Processing 入門

細井博之

We believe that learning to program is not about acquiring a certain skillset, but is instead a creative and exploratory process.

—Processing Foundation

われわれは、プログラミングを学ぶこととはある一定の技能を修得することではなく、想像と探求に満ちた過程そのものであると考えている。

Processing 財団

はしがき

Processing は入門者向けのプログラミング環境であることもあり、Processing 用の入門書はすでにいくつか存在する上に、Web 上にも参考となる情報は多い。それにもかかわらず本テキストを書いたのは、大学での 1 年間の授業の内容に沿ったテキストが欲しいと考えたからである。

プログラミングの入門者にとっては、ただ単にプログラミング上のルールを暗記していくのみではなく、これらが実際にどのように用いられるのかを豊富な事例を通して会得していくことが必要であることは言うまでもない。しかし一方では、ただ体験の連続にまかせるのみならず、プログラミング上で何度も現れる共通した手法や関連する考え方を系統立てて整理しつつ理解することも必要であると考えている。

そこで、本書では第 IV 部に配置されたチュートリアル的な例題集に加えて、第 II 部では構文を、第 III 部では Processing 上で用いられる命令を掲載した。この第 II 部と第 III 部は他のプログラミング言語を学ぶことになった際にも役立つように、プログラミングをする上でのものの考え方や頭の働かせ方をできるだけ会得できるように取捨選択したつもりである。第 I 部は、これから初めて Processing に触れる方向けの内容であるから、すでに Processing でプログラムを組んだことのある方は飛ばしてくださってもいいと思う。第 II 部の第 8 章で扱われる構造体（Structure）という構文そのものは本来 Processing 上には存在しないが、その後の第 9 章のクラスへの理解をスムーズにするために、敢えて C/C++などで伝統的に用いられている名称や順序を持ち出した。第 II 部には他にもいくつか、Processing の言語仕様としての厳密さよりはこれから学ぶ人のための理解を優先して順序や体系を変更した箇所がある。また、第 9 章クラスは本来は内容が複雑な上にそれなりに大きな規模のプログラムを書く際に有益なものであるから、初学者向けのテキストでは必ずしも扱う必要はないかもしれない。しかし、例えば音声やカメラなどの外部ライブラリを使う際にはクラスの形で提供されることが多いため、これらを使うための最低限必要な範囲に留めて記述した。第 III 部では、使用頻度の高い主要なものを中心に選んだが、プログラミングに対する理解が進むに従って本テキストに書かれている以外にもど

のような命令があるかが、次第に想像できるようになってくるはずである。これらも、単純な暗記ではなく、全体の流れや法則から未知の部分を推察できるような頭の働かせ方を少しでも身に付けられるように考慮した。第 IV 部では、おおよそチュートリアル的な配置で実例を取り上げた。この実例では、すでに完成された作品例というよりも、第 II 部と第 III 部の具体例となるような「小さめの部品」を多数並べることを心がけた。これは、原子的な小さな部品のひとつひとつを理解しながら、学習者それぞれの自発的な創造力によって原点から構築する過程を優先させたいという著者の願いから生じた選択である。

また、プログラミングを学ぶ際には Web 上の検索エンジンから目的の情報に辿り着く技術も大切である。しかし、Web 上に散在する情報をばらばらに眺めていくと、情報同士の関連性や全体像を把握することが難しくなる。既に十分な経験のある方にとっては、検索によって得られる極狭い範囲の情報からでも全体像を把握することができるが、これは一般に初学者にとっては難しい。この全体像を体系的に捉えることができなければ、自分が必要な情報がどこに存在するのかを推測することも困難となる。そこで、本テキストでは一通りの情報を網羅することで、できるだけ目的地を探すための「地図」となり得るような構成を試みた。

こういった考えのある一方で、著者自身は大学での授業を始めてからまだ日が浅く、どの程度までお役に立ちうるのかについては不安もある。しかし、思いがけずプログラミングの教育に携わるご縁をいただけたことやこの授業の成立に関わって下さった他の先生方や学生にも感謝すると共に、今後も授業等での体験を通して本テキストをより充実した内容に変えていくことができればと願う次第である。

最後に、主に第 III 部についての一次資料となる公式リファレンスは英語でしか提供されていないが、Processing 財団および Processing の開発者の一人でありカリフォルニア大学教授でありまた自身もアーティストでもある Casey Reas 氏より日本語訳の許可をいただいた。ここに重ねて謝意を示したい。

2018 年 3 月

細井博之

目　次

※本書の内容は Processing3.3 を用いて検証や動作確認などを行っています。
Processing のバージョンによっては、一部内容が異なる場合等があります。

第I部　Processingの導入

第I部では、Processingに初めて触れる人に向けたダウンロードや初期設定などの方法を紹介する。

第1章　Processing とは？

Processing とは、主にメディア・アート向けのプログラミング言語である。映像や音声や各種デバイスを用いたインタラクションを容易に制作することができる。また、近年はコンピュータの高性能化や OS の複雑化により、プログラミング自体の敷居が高くなる傾向にあるが、Processing では非常にシンプルな記述や操作で実際的なプログラミングを行えるため、初学者にも最適である。また、Java をベースにしているため、Windows、MacOS 上のそれぞれで同じソースコードが動く上に、Java はもちろん C/C++を始めとした様々なプログラミング言語に移る際にも基本的な考え方を応用させることができる。

Processing 自体は他のプログラミング言語と比べるとやや簡易的なシステムであるため若干の制限はあるが、実際にアート作品を制作する際にこれらの制限がすぐさま大きな問題となることは少なく、Processing のみでも十分に多彩な作品を制作することができる。

第2章　ダウンロードとインストール

Processing は公式サイト（https://processing.org/）から無償でダウンロードすることができる。公式サイトの左側メニューの Download をクリックするとダウンロードページへ移るので、ここで自分の使っている OS と合ったものを選択してダウンロードする。Donation（寄付）を勧める画面が表示されることもあるが、寄付をしなくても無償で使用することができる。

Windows の場合は Zip ファイルを適当な場所へ解凍してから、Processing.exe を実行する。MacOS の場合は、解凍したファイルをアプリケーションフォルダへ移動させてから実行する。

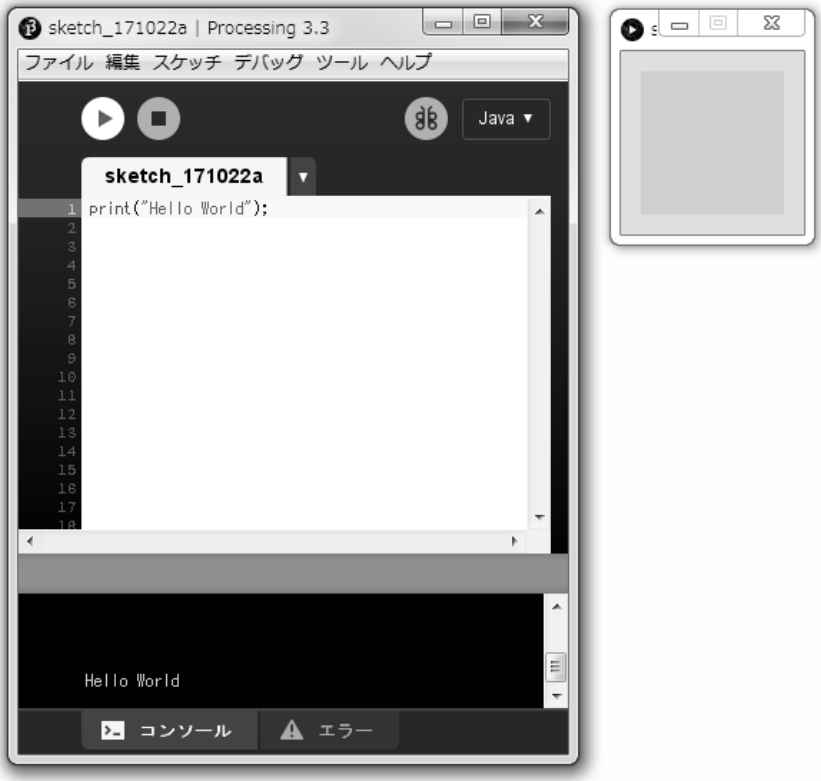
実行すると以下のようなウィンドウが表示される。

第3章　Processing の基本操作

Processing では**スケッチ**と呼ばれる単位でプログラムが表現される。起動すると、空白のソースコードを持つスケッチが自動的に生成されるので、手始めに以下のソースコードを入力してから、上部にある三角の実行ボタンを押してみよう。

```
print(”Hello World”);
```

すると、以下のように下側のコンソール内に Hello World と表示されるはずである。



このとき、小さな実行ウィンドウも自動的に生成されるが、実際の制作では様々なコードを追加して、この実行ウィンドウのサイズを大きくしたり図形を表示したりしていく。下側にあるコンソールは、いわば動作確認のための補助的な表示領域である。

また、コードに誤りがある場合は、下側にある「エラー」タブを選択すると、エラー内容が表示される。以下の例では、行の最後に必要な；（セミコロン）が欠落していることが示されている。

こういったエラーについては、その後修正されるのであればエラーを出すことによるデメリットはとくに無いから、最初のうちはエラーが出ることは気にせずに、とにかく書いては実行して…というプロセスを沢山繰り

返すことが、プログラミングの上達のためには大変に重要である。

なお、Processing は英語圏で開発され、英語圏での使用が前提となっているから、日本語を扱うには多少例外的な操作の要る場合がある。ソースコード内にある日本語を表示するには、メニューから以下の設定画面を表示させ、「エディタとコンソールのフォント」から日本語表示可能なものを選択し、「複雑なテキスト入力を有効にする」のチェックを入れる。

第II部　構文

第 II 部では、語学で言うところの云わば「文法」に相当する構文の全体像を示す。Processing では基本的には Java と同じ構文を持つ上に、C/C++を始めとした様々なプログラミング言語とも共通している。また、アセンブリや関数型言語などの一部の特殊なプログラミング言語を除けば、括弧やインデントなどの記述上の表面的なルールが多少異なるだけで、基本的な考え方は全く同じである。したがって、Processing を通して第 II 部に記されているような基本事項を覚えてしまえば、他のプログラミング言語を扱う際にも大変に役に立つ。

第1章　ウィンドウと文字表示

一般に Processing では映像やアニメーションを持つプログラムを制作するために、Processing の機能としてあらかじめ定義されている `setup()` 関数と `draw()` 関数を持つプログラムが書かれることが多い。このタイプのプログラムは active mode と呼ばれる。しかし、第 II 部では第 5 章の関数を除いてこれらの機能は敢えて使わずに、第 I 部の第 3 章で扱ったように `print()` 関数を使ってコンソールに文字列を表示して動作を確認していく。これは static mode と呼ばれ、現代のプログラミング言語からするとやや補助的なのものである。したがって、各構文の解説に現れる実行結果はすべてコンソール内に表示されるものと解釈して欲しい。active mode を使った一般的なプログラミング方法については、主に第 IV 部を参照していただきたい。

第2章　変数

§1 宣言と初期化

```
int value;
value = 15;
print(value);
```

変数を使うと、プログラムの実行中に数字などの情報を記憶させておくことができる。

変数を使うためには、使う分の変数をあらかじめ作っておく必要があり、これを変数の**宣言**と呼ぶ。宣言するには int というキーワードに続けて変数名を記す。上記の例では、1 行目で value という名前の変数を宣言している。

1 行目で作られたばかりのこの value という変数の中の数字は、自由に書き換えることができる。2 行目では、この value という変数の中身を 15 に書き換えている。これを**代入**と呼び、=（イコール）の右辺の内容が=（イコール）の左辺の変数の中に代入される。これは大変重要なルールであり、また数学のような左辺と右辺の内容が等しいことを示すのではなく、プログラミングにおいては代入という**動作が実際に行われる**ことが重要である。

3 行目では、この変数 value の内容をコンソールに表示している。2 行目で変数の内容は 15 に書き換えられているから、上記の例を実際に実行してみると、コンソールにはやはり 15 と表示される。

ところで、この宣言と最初の代入はセットで同時に行われることが大変に多いので、以下のようにまとめて記述する方法があり、一般的によく用いられる。

```
int hensu = 64;
```

また、変数の使用前にはこの**初期化**と呼ばれる最初の代入を必ず行う必要がある。プログラミング言語によっては、初期化をしない場合は自動的に0が代入されていたり、どんな数字が入っているかは実行してみるまで分からないものもあるが、Processingでは初期化を必須にすることにより、プログラミング上のミスが減るように設計されている。

§2 計算と演算子

代入時の右辺には、単純な数字以外にも様々な数式を書くことができる。

```
int foo;
foo = 1 + 2 + 3;
```

この例では、変数fooには、右辺を計算した結果の6が代入されている。加算のみではなく四則演算を用いることができるが、乗算は × ではなく*（アスタリスク）を、除算は ÷ ではなく/（スラッシュ）を使うことに注意が必要である。

また、乗算と除算が先に計算され、括弧を付けることで計算の順序を明示的に制御できる。例えば次のような例では変数barには1が代入される。

```
int bar;
bar = (2 * 2 + 1) / 5;
```

また、右辺の計算に別の変数を使うこともできる。例えば、三角形の面積は**底辺×高さ÷2**で求まるが、次のように複数の変数を同時に宣言したり、右辺の中に別の変数を使うこともできる。

```
int width, height;
width = 9;
height = 3;
int area = width * height / 2;
print(area);
```

底辺が 9、高さが 3 であるから、面積を入れるために宣言した変数 area の中身は 13.5 になっているはずである。しかしこのコードを実行すると、コンソールには 13 と表示されてしまう。実は、キーワード int によって宣言されたこれらの変数には整数しか入れることができないため、小数点以下が切り捨てられてしまったのである。これらの解決法は、また後のセクションで示す。

右辺での計算に用いる +（加算）や -（減算）、*（乗算）や /（除算）のことを**演算子**と呼ぶ。四則演算以外にもやや派生的なものもあるので、以下に基本的なものの一覧を示す。

+	加算	数字を加える
-	減算	数字を引く
*	乗算	数字を掛ける
/	除算	数字を割る
++、--	インクリメント デクリメント	1 を加える、1 を引く
+=、-= *=、/=	加算代入、減算代入、 乗算代入、除算代入	四則演算を行った結果を代入する
%	剰余	整数で除算した余りを求める

各演算子の使用例は次のようになる。

```
int x = 0;
x++;
x += 10;
x = x % 4;
print(x);
```

2 行目では変数 x に 1 が加えられる。プログラミングでは単純に 1 を加えたり引いたりすることが頻繁に行われるため、このような専用の演算子が用意されている。3 行目は x = x + 10; と書いたのと同じで、これも一

種の省略記法の演算子である。4行目では変数xを4で割った場合の余りを求めている。結果として、コンソールには3が表示される。

なお、累乗や平方根などは演算子ではなく関数として提供されているため、第III部以降で説明されている。

また、ある数を0で除算することはできない。0で除算しようとすると**そこでプログラムは停止してしまう**ため、除算を使う場合には常に注意が必要である。

§3 型

変数には、**型**と呼ばれる変数の種類が存在する。前ページでは最も標準的な型である`int`型（整数を意味する英単語integerから由来する）を使ってきたが、例えば小数を扱うことのできる`float`型の変数を使うには、次のように書く。

```
float value = 10.0 / 4.0;
print(value);
```

この例では期待通り2.5と表示される。もちろん、変数`value`がfloat型であることも必要だが、右辺の数字にも小数点が加えられていることも重要である。

例えば、この右辺の小数点を消去した次のような例を実行してみると、結果は小数点以下が切り捨てられて2と表示されてしまう。

```
float value = 10 / 4;
print(value);
```

これは、実は変数に格納される前の右辺のそれぞれの数字にも型があり、`int`型である10が`int`型である4で除算され、右辺は`int`型である2が導き出され、結果としては2が`float`型である2.0に変換されたものが左辺の変数`value`に代入されるからである。

主な変数の型には、次のようなものがある。

キーワード	名　称	情報量
int	整数型	4byte（32bit）
byte	バイト型	1byte（8bit）
long	長整数型	8byte（64bit）
float	単精度浮動小数点型	4byte（32bit）
double	倍精度浮動小数点型	8byte（64bit）
boolean	ブーリアン型	1bit
char	文字型	1byte
String	文字列型	1 文字あたり 1byte
color	カラー型	4byte（32bit）

● int 型

-2,147,483,648～2,147,483,647 の範囲で、整数のみを入れることのできる、最も基本的な型。特に理由がなければ、通常はこの型が使用されることが多い。

● byte 型

-127～128 の範囲で、整数のみを入れることができる。1byte というコンピュータの世界では最も基本となる容量を持つため、主に互換性を持たせるために存在する。int 型よりもサイズが小さいため、配列などを使って変数を大量に使用する場合には int 型よりも省メモリで高速計算のできる可能性もあるが、そういった恩恵が受けられるのはある程度限られた特殊ケースであるから、初学者が敢えてこの型を使う理由は少ないと言える。

● long 型

-9,223,372,036,854,775,808～9,223,372,036,854,775,808 の範囲で、整数のみを入れることができる。byte 型とは逆に int 型をさらに拡張した型である。int 型では収まらないような大きさの数値を扱う場合に使用する。

第III部 命令

第 III 部では、語学で言えば「単語」に相当する命令の中から基本的なものを紹介する。様々な命令を使うことで、画面に文字や図形を表示したり、音を鳴らしたり、マウスやキーボードからの入力を得たりすることが可能となる。

また、ここでは便宜上「命令」と述べたが、これらは実際には Processing 上にあらかじめ用意されている関数や変数やクラスである。

第1章　図形描画

line()　画面上に線を引く

例
```
line(10, 10, 50, 70);
```

説明
次のように引数を指定して、画面上に線を描画する。
line(開始x座標, 開始y座標, 終端x座標, 終端y座標)
stroke()やstrokeWeight()で色や太さを変更できる。3次元空間上に線を引く場合は6つの引数を指定する。

rect()　長方形を描く

例
```
rect(30, 20, 55, 55);
```

説明
次のように引数を指定して、画面上に長方形を描画する。
rect(左上のx座標, 左上のy座標, 幅, 高さ)
fill()やstroke()で輪郭や塗りつぶしの色を指定する。また、引数を追加すると角を丸めることもできる。

ellipse()　円を描く

例
```
ellipse(56, 46, 70, 15);
```

説明
次のように引数を指定して、画面上に円を描く。
ellipse(中心のx座標, 中心のy座標, 幅, 高さ)
fill()やstroke()で輪郭や塗りつぶしの色を指定する。

point()　点を描く

例
```
point(30, 20);
```

説明
次のように引数を指定して、画面上に点を描く。
point(x座標, y座標)
stroke()で色を指定する。

第2章　色の設定

background()　背景を塗りつぶす

例

```
background(128); //白黒で指定
background(255, 127, 0); //RGB で指定
```

説明 背景を塗りつぶす。通常は draw()関数内の最初で使用される。PImage クラスのオブジェクトを指定すると画像を背景にできる。

fill()　塗りつぶす色を設定する

例

```
background(128); //白黒で指定
background(255, 127, 0); //RGB で指定
```

説明 図形を塗りつぶす色を指定する。一度指定すると、それ以後に描画される図形はすべてこの色で塗りつぶされる。

noFill()　塗りつぶしの解除

例

```
noFill();
```

説明 図形を塗りつぶさないようにする。一度指定すると、それ以後に描画される図形はすべて塗りつぶされないようになる。

stroke()　線の色を設定する

例

```
stroke(128); //白黒で指定
stroke(255, 127, 0); //RGB で指定
```

説明 図形の輪郭線の色を指定する。一度指定すると、それ以後に描画される図形の輪郭線はすべてこの色となる。

第1章　ウィンドウと図形

§1 ウィンドウの生成

第 IV 部では一貫して active mode と呼ばれる Processing 上で最も一般的に用いられる方法を使う。この active mode では最低限 `setup()` 関数と `draw()` 関数が必要なので、最もシンプルな形は次のようになる。

```
void setup()
{
}

void draw()
{
}
```

このプログラムを実行すると単純に次のように 100×100 のサイズのウィンドウが生成されるのみで、他には何も起こらない。

基本的には、この最もシンプルなプログラムを基に、少しずつソースコードを書き換えてゆくのである。次の例では、`setup()` 関数の中に `size()` 関数を加えて、ウィンドウサイズを 640×480 の大きさにしている。

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

void draw()
{
}
```

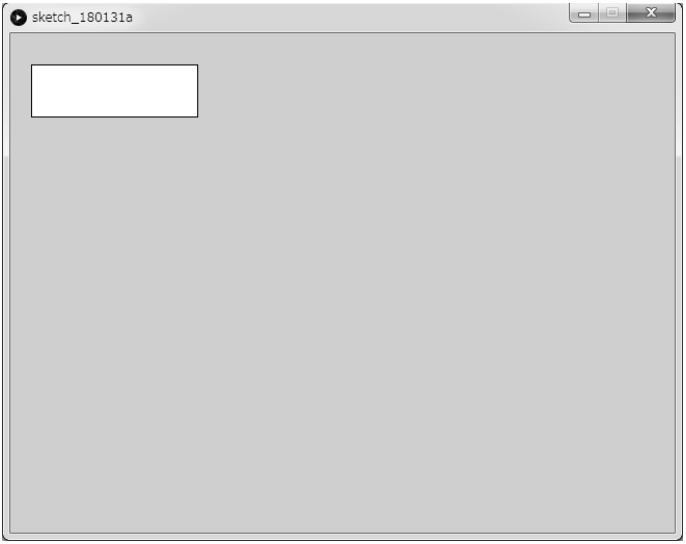
setup()関数はプログラム実行時に最初に一度だけ実行される関数で、名前の通りウィンドウサイズなど主にセットアップに関連した内容を記述する。

§2 図形の描画

図形の描画を行うには、次の例のようにdraw()関数の中にコードを追加してゆく。

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

void draw()
{
 rect(20, 30, 160, 50);
}
```

draw()関数の中に長方形を描画するrect()関数が追加されたため、このコードを実行すると次のようにウィンドウ内の左上に長方形が表示される。



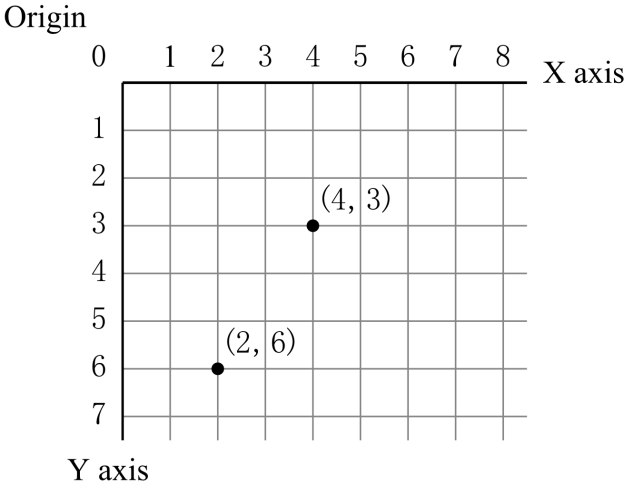
先に示した例の中では rect() 関数には次のように引数と呼ばれる値が 4 つ指定されている。

```
rect(20, 30, 160, 50);
```

実は、この数字によって長方形をどこに描くか、またどのくらいの大きさで描くかを決めているのである。rect() 関数の場合は、引数は次のように決められている。

```
rect(左端の位置, 上端の位置, 幅, 高さ)
```

では試しに、rect() 関数に渡されるこれらの値を書き換えて実行してみよう。ちなみに、rect() 関数に限らず Processing で画面上の位置を指定する場合には、以下のような座標系を用いる。



ちょうど、数学で一般的に用いられる座標系と比べて、Y 軸の方向が逆になっている。これは、コンピュータ上では左上から右下へ向かって順番に文字が表示されることに由来する。

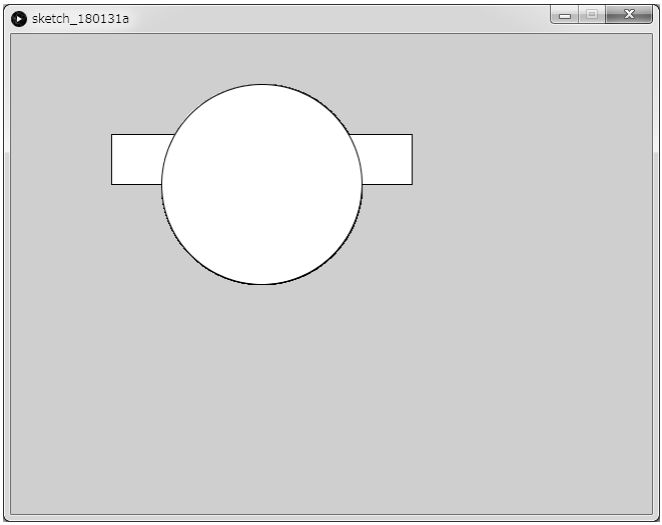
ひとまずは、何度も書き換えては実行する、というプロセスを繰り返して感覚を掴んでいくことが、始めのうちは特に大切である。

§3 様々な図形の描画

§2 では単一の図形を表示するのみだったが、ここでは複数の図形を表示してみる。次の例では、rect()に加えてellipse()を使用して円を描画している。

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

void draw()
{
 rect(100, 100, 300, 50);
 ellipse(250, 150, 200, 200);
}
```

実行すると、長方形に加えて円が表示される。

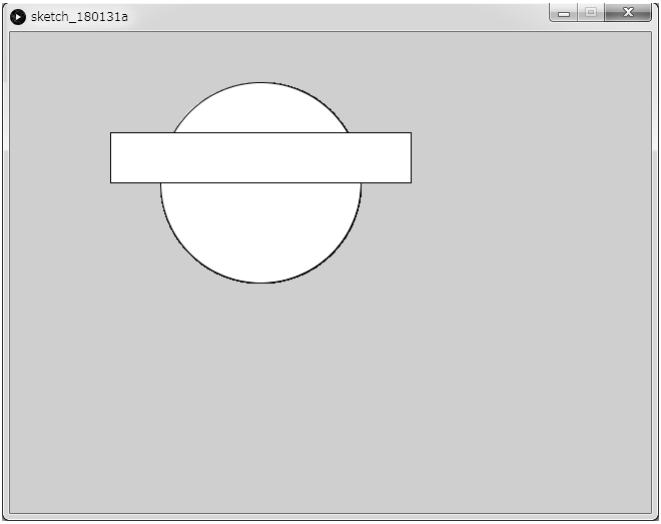


ellipse()関数は円を表示する関数で、4 つの引数はそれぞれ次のように役割が与えられている。

ellipse(中心の x 座標, 中心の y 座標, 幅, 高さ)

前の例で rect() と ellipse() の順番を入れ替えると、次のようになる。

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

void draw()
{
 ellipse(250, 150, 200, 200);
 rect(100, 100, 300, 50);
}
```



先ほどの例と比べると、draw() 関数の中の順序を入れ替えたため、長方形と円の前後関係もまた逆になっていることに注意が必要である。

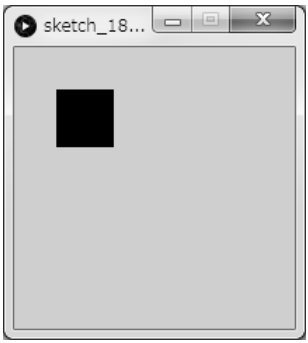
これは、プログラミングにおいては ellipse() や draw() などは単純に図形を示すというよりも、図形を**描画するという動作**を示すためである。つまり、先に ellipse() によって円が描画された後で rect() によって長方形が描画されるため、結果的には後に描画された長方形の方が手前側に表示されるのである。これは例えば推理小説を読む人がいきなり最後のページを開いて犯人を知ることはなく必ず一行ずつ順番に読むように、プログラミングにおいても特別な例外がなければ、**ソースコードは上から下へと順番に実行される**という原則があるためである。

§4 色の変更

rect()やellipse()といった関数には位置や大きさを決める引数はあるが、色を決める引数はない。実は、Processing ではあらかじめ別の関数によって色を決めておくのである。次の例では、fill()関数によって塗りつぶす色を決めている。

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

void draw()
{
 fill(0);
 rect(30, 30, 40, 40);
}
```

このように、fill()関数に引数を 1 つだけ渡すと、0 を黒、255 を白としたグレースケールで図形を塗りつぶす色を指定することができる。



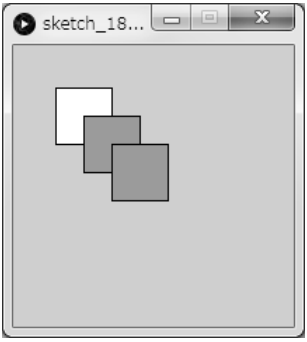
また、白黒ではなくカラーで指定したい場合は、次のように RGB（赤、緑、青）の順番で 3 つの引数を 255 段階の値で渡す。

```
fill(255, 127, 0);
```

この例では赤が強め、緑が中くらい、青がなしであるから、オレンジ色となる。

さて、ここで fill()関数の効能について、次の例を考えてみる。

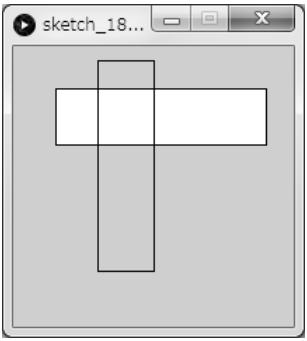
```
void setup()
{
 size(200, 200);
}

void draw()
{
 fill(255, 255, 255);
 rect(30, 30, 40, 40);

 fill(255, 127, 0);
 rect(50, 50, 40, 40);
 rect(70, 70, 40, 40);
}
```



この例を実行してみると、3つ目となる右下の四角形もまたオレンジで描画されることに注目して欲しい（ここでは灰色に印刷されているが）。実は、一度 fill()関数によって描画色を設定すると、その設定が以後残り続けるのである。言い換えると、Processing の内部には「何色で塗りつぶすのか」という情報が常に保たれていて、fill()関数によってそれを書き換えていると言える。このことに留意しておかないと、コードが複雑になったときに何故か思い通りの色で描画できない、といった迷宮に迷い込むことになるので、注意が必要である。

また、次のように noFill()を指定すると、塗りつぶしそのものを行わないという状態を指定できる。

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

void draw()
{
 fill(255, 255, 255);
 rect(30, 30, 150, 40);

 noFill();
 rect(60, 10, 40, 150);
}
```

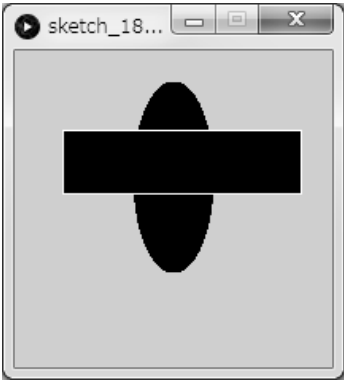


noFill()が指定された後に描画される縦長の長方形は内側が塗りつぶされないため、先に描画された横長の長方形が透けている。

また、次のようにstroke()やnoStroke()を用いると、図形の外側の輪郭線の色も同様にコントロールできる。

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

void draw()
{
 noStroke();
 ellipse(100, 80, 50, 120);

 stroke(255);
 fill(0);
 rect(30, 50, 150, 40);
}
```

長方形には白い輪郭線が与えられているのに対して、楕円形には輪郭線が描画されていない。

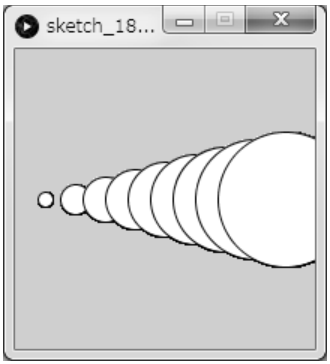


§5 大量の図形の描画

第 1 章としてはやや複雑な内容となるが、ここで図形を大量に描画する方法について軽く触れたい。§4 までの内容は説明のためのあまりにも単純な例に過ぎなかったが、次のようなコードでは多少なりともプログラミングを行う意義が見い出せそうである。

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

void draw()
{
 for (int i = 0; i < 10; i++)
 {
  ellipse(i * 20, 100, i * 10, i * 10);
 }
}
```

for は繰り返しを行う構文で、for()に続く{ }の中身が繰り返される。ここでは、ellipse()を 10 回繰り返している。また、繰り返される度に変数 i の内容も 0, 1, 2, 3… と増えていく結果、次のように円の位置や大きさを変えながら描画を繰り返すことができる。

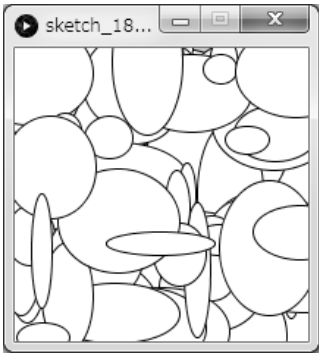


次に、乱数を使った例を示す。random()関数を使うと、まるでおみくじを引く時のように、予測のつかない何らかの値を返してくれる。

```
random(10, 50);
```

例えばこう書けば、10～50（50 は含まない）の範囲で乱数が得られる。これを用いて次のような例を実行してみる。

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

void draw()
{
 ellipse(random(0, 200), random(0, 200), random(10, 100),
random(10, 100));
}
```



ellipse()関数に渡す4つの引数すべてにrandom()で得られる乱数を渡すようにしたが、実行してみると大量の円が次々に描画される。

これは実は、元々はdraw()関数自体が1秒間に60回の割合で次々に呼び出されていたのである。今までの例では、導入として常に同じ場所に同じ図形を描画していたので、それに気付くことはなかった。

§6 作例

この第1章の締めくくりとして、draw()関数が常に呼び出され続ける性質を利用して次のような作例を作ってみた。

```
void setup()
{
 size(640, 480);
 background(255);
}

void draw()
{
 noStroke();
 for (int i = 0; i < 40; i++)
 {
  float x = random(0, 640);
  float h = pow(random(0, 1), 1.5) * 250; //中心からの高さ
  if (int(random(0, 2)) == 0) // 1/2 の確率で分岐
   h = -h; //高さを反転
  float y = 240 + h; //描画位置の y 座標
  fill(random(127, 256), random(0, 127), 0, 10 - abs(h) *
0.03);
  float r = random(3, 24); //円の半径
  ellipse(x, y, r, r);
 }
}
```

プログラムの内容としては、半透明の薄い円をひたすら重ね塗りしていくものである。まず、16 行目の `fill()` で塗りつぶす色を指定しているが、4 つ目の引数でアルファ値を指定している。これは 0 を完全な透明、255 を完全な不透明として透明度を指定するものである。また、12 行目の変数 `h` で中心からの高さを乱数によって決めているが、13 行目の `if` で 2 分の 1 の確率で分岐して符号を反転させている。これによって、描画位置を上下対照に分けている。実行すると、次のような柔らかな模様が描かれる。

§4 音の合成

前回のセクションまではあらかじめ用意されていた音声ファイルやマイクから入力された音声を利用していたが、次の例ではプログラム内部で波形そのものを計算によって作り出している。マウスをクリックすると、計算によって作り出された 440Hz のサイン波が再生される。

```
import ddf.minim.*;
import javax.sound.sampled.*;

Minim minim;
AudioSample as;

void setup()
{
 minim = new Minim(this);

 //生成する波形を格納する配列
 float[] samples = new float[1024 * 8];

 float freq = 440.0; //生成する波形の周波数

 //sine 波の波形を生成
 for (int i = 0; i < samples.length; i++) //サンプルを繰り返す
  samples[i] = sin(i * (freq / 44100.0) * TWO_PI);

 //オーディオフォーマットのデータを作成
 AudioFormat format = new AudioFormat(
  44100,  //サンプリングレート 44.1KHz
  16,     //1 サンプルあたり 16Bit
  1,      //1 チャンネル
  true,   //サンプルデータを符号有りとして解釈
  true    //サンプルデータをビッグエンディアンとして解釈
 );

 //オーディオサンプルオブジェクトを生成
 as = minim.createSample(
  samples,  //生成する波形データ
  format,   //波形フォーマット
  512       //出力時のバッファサイズ
 );
}

void draw()
{}

void mousePressed()
{
 as.trigger(); //生成された波形を再生する
}
```

12行目のsamplesという配列変数に、生成する各サンプルデータを格納しておく。この配列の長さは今回は8092なので、約0.18秒の長さを持つサンプルとなる。

14行目から18行目にかけて、sin()関数を使って440Hzのサイン波をsamplesという配列変数の中に生成している。変数freqによって生成する波形の周波数を変更できるようになっている。

20行目から27行目にかけては、これから生成するサンプルのフォーマット情報を作成している。4つ目の引数の符号と5つ目の引数のバイトオーダーに関しては、サンプルに書き込まれる時と読み出される時で同じ方法が用いられるため、通常はどの設定にしても変わりはない。

いわゆるシンセサイザと呼ばれる電子楽器は内部でこのような処理をしており、特にデジタル回路のシンセサイザにおいては考え方は全く同じと言って良い。たった1秒間の音でも44,100サンプル分の計算をする必要があるため、21世紀以降の高速なコンピュータが普及するまでは計算量の多さを克服することは大変な課題であった。20世紀後半にFM音源と呼ばれる方式が普及したのも、FM合成と呼ばれる手法は計算量の少ない割に多彩な音色を出すことができたためである。

今回の例では、最も単純な三角関数を使用しているが、これは周波数の異なる三角関数を単純に足し合わせるだけでも、音色を変化させることができる。例えば、先の例のソースコードの18行目を

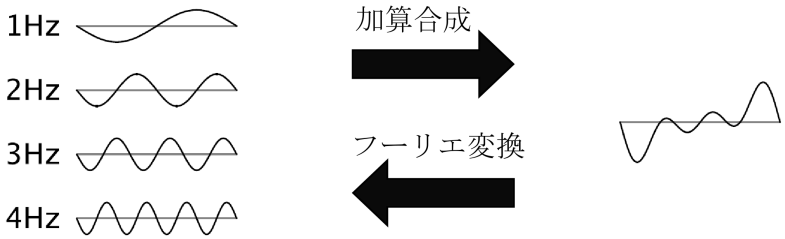
```
samples[i] = sin(i * (freq / 44100.0) * TWO_PI) + sin(i
* (freq * 2 / 44100.0) * TWO_PI);
```

と書き換えると、440Hzに加えて倍音となる880Hzのサイン波も鳴らすことができる。

このように様々な周波数のサイン波を加えるだけで様々な音色が作れるどころか、フーリエ級数という理論によれば三角関数の重ね合わせだけでこの世の中にあるどんな音でも理論上は作ることができるらしい。次のセクションでは、この理論を応用して周波数成分の解析を行うFFT（高速フーリエ変換）という有名な手法をProcessing上で扱ってみる。

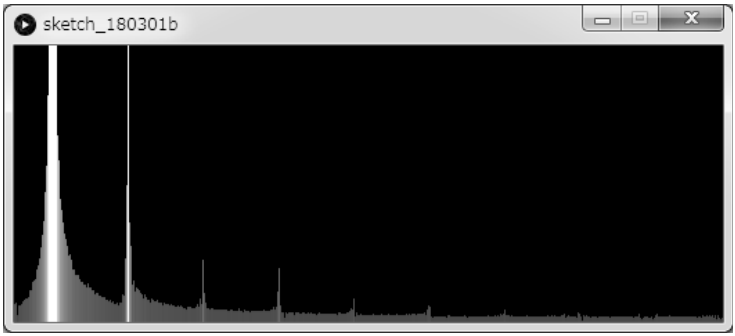
§5 FFT（高速フーリエ変換）

三角関数をたくさん重ね合わせて複雑な波形を作るのとは対象的に、既にある複雑な波形を三角関数の重ね合わせに分解する手法を**フーリエ変換**と呼ぶ。ちょうど図で表すと、三角関数の加算合成と逆方向の操作となる。



セクション3の最初の例を次のように書き換えて、高速フーリエ変換を行った結果をグラフ状に描画するようにしてみた。

```
import ddf.minim.*; //Minimライブラリを使う(インポート)する
import ddf.minim.analysis.*; //FFTのため

Minim minim; //Minimオブジェクト
AudioInput ai; //オーディオ入力オブジェクト
FFT fft; //FFT(高速フーリエ変換)オブジェクト

void setup()
{
 size(512, 200);
 minim = new Minim(this);//Minimオブジェクト生成
 ai= minim.getLineIn(); //AudioInput オブジェクトを取得
 fft = new FFT(ai.bufferSize(), ai.sampleRate()); //FFTオブジェクトを生成
}

void draw()
{
 background(0);
 stroke(255);

 fft.forward(ai.left); //FFT(高速フーリエ変換)を実行

 //スペクトラムを描画する
 for (int i = 0; i < fft.specSize(); i++) //周波数成分を繰り返す
 {
  stroke(64 + fft.getBand(i) * 4);
  line(i, height, i, height - fft.getBand(i) * 4); //成分
 }
}
```



笛の音を分析してみた様子。左側が低周波数、右側が高周波数となるが、左側から順番に見えるいくつかの柱は、倍音成分の表れである。

ソースコードとしては、Minim にあらかじめ用意されている FFT クラスを用いることで簡単に行うことができる。FFT クラスのコンストラクタの最初の引数には FFT の分析を行うバッファサイズを指定するが、これは 2 のべき乗（2, 4, 8, 16, 32…）である必要がある。2 つ目の引数にはサンプリングレートを指定する。draw()の最初の方で FFT クラスの forward()関数を実行して、フーリエ変換の処理を行う。この関数の引数には分析対象となる波形データを指定するので、今回はマイク入力となる AudioInput クラスのオブジェクトの左チャンネルの音声を指定した。分析されたデータは、FFT クラスの getBand()で得ることができる。この関数の引数に、得たい周波数を指定するが、周波数の最大値は FFT クラスの specSize()関数で調べることができる。基本的には周波数の低い側から順番に 1Hz, 2Hz, 3Hz, 4Hz…と順番に並んでいるが、一番最初の 1Hz の手前には直流成分が入っている。また、これはサンプル対象となるバッファサイズ内での周波数であるから、例えばバッファサイズが 512 でサンプリングレートが 44100Hz の場合には、分析の示す一番低い周波数は実際には約 86Hz である。

フーリエ変換自体は音声のみならず自然科学全般の非常に広範囲に応用されているが、音声の分野ではこのように主にスペクトルアナライザとして用いられる。

索　引

M

N

O

P

R

S

T

V

W

Y

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

た

ち

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

め

も

よ

ら

り

る

ろ

●著者紹介 ／ 細井博之

名古屋芸術大学音楽学部卒業、愛知県立芸術大学大学院音楽研究科修了。在学中よりセントラル愛知交響楽団、愛知室内オーケストラ、愛知県立芸術大学管弦楽団を始めソロや室内楽も含め様々な演奏家によって作品が演奏される。近年は東京・パリでも作品が演奏される一方で、名古屋ではコンサート「オセロ」や作曲家グループ「Inexplicable Owl」にも参加し作品を創出している。第 2 回 TIAA 全日本作曲家コンクール入賞、第 30 回読売中部新人演奏会出演、～新進アーティストの発見 in あいち～アーツ・チャレンジ 2010 音楽部門入選。作曲を田中範康、岩本渡、光部雅人、北爪道夫、久留智之の各氏に師事。主な作品にピアノ協奏曲、チェロとピアノのためのソナタ、ヴァイオリンとピアノのためのソナタ「春」などがある。名古屋芸術大学非常勤講師。

Processing 入門

2018 年 3 月 28 日　初版

著者　細井博之