情報科学第10週

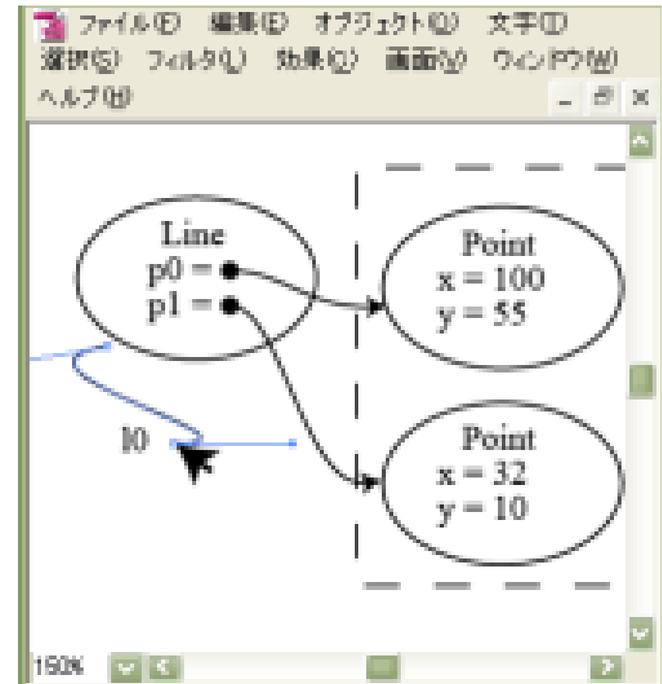
# レコードとオブジェクト

# 扱うデータの種類

- これまで扱ってきたデータ
  - 数値
  - 数値の配列
  - 文字列
- 科学技術計算ではこれで充分なこともある
  - 多くの数値モデルは行列 → 配列で表現
  - 遺伝子情報 → 文字列で表現

# 複雑なデータ

- もっと複雑なデータを扱うソフトウェアは沢山ある
- 例: お絵描きソフト
  - 円、曲線、直線、文字列、四角形などの図形を配置、変形、表示
  - 図形の形状(位置や大きさ)、属性(色や模様)などを処理しなければいけない

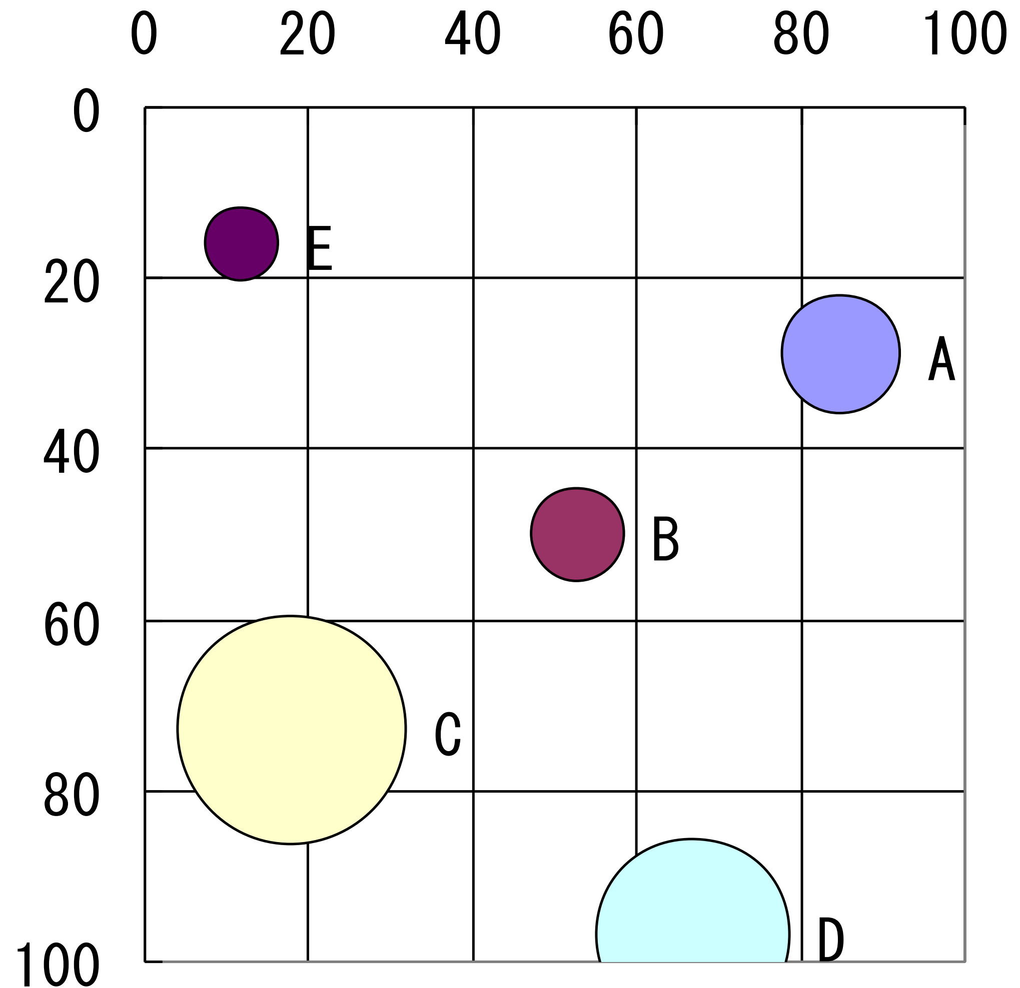


# 複雑なデータの表わし方(1)
# 例:「円」を扱うプログラム

- どう表わすか?

| X | Y | 半径 | 名前 |
|---|---|---|---|
| 85 | 29 | 26 | A |
| 53 | 50 | 17 | B |
| 18 | 73 | 94 | C |
| 67 | 97 | 67 | D |
| 12 | 16 | 11 | E |
| x | y | r | name |

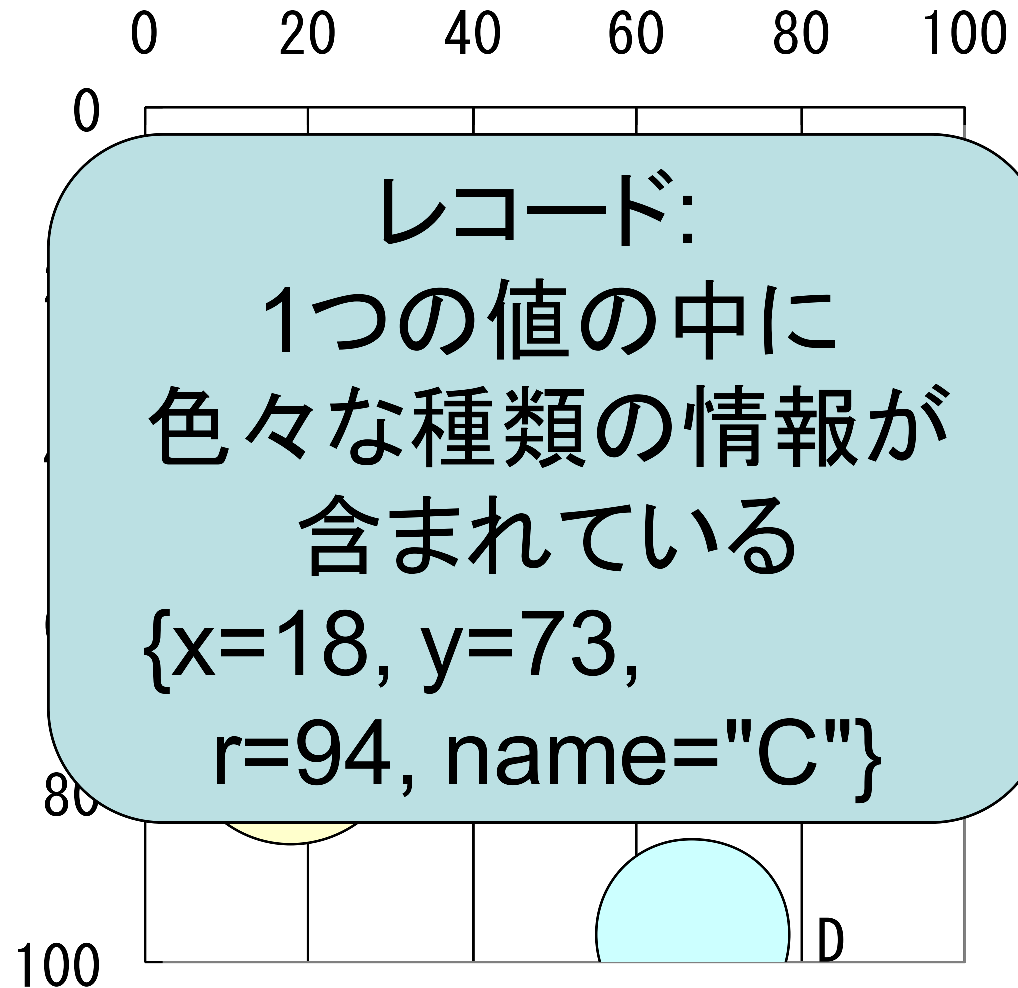
1. それぞれを配列にする



# 複雑なデータの表わし方(2)
# 例:「円」を扱うプログラム

- どう表わすか?

| X | Y | 半径 | 名前 |
|---|---|---|---|
| 85 | 29 | 26 | A |
| 53 | 50 | 17 | B |
| 18 | 73 | 94 | C |
| 67 | 97 | 67 | D |
| 12 | 16 | 11 | E |




2. 横方向に見る

レコード

# 複雑なデータの例

<u>お絵描きソフトウェア</u>

- 点: X座標、Y座標
- 円: 中心点、半径
- 線分: 始点、終点
- 色: 赤、緑、青成分

<u>数学</u>

- 複素数: 実部、虚部
- 分数: 分母、分子

<u>カレンダー</u>

- 日付: 年、月、日
- 時刻: 時、分、秒

<u>住所録</u>

- 住所: 郵便番号、都道府県、市町村、町名、番地
- 連絡先: 氏名、住所、生年月日、メールアドレス

# レコード

```
class Point
  attr_accessor("x", "y")
end
…
```

インスタンス変数の宣言

配布プログラムにあるのでダウンロードしておこう

point.rb

```
irb(main):004:0> load("./point .rb")
=> true
irb(main):005:0> p = Point.new()
=> #<Point:0x40332080>
irb(main):006:0> p.x = 3
=> 3
irb(main):007:0> p.y = 4
=> 4
irb(main):008:0> p
=> #<Point:0 x40332080 @y=4, @x=3>
```

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

インスタンス変数の参照

- Rubyでは，レコードもオブジェクトの一種
- レコードのクラスは，インスタンス変数の宣言のみから成る

# 点(Point)に対する関数を定義しよう

定義したいこと

- 点を作る (1つの式でやりたい)
- スカラー倍する
- 2点のベクトル和
- 2点の内分点を求める
- 画像中に点を打つ
- 画像中に2点を結ぶ直線を引く
- 3点を使って画像中の曲線を引く

```ruby
…
def point_make(u,v)
  p = Point.new()
  p.x = u
  p.y = v
  p
end
…
```

オブジェクトを返す

point.rb

```
…
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
…
```

point.rb

```ruby
…
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
```

配列aの縦の長さ

配列aの横の長さ

Rubyの配列の添え字が実数でもいいことを利用している

0.5は四捨五入をするため

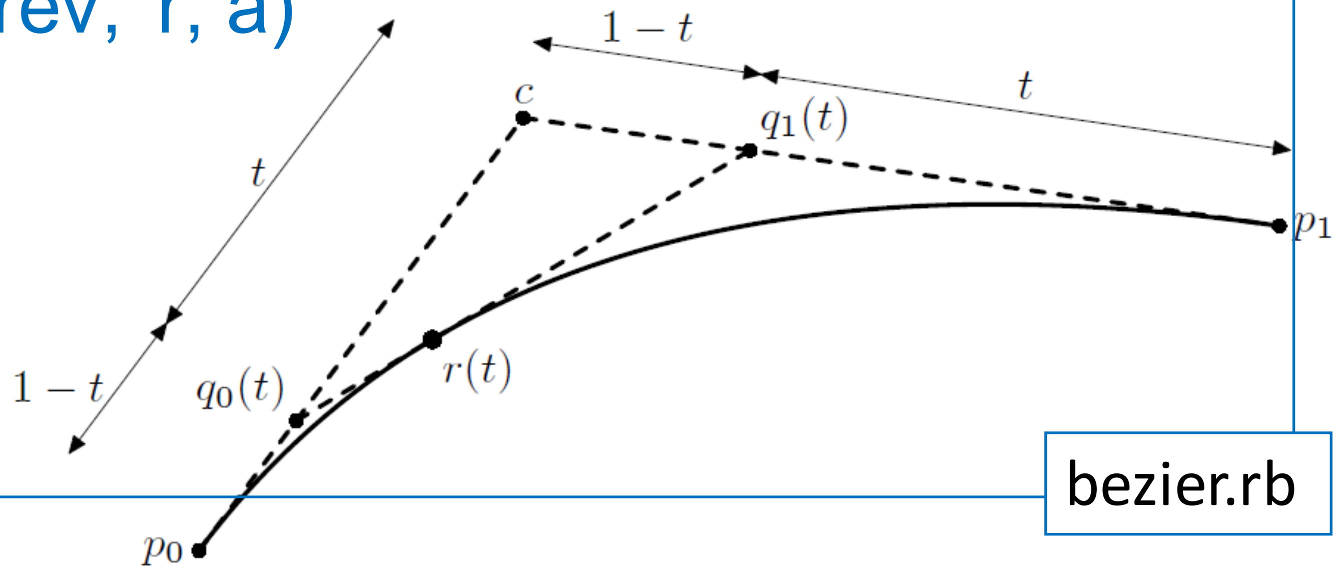
point.rb

```ruby
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
```

line.rb

```ruby
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
```




bezier.rb

# 練習

- point.rb、line.rb、bezier.rb をロード。以下を順に isrb で実行せよ。

```
p0 = point_make(10,10)
p1 = point_make(90,90)
c = point_make(10,90)
a = make2d(100,100)
bezier_draw(p0,c,p1,a)
show(a)
```

- 時間が余った場合、kana.rbをダウンロードし、p.168練習8.1(a)、p.169練習8.2(a)(b)を解いてみよ

# レコードの意義

- ひとまとまりの複数のデータを表す変数が1個ですむ
  - Pointの場合，x座標とy座標をまとめて扱える
  - もっと複雑なデータの場合はもっとうれしい
- このようなデータのまとまりを値として扱える
  - 関数の値として返すことができる
    - 2つの座標を返すことは通常の関数では難しい
  - 内部のデータがどうなっているかを知らなくても使うことができる
    - もし，Pointの座標系が極座標系だったとしても，関数の使い方に変更を加える必要がない

# 練習8.3 (レコードを使わない曲線)

```
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
```

```
def bezier_nr(x0,y0,xc,yc,x1,y1,a)
  n = 10
  prevx = x0
  prevy = y0
  for i in 1..n
    t = i*1.0/n
    q0x = interpolate(x0, xc,  t)
    q0y = interpolate(y0, yc,  t)
    q1x = interpolate(xc,  x1, t)
    q1y = interpolate(yc,  y1, t)
    rx  = interpolate(q0x, q1x, t)
    ry  = interpolate(q0y, q1y, t)
    line_nr(prevx,prevy, rx,ry, a)
    prevx = rx
    prevy = ry
  end
end
```

# しかし、レコードのままだと

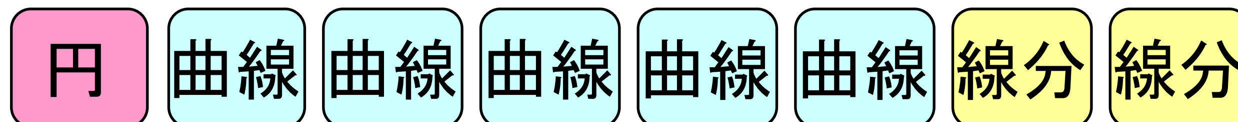
- 色々な種類の値が混在するデータを扱うときに、種類によって操作を使い分けなければいけない

同じ種類のレコードなら扱いは一様

p:

```
for i in 0..(p.length()-1)/2-1
  bezier_draw(p[i*2],p[i*2+1], p[i*2+2], a)
end
```

の



# しかし、レコードのままだと

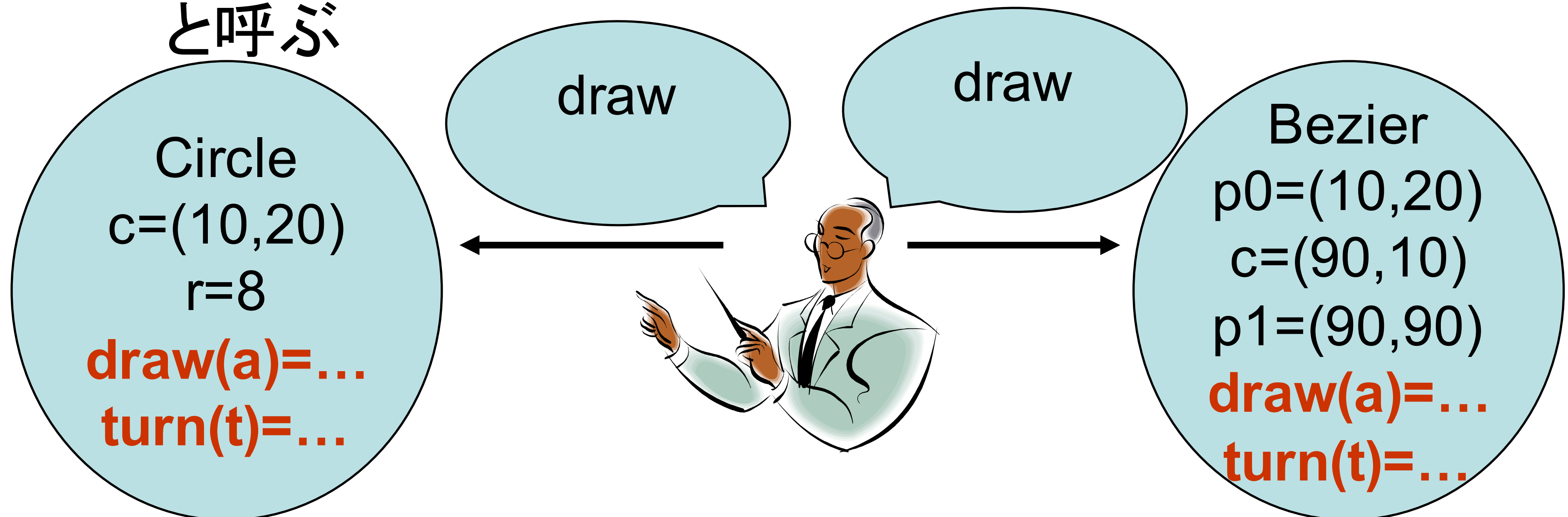
- 色々な種類の値が混在するデータを扱うときに、種類によって操作を使い分けなければいけない

```
for i in 0..(f.length()-1)
  if f[i]が円
    circle_draw(f[i].center,f[i].radius,a)
  else
    if f[i]が曲線
      bezier_draw(..., a)
    else
      if f[i]が線分
        line_draw(..., a)
      else
        ...
      end
    end
  end
end
```

f: 円 曲線 曲線 曲線 曲線 曲線 線分 線分

# オブジェクト指向では

- オブジェクト(ここではレコードのこと)に自分に対する操作の仕方を覚えさせる
  - Circleのレコードに,「表示」,「回転」などの操作を覚えさせる
  - 自分の操作法を知っているレコードを**オブジェクト**と呼ぶ



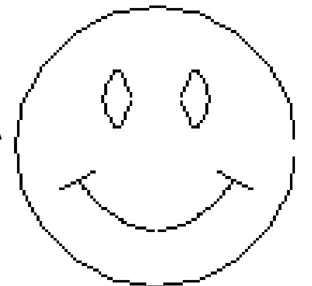
# オブジェクト指向では

- 各オブジェクトが「draw」の方法を知っている
- 扱う側は同じ命令を出すだけでよい

```
for i in 0..(f.length()-1)
  f[i]に対して「a上にdrawせよ」と命令
end
```

```
for i in 0..(f.length()-1)
  f[i]に対して「45度回転」と命令
end
```

f: 円 曲線 曲線 曲線 曲線 曲線 線分 線分



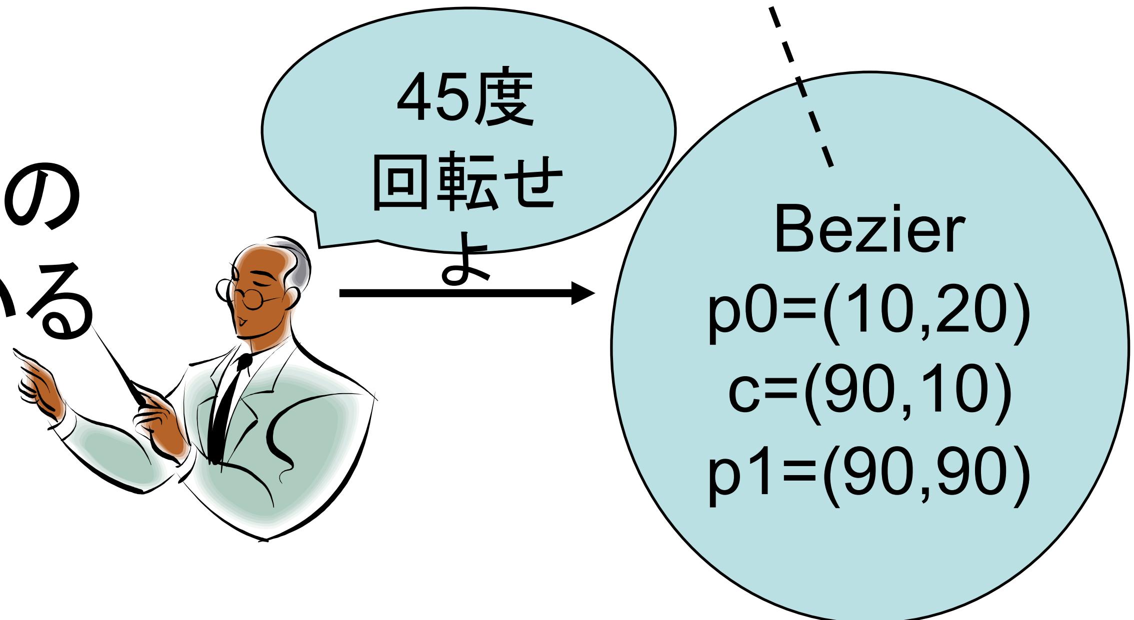
# オブジェクトとクラス

実際には操作方法はオブジェクトではなくクラスに定義される

- オブジェクト: 自分のクラスを知っている
- クラス: 操作方法の定義をまとめている

Bezierクラス

- 値としてp0,c,p1を持つ
- drawの定義
- turnの定義



# クラス

```
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
```

初期化メソッド

自分自身の x

scaleメソッド

自分の操作法を
メソッドと呼ぶ

oo-point.rb

初期化メソッドの引数

```
irb(main):005:0> p = Point .new(3,4)
=> #<Point:0x7ffa3ed8 @x=3, @y=4>
irb(main):006:0> p.x
=> 4
irb(main):007:0> q = p.scale(2)
=> #<Point:0x7ffa3e84 @x=6, @y=8>
irb(main):008:0> p.add(q)
=> #<Point:0x7ff9b0f8 @x=9, @y=12>
irb(main):009:0> p.add(q).scale(0.5)
=> #<Point:0x7ff94ca8 @x=4.5, @y=6.0>
```

新しいオブジェクト

新しいオブジェクト

新しい
オブジェクト

```
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
  …
```

oo-line.rb

# 多相性

```
def drawall(elements,a)
  for i in 0..elements.length()-1
    elements[i].draw(a)
  end
  a
end
…
```

drawall.rb

element[i] のクラスが何であれ,
そのdrawメソッドが呼ばれる

```ruby
…
def drawmoon()
  p0=Point.new(0,85)
  p2=Point.new(99,85)
  f=[Line.new(p0,p2),
       Bezier.new(p0,Point.new(50,60),p2),
       Circle.new(Point.new(66,20),20)]
  a=make2d(100,100)
  drawall(f,a)
  show(a)
end
```

drawall.rb

# 練習8.5

b) Pointクラスにdrawメソッドを追加せよ。

- すなわち、oo-point.rb のPointクラスの中に、

```
def draw(a)
  …
end
```

  を追加。
- point.rb の point_draw を参考に。

c) interpolateメソッドを追加。

- drawmoon() を実行せよ
  - 必要なクラスのファイルをロードする。
  - Circleクラスの定義(練習8.6)を後回しにする場合は式Circle.new(…)を消して実行

```
def interpolate(q,t)
  self.scale(1-t).add(q.scale(t))
end
```

もしくは

```
def interpolate(q,t)
  scale(1-t).add(q.scale(t))
end
```