

例とリファレンス

Processingを学ぶ過程で、たくさんのコードを探検することになるでしょう。いろんなコードを走らせ、書き換え、ときには壊し、新しいものに生まれ変わるものまで拡張してください。そのために、ダウンロードしたProcessingには個々の機能をデモするたくさんの例(example)があらかじめ入っています。例を見たいときは、Fileメニューの「Examples」を選択するか、ツールバーのOpenボタンをクリックしてください。Form、Motion、Imageといった機能ごとに分類されています。リストを眺めて面白そうなトピックを見つけたら、すぐに試してみましょう。

プログラムを見ると、オレンジ色で表示されているコードがあるはずです。オレンジはProcessing固有の言葉であることを示していて、それをマウスで選択してからHelpメニューの「Find in Reference」を実行すると、ブラウザ上にリファレンスが表示されます。Helpメニューの代わりに、右クリック(MacではCtrl+クリック)で開くこともできます。このリファレンスは次のURLでも参照可能です。

<http://www.processing.org/reference/>

Processingリファレンスは個々の機能に関する解説と例文の集まりです。リファレンスにある例文は、Examplesのコードよりもずっと短い4～5行程度のものが主なので、理解しやすいでしょう。プログラミングをしている間はずっと開いておくことをおすすめします。項目はトピックごと、あるいはアルファベット順に整理されていますが、量が多いので、探し物があるときはブラウザ上でテキスト検索を使ったほうが見つけやすいかもしれません。

初心者が使うことを念頭に、明解で理解しやすいリファレンスとなるよう心がけています。ありがたいことに、これまで多くの利用者から間違いを指摘する連絡をもらいました。あなたがもしリファレンスの誤りや改良点を見つけたら、各ページの先頭にある「please let us know」というリンクを使って報告してください。

3

描く

Draw

コンピュータの画面に図形を描くことは、方眼紙の上でそうするのに似ています。最初は技術的手続きばかりですが、新しいアイデアをいくつか導入することで単純な図形の描画からアニメーションやインタラクションへと発展させます。いきなり難しいものに取り組むのではなく、基礎からはじめましょう。

コンピュータの画面はピクセルと呼ばれる発光体が格子状に並んだものです。各ピクセルは座標で示すことができます。Processingにおけるx座標はウィンドウの左端からの距離、y座標は上端からの距離です。あるピクセルの座標は(x, y)のように書き表します。画面サイズが200×200ピクセルのとき、左上隅は(0, 0)で、右下隅は(199, 199)です。この数字はちょっと奇妙ですね。どうして、1から200ではなく、0から199と書くのでしょうか。その理由はコードのなかにあります。後ほど詳しく説明しますが、多くの場合、0からカウントしたほうが計算しやすいのです。

ウィンドウを開いたら、関数と呼ばれるコード要素を使って図形を描きます。関数はProcessingプログラムの基本的な構成単位です。そのふるまいはパラメータで定義されます。たとえば、ウィンドウの幅と高さを設定するsize()がよく使われる関数のひとつで、ほとんどすべてのProcessingプログラムに登場します(サイズを指定しないとウィンドウの大きさは100×100ピクセルになります)。

Example 3-1: ウィンドウを描く

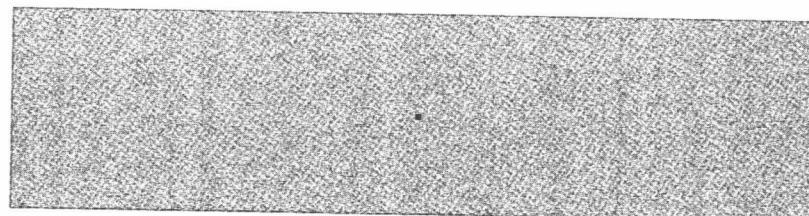
size()関数には2つのパラメータがあります。1つ目は幅、2つ目は高さです。幅800ピクセル、高さ600ピクセルのウィンドウが欲しいときは次のようにします。

```
size(800, 600);
```

実際に実行して、結果を確かめてください。パラメータをぐっと小さくしたり、ディスプレイのサイズより大きくするとどうなるかも試してみましょう。

Example 3-2: 点を描く

ウィンドウ内のピクセルの色を変えたいときはpoint()関数を使います。パラメータは2つで、x座標に続いてy座標を指定します。次の例は小さなウィンドウを開き、その中心となる(240, 60)の位置に点を描きます。

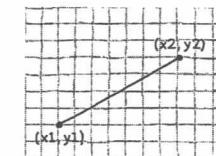


```
size(480, 120);
point(240, 60);
```

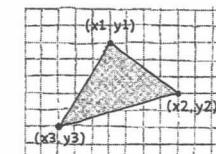
パラメータを変えて、ウィンドウの四隅に点を描いてみましょう。また、複数の点を並べて、水平、垂直、斜めの線を描いてみましょう。

基本的な図形

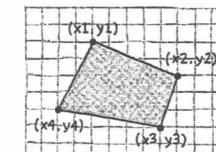
Processingには基本的な図形を描く関数のグループがあります(図3-1)。こうした単純な形を組み合わせて、葉っぱや顔のようにもっと複雑な図形を組み立てることができます。1本の線を描くためには4つのパラメータが必要で、そのうちの2つは始点の位置、もう2つは終点の位置です。



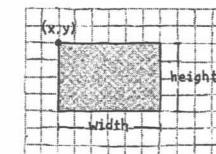
line(x1, y1, x2, y2)



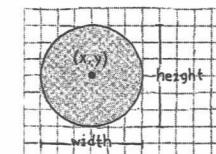
triangle(x1, y1, x2, y2, x3, y3)



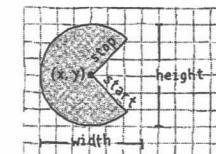
quad(x1, y1, x2, y2, x3, y3, x4, y4)



rect(x, y, width, height)



ellipse(x, y, width, height)



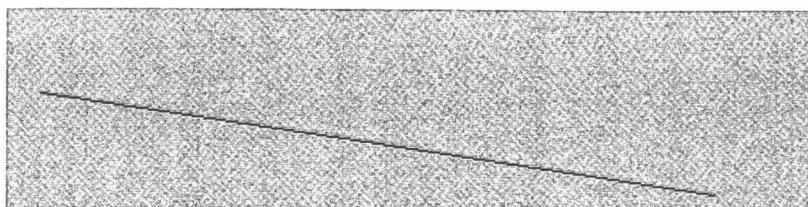
arc(x, y, width, height, start, stop)

図3-1

形と座標

Example 3-3:線を描く

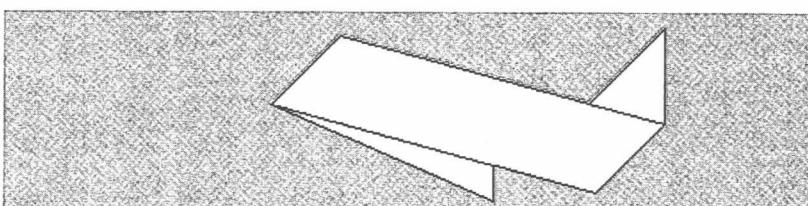
2つの座標 (20, 50) と (420,110) を結ぶ1本の線を描きます。



```
size(480, 120);  
line(20, 50, 420, 110);
```

Example 3-4:基本図形を描く

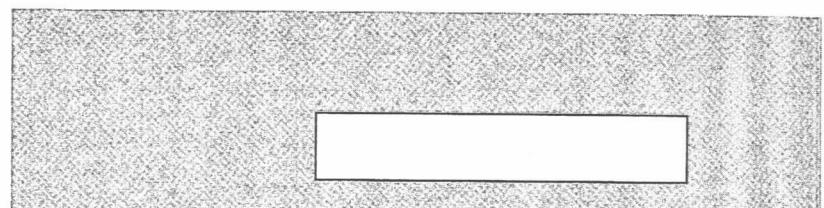
線を描くには4つのパラメータが必要でした。三角形は6つ、四辺形は8つ必要です(1点につき1つのペア)。



```
size(480, 120);  
quad(158, 55, 199, 14, 392, 66, 351, 107);  
triangle(347, 54, 392, 9, 392, 66);  
triangle(158, 55, 290, 91, 290, 112);
```

Example 3-5:長方形を描く

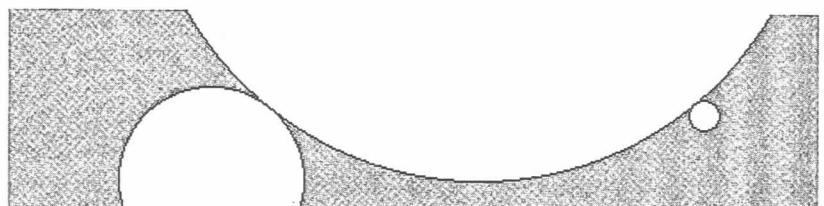
長方形と円は4つのパラメータで設定します。1つ目と2つ目が基準点の座標、3つ目が幅、4つ目が高さです。たとえば、座標 (180, 60) を基準に、幅220ピクセル、高さ40ピクセルの長方形を描くとしたら、次のようにします。



```
size(480, 120);  
rect(180, 60, 220, 40);
```

Example 3-6:円を描く

長方形の基準点は左上の角でしたが、円の場合はその中心です。次の例の最初の円を見てください。中心点のy座標がウィンドウの外側に設定されています。このようにウィンドウからはみ出てもエラーにはなりません。

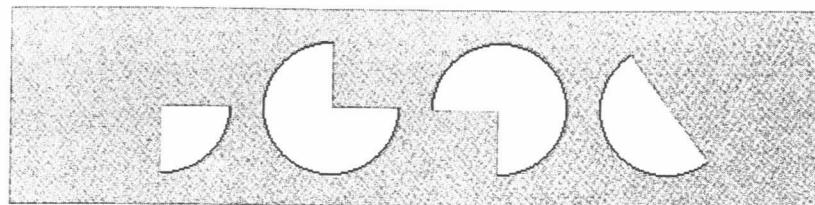


```
size(480, 120);  
ellipse(278, -100, 400, 400);  
ellipse(120, 100, 110, 110);  
ellipse(412, 60, 18, 18);
```

Processingには正方形や正円を描く専用の関数は用意されていません。`rect()` や `ellipse()` の幅と高さを同じ値に設定することで、対応してください。

Example 3-7: 円の一部を描く

`arc()` 関数は円を部分的に描きます。



```
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);
```

1つ目と2つ目のパラメータは中心の位置です。3つ目と4つ目が幅と高さ。5つ目と6つ目は円弧の始まりと終わりの角度を表し、単位は「度」ではなくラジアンです。ラジアンは円周率 (3.14159) を元にした値で、図 3-2 のような関係になります。

この例のように、Processing ではよく使う4つの角度に固有の名前を付けています。`PI`、`QUARTER_PI`、`HALF_PI`、`TWO_PI` の4つで、それぞれ 180 度、45 度、90 度、360 度に対応します。

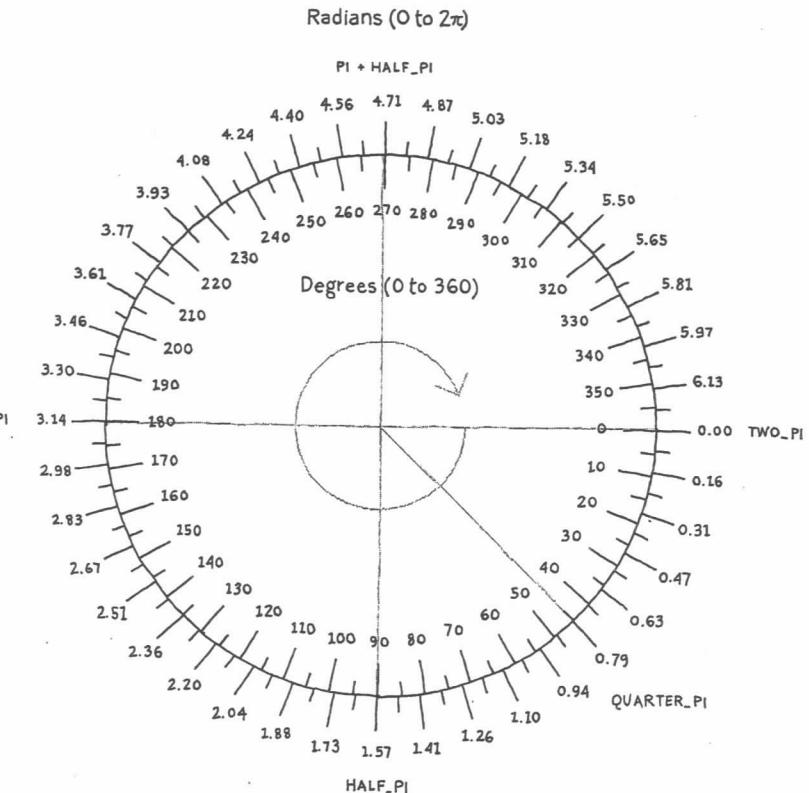


図 3-2 ラジアン (radian) と度 (degree) による角度の表記

Example 3-8: 度を使って描く

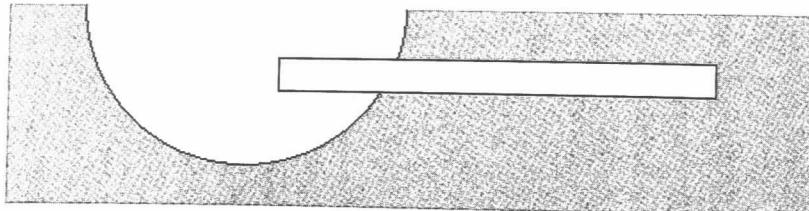
角度を度で指定したい人は、`radians()` 関数を使ってラジアンに変換するといいでしよう。この関数は度で指定した角度を、対応するラジアン値に変えます。次の例は、Example 3-7 と同じ图形を `radians()` を使って描くものです。

```
size(480, 120);
arc(90, 60, 80, 80, 0, radians(90));
arc(190, 60, 80, 80, 0, radians(270));
arc(290, 60, 80, 80, radians(180), radians(450));
arc(390, 60, 80, 80, radians(45), radians(225));
```

描画の順序

プログラムを実行すると、コンピュータは先頭から1行ずつコードを読み込んでいって、最後の行を処理したら停止します。コードの実行順序は重要です。重なり合う複数の図形がある場合、最後に描かれた図形が一番上に表示されます。

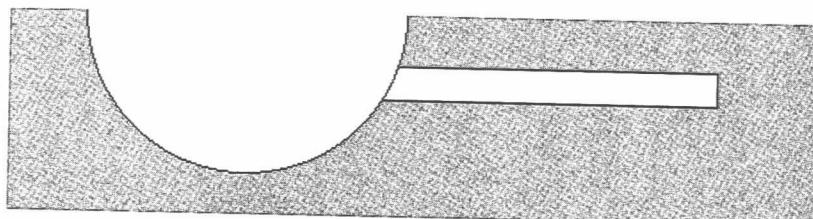
Example 3-9: 描画の順序を意識する



```
size(480, 120);
ellipse(140, 0, 190, 190);
// 長方形は円の後に描かれるので円の上にあらわれます
rect(160, 30, 260, 20);
```

Example 3-10: 順序を逆にする

Example 3-9を修正して、長方形と円の重ね合わせを逆にします。長方形が円の下に隠れました。



```
size(480, 120);
rect(160, 30, 260, 20);
// 円が後に描かれるよう変更されたので長方形の上にあらわれます
ellipse(140, 0, 190, 190);
```

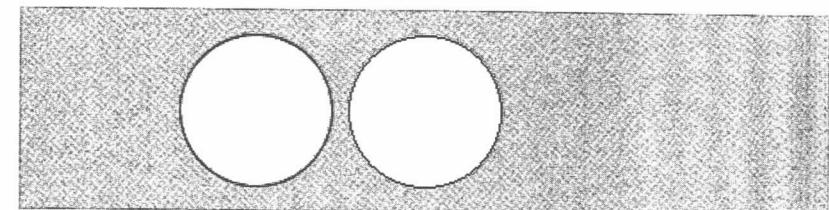
筆で絵を描くときと同じで、最後に描いたものが一番上にあるように見えます。

図形の性質

線の太さと滑らかさ（アンチエイリアシング）はもっとも基本的でよく使う設定項目です。

Example 3-11: 滑らかな線とギザギザの線

隣り合うピクセルの明暗をぼかすことで、ギザギザな線を滑らかな見た目に変えるのがアンチエイリアシング処理です。Processingではこの処理がデフォルトで有効です。無効にしたいときは、`noSmooth()`関数を使ってください。`smooth()`関数を実行すると再度有効になります。

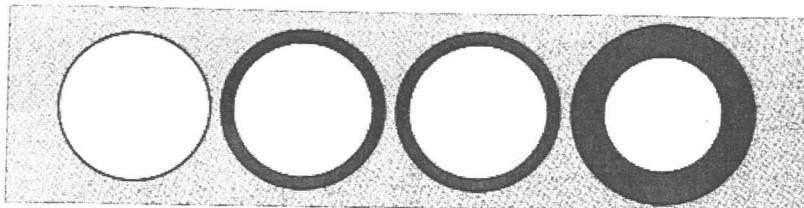


```
size(480, 120);
noSmooth();           // smoothingを無効に
ellipse(140, 60, 90, 90);
smooth();            // smoothingを有効に
ellipse(240, 60, 90, 90);
```

[NOTE] Processing 2.0より前のバージョンでは、`smooth()`関数を使ってエンチエイリアシングを有効にしないと線がギザギザになってしまう場合があります。有効にしたいときは、`setup()`の中に`smooth()`を置いてください。なお、Processingには本書のサンプルコードが付属していて、Processing1.5に付属するサンプルには必要に応じて`smooth()`が書き込まれています。Fileメニューから Examples → Books → Getting Started の順に進むと、本書のサンプルのリストが表示されます。

Example 3-12: 線の太さを変える

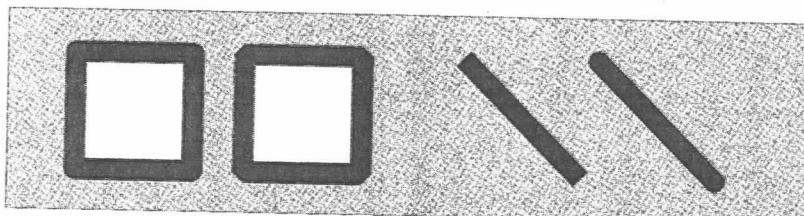
線の太さは1ピクセルがデフォルトですが、`strokeWeight()`関数によって変更できます。`strokeWeight()`のパラメータは1つで、太さ(ピクセル数)を指定します。



```
size(480, 120);
ellipse(75, 60, 90, 90);
strokeWeight(8); // 線の太さを 8 ピクセルに
ellipse(175, 60, 90, 90);
ellipse(279, 60, 90, 90);
strokeWeight(20); // 線の太さを 20 ピクセルに
ellipse(389, 60, 90, 90);
```

Example 3-13: 線の属性を変更する

太さ以外の属性を変える関数を2つ取り上げます。`strokeJoin()`は線の繋ぎ方(角の形)を指定する関数です。`strokeCap()`は線の両端の形状を指定する関数です。



```
size(480, 120);
strokeWeight(12);
strokeJoin(ROUND); // 角を丸く
rect(40, 25, 70, 70);
strokeJoin(BEVEL); // 角を斜めにカット
rect(140, 25, 70, 70);
strokeCap(SQUARE); // 線の端を直角に
line(270, 25, 340, 95);
strokeCap(ROUND); // 線の端を丸く
line(350, 25, 420, 95);
```

`rect()`と`ellipse()`を使って図形を配置するときの基準点は`rectMode()`と`ellipseMode()`という関数で変更できます。長方形の基準点を中心にして、円の基準点を左上隅に設定するといったことが可能です。

属性の変更は、そのあと描かれるすべての図形に影響します。もう一度、Example 3-12を見てください。2個目と3個目の円は線の太さが同じですが、`strokeWeight()`は1回しか実行していません。1回の変更が両方に及ぼしています。

色

これまでの例では、どの図形も白か黒の輪郭で描かれ、ウインドウの背景はいつも明灰色でした。`background()`、`fill()`、`stroke()`といった関数を使って、変更してみましょう。パラメータは0から255の値を取ります。0は黒、128は中間的な灰色、255は白です。図3-3は256段階の濃淡と数値の関係を示したものです。

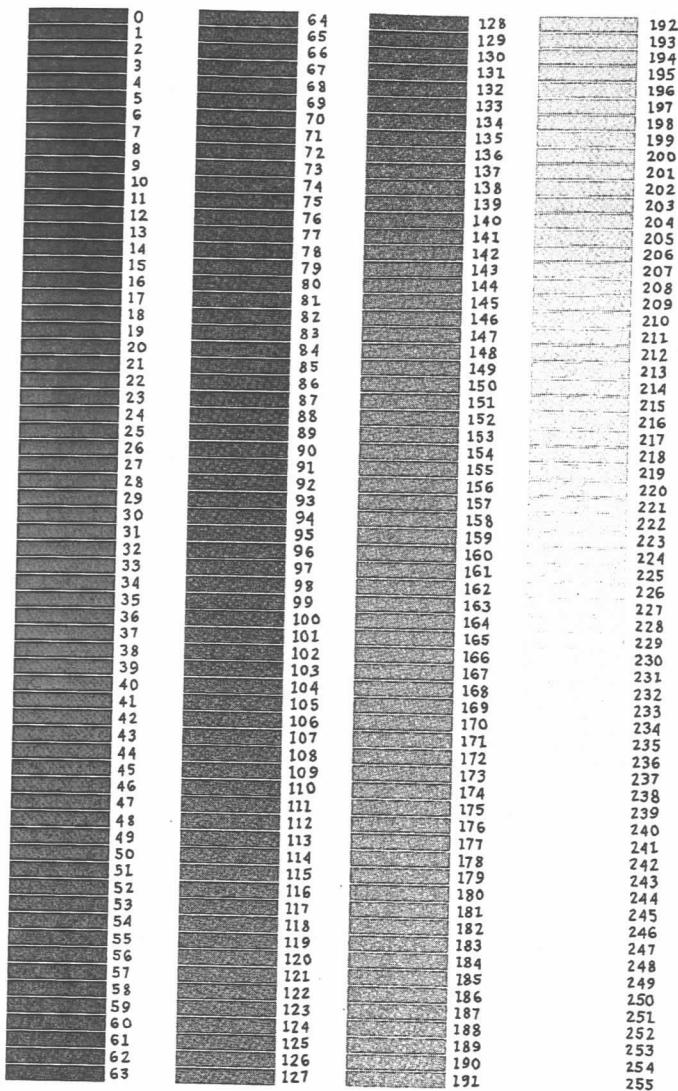
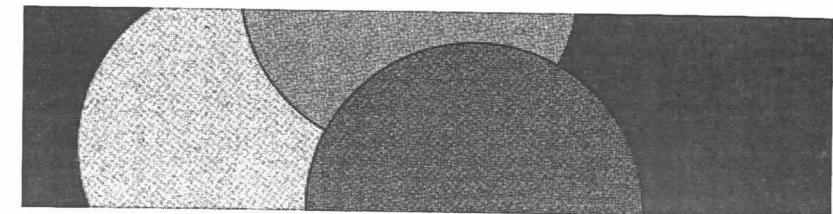


図3-3 0から255までのグレー階調

Example 3-14: グレーを塗る

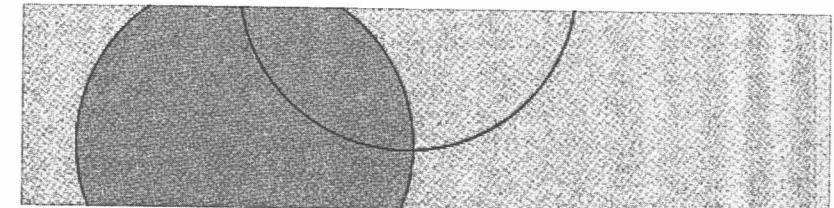
黒を背景に、濃さの異なる3種類のグレーを表示します。



```
size(480, 120);
background(0); // 背景を黒に
fill(204); // 明灰色を選択
ellipse(132, 82, 200, 200); // 明灰色の円
fill(153); // 灰色を選択
ellipse(228, -16, 200, 200); // 灰色の円
fill(102); // 暗灰色を選択
ellipse(268, 118, 200, 200); // 暗灰色の円
```

Example 3-15: 塗りと線の変更

noFill()関数を実行したあとは、図形の内部を塗りつぶしません。noStroke()関数は線の描画をオフにします。輪郭となる線も表示されません。



```

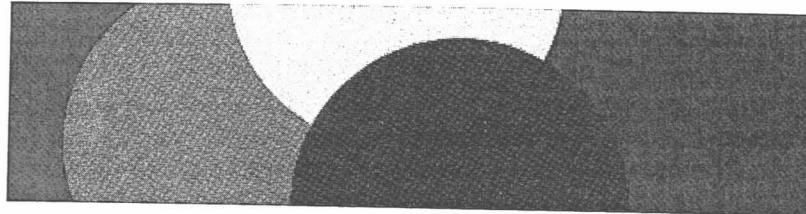
size(480, 120);
fill(153); // 灰色
ellipse(132, 82, 200, 200); // 灰色の円
noFill(); // 塗りを無効に
ellipse(228, -16, 200, 200); // 輪郭線だけの円
noStroke(); // 輪郭線を無効に
ellipse(268, 118, 200, 200); // 見えない!

```

この例のように、塗りと線の両方を無効にしてしまうと、なにも表示されなくなってしまいます。誤ってそうしないように気をつけましょう。

Example 3-16: 色付きの図形を描く

赤、緑、青の三原色に対応する3つのパラメータを設定して、色を表示してみましょう。この本はモノクロなので全部灰色に見えますが、画面には複数の色が現れます。



```

size(480, 120);
noStroke();
background(0, 26, 51); // 背景を濃い青に
fill(255, 0, 0); // 赤
ellipse(132, 82, 200, 200); // 赤い円
fill(0, 255, 0); // 緑
ellipse(228, -16, 200, 200); // 緑の円
fill(0, 0, 255); // 青
ellipse(268, 118, 200, 200); // 青い円

```

赤、緑、青の3色 (RGBと呼ばれることがあります) に対応する3つの値によって、画面上の色は決定されます。それぞれの値が0から255の範囲を持つのはグレースケールのときと同じです。RGB値は直観的ではないため、Processingにはカラーセレクタが用意されています。Toolsメニューの「Color Selector」を実行すると、グラフィックツールでお馴染みのカラーパレットが表示されます（図3-4）。色選び、RGBそれぞれの値をbackground()、fill()、stroke()といった関数のパラメータとして使います。

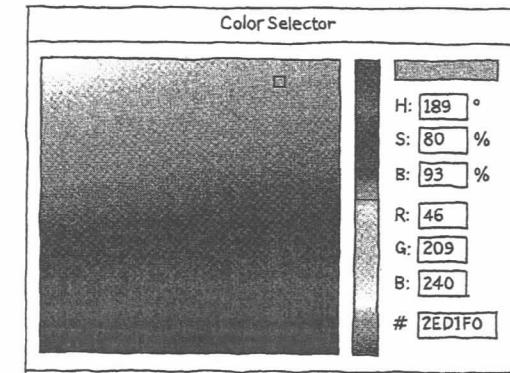
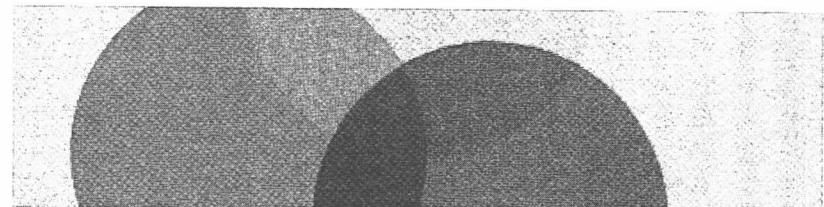


図3-3 0から255までのグレー階調

Example 3-17: 透明度の設定

fill()やstroke()に4つ目のパラメータを加えることで、透明度を設定できます。このパラメータをアルファ値といいます。値の範囲は0から255で、0にすると完全に透明となり表示されません。0と255の間では半透明になり、下の色が透けて混ざり合います。255の場合は完全に不透明、つまりアルファ値を指定しないときと同じです。



```

size(480, 120);
noStroke();
background(204, 226, 225); // 明るい青
fill(255, 0, 0, 160); // 赤

```

```

ellipse(132, 82, 200, 200); // 赤い円
fill(0, 255, 0, 160); // 緑
ellipse(228, -16, 200, 200); // 緑の円
fill(0, 0, 255, 160); // 青
ellipse(268, 118, 200, 200); // 青い円

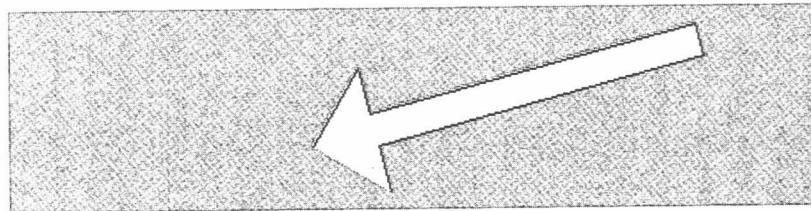
```

形を作る

Processing が扱える形状は、単純な幾何学的図形だけではありません。複数の点をつなぎ合わせて、新しい形を定義してみましょう。

Example 3-18: 矢印を描く

`beginShape()` 関数は新しい形を描き始める合図です。`vertex()` 関数を使って輪郭をたどるように xy 座標をひとつひとつ定義していきます。最後に描き終わりの合図として `endShape()` 関数を実行します。



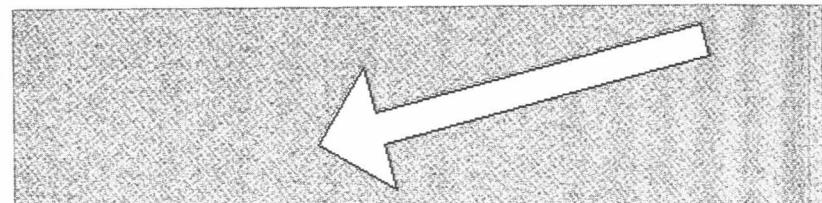
```

size(480, 120);
beginShape();
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape();

```

Example 3-19: 隙間を閉じる

Example 3-18 のプログラムでは、最初の点と最後の点がつながりませんでした。`endShape()` のパラメータとして `CLOSE` を付け加えると、閉じた形になります。



like this:

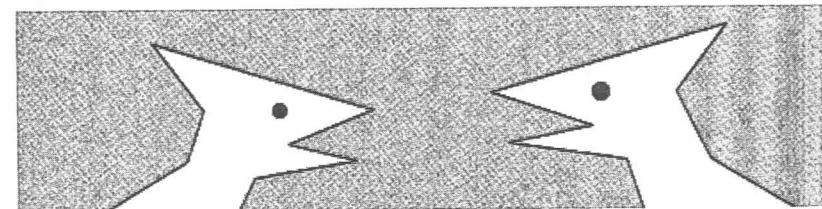
```

size(480, 120);
beginShape();
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape(CLOSE);

```

Example 3-20: 生物の創造

`vertex()` の威力は、複雑な輪郭を持つ形状を定義できるところにあります。Processing は瞬時に無数の線を描いて、創造の泉から湧き出た空想の産物に形を与えてくれるでしょう。次の例は、まだ控えめな規模ですが、これまでの例よりは複雑です。



```
size(480, 120);

// 左の生き物
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);
ellipse(155, 60, 8, 8);

// 右の生き物
fill(255);
beginShape();
vertex(370, 120);
vertex(360, 90);
vertex(290, 80);
vertex(340, 70);
vertex(280, 50);
vertex(420, 10);
vertex(390, 50);
vertex(410, 90);
vertex(460, 120);
endShape();
fill(0);
ellipse(345, 50, 10, 10);
```

コメント

コード中のダブルスラッシュ(//)はコメントの目印で、その後ろに書かれたテキストは実行時に無視される部分です。どんな動作をするコードなのかをコメントとして書いておくと後々のためにになります。また、他人がそのプログラムを読むときにも、あなたの思考プロセスを伝えてくれるコメントの存在がとても重要となるでしょう。

コメントには別の使い方もあります。複数の選択肢を試したいときに便利な方法です。実例で説明しましょう。円の色を選んでいます。最初に明るい赤を試してみました。

```
size(200, 200);
fill(165, 57, 57);
ellipse(100, 100, 80, 80);
```

この色を見て、違う赤を試したくなつたとします。ただし、古い赤も消したくはありません。そういうときは、`fill()`の行を複製(コピーアンドペースト)して、新しい行に別の赤を定義し、古い行は「コメントアウト」します。

```
size(200, 200);
//fill(165, 57, 57);
fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

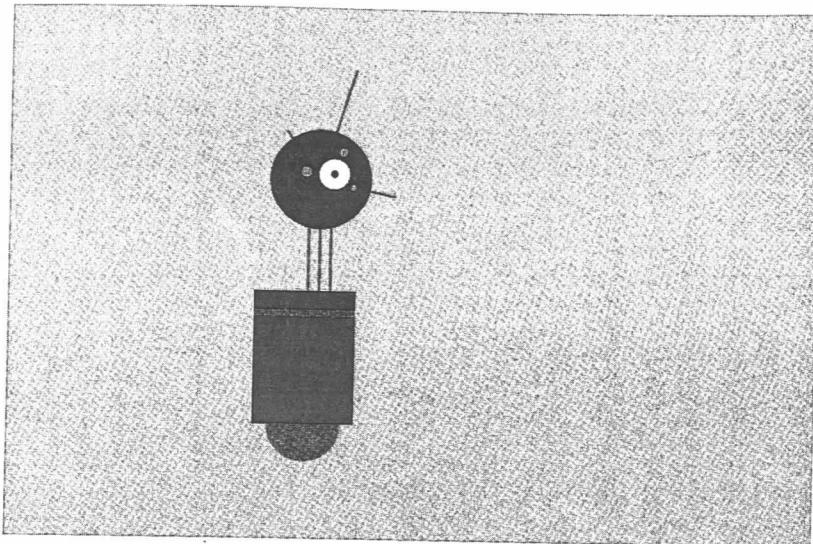
行頭に//を挿入して、その行を一時的に無効化しました。前の赤に戻したくなつたら、//を消して、新しい`fill()`の行をコメントアウトします。

```
size(200, 200);
fill(165, 57, 57);
//fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

[NOTE] ショートカットCtrl-/ (Cmd-/)はカーソル行に//を挿入してコメントアウトします。すでにコメントになっている行に対して使うと、//が削除されます。複数の行を選択している状態でこのショートカットを実行すると、複数行が同時にコメントアウトされます。

Processingでスケッチを作っていると、いろんなアイデアを繰り返し試している自分に気づくはずです。コメントでメモを書いたり、無効にしたコードを残しておくことで、選択の過程を記録することができます。

Robot 1: Draw



これはProcessing Robot の P5 です。本書には、このロボットを描く8通りのプログラムが載っていて、それぞれが異なるプログラミングテクニックの使用例になっています。

P5のデザインは、スプートニク1号(1957)、Stanford Research Instituteの Shakey (1966-1972)、David Lynchの「デューン／砂の惑星」(1984)に登場したファイタードローン、そして「2001年宇宙の旅」(1968)の HAL 9000 にインスピアされています。

最初のロボットは3章で紹介した描画関数を使ったものです。`fill()`と`stroke()`でグレースケールの設定をし、`line()`、`ellipse()`、`rect()`で首、アンテナ、胴体、頭部を描いています。これらの関数にもっと慣れるため、パラメータを変更してロボットのデザインに手を加えてください。

```
size(720, 480);
strokeWeight(2);
background(204);
ellipseMode(RADIUS);

// 首
stroke(102);           // 線の色をグレーに
line(266, 257, 266, 162); // 左
line(276, 257, 276, 162); // 中央
line(286, 257, 286, 162); // 右

// アンテナ
line(276, 155, 246, 112); // 小
line(276, 155, 306, 56); // 高
line(276, 155, 342, 170); // 中央

// 胴体
noStroke();           // 輪郭なし
fill(102);             // 塗り色をグレーに
ellipse(264, 377, 33, 33); // 反重力球体
fill(0);               // 塗り色を黒に
rect(219, 257, 90, 120); // 胴体
fill(102);             // 塗り色をグレーに
rect(219, 274, 90, 6); // グレーのストライプ

// 頭部
fill(0);               // 塗り色を黒に
ellipse(276, 155, 45, 45); // 頭
fill(255);             // 塗り色を白に
ellipse(288, 150, 14, 14); // 大きい目
fill(0);               // 塗り色を黒に
ellipse(288, 150, 3, 3); // 瞳孔
fill(153);             // 塗り色を明るいグレーに
ellipse(263, 148, 5, 5); // 小さい目1
ellipse(296, 130, 4, 4); // 小さい目2
ellipse(305, 162, 3, 3); // 小さい目3
```

4

変数

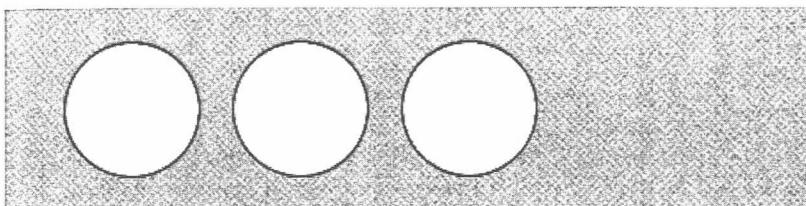
Variables

値をメモリに保存しておき、プログラムのなかで再利用できるようにするのが変数の役割です。プログラムの実行中、変数は何度も繰り返し使用でき、その値は簡単に変えられます。

変数を使う最大の理由は、不要な繰り返しを避けるためです。プログラムの中で同じ数字が2回以上登場したら、その値を変数に収めることを検討するべきです。そうすることで、わかりやすく修正の容易なコードになるでしょう。

Example 4-1: 同じ値の再利用

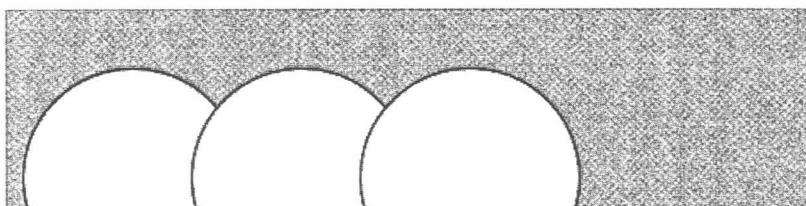
y座標と直径を変数に格納し、3つの円に対して使用します。すべての円が同じ値を受け取ることになります。



```
size(480, 120);
int y = 60;           // y座標
int d = 80;           // 直径 (diameter)
ellipse(75, y, d, d); // 左
ellipse(175, y, d, d); // 中央
ellipse(275, y, d, d); // 右
```

Example 4-2: 値の変更

変数yとdを変更するだけで、3つの円すべてに反映されました。



```
size(480, 120);
int y = 100; // y座標
int d = 130; // 直径
ellipse(75, y, d, d); // 左
ellipse(175, y, d, d); // 中央
ellipse(275, y, d, d); // 右
```

変数を使わずに同じことをしようとしたら、y座標を3か所、直径は6か所も直すめになります。Example 4-1と4-2のコードを比べると、最後の3行は同じで、修正が必要なのは途中の2行だけです。変数を使うことで、変更が必要な部分とそうでない部分を分離できます。ある図形の色を、目で確かめながら選びたいとしましょう。その場合も、色の設定に必要なパラメータは変数にして1か所にまとめておくと、最低限の手間で多くの選択肢を効率よく試すことができます。

変数の作成

自分で変数を作るときは、名前、データ型、そして値を考えて決めます。

名前はその変数に格納される値の意味が伝わる、一貫性があって、長すぎない言葉を選びましょう。たとえば、radiusは単にrとするより意味が明確なので、後日コードを見直すときにもわかりやすいはずです。

ある変数に格納できる値の種類や範囲はデータ型によって決まります。たとえば、整数型(int)の変数は小数点を持たない数を記憶できます。格納できる値を分類すると、整数のほかに、浮動小数点、文字、単語、画像、フォントなどがあり、それぞれにデータ型が用意されています。

変数を使う前にかならず必要となるのが宣言です。ここでいう宣言とは、どんな値を記憶するのかを示すためにデータ型(前述のintがそのひとつ)を定義し、コンピュータのメモリに格納する領域を確保する処理のことです。データ型と名前が決まれば、次のようにして値をセットできます。

```
int x; // xをint型変数として宣言
x = 12; // xに値を割り当てる
```

次の例は、同じことをより短いコードでやっています。

```
int x = 12; // xをint型として宣言し、値を割り当てる
```

変数名の前にデータ型を書く必要があるのは、最初の宣言のとき1回だけです。コンピュータはデータ型のついた変数を見つけると、新しい変数の宣言として処理しようとしますが、同じ名前の変数を2度宣言するとエラーになります。

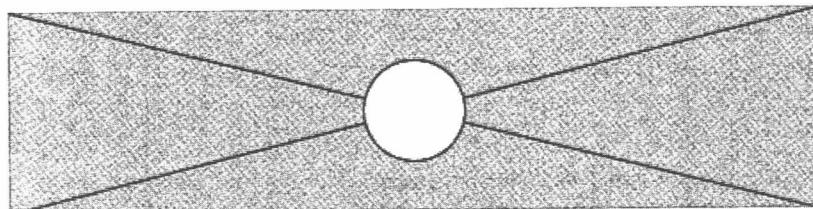
```
int x;      // xをint型変数として宣言  
int x = 12; // エラー! xという名前の変数を2度作ろうとしました
```

変数の処理

Processingは実行中のプログラムの状態を表す特殊な変数を持っています。現在のウィンドウの幅と高さを記憶しているwidthとheightがその一例で、これらはsize()関数を実行したときにセットされます。widthとheightを使うことで、ウィンドウの大きさに対して相対的に図形の位置や大きさを決めることができます。

Example 4-3: ウィンドウサイズに合わせる

size()関数のパラメータを変更して、見た目がどう変化するか確認しましょう。



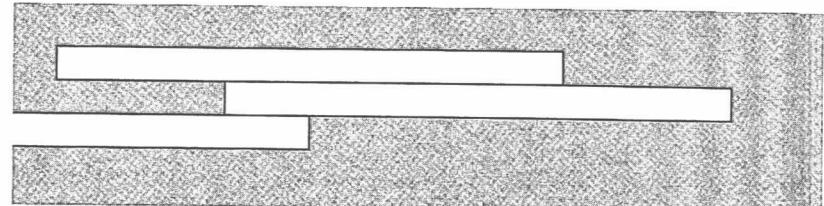
```
size(480, 120);  
line(0, 0, width, height); // (0, 0)から(480, 120)へ線を描く  
line(width, 0, 0, height); // (480, 0)から(0, 120)へ線を描く  
ellipse(width/2, height/2, 60, 60);
```

似た働きの変数は他にもあります。マウスやキーボードの状態を保持する変数については5章で説明します。

変数と計算

プログラミングと数学は同じものだと考える人がいます。たしかに、ある種のコーディングにおいて数学は有効ですが、大部分の課題は算数レベルの計算で解決できます。

Example 4-4: 基本的な計算



```
size(480, 120);  
int x = 25;  
int h = 20;  
int y = 25;  
rect(x, y, 300, h);      // 上  
x = x + 100;  
rect(x, y + h, 300, h); // 真ん中  
x = x - 250;  
rect(x, y + h*2, 300, h); // 下
```

コードを見ると、+や-、*といった記号が使われています。これらは演算子（オペレータ）と呼ばれ、2つの数値の間において式を表すためにあります。たとえば、5 + 9や1024 - 512が式です。基本的な演算子は次のとおり。

- + 足す（加算）
- 引く（減算）
- * かける（乗算）
- / 割る（除算）
- = 代入

Processingには演算子の優先順位に関するルールがあって、それによって式のどこから計算が始まり、どういう順番で進んでいくかが決まります。次のような短い式を解釈するときも、優先順位に関する知識が大きな意味を持ちます。

```
int x = 4 + 4 * 5; // xに24が代入されます
```

乗算 (*) が加算 (+) よりも優先されるルールにより、この式では $4 * 5$ が最初に評価されます。そこに 4 を足すので、答は 24 となり、それが変数 x に代入されます。代入が最後に行われるのは、= がこの式のなかでもっとも優先順位の低い演算子だからです。
カッコを用いて、この式をもっと明確に書くことができます。結果は同じです。

```
int x = 4 + (4 * 5); // x に 24 を代入
```

足し算を先に実行する式を書きたいときは、カッコを付け足すだけです。カッコは乗算よりも高い優先順位を持っていて、計算の順番を変える働きがあります。

```
int x = (4 + 4) * 5; // x に 40 を代入
```

カッコは全演算子のなかで最高の優先順位です。完全な優先順位の表はリファレンス(202 ページ) にあります。

プログラマーが頻繁に使う式の形があり、そういう式は少ない文字数で書けるよう短縮形が用意されています。次の例は、ひとつの演算子で計算と代入を行います。

```
x += 10; // この式は x = x + 10 と同じ  
y -= 15; // こちらは y = y - 15 と同じ
```

次の例も短縮形です。変数に 1 を足したいとき、あるいは 1 を引きたいときによく使います。

```
x++; // x = x + 1 と同じ意味  
y--; // y = y - 1 と同じ
```

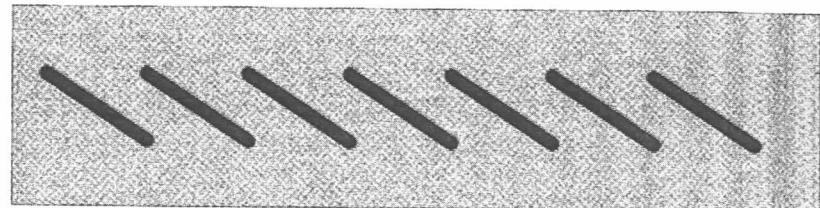
上記以外の短縮形については、リファレンスを見てください。

繰り返し

何度かプログラムを書くうちに、数行のコードが繰り返されるパターンの存在に気づくでしょう。for ループと呼ばれる構造を使うと、同じようなコードを並べるかわりに、ひとつのコードを繰り返し実行することができ、反復するパターンをより短いコードに濃縮できます。その結果、プログラムがモジュール化され、変更しやすくなります。

Example 4-5: 何度も同じことを繰り返す

for ループを使う前の繰り返しパターンの例です。



```
size(480, 120);  
strokeWeight(8);  
line(20, 40, 80, 80);  
line(80, 40, 140, 80);  
line(140, 40, 200, 80);  
line(200, 40, 260, 80);  
line(260, 40, 320, 80);  
line(320, 40, 380, 80);  
line(380, 40, 440, 80);
```

Example 4-6: for ループを使う

for ループを使って記述すると、このようにずっと短くなります。

```
size(480, 120);  
strokeWeight(8);  
for (int i = 20; i < 400; i += 60) {  
    line(i, 40, i + 60, 80);  
}
```

for ループを使ったコードは、いくつかの点で、これまでに書いたものと違います。{} に注目してください。2 つの波カッコに挟まれたコードはブロックと呼ばれ、for ループによって繰り返し実行される部分です。

次に、for の後ろのカッコ内を見てみましょう。セミコロンで区切られた 3 つの文によって、ブロック内のコードの実行回数をコントロールしています。3 つの文に、init (initialization=初期化)、test、update と名前を付けて、人間にとって読みやすいよう書き直すとこうなります。

```

for (init; test; update) {
    繰り返し実行されるコード
}

```

initの部分で、このforループで使う変数の宣言と初期値の代入を行うのが一般的です。変数名には*i*を使うことが多いのですが、深い意味はないので別の名前でもかまいません。testには変数の値を評価するための式を書き、updateには変数を変化させるための式を書きます。

図4-1に、どういう順番でforループが実行され、ブロック内の繰り返し処理がコントロールされるかを示します。

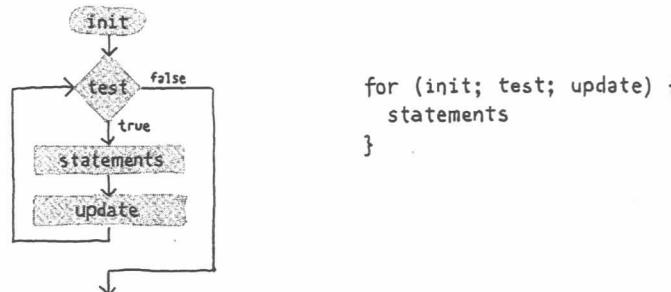


図4-1 forループの流れ図

testの部分についてはもう少し説明が必要でしょう。ここには必ず2つの値を比べるための式があります。この例では、 $i < 400$ がそれです。<のような演算子は関係演算子と呼ばれます。一般的な関係演算子は次のとおりです。

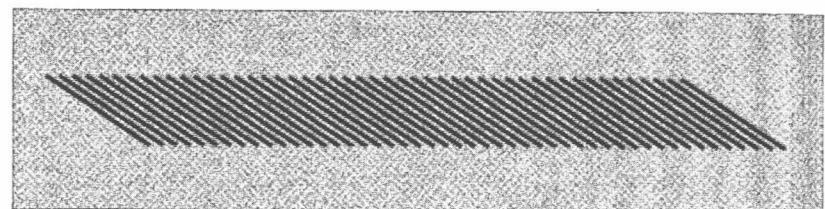
- > より大きい
- < より小さい
- >= 以上
- <= 以下
- == 等しい
- != 等しくない

関係演算子を含む式を評価した結果は真(true)か偽(false)のどちらかです。たとえば、 $5 > 3$ の結果は真となります。初めのうちは声に出しながら読むとわかりやすいかもしれません。「5は3より大きいですか?」「はい」。答が「はい」ならば真です。 $5 < 3$ の例でも試してみましょう。「5は3より小さいですか?」「いいえ」。結果は偽ですね。

評価の結果が真ならばブロックの中身が実行され、偽の場合はブロック内を飛ばしてループが終了します。

Example 4-7:forループで柔軟体操

forループを使う最大のメリットは、手早くコードを変更できる点にあります。ブロック内のコードを1か所変更すると、それが繰り返し実行されて効果が増幅されるわけです。次の例は、Example 4-6に少しだけ手を加えて、パターンの変化を作り出しました。

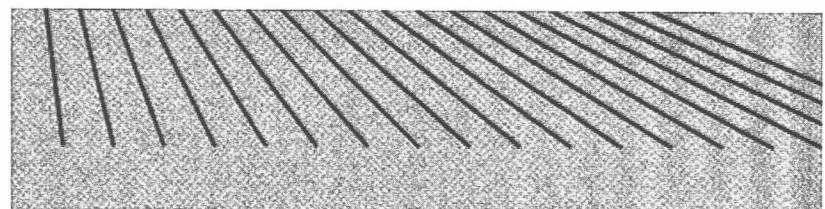


```

size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 8) {
    line(i, 40, i + 60, 80);
}

```

Example 4-8:扇状に広がるライン

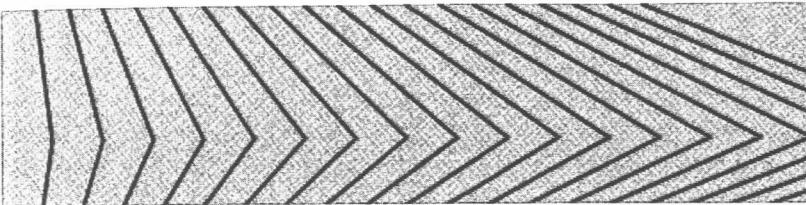


```

size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 20) {
    line(i, 0, i + i/2, 80);
}

```

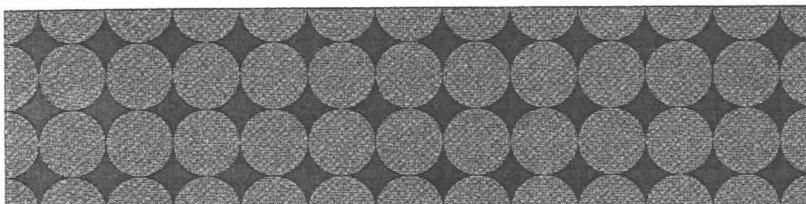
Example 4-9:よじれるライン



```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 20) {
    line(i, 0, i + i/2, 80);
    line(i + i/2, 80, i*1.2, 120);
}
```

Example 4-10:ループにループを埋め込む

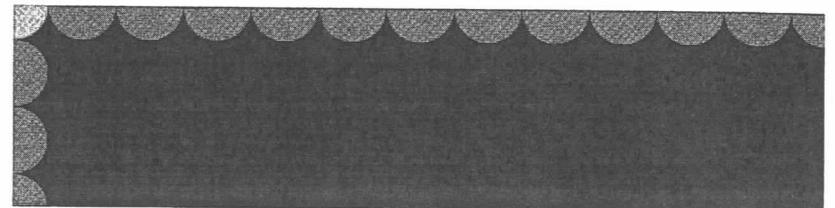
forループのなかにforループを埋め込むと、それぞれのループ回数を掛け算した結果が、全体の繰り返しの回数となります。最初に短い例をあげ、続くExample 4-11でそれを分解します。



```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y <= height; y += 40) {
    for (int x = 0; x <= width; x += 40) {
        fill(255, 140);
        ellipse(x, y, 40, 40);
    }
}
```

Example 4-11:横と縦の列

この例のforループは、一方に埋め込まれるのではなく、並んでいます。実行すると、片方のforループが横に13個の円を描き、もう一方のforループが縦に4個の円を描きます。

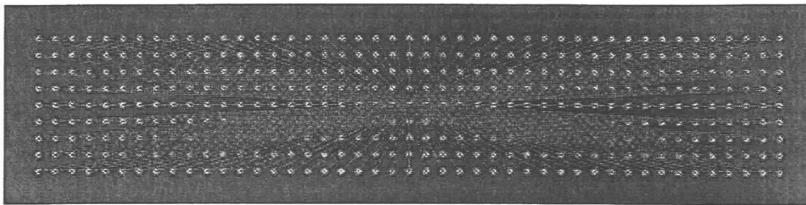


```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y < height+45; y += 40) {
    fill(255, 140);
    ellipse(0, y, 40, 40);
}
for (int x = 0; x < width+45; x += 40) {
    fill(255, 140);
    ellipse(x, 0, 40, 40);
}
```

Example 4-10のようにforループがforループのなかにある場合、繰り返しの回数は2つのループ回数の掛け算になります。つまり、 4×13 で計52回、ブロック内のコードが実行されるわけです。

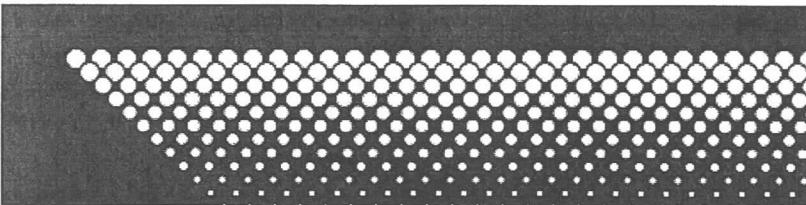
繰り返しを使って模様を描きたいときはExample 4-10を出発点にするといいでしょう。次の2つは、このコードを拡張してできることを示すための小規模な例です。Example 4-12は格子状に並んだ各点から中心へ向かう線を描いたもの。Example 4-13では円が縮みながら右下へずれていきます。y座標の値をx座標に足すことで、このようなパターンが出現します。

Example 4-12: ピンと線



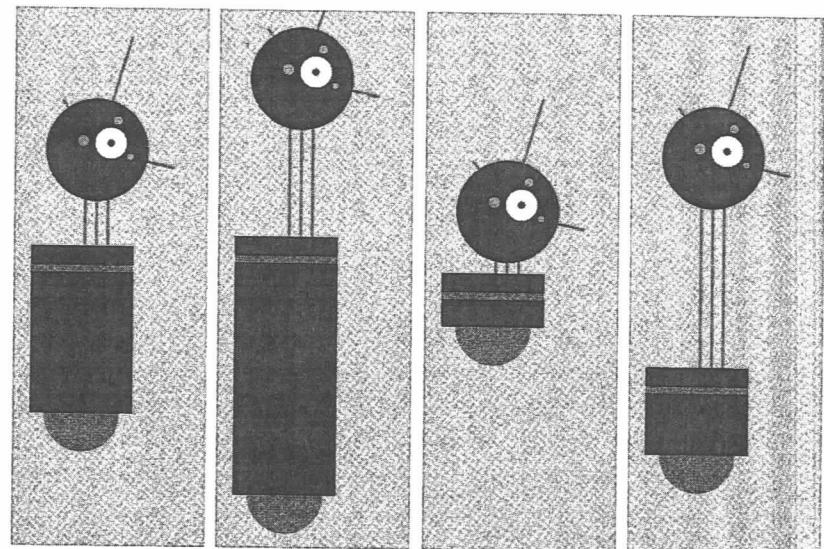
```
size(480, 120);
background(0);
fill(255);
stroke(102);
for (int y = 20; y <= height-20; y += 10) {
    for (int x = 20; x <= width-20; x += 10) {
        ellipse(x, y, 4, 4);
        // 描画領域の中心に向かう線を描く
        line(x, y, 240, 60);
    }
}
```

Example 4-13: 網点



```
size(480, 120);
background(0);
for (int y = 32; y <= height; y += 8) {
    for (int x = 12; x <= width; x += 15) {
        ellipse(x + y, y, 16 - y/10.0, 16 - y/10.0);
    }
}
```

Robot 2: Variables



変数が導入されたプログラムは Robot 1 (3 章) よりも難しく見えますが、影響しあう数値が1か所に集められているので、変更はむしろずっと簡単です。この例では、ロボットの外見を決定づける変数がプログラムの先頭部分に集められていて、位置、胴体の長さ、首の長さなどが設定できるようになっています。首の位置は胴体の長さ (bodyHeight) と首の長さ (neckHeight) で決まるというように、これらの変数は互いに影響しあっているので、まとまっているほうがわかりやすいわけです。

y = 390	y = 460	y = 310	y = 420
bodyHeight = 180	bodyHeight = 260	bodyHeight = 80	bodyHeight = 110
neckHeight = 40	neckHeight = 95	neckHeight = 10	neckHeight = 140

自分でプログラムのなかの数値を変数に置き換えるときは、少しづつ一步一歩進めるようにしましょう。変数を作るのは1個ずつ。新たに変数を作ったら、次の変数を作る前にプログラムを実行して動作を確認してください。そうして移行の複雑さを最小限にとどめることができです。

```
int x = 60;           // x座標
int y = 420;          // y座標
int bodyHeight = 110; // 胴の高さ(長さ)
int neckHeight = 140; // 首の高さ(長さ)
```

```
int radius = 45;      // 頭の半径
int ny = y - bodyHeight - neckHeight - radius; // 首のy

size(170, 480);
strokeWeight(2);
background(204);
ellipseMode(RADIUS);

// 首
stroke(102);
line(x+2, y-bodyHeight, x+2, ny);
line(x+12, y-bodyHeight, x+12, ny);
line(x+22, y-bodyHeight, x+22, ny);
// アンテナ
line(x+12, ny, x-18, ny-43);
line(x+12, ny, x+42, ny-99);
line(x+12, ny, x+78, ny+15);
// 胴体
noStroke();
fill(102);
ellipse(x, y-33, 33, 33);
fill(0);
rect(x-45, y-bodyHeight, 90, bodyHeight-33);
fill(102);
rect(x-45, y-bodyHeight+17, 90, 6);
// 頭
fill(0);
ellipse(x+12, ny, radius, radius);
fill(255);
ellipse(x+24, ny-6, 14, 14);
fill(0);
ellipse(x+24, ny-6, 3, 3);
fill(153);
ellipse(x, ny-8, 5, 5);
ellipse(x+30, ny-26, 4, 4);
ellipse(x+41, ny+6, 3, 3);
```

5

反応

Response

マウスやキーボードといったデバイスからの入力に反応するプログラムは止まらずに動き続けている必要があります。そのためには、draw()関数のなかにコードを配置します。