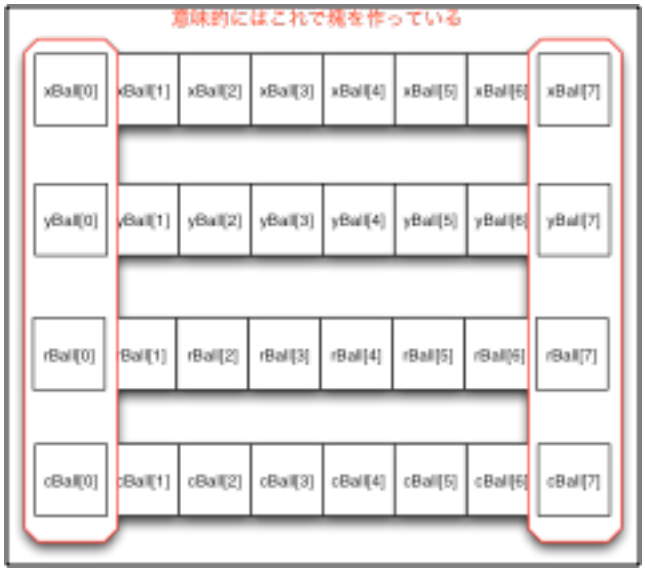
```
int numberOfBalls = 100;// 円の個数
float[] xBall;// 円の中心の X 座標
float[] yBall; // 円の中心の Y 座標
float[] rBall; // 円の半径
color[] cBall; // 円の色
void setup(){
  size(400,400);
  smooth();
  smooth();
  colorMode(HSB,359,99,99);
// 配列の確保
  xBall = new float[numberOfBalls];
  yBall = new float[numberOfBalls];
  rBall = new float[numberOfBalls];
  cBall = new color[numberOfBalls];
// 円の初期状態の決定
  for(int i=0;i<numberOfBalls;i++){
    rBall[i] = random(10,20);
    xBall[i] = random(rBall[i],width-rBall[i]);
    yBall[i] = -rBall[i];
    cBall[i] = color(random(360),random(50,100),random(50,100));
  }
}
void draw(){
  background(0,0,99);
  for(int i=0;i<numberOfBalls;i++){
// 円を移動させる
    yBall[i] += 1;
    if(yBall[i]-rBall[i] > height){
      yBall[i] = -rBall[i];
    }
// 円を描く
    stroke(cBall[i]);
    fill(cBall[i]);
    ellipse(xBall[i],yBall[i],2*rBall[i],2*rBall[i]);
  }
}
```

サンプル 12-1 よりは少し複雑になっているように見えますが、沢山の円を処理するために、for 命令による繰り返し処理が付け加わっているだけです。そのために、xBall が xBall[i] などと置き換わっていますが。

サンプル 12-2 では、配列変数 xBall, yBall, rBall, cBall 毎に塊を作っています。しかし、プログラムでの意味的には、xBall[0], yBall[0], rBall[0], cBall[0] は、1 つの円の中心座標、半径、色を保存しています。添え字の番号の値が同じものどうしで組を作って、プログラム内での意味を表しています。このサンプルのように、複数の変数が集まって、一つの意味のある情報を表すことがあります。Processing では、このような複数の情報をまとめて、新たなデータ型を作る仕組みが用意されています。それがクラスと呼ばれる仕組みです。最近のプログラミング言語は、この仕組みを持っていることが一般的になっています。

データをまとめる仕組みとしてのクラス

簡単のために、サンプル 12-1 をクラスの仕組みを用いて書き直してみます。サンプル 12-1 でも、xBall, yBall, rBall, cBall が一塊となって、意味のある情報を表しています。どの情報かを区別するために、名前をつける必要があります。クラスを構成する個々の情報（データ）のことをメンバ (member) やメンバ変数と呼び、その名前のことをメンバ名と呼んでいます。そこで、xBall は円の中心の X 座標の値なので xCenter という名前で表すことにします。同様に、yBall は円の中心の Y 座標の値なので yCenter という名前で表すことにします。また、rBall は円の半径なので radius、cBall は円の色なので col とすることにします。また、この 4 つの情報を一塊にしたものを Ball と名付けることにします。この Ball はクラス名と呼ばれます。一般に、クラス名は大文字から始まる名前にし



ます。また、メンバ変数にはどのようなデータを記録するのかを指定するために、データ型を指定する必要があります。今回は、xCenter, yCenter, radius は float 型、color は color 型とします。
つまり、PImage や PFont などはクラスという仕組みで作られたデータ型でした。

クラスの宣言その 1

クラスの宣言の一般形	Ball クラスの宣言

class クラス名 {
　メンバ変数型 0 メンバ名 0;
　メンバ変数型 1 メンバ名 1;
　メンバ変数型 2 メンバ名 02
　　:
　　:
　　:
}
	class Ball {
	　float xCenter;
	　float yCenter;
	　float radius;
	　color col;
	}

この処理を行わないと、再度の実行の際に音声ファイルの再生などが正常に行われないなどの不都合が起きる場合があります。

　このように定義したクラスは通常のデータ型と同じように利用することが出来ます。サンプル 12-1 をこの Ball クラスを使って書きかえたものがサンプル 12-3 です。

サンプルプログラム 12-3

```
class Ball {
  float xCenter; // 円の中心の X 座標
  float yCenter; // 円の中心の Y 座標
  float radius; // 円の半径
  color col;  // 円の色
}
Ball myBall; // Ball 型変数の宣言

void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
// 円の初期状態の決定
  myBall = new Ball();
  myBall.radius = random(10,20);
  myBall.xCenter = random(myBall.radius,width- myBall.radius);
  myBall.yCenter = -myBall.radius;
  myBall.col = color(random(360),random(50,100),random(50,100));
}

void draw(){
  background(0,0,99);
// 円を移動させる
  myBall.yCenter += 1;
  if(myBall.yCenter - myBall.radius > height){
    myBall.yCenter = -myBall.radius;
  }
```

```
// 円を描く
  stroke(myBall.col);
  fill(myBall.col);
  ellipse(myBall.xCenter,myBall.yCenter,2*myBall.radius,2*myBall.
radius);
}
```

Ball クラスの変数を宣言するためには、通常の変数の宣言と同じように「Ball myBall;」などとします。また、実際にデータを保存する場所を作る必要があります。これを行っているのが、「myBall = new Ball();」の部分です。クラスはどのような種類のデータの集まりかを決める鋳型（テンプレート）のようなものです。この鋳型から new 関数を使って、実際にデータを保存する場所を作りだします。この作り出された場所のことをインスタンス (instance) と呼んでいます。鯛焼き器がクラスで、鯛焼きがインスタンス、鯛焼き器を使って鯛焼きを作る作業が new といった感じでしょうか？

姉ヶ崎寧々さんというキャラクタはクラスのようなもので、姉ヶ崎寧々さんは俺の嫁と思っている人の 3 DS にはインスタンスとしての “ 姉ヶ崎寧々さん ” が存在しています。

インスタンスのメンバ変数にアクセスするためには、”.” を使います。例えば、m yBall の xCenter にアクセスするためには、myBall.xCenter などします。その他のメンバの値に対しても、同じようにアクセス出来ます。

サンプル 12-3 は、クラスを使ったプログラム例としては少し不自然なものです。実は、メンバには単なる変数だけなく、関数を持ってくることも出来ます。クラスに付随している関数のことは、メソッド (method) と呼びます。メソッドの定義は、通常の関数の定義と同じです。一つ異なっている点は、class クラス名 { 〜 } の中に書くことになっている点です。また、メンバ変数の初期化などはコンストラクタ (constructor) と呼ばれる特殊なメソッド（戻り値無し、名前はクラス名と同じ）を利用します。

クラスの宣言その 2

クラスの宣言の一般形	Ball クラスの宣言

class クラス名 {
　メンバ変数型 0 メンバ名 0;
　メンバ変数型 1 メンバ名 1;
　メンバ変数型 2 メンバ名 02
　　:
　　:
　クラス名 (){
〜
　}
}
	class Ball {
	　float xCenter;
	　float yCenter;
	　float radius;
	　color col;
	　Ball(){
	〜
	　}
	}

コンストラクタを使ってサンプル 12-3 を書きかえたものがサンプ

ル 12-4 です。この例では、class Ball { 〜 } 内の Ball(){ 〜 } の部分がコンストラクタです。コンストラクタやそのメソッドが付随している class クラス名 { 〜 } の部分でメソッドの定義を書く場合には、直接メンバ名を書けば、メンバ変数にアクセスすることが出来ます。

サンプルプログラム 12-4

```
class Ball {
  float xCenter; // 円の中心の X 座標
  float yCenter; // 円の中心の Y 座標
  float radius; // 円の半径
  color col;  // 円の色
  Ball(){
    radius = random(10,20);
    xCenter = random(radius,width- radius);
    yCenter = -radius;
    col = color(random(360),random(50,100),random(50,100));
  }
}

Ball myBall; // Ball 型変数の宣言
```

```
void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
// 円の初期状態の決定
  myBall = new Ball();
}

void draw(){
  background(0,0,99);
// 円を移動させる ,update
  myBall.yCenter += 1;
  if(myBall.yCenter - myBall.radius > height){
    myBall.yCenter = -myBall.radius;
  }
// 円を描く ,draw
  stroke(myBall.col);
  fill(myBall.col);
  ellipse(myBall.xCenter,myBall.yCenter,2*myBall.radius,2*myBall.
radius);
}
```

サンプル 12-4 の「円を移動させる」や「円を描く」などの部分は、一つのインスタンスだけの情報を利用して作られています。このような場合には、クラスのメソッドとして書くことが一般的です。そこで、このような方針でサンプル 12-4 を書きかえたものがサンプル 12-5 です。クラスに付随するメソッドを呼び出す場合にも、メンバ変数と同じように "." を使って使用します。

クラスの宣言その 3

クラスの宣言の一般形	Ball クラスの宣言

class クラス名 {
　メンバ変数型 0 メンバ名 0;
　メンバ変数型 1 メンバ名 1;
　メンバ変数型 2 メンバ名 02
　　:
　　:
　　:
　クラス名 (){
　　〜
　}
　戻り値型 0　メソッド名 0(仮引数の並び){
　　〜
　}
　戻り値型 1　メソッド名 1(仮引数の並び){
　　〜
　}
　　:
}
	class Ball {
	　float xCenter;
	　float yCenter;
	　float radius;
	　color col;
	　Ball(){
	　　〜
	　}
	　void update(){
	　　〜
	　}
	　void draw(){
	　　〜
	　}
	}

サンプルプログラム 12-5

```
class Ball {
  float xCenter; // 円の中心の X 座標
  float yCenter; // 円の中心の Y 座標
  float radius; // 円の半径
  color col;  // 円の色
// コンストラクタの定義
  Ball(){
    radius = random(10,20);
    xCenter = random(radius,width- radius);
    yCenter = -radius;
    col = color(random(360),random(50,100),random(50,100));
  }
// メソッドの定義
  void update(){
    yCenter += 1;
    if(yCenter - radius > height){
      yCenter = -radius;
    }
  }
  void draw(){
    stroke(col);
    fill(col);
    ellipse(xCenter,yCenter,2*radius,2*radius);
  }
}
```

```
Ball myBall; // Ball 型変数の宣言
void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
// 円の初期状態の決定
  myBall = new Ball();
}
void draw(){
  background(0,0,99);
// 円を移動させる ,update
  myBall.update(); // myBall の update メソッドの呼び出し
// 円を描く ,draw
  myBall.draw();  // myBall の draw メソッドの呼び出し
}
```

今までのクラスを使ったサンプルでは、一つのタブの中に全てのプログラムを書いていました。しかし、通常はクラス毎に別々のタブに記述します。新たにタブを作るためには、次の様に行います。

1. ウインドウの右上にある矢印状のボタンを押します。
2. すると、メニューが出てきますので、「New Tab」を選択します。
3. そして、新たに作るタブの名前を入力し、OK ボタンを押します。
4. 新しいタブが作られます。
5. タブの名前は、クラスの名前と同じにするのが一般的です。

Ball クラスを使って、サンプル 12-2 を書きかえてみます。この結果がサンプル 12-6 です。

サンプルプログラム 12-6

Ball クラスのタブ

```
class Ball {
  float xCenter; // 円の中心の X 座標
  float yCenter; // 円の中心の Y 座標
  float radius; // 円の半径
  color col;  // 円の色
// コンストラクタの定義
  Ball(){
    radius = random(10,20);
    xCenter = random(radius,width- radius);
    yCenter = -radius;
    col = color(random(360),random(50,100),random(50,100));
  }
// メソッドの定義
  void update(){
    yCenter += 1;
    if(yCenter - radius > height){
      yCenter = -radius;
    }
  }
```

```
  void draw(){
    stroke(col);
    fill(col);
    ellipse(xCenter,yCenter,2*radius,2*radius);
  }
}
```

メインのタブ

```
int numberOfBalls=100;
Ball[] myBalls; // Ball 型変数の宣言
void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
// 配列の確保
  myBalls = new Ball[numberOfBalls];
  for(int i=0;i<numberOfBalls;i++){
    myBalls[i] = new Ball();
  }
}
void draw(){
  background(0,0,99);
  for(int i=0;i<numberOfBalls;i++){
    myBalls[i].update(); // myBalls[i] の update メソッドの呼び出し
    myBalls[i].draw();  // myBall[i] の draw メソッドの呼び出し
  }
}
```

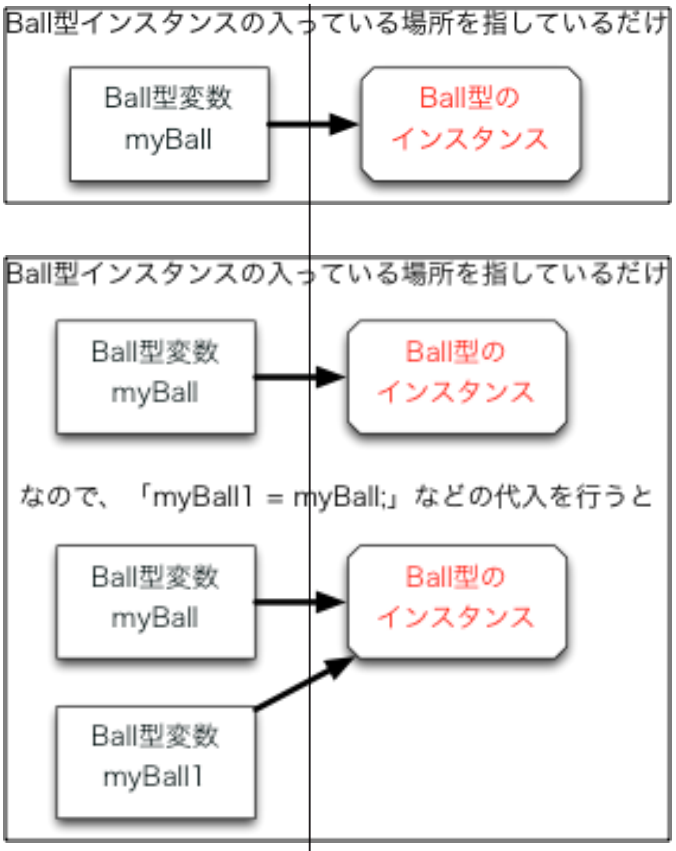
このようにクラスを利用してプログラムを作成すると、プログラムの見通しが良くなります。また、コンストラクタにも引数を渡すことが出来ます。

クラスの宣言その 4

クラスの宣言の一般形	Ball クラスの宣言

class クラス名 {
メンバ変数型 0 メンバ名 0;
メンバ変数型 1 メンバ名 1;
メンバ変数型 2 メンバ名 02
:
:
クラス名 (){
〜
}
戻り値型 0　メソッド名 0(仮引数の並び){
〜
}
戻り値型 1　メソッド名 1(仮引数の並び){
〜
}
:
}
	class Ball {
	float xCenter;
	float yCenter;
	float radius;
	color col;
	Ball(){
	〜
	}
	void update(){
	〜
	}
	void draw(){
	〜
	}
	}

今まで、説明をサボってきましたが、クラス型の変数は、そのクラスのインスタンスへの参照となっています。普通は気にしなくても大丈夫ですが、時々問題が起きることがあります。つまり、Ball 型変数同士の代入を行っても、変数が指しているインスタンスの情報そのものが複製される訳ではありません。インスタンスの情報もコピーするような代入を浅いコピー (shallow copy) と呼んでいます。メンバ変数の型が何らかのクラス型になっている場合には、単に参照がコピーされるだけです。Processing でも、浅いコピーを実現するための clone メソッドが用意されています。逆に、完全なコピーを作るような代入を深いコピー (deep copy) と呼ばれています。深いコピーを実現するためには、複製を作るために時間がかかるので、どうしても深いコピーを使いたいときだけ利用します。ちょっと難しい話なので、詳しくは触れません。



サンプル 12-6 をまねして、円の代わりに正方形が落ちてくるようなプログラムを作成してみます。サンプル 12-7 を見るとわかるように、クラスを使ってプログラムを作成しておくと、どの部分を変更すれば良いかが見やすくなっていることがわかると思います。

サンプルプログラム 12-7

Square クラスのタブ

```
class Square {
  float xCenter; // 中心の X 座標
  float yCenter; // 中心の Y 座標
  float length; // 一辺の長さ
  color col;  // 色
// コンストラクタの定義
  Square(){
    length = random(10,20);
    xCenter = random(length/2,width- length/2);
    yCenter = -length/2;
    col = color(random(360),random(50,100),random(50,100));
  }
// メソッドの定義
  void update(){
    yCenter += 1;
    if(yCenter - length/2 > height){
      yCenter = -length/2;
    }
  }
```

```
  void draw(){
    rectMode(CENTER);
    stroke(col);
    fill(col);
    rect(xCenter,yCenter,length,length);
  }
}
```

メインのタブ

```
int numberOfSquares=100;
Square[] mySquares; // Square 型変数の宣言
void setup(){
  size(400,400);
  smooth();
  colorMode(HSB,359,99,99);
// 配列の確保
  mySquares = new Square[numberOfSquares];
  for(int i=0;i<numberOfSquares;i++){
    mySquares[i] = new Square();
  }
}
void draw(){
  background(0,0,99);
  for(int i=0;i<numberOfSquares;i++){
    mySquares[i].update(); // mySquares[i] の update メソッドの呼び出し
    mySquares[i].draw();  // mySquare[i] の draw メソッドの呼び出し
  }
}
```

機能拡張・継承

オブジェクト指向の提供する仕組みの２つめは、機能拡張に関わるものです。これは、ポケモンを進化させるように、元になるクラスを進化させ、機能を拡張しようとするものです。この進化の際には、もとのクラスの特徴（メンバ変数やメソッド）は保持され、新しい特徴が付け加わったり、元の特徴が進化したりします。オブジェクト指向の言葉では、元になるクラスを親クラス（スーパークラス）、進化して出来た新しいクラスを子クラスと呼んでいます。進化させることを継承する (inherit) と呼びます。

サンプル 12-8 では、JitteringObject というクラスを定義しています。このクラスは赤色の点を表示するものです。update メソッドは、乱数で点の位置を変更します。

サンプルプログラム 12-8

```
class JitteringObject {
  float xCenter;
  float yCenter;
  JitteringObject(float x0,float y0){
    xCenter = x0;
    yCenter = y0;
  }
  void update(){
    xCenter = constrain(xCenter+random(-1,1),0,width);
    yCenter = constrain(yCenter+random(-1,1),0,height);
  }
  void draw(){
    stroke(255,10,10);
    point(xCenter,yCenter);
  }
}
JitteringObject myPoint;
void setup(){
  size(400,400);
  smooth();

  myPoint = new JitteringObject(width/2,height/2);
}
void draw(){
  background(255);
  myPoint.update();
  myPoint.draw();
}
```

このサンプルプログラムを変更して、点の代わりに正方形を表示するようにするための、一つの方法として、サンプル 12-9 のように行う方法があります。

サンプルプログラム 12-9

```
class JitteringRect {
  float xCenter;
  float yCenter;
  float len;
  JitteringRect(float x0,float y0,float l0){
    xCenter = x0;
    yCenter = y0;
    len = l0;
  }
  void update(){
    xCenter = constrain(xCenter+random(-1,1),0,width);
    yCenter = constrain(yCenter+random(-1,1),0,height);
  }
≈
```

```
  void draw(){
    stroke(255,10,10);
    fill(255,10,10);
    rectMode(CENTER);
    rect(xCenter,yCenter,len,len);
  }
}

JitteringRect myRect;
void setup(){
  size(400,400);
  smooth();

  myRect = new JitteringRect(width/2,height/2,10);
}
void draw(){
  background(255);
  myRect.update();
  myRect.draw();
}
```

サンプル 12-8 とサンプル 12-9 を見比べると共通する部分が多いことに気がつきます。例えば、サンプル 12-9 では、サンプル 12-8 のコンストラクタ（JitteringObject() と JitteringRect()）の部分と draw メソッドが変わっています。このようなときに利用するのが継承という仕組みです。この継承を使ったものがサンプル 12-10 です。

サンプルプログラム 12-10

```
class JitteringObject {
  float xCenter;
  float yCenter;
  JitteringObject(float x0,float y0){
    xCenter = x0;
    yCenter = y0;
  }
  void update(){
    xCenter = constrain(xCenter+random(-1,1),0,width);
    yCenter = constrain(yCenter+random(-1,1),0,height);
  }
  void draw(){
    stroke(255,10,10);
    point(xCenter,yCenter);
  }
}
```

```
class JitteringRect extends JitteringObject{
  float len;
  JitteringRect(float x0,float y0,float l0){
    super(x0,y0);
    len = l0;
  }
  void draw(){
    stroke(255,10,10);
    fill(255,10,10);
    rectMode(CENTER);
    rect(xCenter,yCenter,len,len);
  }
}
JitteringRect myRect;
void setup(){
  size(400,400);
  smooth();
  myRect = new JitteringRect(width/2,height/2,10);
}
void draw(){
  background(255);
  myRect.update();
  myRect.draw();
}
```

「class JitteringRect extends JitteringObject」の部分で、どのクラスから進化をさせて、新しいクラスを作るかを指定します。この場合では、JitteringObject クラスから進化をさせて（拡張さえて、extends）、新しいクラス JitteringRect を作っています。元のクラスで定義されているメンバ変数やメソッドはそのまま利用することが出来ます。また、draw のように、子クラスでメソッドを書き換えることも出来ます。また、super と親クラスのコンストラクタを呼び出すことが出来ます。

なぜオブジェクト指向の話をしたのか

オブジェクト指向の話をきちんと理解するためには、もう少し色々な説明をする必要があります。また、自分でクラスを作る際には、色々と知っていると良いことが沢山あります。これらの話題は、2 年生のプログラミング関連の授業で詳しく説明されます。ですので、わざわざオブジェクト指向の話をする必要はなかったかもしれません。しかし、簡単でもオブジェクト指向の話をしたのには、訳があります。

前回にやった音を発生させる授業では、Minim クラスを使っていました。Processing を利用したプログラミングでは、全てのプログラムを自分で作成することもありますが、他の人が作ったプログラムを利用しながら、プログラムを作成することがあります。この他人が

作ったプログラムを利用する際に必要となるのが、オブジェクト指向の言葉です。そこで簡単ですが、オブジェクト指向の話をしました。きちんと説明しませんが、サンプル 12-11 は円柱を表示するプログラムです。

サンプルプログラム 12-11

```
// size( 幅 , 高さ ,OPENGL) の際のおまじない
import processing.opengl.*;

void setup(){
// size( 幅 , 高さ ,OPENGL) や size( 幅 , 高さ ,P3D) とすると、
// 3 次元での表示が可能となる
  size(400,400,OPENGL);
  smooth();
}

void draw() {
  background(0);
  directionalLight(255,255,255,1,0,-1);
  noStroke();
  translate(width/2, height/2, 0);
  beginShape(QUAD_STRIP);
  for (int a = 0; a <= 360; a += 20) {
    float x = 100 * cos(radians(a));
    float z = 100 * sin(radians(a));
    normal(x, 0, z);
    vertex(x, -100, z);
    normal(x, 0, z);
    vertex(x, 100, z);
  }
  endShape();
}
```

これは、単純に円柱を表示するだけのプログラムです。マウスを動かしても円柱は動きません。マウスの動きに合わせて円柱を動かすためには、自分でプログラムを作成するという方法もありますが、他の人の使ったプログラムを利用するという方法もあります。ここでは、PeasyCam というものを利用してみます。このライブラリの web ページは http://mrfeinberg.com/peasycam/ です。このライブラリを利用すると簡単にカメラの移動を行うことができるようになります。この変更を加えてものがサンプル 12-12 です。

サンプルプログラム 12-12

```
import processing.opengl.*;
import peasy.test.*;
import peasy.org.apache.commons.math.*;
import peasy.*;
import peasy.org.apache.commons.math.geometry.*;

PeasyCam cam;
```

```
void setup(){
  size(400,400,OPENGL);
  smooth();
// Peasy の機能を使うと、マウスの動きによって物体の回転ができる
  cam = new PeasyCam(this, width/2, height/2, 0,400);
// Peasy でカメラが表示する範囲を指定
  cam.setMinimumDistance(50);
  cam.setMaximumDistance(500);
}

void draw() {
  background(0);
// 光源の設定
  directionalLight(255,255,255,1,0,-1);
  noStroke();
  translate(width/2, height/2, 0);
  beginShape(QUAD_STRIP);
  for (int a = 0; a <= 360; a += 20) {
    float x = 100 * cos(radians(a));
    float z = 100 * sin(radians(a));
    normal(x, 0, z);
    vertex(x, -100, z);
    normal(x, 0, z);
    vertex(x, 100, z);
  }
  endShape();
}
```

一般的に、外部の人の作ったライブラリのインストールは簡単です。ユーザーのスケッチフォルダの中に libraries というフォルダを作り、その中にコピーするだけです。スケッチブックフォルダの場所は、Processing エディタメニューの [File]-[Preferences] で表示される設定ダイアログの [Sketchbook location:] に指定されているフォルダです。なお、コピーした後に、Processing のプログラムを再起動して下さい。

サンプル 12-12 の場合には、http://mrfeinberg.com/peasycam/ の download から zip ファイルをダウロード（2015 年 7 月 9 日時点では、peasycam_201.zip）を解凍して得られるフォルダ peasycam をユーザーのスケッチフォルダの中に libraries にコピーするだけです。Proxy を使っていない環境であれば、Processing の機能を利用して（Sketch > Import Library... > Add Library...）でもインストールすることができます。なお、3 次元の物体の表示には、物体の位置や大きさ、光源の位置、視点の位置などさまざまな要素が絡み合うので、ちょっと複雑です。このサンプルでは、OBJ 形式と呼ばれている 3 次元形状の情報を保存するデータ形式のファイルから、表示する形状を読み込んでいます。

サンプルプログラム 12-13

```
import peasy.test.*;
import peasy.org.apache.commons.math.*;
import peasy.*;
import peasy.org.apache.commons.math.geometry.*;
import processing.opengl.*;

PShape model;
PeasyCam cam;

void setup() {
  size(400, 400, OPENGL);
// tachikoma.obj というファイルを読み込む
  model = loadShape("tachikoma.obj");
// マウスの移動で表示物体を回転させる
  cam = new PeasyCam(this, width/2, height/2, 0,400);
  cam.setMinimumDistance(50);
  cam.setMaximumDistance(500);
}

void draw() {
  background(0, 0, 100);
  lights();
  translate(width/2, height/2, 0);
// 0.5 という数字を変更すると、表示されている物体の大きさが変わる
  scale(0.5);
  shape(model,0,0);
}
```

ファイルからのデータの読み出し

Processing 言語では、ファイルからのデータの読み出しや、書き込みを行うことが出来ます。Processing 言語のバージョンが 2.2 以上の場合には、ファイルからのデータの読み出しや書き出しを行う関数が含まれています。しかし、バージョンが 1.5 の場合には、それほど多くはありません。ここでは、どちらでも使える、基本的な関数を紹介します。

ファイルからのデータの読み出しには、

1) 一気に行毎に全てのデータを読み出す、

2) 一行ずつデータを読み出す、

3) 一文字ずつデータを読み出す、

の 3 通りの方法があります。

一番簡単の方法は、1) の方法です。つぎのサンプルは ”test.txt” というファイルからデータを読み出し、それを表示するプログラムです。

サンプルプログラム 12-14

```
String[] moji;
PFont font;

void setup(){
  size(400,200);
  font = loadFont("Serif-48.vlw");
  textFont(font,36);
// text.txt からデータを読み出す
  moji = loadStrings("text.txt");
}
void draw(){
  background(255);
  fill(0);
  textAlign(CENTER,CENTER);
  text(moji[second() % moji.length],0,0,width,height);
}
```

text.txt の内容

Kongo	Fuso	Nagato
Hiei	Yamashiro	Mutsu
Haruna	Ise	Yamato
Kirishima	Hyuga	Musashi

ファイルからデータを読み出すために、loadStrings 関数を使用しています。この関数は、ファイルからデータを読み出し、そのデータを String 型の配列に保存するような関数です。このサンプルでは、text.txt というファイルからデータを読み出し、このファイルのデータを moji という String 型の配列に保存市営ます。moji[0] には、ファイルの 1 行目のデータが文字列として記録されています。同様に、moji[1] にはファイルの 2 行目、moji[2] にはファイルの 3 行目の情報が記録されています。ファイルの最後の行のデータは moji[moji.length-1] に記録されています。loadStrings 関数では、ファイルのデータを一気に読み出し、メモリ中に保存するので、余り大きなファイルを読み込むことは難しいと思います。

String[] loadStrings（ファイル名）
ファイル名で指定されたファイルからデータを読み出し、その結果を String 型の配列変数に保存する。読み出された情報は String 型となっています。また、一般的には、”Show sketch folder” で表示されるフォルダ内にある data フォルダに保存されているファイルからデータを読み出します。つまり、読み出したファイルはこの data フォルダに保存しておくことが必要となります。

loadStrings 関数を使った方法では、ファイルから読み出したデータはすべて文字列 (String) となってしまいます。数値データが入ってい

る場合には、どのようにしたら良いでしょうか？一番単純な方法は、文字列として読み込んだデータを int 型や float 型に変換する方法です。
次のサンプルでは、loadStrings 関数を使って fuel.txt というファイルからデータを読み出し、それのデータを int 関数によって、int 型の値に変換しています。

サンプルプログラム 12-15

```
String[] moji;
int[]  fuel;
int barWidth = 20;
void setup(){
  moji = loadStrings("fuel.txt");
  fuel = new int[moji.length];
  for(int i=0;i < moji.length;i++){
    fuel[i] = int(moji[i]);
  }
  size(moji.length*barWidth,max(fuel));
}
void draw(){
  background(255);
  stroke(255);
  for(int i=0;i<fuel.length;i++){
    fill(255-fuel[i]);
    rect(barWidth*i,height-fuel[i],barWidth,fuel[i]);
  }
}
```

fuel.txt の内容

80	85	100
80	85	100
80	85	250
80	85	250

このサンプルでは、メモ帳などで読み込みに使うデータを作っていました。ゲームなどを作る際には、Excel などを利用してデータファイルを作成することがあります。拡張子が .”xlsx” などのファイルを読み込むことは面倒ですが、拡張子が ”.csv” というファイルでは、割と簡単にデータを読み出すことができます。
次のような表の情報を保存した “.cvs” ファイル (“battleships.cvs”) では、1 行の情報が ”,” で区切られて並んでいます。例えば、このファイル（battleships.csv）の情報を loadStrings 関数で読み込む（lines = loadStrings(“battleships.cvs”);）とすると、lines[0] は ” name,fuel,durableness” という文字列が記録されることになります。同様にして、lines[1] は ”Kongo,80,63”、lines[2] は “Hiei,80,63”、

lines[3] は “Haruna,80,63”、lines[12] は “Musashi,250,94” という文字列が記録されることになります。配列 lines に記録されている情報は、名前 , 燃料 , 耐久という情報が並んでいます。つまり、”,” で区切られている一つ一つの項目に意味があります。そこで、この意味のある情報をバラバラにして取り出すことが必要になります。このような目的に使える関数として split 関数があります。この関数は文字列を指定した文字を区切り文字を基準に分割する関数です。

excel の表情報

name	fuel	durableness
Kongo	80	63
Hiei	80	63
Haruna	80	63
Kirishima	80	63
Fuso	85	67
Yamashiro	85	67
Ise	85	74
Hyuga	85	74
Nagato	100	80
Mutsu	100	80
Yamato	250	93
Musashi	250	94

CVS ファイルの中身

name,fuel,durableness
Kongo,80,63
Hiei,80,63
Haruna,80,63
Kirishima,80,63
Fuso,85,67
Yamashiro,85,67
Ise,85,74
Hyuga,85,74
Nagato,100,80
Mutsu,100,80
Yamato,250,93
Musashi,250,94

String[] split(文字列 , 区切り文字)
文字列を区切り文字で分割し、その結果を String 型の配列に保存する関数です。

例えば、String[] items = split(“Kongo,80,63”,”,”) とすると、“Kongo,80,63” という文字列を区切り文字 ”,” で分割するので、items[0] には ”Kongo”、items[1] には ”80”、items[2] には ”63” という文字列が保存されることになります。この利用したサンプルを次に示します。

サンプルプログラム 12-16

```
PFont font;
String[] name;int[] fuel,durable;
int yourID;
int myID;
boolean playing = false;
void setup(){
  size(300,210);
  String[] lines = loadStrings("battleships.csv");
  font = loadFont("Serif-48.vlw");
  textFont(font,24);
  name = new String[lines.length-1];
  fuel = new int[lines.length-1];
  durable = new int[lines.length-1];
  for(int i=0;i<lines.length-1;i++){
    String[] items = split(lines[i+1],",");
    println(items);
    name[i] = items[0];
    fuel[i] = int(items[1]);
    durable[i] = int(items[2]);
  }
}
void mouseClicked(){
  yourID = int(random(name.length));
  myID = int(random(name.length));
  playing = true;
}
```

```
void draw(){background(255);
  fill(0);
  if(playing){
      textAlign(CENTER,CENTER);
      text("You choose "+name[yourID]+".",0,0,width,70);
      text("I choose "+name[myID]+".",0,70,width,70);
      String msg;
      if(durable[myID] > durable[yourID]){
        msg = "I win!!";
      }else if(durable[myID] == durable[yourID]){
        msg = "Draw!!";
      }else{
        msg = "You win!!";
      }
      text(msg,0,140,width,70);
  }
}
```

このように loadStrings 関数などを使うとファイルからデータを読み出すことが出来るようになります。

ファイルへの保存

ファイルからのデータの書き出しには、読み出しの時と同じように、

1) 一気に行毎に全てのデータを書き出す、
2) 一行ずつデータを書き出す、
3) 一文字ずつデータを読み出す

の３通りの方法があります。一番簡単の方法は、1) の方法です。反対に、一番面倒な方法は 3) の方法です。2) の 1 行ずつ書き出すという処理を行うことが多いように思います。ファイルに一気にデータ（String 型）を書き出すためには、saveStrings 関数を利用します。

サンプルプログラム 12-17

```
String fruits = "apple banana orange strawberry";
String[] items = split(fruits," ");
saveStrings("fruits.txt",items);
```

1 行ずつ書き出すためには、最初に書き出すファイルを指定し、PrinterWriter と呼ばれるデータ型の変数を用意します。この変数から println メソッドなどを利用して、データを書き出して行きます。本来であれば、エラー発生に備えた処理を記述する必要があります。

サンプルプログラム 12-18

```
PrintWriter output;
void setup(){
  size(400,400);
  output = createWriter("pos.txt");
}
void draw(){
  background(255);
  fill(255,0,0);
  ellipse(mouseX,mouseY,10,10);
  output.println(mouseX+","+mouseY);
}
void mouseClicked(){
// マウスをクリックすると、書き出し終了
  output.flush(); // メモリに残っている内容を書き出す
  output.close();// 書き出し終了
  exit();
}
```

ファイルへの書き出しは、駆け足でごめんなさい。