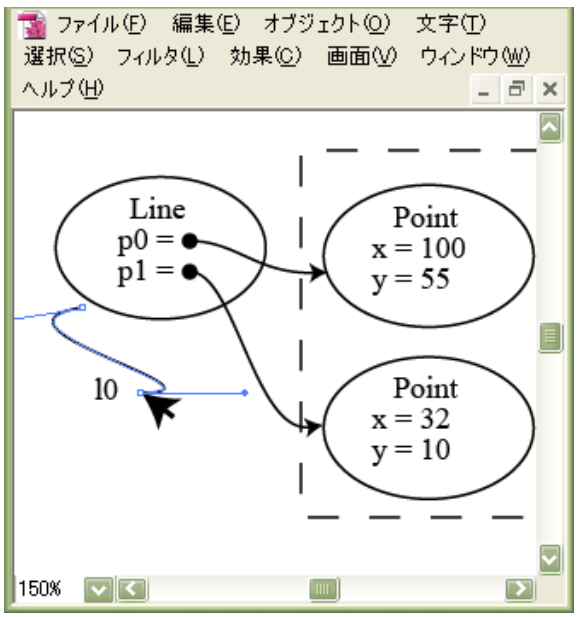
第8章　レコードとオブジェクト

これまでの章で見たプログラムが扱っていたデータは、数値、配列、文字列が中心だった。数学的な問題や文書の処理などであればこのようなデータだけでも充分な場合も多いだろう。しかし現実の情報システムは、より複雑なデータを扱わなければいけないことも多々ある。そのような場合には、**プログラム中でどのように複雑なデータを表わすか**が大きな問題となってくる。

この章では、複雑なデータを１つの値としてまとめた**レコード**とそれを発展させた**オブジェクト**を紹介する。オブジェクトはデータだけでなく、データの使い方までを１つの値にまとめたものであり、今日のソフトウェア開発でよく用いられている方法である。

8.1　複数の値をまとめるレコード

まずはどのような場合に複数の値をまとめたくなるのかを描画ソフトウェアを例に見てゆこう。描画ソフトウェアとは、マウスやペンなどを用いて線分、四角形、円、文字列などの図形要素を画面上に配置し、説明図や挿絵などを作画するものである。



このようなソフトウェアでは、線分や円などの 1 つ 1 つの図形について

- 位置
- 形状 (円であれば半径、四角形であれば幅や高さ)
- 色や模様

などの情報を記憶していて、それに基づいて表示や編集といった処理をする。画面に図を表示するときは、1 つ 1 つの図形が持っている情報を元に画面上に点を打ってゆき、利用者が図形を移動させる命令を出すと、その図形の位置情報を変更し、画面に図を表示し直すといった具合である。

このように 1 つの図形として見えるものには色々な種類の情報が含まれる。そのかたわら、「図形を削除する、コピーする」といった命令は図形を 1 つのデータとして扱う。このような場合、プログラムの中で**色々な種類の情報をまとめたデータ**を 1 つの値として扱うために、**レコード**というデータを作り操作する。

8.1.1　2 次元の点

簡単な例として、2 次元座標の点をレコードによって 1 つの値として表わそう。言うまでもなく 2 次元座標の点は、X 成分と Y 成分という 2 つの数値からなる。これを 1 つの値として表わすために、Ruby では**クラス**を定義する。

> 注意:　Ruby にはレコードを表わすための専用の手段は用意されていないので、本書では第 8.2 節以降で紹介する**オブジェクト指向プログラミング**のための機能を使ってレコードを表わしている。従って以降では、オブジェクト指向プログラミングの言葉を用いて説明をしている。

```
1 class Point
2   attr_accessor("x", "y")
3 end
```

ファイル 8.1: point.rb (一部)

この定義は「2 次元の点 (Point) には X 座標 (x) と Y 座標 (y) の情報が含まれる」ことを表わしている。この定義全体は**クラス定義**といい、x, y などの中に含まれる情報の名前を**インスタンス変数**という。

クラスが定義されると、Point.new() という式によって点を表わすデータ (**オブジェクト**という) を作ることができるようになる。例えば座標 (3,4) を表わす点を作るには、次のような命令を実行すればよい。

```
1  irb(main):004:0> load("./point.rb")
2  => true
3  irb(main):005:0> p = Point.new()
4  => #<Point:0x40326168>
5  irb(main):006:0> p.x = 3
6  => 3
7  irb(main):007:0> p.y = 4
8  => 4
9  irb(main):008:0> p
10 => #<Point:0x40326168 y=4, x=3>
```

3 行目は、Point クラスのオブジェクトを 1 つ作り、変数 p に代入している。続く行に表示されている#<Point:0x...>は作られたオブジェクトのクラス名と、値がしまわれているメモリ上の番地を表わしている。この時点でオブジェクトの中身は空である。

5 行目と 7 行目は X 座標として 3 を、Y 座標として 4 を代入している。ここで用いられている p.x という書き方は、変数 p にしまわれているオブジェクトの中の x というインスタンス変数を表わす。変数と同じように代入や参照をすることができるが、しまわれる場所がオブジェクト の中である点が異なる。9 行目のように p にしまわれているオブジェクトをもう一度表示すると、今度はオブジェクトの中に x や y の値がしまわれていることが分かる。

オブジェクトの中にしまわれているインスタンス変数を参照する場合には、次のような式を書く。

```
1  irb(main):013:0> p.y
2  => 4
3  irb(main):014:0> sqrt(p.x ** 2 + p.y ** 2)
4  => 5.0
```

これらの式は、p にしまわれているオブジェクトの中にあるインスタンス変数 x, y を参照している。

以下では、この 2 次元の点を用いて座標を計算して作図をする。そこで、作図の際に必要とな関数を定義しておこう (ファイル 8.2)。

点を作る: 座標 (u, v) を表わす点を作るには、すでに見たように

p = Point.new() ↵ p.x = u ↵ p.y = v

という 3 つの命令を実行すればよい。ただし、この操作は何度も行われるので、1 つの関数 point_make(u,v) としてまとめてしまおう。ファイル 8.2 の 4 行目からの定義がそれである。考え方は、0 で埋められた配列を作る関数 (p.38, 練習 3.2) などと同じである。

```
4  def point_make(u,v)
5    p = Point.new()
6    p.x = u
7    p.y = v
8    p
9  end
10
11 def point_scale(p,s)
12   point_make(p.x*s,p.y*s)
13 end
14
15 def point_add(p,q)
16   point_make(p.x+q.x, p.y+q.y)
17 end
18
19 def point_interpolate(p,q,t)
20   point_add(point_scale(p,1-t),
21             point_scale(q,t))
22 end
23
24 def point_draw(p,a)
25   if 0 <= p.y+0.5 && p.y+0.5 < a.   length() &&
26      0 <= p.x+0.5 && p.x+0.5 < a[0].length()
27     a[p.y+0.5][p.x+0.5]=1
28   end
29 end
```

ファイル 8.2: point.rb (続き)

スカラー倍: 座標 (x, y) を表わす点 p を s 倍した座標 (sx, sy) を表わす点を作る関数 point_scale(p,s) は 11 行目から定義されている。この関数は**新しく点を作る**。つまり p が $(3, 4)$ を表わしているときに q = point_scale(p,2) という命令を実行しても p の位置は変わらず、新しく作られた $(6, 8)$ を表わす点が q に代入される。

ベクトル和: 座標 (x, y) を表わす点 p と座標 (u, v) を表わす点 q のベクトル和である座標 $(x + u, y + v)$ を表わす点を作る関数は point_add(p,q) は 15 行目から定義されている。これも新しい点を作っていることに注意しよう。

線形補間: 2 つの点 p, q を t : $(1-t)$ に内分する点を求める関数 point_interpolate(p,q,t) は point_scale, point_add を組み合わせることで定義されている (19 行目から)。この内分点はベクトル演算では $(1-t)\mathrm{p}+t\mathrm{q}$ という式で求めることができるので、point_scale と point_add の組み合わせによって定義できる。

点を打つ: 画像を表わす 2 次元配列 a の点 p の位置に点を打つ関数 point_draw(p,a) は 24 行目から定義されている。ただし、点 p の座標に数値誤差がある場合を考えて各座標には 0.5 を足すことで四捨五入した位置に点を打つ。また p は配列の中に収まるとは限らないので、座標が配列の大きさに収まる場合にのみ点を打つ。

8.1.2　直線

レコードを使うことによって、複雑とまでははは言えないが、2 つの座標データを持つ 2 次元の点を 1 つのデータにまとめることができた。この効果は、点を使った様々な処理が単純に分かりやすく表わせることで実感できる。

そこで 2 次元の画像に直線と曲線を描く方法を見てゆこう。画像は第 2.1 節で紹介したように 2 次元配列を使って表わし、isrb 中の関数 show によって表示することにする。

まずは 2 点 $p_0=(x_0,y_0)$, $p_1=(x_1,y_1)$ の間の線分を作図する関数である。

```
1 load("./max.rb")
2 load("./abs.rb")
3
4 def line_draw(p0,p1,a)
5   n=max(abs(p1.x - p0.x), abs(p1.y - p0.y))
6   for i in 0..n
7     point_draw(point_interpolate(p0,p1,i*1.0/n), a)
8   end
9 end
```

ファイル 8.3: line.rb

定義を見る前に線分の描き方を考えておこう。まずこの線分は

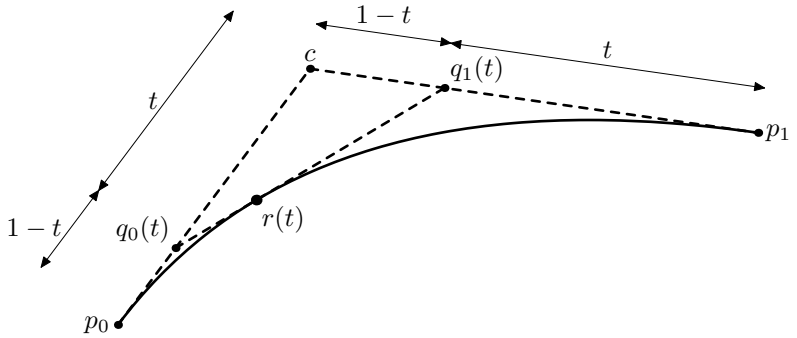
$$p(t)=(1-t)p_0+tp_1 \quad (0\le t\le 1) \tag{8.1}$$

と書くことができる。従って t の値を 0 から 1 まで Δt ずつ変化させて $p(t)$ に対応する位置に点を打てばよい。Δt は、p_0,p_1 の X,Y 座標ごとの差の絶対値の大きい方を n としたときに $\Delta t=1/n$ とすれば、$p(t)$ と $p(t+\Delta t)$ は水平方向あるいは垂直方向に 1 離れるので、点が途切れることなくつながる。

この考え方を関数としたものが line_draw である。$p(t)$ を求めるところと点を打つところは、すでに定義した point_interpolate と point_draw が使えるので、式 8.1 をほとんどそのまま関数として定義できている。これがレコードを使う効果である。

8.1.3　曲線

曲線を描く方法には色々なものがあるが、ここでは、3 点 p_0, c, p_1 が与えられたときに次の図のような曲線を描くことにする。この曲線は 2 次の Bézier 曲線と呼ばれている。



この曲線は以下の式によって定められる $r(t)$ を動かしたときの軌跡である。

$$\begin{aligned} q_0(t) &= (1-t)p_0 + tc \\ q_1(t) &= (1-t)c + tp_1 \qquad (0 \le t \le 1) \\ r(t) &= (1-t)q_0(t) + tq_1(t) \end{aligned} \tag{8.2}$$

つまり、p_0, c と c, p_1 という 2 点を $(1-t) : t$ に内分する点をそれぞれ求め、その内分点どうしの内分点が $r(t)$ である。

ファイル 8.4 に示すのが 3 点 p0, c, p1 によって定められる Bézier 曲線を 2 次元配列 a 上に描く関数 bezier_draw(p0,c,p1,a) である。この関数は曲線を n 本の線分で近似して作図している。i 番目の線分を描くときはまず $r(\mathtt{i}/\mathtt{n})$ の座標を求め、prev にしまわれている $\mathtt{i}-1$ 番目の点との間に線分を引いている。その後、prev に現在の点を代入することで次回の繰り返しにおける prev が $\mathtt{i}-1$ 番目の点になるようにしている。

練習 8.1 (連続する曲線による図) 点の列が与えられたときに、順に曲線でつないでゆくと、一筆描きの図を簡単に描くことができる。ファイル 8.5 に示した関数 kana() は、配列として与えられた XY 座標から点の配列 p を作り、p の $2i, 2i+1, 2i+2$ 番目の 3 点による Bézier 曲線を描くものである。

a) この関数を isrb の中で実行して、どのような図形が表示されるかを確認せよ。

```
1  load("./line.rb")
2  def bezier_draw(p0,c,p1,a)
3    n = 10
4    prev = p0
5    for i in 1..n
6      t = i*1.0/n
7      q0 = point_interpolate(p0, c,  t)
8      q1 = point_interpolate(c,  p1, t)
9      r  = point_interpolate(q0, q1, t)
10     line_draw(prev, r, a)
11     prev = r
12   end
13 end
14
```

ファイル 8.4: bezier.rb

b) この関数を参考にして、自分でデザインした図形を描く関数mypicture()を定義せよ。

なお、5 行目からの make_points(xy) は、X 座標と Y 座標が交互に並んだ配列から、Point オブジェクトの配列を作る関数である。

練習 8.2 (3 次曲線) 式 8.2 を展開すると $r(t)$ は t の 2 次関数であることが分かる。つまりこの式によって描かれるのは 2 次曲線 (放物線) である。ところで 3 次の Bézier 曲線は、4 点 p_0, c_0, c_1, p_1 が与えられたときに以下のような式によって定められる $s(t)$ の軌跡である。

$$q_0(t) = (1-t)p_0 + tc_0, \quad q_1(t) = (1-t)c_0 + tc_1, \quad q_2(t) = (1-t)c_1 + tp_1$$
$$r_0(t) = (1-t)q_0 + tq_1, \quad r_1(t) = (1-t)q_1 + tq_2$$
$$s(t) = (1-t)r_0 + tr_1$$

a) この式をもとに、3 次の Bézier 曲線を描く関数bezier_draw3(p0,c0,c1,p1,a)を定義せよ。

b) 次の点列を使って一続きの曲線を描く関数 kana3() を定義せよ。

$$(323, 25), (210, 553), (-1, 361), (0, 242), (-1, -71), (242, -24),$$
$$(269, 234), (307, 448), (390, 347), (399, 302)$$

(練習 8.1 の図形は 5 本の曲線から成っていたのに対して、この図形は 3 本の曲線である。) ただし先頭を 0 番目として、$3i, 3i+1, 3i+2, 3i+3$ 番目の点を使って 1 つの Bézier 曲線を描くものとする。

```
load("./point.rb")
load("./bezier.rb")
load("./make2d.rb")

def make_points(xy)
  points = Array.new(xy.length()/2)
  for i in 0 .. points.length()-1
    points[i] = point_make(xy[2*i], xy[2*i+1])
  end
  points
end

def kana()
  a=make2d(400,400)
  p=make_points([208,70, 163,210, 56,317, 0,264,
                 0,196, 115,50,  250,53, 353,67,
               390,149, 412,268, 237,347])
  for i in 0..(p.length())/2-1
    bezier_draw(p[i*2], p[i*2+1], p[i*2+2], a)
  end
  show(a)
end
```

ファイル 8.5: kana.rb

c) この関数を参考にして、自分でデザインした図形を描く関数mypicture3()を定義せよ。

8.1.4　レコードを使う意義

この章の最初で、レコードは複雑なデータを 1 つの値としてまとめるものだと説明した。関数 bezier_draw(ファイル 8.4) の定義に即して考えてみると、レコードを使うことには次のような意義があることが分かる。

1 つの点を表わすために変数が 1 つで済む。 レコードを使わない場合、X 座標、Y 座標をしまう別々の変数が必要であるが、bezier_draw では p0 や q0 のように 1 つの変数で 1 つの点を表わすことができている。

計算した点を答えとするような関数を定義できる。 例えば内分点を求める point_interpolate(p0,p1,t) はレコードによって表わされた 1 つの

点を答える関数であった。レコードを使わない場合、1 つの関数は 1 つの値を答えることしかできないので、X 座標と Y 座標についてそれぞれ内分する関数を使うことになるだろう。

ここでは簡単のために 2 次元の点を使って説明していたため X 座標と Y 座標の 2 つのデータを 1 つのデータにできた効果しか見られなかった。そのため一見、大きな違いはないと思われるかも知れない。しかし、より複雑なデータを表現する場合には XY 座標だけなく沢山の種類のデータを 1 つにまとめることになるため、レコードを使う意義が大きくなる。

練習 8.3 (レコードを使わない曲線) $p_0 = (\mathtt{x0}, \mathtt{y0}), c = (\mathtt{xc}, \mathtt{yc}), p_1 = (\mathtt{x1}, \mathtt{y1})$ によって定義される 2 次の Bézier 曲線を描く関数 `bezier_nr(a,x0,y0,xc,yc,x1,y1)` をレコードを使わずに定義せよ。ファイル 8.4 と比べて定義の分かりやすさは違うだろうか? (注意: 大きさ 2 の配列に X 座標と Y 座標をしまうことで 1 つの点を表わす方法もあるが、このようなやり方もしないものとする。これは配列によってレコードを表わしていることに相当する。)

練習 8.4 (円) 次の手順に従って円を描く関数を定義せよ。

a) 点 `p` を原点中心に角度 `theta` だけ回転させた座標を求める関数 `rotate(p,theta)` を定義せよ。ただし座標 (x, y) を原点まわりに θ だけ回転した座標は以下のように求められる。

$$(x\cos\theta - ysin\theta, x\sin\theta + y\cos\theta)$$

b) 点 `p` を中心に半径 `r` の円を、2 次元配列 `a` に描くような関数 `circle_draw(p,r,a)` を定義せよ。(ヒント: 曲線を描くときと同じように、何本かの直線で近似すればよい。例えば、N 本の直線で近似させるのであれば、i 番目の点は点 $(r, 0)$ を原点中心に $2\pi i/N$ だけ回転させ、それに `p` を加えることで決められる。)

8.2　値と操作をまとめるオブジェクト指向

現実の情報システムは沢山の種類の値を扱わなければいけないことが多い。

一方、レコードを使って複雑な値を表わす場合、その値に対する計算や操作を同時に定義しなければいけないことはすでに見た。例えば、2 次元の点であれば、点を作る、スカラー倍、ベクトル和などである。

値の種類が増えると、それぞれの値について計算や操作を定義することになる。例えば、直線、曲線、円などの図形であれば次のようなクラスや関数を定義することになるだろう。

直線クラス	曲線クラス	円クラス
直線を作る	曲線を作る	円を作る
直線を表示する	曲線を表示する	円を表示する
直線を回転する	曲線を回転する	円を回転する
直線を拡大縮小する	曲線を拡大縮小する	円を拡大縮小する
直線を平行移動する	曲線を平行移動する	円を平行移動する
	⋮	

そしてこれらの値が混在しているときには、値の種類に応じて関数を使い分けなければいけない。例えば、直線、曲線、円をそれぞれレコードを使って表わし、それらが混在する配列 f を表示する関数 drawall(f,a) を定義することを考える。図形の種類によって表示させる方法が違うので、直線であれば line_draw 曲線であれば bezier_draw を使うように定義しなければならないので、関数の定義は次のようになるだろう。

```
1  def drawall(f,a)
2    for i in 0..f.length()
3      if f[i]が直線の場合
4        line_draw(f[i]の両端点,a)
5      else
6        if f[i]が曲線の場合
7          bezier_draw(f[i]の端点と制御点,a)
8        else
9          if f[i]が円の場合
10           circle_draw(f[i]の中心と半径,a)
11         end
12       end
13     end
14   end
15 end
```

また例えば「図形を点 (x, y) を中心に角度 θ 回転させる」場合には図形の種類に関らず

図形を $(-x, -y)$ だけ平行移動して
原点中心に角度 θ 回転させ
図形を (x, y) だけ平行移動する

という操作をすればよいように思える。しかし、平行移動や回転をさせる関数は図形の種類ごとに定義されているので、drawall の場合と同様に、図形の種類で場合分けをしなければならない。

オブジェクト指向は、データに対する操作をデータに覚えさせるようなプログラムの作り方である。図形の編集で言えば、「表示」「移動」などの操作方法は各図形が覚えており、図形を扱う側は図形の種類による違いを意識する必要がなくなる。

オブジェクト指向はまた、プログラムを再利用するのに適している。ここで言う再利用とは、似た種類のデータ (オブジェクト) に対する操作をまとめて定義してしまうような方法である。例えば直線と曲線は多くの性質や操作が共通しているので、それらをまとめることができる。

8.2.1　点を表わすオブジェクト

では第 8.1 節でレコードとして表わされていた 2 次元の点をオブジェクトとして定義し直してみよう。定義全体はファイル 8.6 のような Point という名前のクラスの定義になる。レコードのときと同じくインスタンス変数 (2 行目の attr_accessor) の定義があるが、それに加えて Point クラスのオブジェクトに対する計算や操作 (4 行目以降) をまとめたものになっている。

```
1  class Point
2    attr_accessor("x", "y")
3
4    def initialize(u,v)
5      self.x = u
6      self.y = v
7    end
8
9    def scale(s)
10     Point.new(self.x * s, self.y * s)
11   end
12
13   def add(q)
14     Point.new(self.x + q.x, self.y + q.y)
15   end
16 end
```

ファイル 8.6: oo-point.rb

4 行目以降にある関数のような定義は、クラス定義の中に書かれているために、これまでの関数と少しだけ違った動きをする。本書では、このようなクラス定義中にあるものを**メソッド**と呼び、これまでのようにクラス定義の外で定義された**関数**と区別する。

4 行目から定義されている initialize というメソッドは、**初期化メソッド**とよばれるもので、オブジェクトが作られた際に自動的に呼び出されるものである。この定義があると、下に示すように Point.new(3,4) という式だけでインスタンス変数 x, y に 3,4 が代入された Point オブジェクトを作ることができる。

```
1 irb(main):005:0> p = Point.new(3,4)
2 => #<Point:0x4032167c y=4, x=3>
3 irb(main):006:0> p.y
4 => 4
```

この初期化メソッドの中では self.x に代入しているが、self とは**現在注目しているオブジェクト自身**を表わす特別な変数である。つまり Point.new(3,4) を計算するときは

1. Point クラスのオブジェクトが作られる
2. 作られたオブジェクトの初期化メソッドが引数 3,4 とともに実行される
3. 初期化メソッドの中で self は、ここで作られたオブジェクト を表わしているので、そのオブジェクトのインスタンス変数 x, y に 3, 4 が代入される

という操作が行われることになる。前節のファイル 8.2 に定義された関数 point_make の定義と見比べて欲しい。前節の定義が「点 p を作るときに p のインスタンス変数 x, y に代入する」ものだったのに対し、オブジェクト指向では「わたしが作られるときは、わたし (self) のインスタンス変数 x, y に代入する」ものになっている。このようにオブジェクト指向は**オブジェクトの視点から定義をする**方法なのである。

9 行目から定義されているメソッド scale(s) は、ある点を s だけスカラー倍した点を作る。このメソッドを使う書くには、例えば p が Point オブジェクトだったら、p.scale(2) という式になる。

配列 a の長さを求める式 a.length() もこの書き方と同じである。

```
1 irb(main):008:0> q = p.scale(2)
2 => #<Point:0x40312d98 y=8, x=6>
```

メソッド scale の定義と前節の point_scale(p,s)(ファイル 8.2 の 11 行目) を見比べてほしい。前節では「p を s 倍した点を新しく作る」定義だったものが、オブジェクト指向では「わたし (self) を s 倍した点を新しく作る」定義になっている。結果としては p を self に置き換えて、点を作るために Point.new(x,y) という式を使うようになっただけだということが分かるだろう。

2 つ点のベクトル和を求める add メソッドは 13 行目のような定義になる。Ruby のメソッドは 1 つのオブジェクトに対するものなので、このような 2 つ

のオブジェクトを用いた計算は、1 つ目のオブジェクトのメソッドの引数に 2 つ目のオブジェクトを与えるような書き方になる。結果として点 p, q のベクトル和は p.add(q) のように p に対してメソッドを呼び出すことになる。

.add を + 記号に置き換えれば「$p+q$」と読めるだろう。

```
1 irb(main):010:0> p.add(q)
2 => #<Point:0x4030f544 y=12, x=9>
3 irb(main):011:0> p.add(q).scale(0.5)
4 => #<Point:0x4030a3a0 y=6.0, x=4.5>
```

練習 8.5 (点オブジェクトのメソッド) Point クラスの定義 (ファイル 8.6) に、以下の操作を行うメソッド定義を追加せよ。

a) 点 q との差を求める sub(q)

b) 2 次配列 a 上に点を打つ draw(a)

c) 点 q と $(1-t):t$ の内分点を求める interpolate(q,t)

d) 原点中心に角度 theta だけ回転した位置を求める rotate(theta)

ただし座標 (x, y) を原点まわりに θ だけ回転した座標は以下のように求められる。

$$(x\cos\theta - ysin\theta, x\sin\theta + y\cos\theta)$$

8.2.2　図形を表わすオブジェクト

続けて直線や曲線といった図形を表わすオブジェクトをいくつか定義してみよう。まず直線オブジェクトの定義はファイル 8.7 のようになる。

ここでは簡単のために初期化メソッドのほかは次のメソッドだけが定義されている。

- draw(a) というメソッドを呼び出すと、2 次元配列 a に直線を描く
- turn(theta) というメソッドを呼び出すと、原点中心に角度 theta だけ回転した位置へ移動する

初期化メソッドと draw の定義自体は前節の line_make と line_draw をオブジェクト指向のやり方で書き直しただけなので説明は不要だろう。

メソッド turn は、新しく直線を作るのではなく、呼び出された直線オブジェクトの状態を変える。Point オブジェクトに定義されている rotate(theta) メソッドは、角度 theta だけ回転した座標の点を新しく作るので、21 行目や次の行のように p0, p1 に新しく計算した点を代入している。

練習 8.6 (円クラス) 円を表わすクラス Circle を定義し、次のようなメソッドを定義せよ。

```
load("./max.rb")
load("./abs.rb")
class Line
  attr_accessor("p0", "p1")

  def initialize(q,r)
    self.p0 = q
    self.p1 = r
  end

  def draw(a)
    n = max(abs(self.p1.x - self.p0.x),
            abs(self.p1.y - self.p0.y))
    for i in 0..n
      p = self.p0.interpolate(self.p1, i*1.0/n)
      p.draw(a)
    end
  end

  def turn(theta)
    self.p0 = self.p0.rotate(theta)
    self.p1 = self.p1.rotate(theta)
  end
end
```

ファイル 8.7: oo-line.rb

- 中心を p に、半径を r とする初期化メソッド initialize(p,r)
- 2 次配列 a に円を描く draw(a)
- 原点中心に theta 回転した位置に移動する turn

8.2.3　継承によって図形の種類を追加する

曲線は直線とよく似た性質を持っており、(2 次の Bézier 曲線であれば) 直線に制御点を 1 つ追加しただけのものと見なせる。このような場合、曲線を表わす Bezier クラスを定義する際に、Line のクラス定義を**継承**することで、重複する定義を省略することができる。

それではクラス Bezier の定義 (ファイル 8.8) を見てゆこう。

```
1  load("./oo-line.rb")
2  class Bezier < Line
3    attr_accessor("p0", "c", "p1")
4
5    def initialize(q,r,s)
6      super(q,s)
7      self.c  = r
8    end
9
10   def draw(a)
11     n = 10
12     prev = self.p0
13     for i in 1..n
14       t = i*1.0/n
15       q0 = self.p0.interpolate(c,  t)
16       q1 = self.c. interpolate(p1, t)
17       r  = q0.interpolate(q1, t)
18       Line.new(prev, r).draw(a)
19       prev = r
20     end
21   end
22
23   def turn(theta)
24     super(theta)
25     self.c = self.c.rotate(theta)
26   end
27 end
```

ファイル 8.8: oo-bezier.rb

クラスの継承　クラス定義の先頭 (2 行目) には、クラス名 Bezier に続けて< Line と書かれている。これは、ここで定義しているクラスが Line クラスの定義を継承することを意味している。この指示によって Bezier クラスは Line クラスが持っていた全てのメソッド定義をはじめから持っていることになる。

このとき Bezier クラスを**子クラス**、Line クラスを**親クラス**と呼ぶ。

メソッドの継承と上書き　Line クラスにあった initialize, draw, turn などのメソッドの定義は Bezier クラスにそのまま継承されるので、Bezier ク

ラスの中に何も定義しない場合は、Bezier クラスは Line クラスと全く同じ働きをする。

そこで draw メソッドによって曲線を表示するために、Bezier クラス独自のメソッドを 10 行目 から定義する。このように、親クラスにあるメソッドと同じ名前でメソッドを定義することを**上書き**という。

Bezier オブジェクトに対して draw メソッドを呼び出すと上書きされた方のメソッドが実行される。これによって「他は Line と同じだけど表示方法だけが違う」定義が可能になる。

ここでは初期化メソッド (initialize) とメソッド turn も同様に上書きしている。

親クラスのメソッドの実行　23 行目からの turn メソッドに注目してほしい。Line クラスの turn メソッドは 2 つの端点の位置を原点まわりに回転させるものだった。Bézier 曲線は 2 つの端点に加えて制御点があるので、回転をする場合には制御点も一緒に移動させなければいけない。

このような場合、まず Line クラスの turn を使うことで両端点を回転させ、Bezier クラスに追加された制御点を回転させることで全体を回転させることができる。ここでは 24 行目に書かれた命令 super(theta) は、親クラスに定義されている同じ名前のメソッド (turn) を実行させている。

練習 8.7 (点のまわりの回転) 点 p のまわりに図形を theta だけ回転させるメソッド turn_at(p,theta) を定義したい。点 p のまわりで回転させるには (1) 図形を −p だけ平行移動させ (2) 原点中心に回転させ、(3)p だけ平行移動すればよい。もし move(p) というメソッドで図形を p だけ平行移動できたとすると、次のようなメソッドが定義されていればよいことになる。

```
1 def turn_at(p,theta)
2   self.move(p.scale(-1)) #-pだけ平行移動
3   self.turn(theta)       #原点中心に回転
4   self.move(p)           #pだけ平行移動
5 end
```

この定義を Line, Bezier および練習 8.6 で定義した Circle の 3 つのクラスに共通して使いたいので、次のように定義せよ。

a) 図形を点 p だけ平行移動させるメソッド move(p) を Line, Bezier および練習 8.6 で定義した Circle の 3 つのクラスにそれぞれ定義せよ。

b) Figure というクラスを定義してその中に上に示したメソッド turn_at の定義を書け。

c) クラス Line と Circle が Figure の子クラスとなるようにそれぞれのクラス定義を変更せよ。これで例えばファイル 8.9 の turning_figure

の e ように 1 つのメソッド定義で異なる種類の図形を回転させることができるようになる。

```
include(Math)
def turning_figure()
  f1 = Line.new(Point.new(0,0),Point.new(99,99))
  f2 = Circle.new(Point.new(50,50),25)
  a = make2d(100,100)
  for i in 0..10
    f1.draw(a)
    f2.draw(a)
    f1.turn_at(Point.new(100,100),  PI/40)
    f2.turn_at(Point.new(100,100), -PI/40)
  end
  show(a)
end
```

ファイル 8.9: turn_at.rb

練習 8.8 (3 次曲線のクラス) 3 次の Bézier 曲線を表わすクラス Bezier3 を、Bezier を継承して定義せよ。(3 次の Bézier 曲線の描き方は練習 8.2 を参照せよ。)

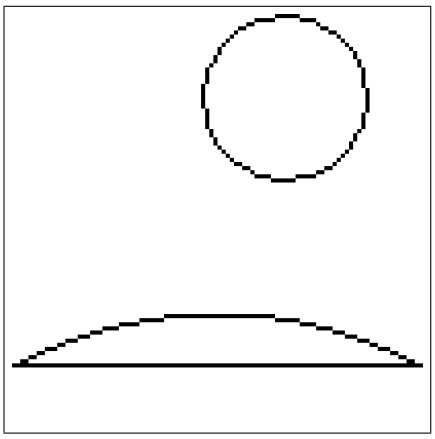
8.2.4　多相性によって異なる種類の図形を統一的に扱う

オブジェクトがどのような計算・操作されるべきかを自分で覚えていることで、色々な種類 (クラス) のオブジェクトを統一的に扱うことが可能になる。このことを**多相性**(polymorphism) という。

ファイル 8.10 に示した関数 drawmoon を見てほしい。この関数は、直線 (Line)、曲線 (Bezier)、円 (Circle) の 3 種類のオブジェクトが入った配列 f を作り、次に示す関数 drawall(ファイル 8.11) を使って表示している。

```
1  def drawmoon()
2    p0=Point.new(0,85)
3    p1=Point.new(99,85)
4    f=[Line.new(p0,p1),
5       Bezier.new(p0,Point.new(50,60),p1),
6       Circle.new(Point.new(66,20),20)]
7    a=make2d(100,100)
8    drawall(f,a)
9    show(a)
10 end
```

ファイル 8.10: drawall.rb (一部)



```
11 def drawall(elements,a)
12   for i in 0..elements.length()-1
13     elements[i].draw(a)
14   end
15   a
16 end
```

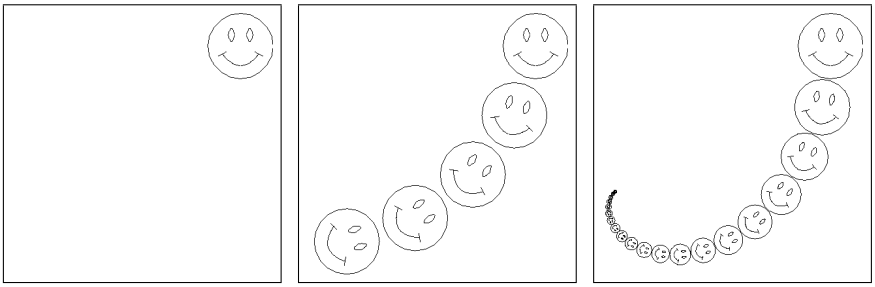
ファイル 8.11: drawall.rb (続き)

関数 drawall(f,a) の定義を見ると、配列の内容 1 つ 1 つについて draw メソッドを呼び出している (13 行目) だけである。つまり、配列の中にある値が直線であれ何であれ、そのオブジェクト の draw メソッドを使えば適切に表示されることになっているので、drawall を定義するときはこれで上手くゆく。これが多相性によって可能になったことである。

練習 8.9 (作図) ファイル 8.10 や配布プログラム `oo-face.rb`(後述) を参考に、適当な絵を表わす配列を返す関数 `sketch` を定義せよ。正しく定義されたものは `drawall` を使って表示させることができるはずである。

練習 8.10 (図形の回転) 配列 `f` にしまわれた全ての図形をそれぞれ原点のまわりに `theta` だけ回転させる `turnall(f,theta)` を定義せよ。

練習 8.11 (図形の変形) 配布プログラム `oo-face.rb` に定義されている関数 `face` は、円、曲線、直線がしまわれた配列を作る。その配列を `drawall` を使って 400×400 の画像に表示すると以下の左の図のようになる。



関数 `face` によって作られる図形を変形して、上の中央と右のような図を以下の手順に従って作成せよ。

a) 図形を原点を中心として `s` 倍に拡大・縮小させるメソッド `zoom(s)` ををクラス `Line`, `Circle`, `Bezier` に定義せよ。ただし拡大・縮小の中心は原点だとする。

b) ファイル 8.12 に定義された関数 `faces`, `whirl` を使うとそれぞれ上の中央と右の図が表示されるはずである。確めよ。

c) ここに示したものを参考に、回転・移転・拡大縮小を使って面白いパターンを描く関数 `art()` を定義せよ。

8.3　定義のまとめ

クラスの定義: クラス定義は次のような形をしている。

```
class 名前1 < 名前2
  attr_accessor("変数名1","変数名2",...)
  メソッド定義1
  メソッド定義2
  ⋮
end
```

```
include(Math)
def faces()
  a = make2d(400,400)
  elements = face()
  for i in 1..5
    drawall(elements, a)
    turnall(elements, PI/10)
  end
  show(a)
end

def whirl()
  a = make2d(400,400)
  elements = face()
  for i in 1..20
    for j in 0..elements.length()-1
      elements[j].draw(a)
      elements[j].turn(PI/15)
      elements[j].zoom(0.85)
      elements[j].move(Point.new(55,40))
    end
  end
  show(a)
end
```

ファイル 8.12: oo-whirl.rb

名前$_1$ は新しく定義されるクラスの名前である。名前$_2$ はすでであるクラスの名前で**親クラス**と呼ばれる。「< 名前$_2$」の部分は省略することができる。

インスタンス変数: 1 つ 1 つのオブジェクトの中にある変数をインスタンス変数 という。クラス定義に

attr_accessor("変数名$_1$","変数名$_2$",...)

と書くとそのクラスのオブジェクトは変数名$_i$ という名前のインスタンス変数 を持つようになる。attr_accessor に書く変数名は " " で囲まなければいけないことに注意せよ。

初期化メソッド: あるクラスのオブジェクトが作られるときに呼び出されるメソッドを初期化メソッドという。Ruby では `initialize` という名前のメソッドを定義すると、それが初期化メソッドとなる。

自分自身を表わす変数 `self`: メソッド定義の中では、現在そのメソッドを実行しているオブジェクト自身を `self` という名前の変数で表わす。

親クラスのメソッドを呼び出す `super`: メソッド中で `super(`式$_1$`,`式$_2$`,...)` という式が計算されると、現在実行中のメソッドと同じ名前の親クラスに定義されたメソッドが実行される。

オブジェクトの作成: あるクラスのオブジェクトを作る式は

クラス名`.new(`式$_1$`,`式$_2$`,...)`

という形をしている。この式が計算されると、クラス名のオブジェクト が 1 つでき、そのオブジェクトの `initialize` メソッドが自動的に実行される。

インスタンス変数の読み書き: 式`.`変数名という式は、式が表わすオブジェクトの変数名というインスタンス変数を参照する。

式$_1$`.`変数名`=`式$_2$という命令は、式$_1$が表わすオブジェクトの変数名というインスタンス変数に式$_2$の値を代入する。

メソッドの実行: 式$_0$`.`メソッド名`(`式$_1$`,`式$_2$`,...)` という式は、式$_0$が表わすオブジェクトのメソッドを呼び出す。式$_0$が表わすオブジェクトのクラスにメソッド名を持つメソッドが定義されている場合は、そのメソッドが実行される。定義されていない場合は、親クラスを順に辿ってゆき最初に見つけたメソッド名を持つメソッド を実行する。