

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。

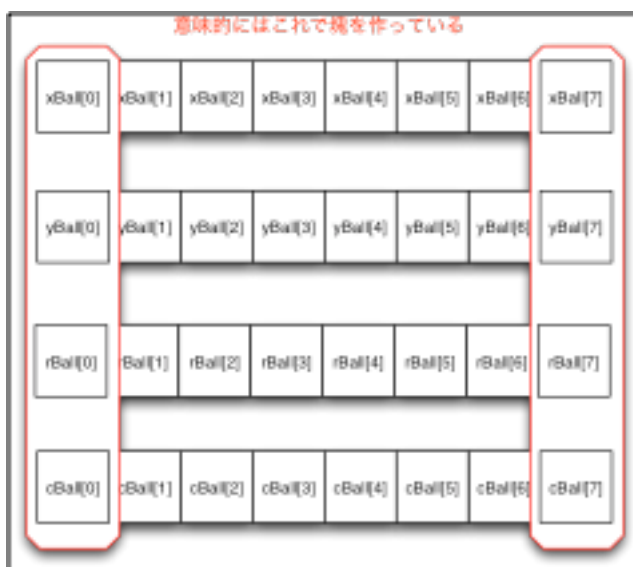
サンプル 12-2 では、配列変数 xBall, yBall, rBall, cBall 毎に塊を作っています。しかし、プログラムでの意味的には、xBall[0], yBall[0], rBall[0], cBall[0] は、1つの円の中心座標、半径、色を保存しています。添え字の



番号の値が同じものどうして組を作って、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、**簡**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
<pre>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 : : : }</pre>	<pre>class Ball { float xCenter; float yCenter; float radius; color col; }</pre>

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変数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定める鋳型（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3DS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、myBall の xCenter にアクセスするためには、myBall.xCenter などとします。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でなく、関数を持ってくことも出来ます。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
<pre>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0{ ~ } }</pre>	<pre>class Ball { float xCenter; float yCenter; float radius; color col; Ball(){ ~ } }</pre>



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
<pre>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 0{ ~ } 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){ ~ } : }</pre>	<pre>class Ball { float xCenter; float yCenter; float radius; color col; Ball(){ ~ } void update(){ ~ } void draw(){ ~ } }</pre>

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. ウィ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
    }
}
}

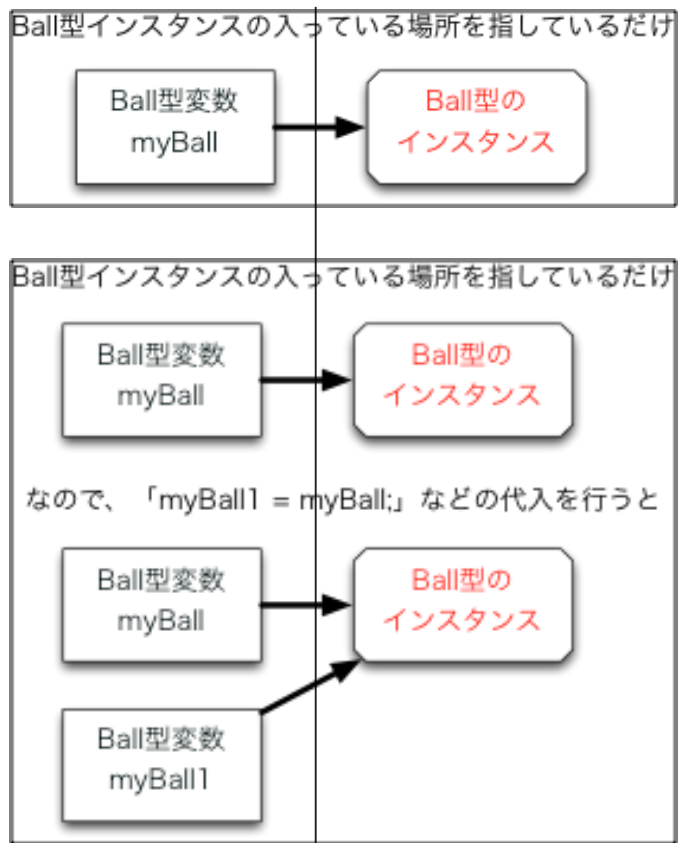
```

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
<pre> 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){ ~ } : } </pre>	<pre> class Ball { float xCenter; float yCenter; float radius; color col; Ball(){ ~ } void update(){ ~ } void draw(){ ~ } } </pre>

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、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んと思います。

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、機能拡張に関するものです。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（拡張させて、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、色々知っているとい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。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

```
import processing.opengl.*;
void setup(){
  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.*;
PeasyCam cam;

void setup(){
  size(400,400,OPENGL);
  smooth();
  cam = new PeasyCam(this, width/2, height/2, 0,400);
  cam.setMinimumDistance(50);
  cam.setMaximumDistance(500);
}
```

```

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();
}

```

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 ファイルをダウンロード（2013 年 7 月 15 日時点では、peasycam_200.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。他にも OBJ loader というライブラリがあります。これを利用すると、OBJ 形式のモデルファイルの形を表示することが出来ます。OBJ Loader の web ページは <http://code.google.com/p/saitoobjloader/> です。ここから OBJLoader のプログラムをダウンロードし、解凍します。この場合には、解凍されて出来るフォルダにはバージョン番号が付いている（OBJLoader_023）ので、このフォルダを OBJLoader に変更して、ユーザーのスケッチフォルダの中に libraries にコピーします。すると、サンプル 12-13 が実行できるようになります。

サンプルプログラム 12-13

```
import processing.opengl.*;
import saito.objloader.*;
import peasy.*;
OBJModel model;
PeasyCam cam;
void setup() {
  size(400, 400, OPENGL);
  model = new OBJModel(this);
  model.load("tachikoma.obj");
  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);
  scale(0.3);
  noStroke();
  model.enableTexture();
  model.shapeMode(QUADS);
  model.draw();
}
```