第8章　レコードとオブジェクト　補足

8.1　レコード

値や処理をまとめることで，プログラムを簡潔に記述できるようになる．

レコード： 複数の値をまとめたデータ

2次元の点を表すレコード Point は，次のようなプログラムになる

```
# Point レコード ( point.rb )
class Point
  attr_accessor("x", "y")
end
```

Point の情報は p.x や p.y という記法でアクセスできる．また Point に対する処理（たとえば描画など）を定義できる．

```
# Point の処理 ( point.rb )
def point_make(u,v)
  p = Point.new()
  p.x = u
  p.y = v
  p
end
def point_scale(p,s)
  point_make(p.x*s,p.y*s)
end
def point_add(p,q)
  point_make(p.x+q.x, p.y+q.y)
end
def point_interpolate(p,q,t)
  point_add(point_scale(p,1-t),
            point_scale(q,t))
end
def point_draw(p,a)
  if 0 <= p.y+0.5 && p.y+0.5 < a.   length() &&
     0 <= p.x+0.5 && p.x+0.5 < a[0].length()
    a[p.y+0.5][p.x+0.5]=1
  end
```

```
end
```

線形補間： 2つのデータを1次式で補間すること

2点 p_0, p_1 を線形補間することで，2点を結んだ直線上の点を計算できる．

$$p(t) = (1-t)p_0 + tp_1 \qquad (0 \le t \le 1)$$

直線の描画は，与えられた2つの点の間の点を，すべて線形補間で計算し，その点の描画を繰り返すことで実現する．

```
# 直線の描画 ( line.rb )
load("max.rb")
load("abs.rb")
def line_draw(p0,p1,a)
  n=max(abs(p1.x - p0.x), abs(p1.y - p0.y))
  for i in 0..n
    point_draw(point_interpolate(p0,p1,i*1.0/n), a)
  end
end
```

ベジエ (Bézier) 曲線： 線形補間の反復で生成される曲線

2次のベジエ曲線は，図8.1のように3点 p_0, c, p_1 の間で線形補間を2段階 (合計3回) 行うことで得られる点によって構成される．結果として，$r(t)$ は3点 p_0, c, p_1 と t に関する2次式で表現されることから，2次のベジエ曲線と呼ばれる．

$$\begin{aligned} q_0(t) &= (1-t)p_0 + tc \\ q_1(t) &= (1-t)c + tp_1 \\ r(t) &= (1-t)q_0(t) + tq_1(t) = (1-t)^2 p_0 + 2(1-t)tc + t^2 p_1 \end{aligned}$$

ベジエ曲線の描画は，曲線上の点列を求め，となり合った2点を結ぶ線分列の描画によって実現される．

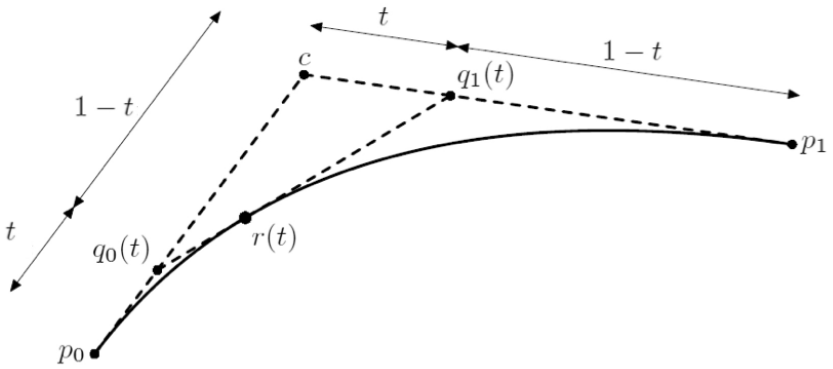

図 8.1: 2次のベジエ曲線.

```
# ベジエ曲線の描画 ( bezier.rb )
load("line.rb")

def bezier_draw(p0,c,p1,a)
  n = 10
  prev = p0
  for i in 1..n
    t = i*1.0/n
    q0 = point_interpolate(p0, c,  t)
    q1 = point_interpolate(c,  p1, t)
    r  = point_interpolate(q0, q1, t)
    line_draw(prev, r, a)
    prev = r
  end
end
```

練習： ベジエ曲線の描画

教科書 p.169 kana.rb の kana メソッドを実行してみよ，

8.2 オブジェクトとカプセル化

オブジェクト： 複数の値とともに操作 (処理) もまとめたデータ

2 次元の点を表すとともに操作も組み込んだオブジェクト Point は，次のようなプログラムとなり，class Point 〜 end の間に，メソッド（関数）が定義されている．

```
# Point クラス (oo-point.rb)
include(Math)

class Point
  attr_accessor("x", "y")
  def initialize(u,v)
    self.x = u
    self.y = v
  end
  def scale(s)
    Point.new(self.x * s, self.y * s)
  end
  def add(q)
    Point.new(self.x + q.x, self.y + q.y)
  end
end
```

メソッド： オブジェクトに対する操作

「オブジェクト.メソッド (引数)」という使い方をする．たとえば，上記の Point オブジェクトの場合には，p.add(q) のような計算 (表記) が可能となる（教科書 p.175 の実行例参照）．

カプセル化： オブジェクトの内部を意識させない処理

p.add(q) という表記では，点 (ベクトル) の加算を指定しているものの，内部で x, y 座標を持っていることは意識しないで良い．もしも p，q が 3 次元空間内の点であれば，x, y 以外に z 座標も必要となるが，p.add(q) という表記には変わりがない．

練習： Point クラスのメソッド定義

教科書 p.176 の**練習 8.5** b, c のメソッド draw(a)，interpolate(q,t) を定義し，次の line_draw.rb の line_draw メソッド実行してみよ．

```
isrb(main):001:0> load("line_draw.rb")
true
isrb(main):002:0> load("make2d.rb")
true
isrb(main):003:0> a=make2d(200,200)
 :（省略）
isrb(main):004:0> show(a)
 :（省略）
isrb(main):005:0> line_draw(Point.new(50,50),Point.new(100,170),a)
0..120
isrb(main):006:0> line_draw(Point.new(40,150),Point.new(180,70),a)
0..140
isrb(main):007:0>
```

線分の描画： 2 点を結ぶ線分の描画

与えられた 2 点間の画素を隙間なく描画する（教科書 p.166 ファイル 8.3, 教科書 p.177 ファイル 8.7）．

```
# 線分の描画（line_draw.rb）
load("abs.rb")
load("max.rb")
load("oo-point.rb")

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

8.3 オブジェクトの継承

オブジェクト間の継承関係を使うことで，性質や処理を再利用（共有）できる．

描画図形： (抽象的な) 描画図形を定義する

線画図形の場合，自分自身を構成する点列（点の配列）があれば，点列を結んで画を描くことができる．points メソッドによって自らの点列が得られるものとする．

```
# 描画図形の定義 (oo-figure.rb)
load("oo-point.rb")
load("line_draw.rb")
# load("line_intersect.rb")

class Figure
  def draw(a)
    pnts = self.points()
    for i in 0..(pnts.length()-2)
      line_draw(pnts[i], pnts[i+1], a)
    end
  end
end
```

継承： オブジェクト (クラス) が他のオブジェクトの性質を引き継ぐこと

以下の例では，Figure クラスをを継承した Line クラスを定義している．

線分の定義： 線分は両端点からなる図形

線分の場合，points メソッドは両端点を配列にして返す．Figure クラスを継承しているので，Figure クラスの draw メソッドで描画できる．

```
# 線分の定義 (oo-line.rb)
load("oo-figure.rb")

class Line < Figure
  attr_accessor("p0", "p1")
  def initialize(q,r)
    self.p0 = q
    self.p1 = r
  end
  def points()
    [self.p0, self.p1]
  end
end
```

練習： Line オブジェクトの draw メソッド

drawall.rb の drawline メソッド実行してみよ．

ベジエ曲線の定義： 2次のベジエ曲線

ベジエ曲線の場合，points メソッドは曲線上の点 $n+1$ 個を配列にして返す．Figure クラスを継承するので，draw メソッドが使える．プログラムでは $n=16$ としている．

```
# ベジエ曲線の定義 (oo-bezier.rb)
load("oo-line.rb")

class Bezier < Line
  attr_accessor("p0", "c", "p1")
  def initialize(q,r,s)
    super(q,s)
    self.c  = r
  end
  def points()
    n = 16
    pnts = Array.new(n+1)
    pnts[0] = self.p0
    pnts[n] = self.p1
    for i in 1..(n-1)
      t = i*1.0/n
      q0 = self.p0.interpolate(c,  t)
      q1 = self.c. interpolate(p1, t)
      pnts[i] = q0.interpolate(q1, t)
    end
    pnts
  end
end
```

練習： Bezier オブジェクトの draw メソッド

drawall.rb の drawground メソッドを実行してみよ．

円の定義： 正 n 角形で近似した円

円の場合 points メソッドは正 n 角形の頂点 $n+1$ 個を配列にして返す（0番目と n 番目は同一の頂点にすることで閉じた図形になる）．Figure クラスを継承するので，draw メソッドで描画できる．プログラムでは $n=32$ としている．なお，このプログラムでは図8.2のように oo-point.rb の rotate メソッドの利用を仮定しているので，教科書 p.176 の**練習 8.5** d を完成させる必要がある．

```
# 円の定義 (oo-circle.rb)
load("oo-figure.rb")
include(Math)

class Circle < Figure
```

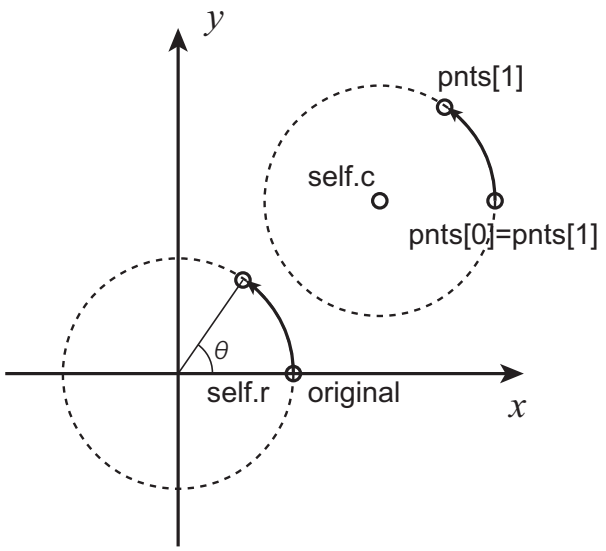

図 8.2: 円 (正多角形) の頂点列.

```
  attr_accessor("c", "r")
  def initialize(q,r)
    self.c = q
    self.r = r
  end
  def points()
    n = 32
    pnts = Array.new(n+1)
    original = Point.new(self.r, 0)
    pnts[0] = original.add(self.c)
    pnts[n] = pnts[0]
    theta = 2 * PI / n
    for i in 1..(n-1)
      pnts[i] = # ここは自分で考えること
    end
    pnts
  end
end
```

8.4 オブジェクトの多相性

複数オブジェクトが同一のオブジェクトを継承することで，共通の性質を持ち，統一的に扱うことが可能となる.

多相性： オブジェクトの変容 (変化)

同じ名前のメソッドを持つことにより，共通の性質 (メソッド) を持ちつつ，異なる振舞をするオブジェクト群を統一的に処理できる．特に，同一オブジェクトを継承することで，共通の性質を持ちつつ，微妙に差のあるオブジェクト群を定義できる.

以下のプログラムでは，線分 Line ，ベジエ曲線 Bezier ，円 Circle はいずれも線画図形 Figure であり，Figure オブジェクトの配列から取り出した図形を draw メソッドで描画すると，それぞれの性質に応じて描画される．実際には，各オブジェクトの振舞が変わるのは points メソッドの差である．その意味では，この例の場合，points メソッドが多相性の中心的な役割をなしている．

```
# 図形の描画 (drawall.rb)
load("make2d.rb")
load("oo-line.rb")
load("oo-bezier.rb")
load("oo-circle.rb")

def drawmoon()
  p0=Point.new(0,85)
  p1=Point.new(99,85)
  f=[Line.new(p0,p1),
     Bezier.new(p0,Point.new(50,60),p1),
     Circle.new(Point.new(66,20),20)]
  a=make2d(100,100)
  drawall(f,a)
  show(a)
end
def drawall(elements,a)
  for i in 0..elements.length()-1
    elements[i].draw(a)
  end
end
```

仮に多相性がなければ，drawall は次のように書かなくてはならない．

```
def drawall(elements,a)
  for i in 0..elements.length()-1
    if elements[i].type == "Line"
      line_draw(elements[i],a)
    else
      if elements[i].type == "Circle"
        circle_draw(elements[i],a)
      else
        if elements[i].type == "Bezier"
          bezier_draw(elements[i],a)
        end
      end
    end
```

```
  end
end
```

8.5 オブジェクト指向によるベクトルの計算

オブジェクト指向ではカプセル化により，内部を見せずに操作が可能となる．たとえば，点をベクトルの一種と考えて，ベクトル同士の演算を定義すると，様々な計算をすっきりと書くことが可能となる．この際，ベクトルの次元 (x, y, z などの座標値) を意識しなくて良い．

加減算： ベクトル間の加減算

すでに教科書のなかで，add や scale が定義されているが，sub は以下のようになる．

```
# Point クラスの定義内，以下 normal() まで同様 （oo-point.rb）
  def sub(q)
    Point.new(self.x - q.x, self.y - q.y)
  end
```

内積： ベクトル間の内積

```
  def dotProduct(v)
    self.x*v.x + self.y*v.y
  end
```

正規化： ベクトルの正規化

内積を使うことで比較的簡潔に書ける

```
  def normalize()
    self.scale(1.0/sqrt(self.dotProduct(self)))
  end
```

直交ベクトル： 直交するベクトルを作る

時計回りに $\pi/2$ だけ回転した単位ベクトルを作る

```
  def normal()
    Point.new(self.y, -self.x).normalize()
  end
```

直線の交点： 2 直線 (線分) の交点計算

図 8.3 のように 2 つの直線 (線分) が交差するには，一方の直線の両端点が他方の直線の左右に存在しなければならない．直線に直交するベクトル (＝直線の法線ベクトル) と直線上の任意の点から，与えられた点が直線のどちら側にあるか，またその距離を計算できる．

```
# 直線の交点計算（line_intersect.rb）
load("oo-point.rb")

def line_intersect(p0,p1,q0,q1)
  pNormal = p1.sub(p0).normal()
  qStart  = pNormal.dotProduct(q0.sub(p0))
  qEnd    = pNormal.dotProduct(q1.sub(p0))
  qNormal = # ここは自分で考えること
  pStart  = # ここは自分で考えること
  pEnd    = # ここは自分で考えること
  if (qStart * qEnd > 0) || (pStart * pEnd > 0)
    nil
  else
    p0.interpolate(p1, ((pStart * 1.0) / (pStart - pEnd)))
  end
end
```

8.6　オブジェクトの継承と多相性（再）

オブジェクト指向の継承と多相性を利用することで，性質（値や処理）を統一的に扱うことができる．

線画図形の交点： 線画図形間の交点計算

線画図形を線分の列（点列）とみなすことで，線分同士の交点計算に帰着できる．

```
# Figure クラスの定義内（oo-figure.rb）
load("line_intersect.rb")

  def intersect(other)
    spnts = self.points()
    opnts = other.points()
    results = Array.new()
    for i in 0..(spnts.length()-2)
      for j in 0..(opnts.length()-2)
```

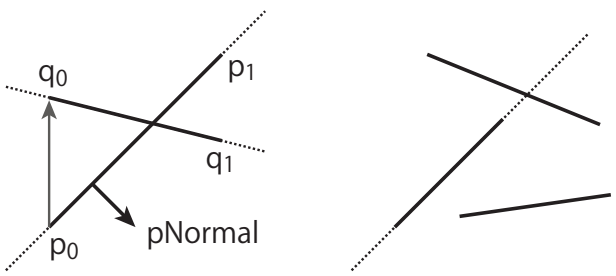

図 8.3: 直線 (線分) の交点．

```
        point =
          line_intersect(spnts[i],spnts[i+1],opnts[j],opnts[j+1])
        if point != nil
          results[results.length()] = point
        end
      end
    end
    results
  end
```

多相性： オブジェクトの変容 (変化)

同じ名前のメソッドを持つことにより，共通の性質 (メソッド) を持ちつつ，異なる振舞をするオブジェクト群を統一的に処理できる．

以下のプログラムでは，線分 Line ，ベジエ曲線 Bezier がともに線画図形 Figure であり，Figure オブジェクト配列内の図形間で交点計算を統一的に処理できる．

```
# 交点の表示 (drawintersect.rb)
load("make2d.rb")
load("oo-line.rb")
load("oo-circle.rb")
load("oo-bezier.rb")

def drawintersect()
  a = make2d(400, 400)
  f = [Line.new(Point.new(50, 50), Point.new(300, 350)),
       Circle.new(Point.new(199, 199), 70),
       Bezier.new(Point.new(50, 200), Point.new(150, 50),
                  Point.new(350, 300)),
       Bezier.new(Point.new(250, 50), Point.new(100, 150),
                  Point.new(200, 350))]
  for i in 0..(f.length()-1)
    f[i].draw(a)
    for j in (i+1)..(f.length()-1)
      results = f[i].intersect(f[j])
      for k in 0..(results.length()-1)
        Circle.new(results[k], 4).draw(a)
      end
    end
  end
  show(a)
end
```

練習： 線画図形間の交点計算

line_intersect.rb の line_intersect メソッドを完成せよ．さらに drawintersect.rb の drawintersect メソッドを実行せよ．