7 Objektorientierung in Java

7.1 Alles ist Objekt

Grundeinheit von Java ist die *Klasse*.

Eine **Klasse** ist ein benannter *Namensraum*. Dieser ist bzw. enthält gleichzeitig einen *Datentyp* gleichen Namens.

Ein **Namensraum** kapselt die in ihm enthaltenen Elemente nach außen ab. Ein Element ist mit dem Schlüsselwort public zur Weitergabe nach außen freigegeben.

Von einem **Datentyp** können *Instanzen*, d.h. *Objekte*, d.h. Variable dieses Typs, erzeugt werden.

Elemente sowohl des Namensraumes als auch des Datentyps sind *Variable* und Unterprogramme, hier *Methoden* genannt.

Variable und **Methoden** des Namensraumes werden durch das Schlüsselwort static gekennzeichnet. Sie können innerhalb des Namensraumes beliebig verwendet werden.

Variable und **Methoden** des Datentyps sind immer an eine Instanz gebunden, sie definieren den Zustand und ermöglichen Zustandsänderungen in dieser Instanz.

Jede *Instanz* hat einen auf sie personalisierten Zugriff auf die **Methoden** des Datentyps, innerhalb einer Methode sind die Variablen des Datensatzes der Instanz verfügbar.

Auf Variable und Methoden des Namensraumes besteht allgemeinen Zugriff.

Die *Variablen* des **Namensraums** existieren nur einmal, die *Variablen* des **Datentyps** werden in jeder Instanz neu erschaffen.

7.2 Variablen

`static int count;` ist eine Variable des Namensraumes

`public static int count;` ist eine öffentliche Variable des Namensraumes

`int number;` ist eine Variable des Datentyps

`public double x,y;` sind öffentliche Variable des Datentyps, z.B. als Punkt in der Ebene

```
public class Punkt{
  public double x,y;
}
```

7.3 Methoden

`static double f(double x ){...; return y;}` ist eine Methode des Namensraumes, die einer reellen Zahl eine reelle Zahl zuordnet, also eine reelle Funktion.

`void ausgabe (){...}` ist eine Methode des Datentyps, deren Aktion von der Belegung der Instanzvariablen abhängen kann.

7.4 new und Konstruktor

Instanzen der (öffentlichen) Klasse `foo` werden mit

```
foo bar=new foo(1,2,"test");
```

erzeugt. Dazu muss foo eine Konstruktormethode ohne Rückgabetyp

```
foo( int a, int b, String name){...}
```

enthalten. Es muss kein Konstruktor angegeben werden, und die Argumentliste kann natürlich variieren.

7.5 Programme

Der Java–Compiler `javac` erzeugt zu jeder Klasse eine Binärcode-Datei.

Wird von der virtuellen Maschine `java` ein Programm `foo` aufgerufen, dann wird die Datei `foo.class` geöffnet und im Namensraum der Klasse `foo` eine öffentliche Methode `main` mit String-Liste als Argument gesucht, also

```
static public void main (String[] s ){...}
```

7.6 Arrays – Tupel bzw. Vektoren in Java

`Typ[] A = new Typ[N];` erzeugt ein Tupel $(A[0], A[1], \ldots, A[N-1])$. Die Elemente werden zu Null gesetzt bzw. der Standardkonstruktor wird ausgeführt. Anschließend kann $A[0], A[1], \ldots$ genauso verwendet werden wie Variable `Typ C1=new Typ(),C2=new Typ,C3=new Typ`

Unterschied: Über ein Array kann iteriert werden

```
double[] A=new double[10];
for(int k=0;k<10;k++) A[k]=1.0/k;
```

int [] B = {1,2,3}; erzeugt ein mit Werten vorbelegtes Tripel $(B[0],B[1],B[2]) = (1,2,3)$

Arrays sind selbst schon Objekte, Instanzen eines automatisch gebildeten Typs. Als solche haben sie zusätzliche Datenfelder und Methoden, wie die Länge

A.length ergibt die Länge N, mit der A angelegt wurde.

7.7 Polynome – Horner–Schema

Ein Polynom ist, algebraisch gesehen, eine endliche Folge von Koeffizienten.

```
public class Pol{
  int degree;
  double[] coeff;

  Pol(int deg){
    degree=deg; coeff=new double[deg+1];}
}
```

Nach der Objekt–Philosophie wird auf die Zustandsvariablen nicht direkt zugegriffen, sondern diese werden mit set– und get–Methoden geändert und gelesen.

```
  public void
    setCoeff(int k, double c){ coeff[k]=c; }
  public double
    getCoeff(int k){ return coeff[k]; }
  public int
    getDegree(){ return degree; }
```

Man kann, solange es übersichtlich bleibt, aber auch die Variablen des Datentyps als public deklarieren und direkt darauf zugreifen. N=P.degree, P.coeff[1]=5, etc.

Auswerten des Polynoms
Statt die Potenzen von x zu bilden und mit den Koeffizienten zu multiplizieren, kann man x auch soweit wie

möglich ausklammern.

$$\left.\begin{aligned} p(x) &= c_n x^n + \cdots + c_1 x + c_0 \\ &= (c_n x^{n-1} + \cdots + c_1)x + c_0 \end{aligned}\right\} \Longrightarrow \left\{\begin{aligned} p_n(x) &= p(x) \\ &= p_{n-1}(x)\,x + c_0 \\ p_{n-1}(x) &= p_{n-2}(x)\,x + c_1 \\ &\cdots \\ p_1(x) &= p_0(x)\,x + c_{n-1} \\ p_0(x) &= c_n \end{aligned}\right.$$

Lese die Zerlegung des Polynoms rückwärts und konstruiere schrittweise den Wert des Polynoms:

$$p(x) = 1\cdot x^3 + 3x^2 - 5x + 2 \text{ auszuwerten in } x = -2$$

k	0	1	2	3
c_{n-k}	1	3	-5	2
$p_{k-1}(x)x$		-2	-2	14
$p_k(x)$	1	1	-7	16

$\Longrightarrow\ p(-2) = 16$

$$\begin{aligned} \text{Ruffini–Regel} \Longrightarrow\ p(x) &= p(-2) + (x-(-2))q(-2,x) \\ &= 16 + (x+2)(1\cdot x^2 + 1\cdot x - 7) \end{aligned}$$

```
public double eval(double x){
      double p=coeff[degree];
      for(int k=degree-1;k>=0; k--){
        p *= x; p+=coeff[k];
  }
  return p;
}
```

7.8 Bisektionsverfahren

Suche Nullstellen einer stetigen Funktion $f : \mathbb{R} \to \mathbb{R}$. Es sei ein Anfangsintervall $[a_0, b_0]$ bekannt mit $f(a_0)f(b_0) < 0$.

Satz – Zwischenwertsatz – Sei $f : [a,b] \to \mathbb{R}$ stetig mit $f(a) < f(b)$ (betrachte sonst $g(x) = -f(x)$). Dann gilt $[f(a), f(b)] \subset f([a,b])$, d.h. für jeden Wert $y \in [f(a), f(b)]$ gibt es ein Urbild $x \in [a,b]$ mit $y = f(x)$.

Folgerung – Nullstellensatz – Sei $f : [a,b] \to \mathbb{R}$ stetig mit $f(a) < 0 < f(b)$. Dann gibt es eine Nullstelle $x^* \in (a,b)$ mit $f(x^*) = 0$.

Lemma – Größe von Polynomnullstellen – Sei $p(x) = cnx^n + \cdots + c_1x + c_0$ ein Polynom mit $c_n \neq 0 \neq c_0$. Dann haben alle Nullstellen, reell wie komplex, Beträge zwischen $r < |x| < R$ mit

$$R = 1 + \max\left(\frac{|c_0|}{|c_n|}, \ldots, \frac{|c_{n-1}|}{|c_n|}\right) \text{ und } \frac{1}{r} = 1 + \max\left(\frac{|c_1|}{|c_0|}, \ldots, \frac{|c_n|}{|c_0|}\right).$$

Der Nullstellensatz und damit der Zwischenwertsatz für $g(x) = f(x) - y$ können mit dem Bisektionsverfahren bewiesen werden. Dabei wird eine Stelle des Vorzeichenwechsels von f immer enger eingegrenzt. Dazu wird rekursiv eine geschachtelte Folge von Teilintervallen bestimmt, startend bei $[a_0, b_0] = [a, b]$.

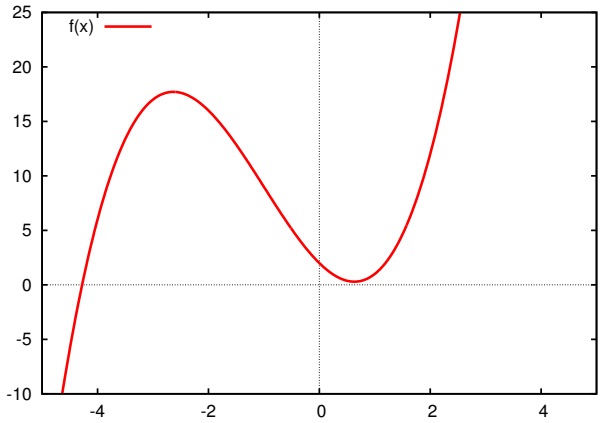
Sei $[a_k, b_k]$ bekannt mit $f(a_k)f(b_k) < 0$. Bestimme den Mittelpunkt $c_k = \frac{1}{2}(a_k + b_k)$ und setze

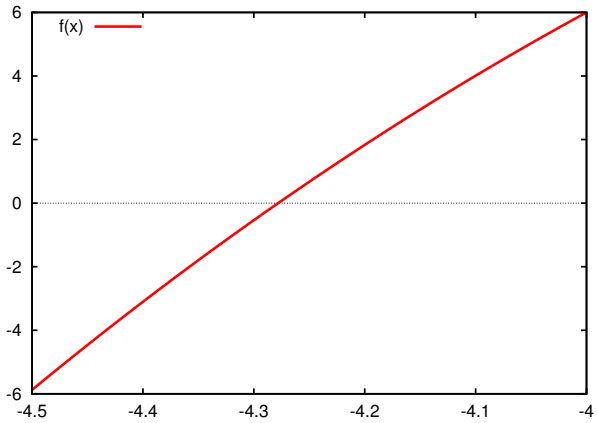
$$[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, c_k] & \text{wenn } f(a_k)f(c_k) \leq 0 \\ [c_k, b_k] & \text{wenn } f(c_k)f(b_k) < 0 \end{cases}$$

Ist $f(c_k) = 0$, dann ist eine Nullstelle gefunden, sonst fortsetzen bis $|b_k - a_k| < \varepsilon$

Die Folge (a_k) ist monoton wachsend und durch jedes b_m nach oben beschränkt. Die Folge (b_k) ist monoton fallend und durch jedes a_m nach unten beschränkt. Beide konvergieren und der Grenzwert x^* muss derselbe sein, da $b_k - a_k = 2^{-k}(b-a) \xrightarrow[k\to\infty]{} 0$. Wegen Stetigkeit ist $f(x^*) = 0$, da sowohl nichtnegativ als auch nichtpositiv.

```
public double
bisect(double a, double b){
  double fa=f(a), fb=f(b);
  while((b-a)>1e-15){
    double c=(a+b)/2, fc=f(c);
    if(fa*fc<0){
      b=c, fb=fc;
    } else {
      a=c; fa=fc;
    }
    if(Math.abs(fc)<1e-12) return c;
  }
  return a;
}
```





	$[a,$	$b]$		$[f(a),$	$f(b)]$
1	$[-5.0000000,$	$-4.0000000]$	$->$	$[-23.000000,$	$6.0000000]$
2	$[-4.5000000,$	$-4.0000000]$	$->$	$[-5.8750000,$	$6.0000000]$
3	$[-4.5000000,$	$-4.2500000]$	$->$	$[-5.8750000,$	$0.67187500]$
4	$[-4.3750000,$	$-4.2500000]$	$->$	$[-2.4433594,$	$0.67187500]$
5	$[-4.3125000,$	$-4.2500000]$	$->$	$[-0.84692383,$	$0.67187500]$
6	$[-4.2812500,$	$-4.2500000]$	$->$	$[-0.077911377,$	$0.67187500]$
7	$[-4.2812500,$	$-4.2656250]$	$->$	$[-0.077911377,$	$0.29937363]$
8	$[-4.2812500,$	$-4.2734375]$	$->$	$[-0.077911377,$	$0.11133051]$
9	$[-4.2812500,$	$-4.2773438]$	$->$	$[-0.077911377,$	$0.016859591]$
10	$[-4.2792969,$	$-4.2773438]$	$->$	$[-0.030488364,$	$0.016859591]$
11	$[-4.2783203,$	$-4.2773438]$	$->$	$[-0.0068050073,$	$0.016859591]$
12	$[-4.2783203,$	$-4.2778320]$	$->$	$[-0.0068050073,$	$0.0050296363]$
13	$[-4.2780762,$	$-4.2778320]$	$->$	$[-0.00088709935,$	$0.0050296363]$
14	$[-4.2780762,$	$-4.2779541]$	$->$	$[-0.00088709935,$	$0.0020714150]$
15	$[-4.2780762,$	$-4.2780151]$	$->$	$[-0.00088709935,$	$0.00059219447]$
16	$[-4.2780457,$	$-4.2780151]$	$->$	$[-0.00014744328,$	$0.00059219447]$
17	$[-4.2780457,$	$-4.2780304]$	$->$	$[-0.00014744328,$	$0.00022237789]$
18	$[-4.2780457,$	$-4.2780380]$	$->$	$[-0.00014744328,$	$3.7467876e-05]$
19	$[-4.2780418,$	$-4.2780380]$	$->$	$[-5.4987559e-05,$	$3.7467876e-05]$
20	$[-4.2780399,$	$-4.2780380]$	$->$	$[-8.7598059e-06,$	$3.7467876e-05]$

Nullstelle bei -4,278039932250977

7.9 Regula falsi

Bestimme die Gerade durch $(a_k, f(a_k))$ und $(b_k, f(b_k))$ und wähle als c_k deren Nullpunkt.

$$c_k = a_k + \frac{b_k - a_k}{f(b_k) - f(a_k)}(0 - f(a_k)) = \frac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}$$

```
public double
regfa(double a, double b){
  double fa=f(a), fb=f(b);
  while((b-a)>1e-15){
    double c=(a*fb-b*fa)/(fb-fa), fc=f(c);
    if(fa*fc<0){
      b=c, fb=fc;
    } else {
      a=c; fa=fc;
    }
    if(Math.abs(fc)<1e-12) return c;
  }
  return a;
}
```

	$[a,$	$b]$		$[f(a),$	$f(b)]$
1	$[-6.0000000,$	$-4.1463415]$	$->$	$[-76.000000,$	$3.0236358]$
2	$[-6.0000000,$	$-4.2172669]$	$->$	$[-76.000000,$	$1.4368280]$
3	$[-6.0000000,$	$-4.2503453]$	$->$	$[-76.000000,$	$0.66369554]$
4	$[-6.0000000,$	$-4.2654924]$	$->$	$[-76.000000,$	$0.30255424]$
5	$[-6.0000000,$	$-4.2723701]$	$->$	$[-76.000000,$	$0.13709337]$
6	$[-6.0000000,$	$-4.2754809]$	$->$	$[-76.000000,$	$0.061949821]$
7	$[-6.0000000,$	$-4.2768854]$	$->$	$[-76.000000,$	$0.027959263]$
8	$[-6.0000000,$	$-4.2775191]$	$->$	$[-76.000000,$	$0.012611552]$
9	$[-6.0000000,$	$-4.2778049]$	$->$	$[-76.000000,$	$0.0056872426]$
10	$[-6.0000000,$	$-4.2779338]$	$->$	$[-76.000000,$	$0.0025643989]$
11	$[-6.0000000,$	$-4.2779919]$	$->$	$[-76.000000,$	$0.0011562377]$
12	$[-6.0000000,$	$-4.2780181]$	$->$	$[-76.000000,$	$0.00052131308]$
13	$[-6.0000000,$	$-4.2780299]$	$->$	$[-76.000000,$	$0.00023504206]$
14	$[-6.0000000,$	$-4.2780352]$	$->$	$[-76.000000,$	$0.00010597185]$
15	$[-6.0000000,$	$-4.2780376]$	$->$	$[-76.000000,$	$4.7778721e-05]$
16	$[-6.0000000,$	$-4.2780387]$	$->$	$[-76.000000,$	$2.1541608e-05]$
17	$[-6.0000000,$	$-4.2780392]$	$->$	$[-76.000000,$	$9.7122872e-06]$

Nullstelle bei x=-4,278039170094968, f(x)=9.71228719715001e-06

Problem: Aufgrund der Konvexität der Funktion wird das linke Intervallende nicht geändert, was die Konvergenz verlangsamt. Auswege: Beschränkung des Mittelpunktes auf das innere Drittel oder die inneren zwei Viertel des Intervalls, Brent's Methode: wenn ein Intervallende lange nicht geändert wurde, dann bestimme einen Mittelpunkt mit einem Verfahren höherer Ordnung,...

Z.B., wenn außen, dann halbiere den Abstand zum Mittelpunkt

```
  if(Math.abs(a+b-2c)>(b-a)/2){
```

```
    c=(a+b+2c)/4;  fc=f(c);
  }
```

	$[a,$	$b]$		$[f(a),$	$f(b)]$
1	$[-4.5731707,$	$-4.0000000]$	$->$	$[-8.0352668,$	$6.0000000]$
2	$[-4.5731707,$	$-4.2450274]$	$->$	$[-8.0352668,$	$0.78942342]$
3	$[-4.3417404,$	$-4.2450274]$	$->$	$[-1.5840577,$	$0.78942342]$
4	$[-4.3417404,$	$-4.2771943]$	$->$	$[-1.5840577,$	$0.020480060]$
5	$[-4.2937427,$	$-4.2771943]$	$->$	$[-0.38302089,$	$0.020480060]$
6	$[-4.2817514,$	$-4.2771943]$	$->$	$[-0.090096799,$	$0.020480060]$
7	$[-4.2787556,$	$-4.2771943]$	$->$	$[-0.017358171,$	$0.020480060]$
8	$[-4.2787556,$	$-4.2780393]$	$->$	$[-0.017358171,$	$5.9520420e-06]$

Nullstelle bei x=-4,278039325242205, f(x)=5.95204195419363e-06

7.10 Komplexe Zahlen als weiteres Beispiel einer Klasse

In den reellen Zahlen gibt es keine Quadratwurzel von -1. In der Ebene ist die Multiplikation mit -1 die Rotation um $180°$. Hintereinander Ausführen von Rotationen entspricht der Addition der Winkel. Zwei Rotationen um den halben Winkel ergeben die Rotation um den vollen Winkel. Also ist, in diesem erweiterten Kontext, die Rotation um $90°$ eine Quadratwurzel $\mathbf{i}$ von -1. Die allgemeine Rotationsmatrix für den Winkel α ist

$$\begin{pmatrix}\cos\alpha & -\sin\alpha\\ \sin\alpha & \cos\alpha\end{pmatrix}, \text{ also für } \alpha = 90°: \quad \mathbf{i} = \begin{pmatrix}0 & -1\\ 1 & 0\end{pmatrix}$$

Die komplexen Zahlen $\mathbb{C}$ sind die Drehstreckungen der Ebene $\mathbb{R}^2$,

$$z = x + \mathbf{i}y = \begin{pmatrix}x & -y\\ y & x\end{pmatrix} = r\begin{pmatrix}\cos\phi & -\sin\phi\\ \sin\phi & \cos\phi\end{pmatrix} = re^{\mathbf{i}\phi}$$

Die komplexe Zahl selbst ist als Punkt in der Ebene durch die erste Spalte der Matrix definiert. Das algebraische Verhalten wird jedoch durch die volle Matrix bestimmt.

Eine Java-Klasse für die komplexen Zahlen besteht also erst einmal aus zwei reellen Koordinaten, die als Real- und Imaginärteil interpretiert werden.

```
public class Cpx{
 double re,im;

 Cpx(double rr, double ii){re=rr;im=ii;}

 String toString(){ return ""+re+"+i*"+im; }
 }
```

7.11 Operationen

Das Verhalten als komplexe Zahl wird durch die Methoden dieser Klasse festgelegt:

```
public Cpx addTo(Cpx b){ re+=b.re; im +=b.im; return this;}

static public Cpx add(Cpx a, Cpx b){
      return (new Cpx(a)).addTo(b); }

public Cpx mulTo(Cpx b){
  double h=re; re=re*b.re-im*b.im;
  im=h*b.im+im*b.re; return this; }

static public Cpx mul(Cpx a, Cpx b){
      return (new Cpx(a)).mulTo(b); }
```

7.12 Anwendung Julia–Menge

Jede lineare Rekursion $x_{n+1} = a x_n + b$ kann durch Multiplikation mit $(a-1)$ und Addition von b auf die Gestalt

$$z_{n+1} = (a-1)x_{n+1} + b = a\left((a-1)x_n + b\right) = a\, z_n$$

gebracht werden, woraus sich das Verhalten der Folge eindeutig ablesen läßt.

Interessanter sind quadratische Rekursionen $x_{n+1} = a x_n^2 + b x_n + c$, $a \neq 0$. Wieder kann man versuchen durch Skalieren und Verschieben der Folge die Anzahl der Parameter zu reduzieren. Multiplikation mit a ergibt

$$(a x_{n+1}) = (a x_n)^2 + b\,(a x_n) + ac$$

Quadratisches Ergänzen und Korrektur der linken Seite auf denselben Faktor ergibt

$$(a x_{n+1} + \tfrac{1}{2}b) = (a x_n + \tfrac{1}{2}b)^2 + \tfrac{1}{2}b - \tfrac{1}{4}b^2 + ac,$$

also die Elimination des linearen Summanden. Mit $z_n = a x_n + \frac{1}{2}b$ ist der quadratische Koeffizient 1. Zusammenfassen des Konstanten Koeffizienten ergibt die Normalform der Rekursion

$$z_{n+1} = f(z_n) = z_n^2 + c\,.$$

Lemma – Ist $|z| > 2^{2^k} + |c|$, $k = 0, 1, \ldots$, so $|f(z)| > 2^{2^{k+1}} + |c|$.

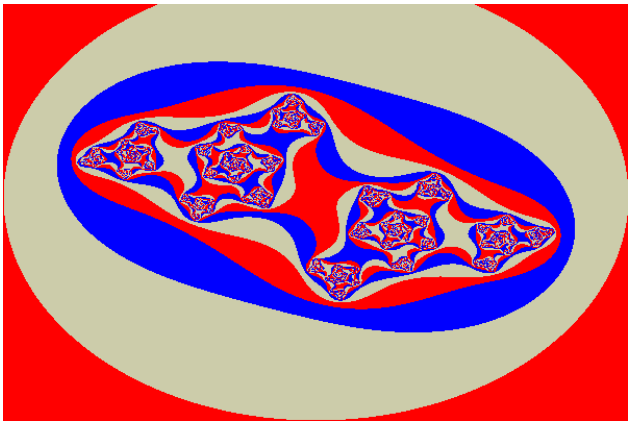
Abhängig vom Startpunkt z_0 ist die Folge $\mathcal{O}_+(z_0) = (z_n)_{n\in\mathbb{N}}$ beschränkt oder unbeschränkt, für große z_0 immer unbeschränkt.

Die **Fatou–Menge** $\mathcal{F}_\infty = \{z \in \mathbb{C} : \mathcal{O}_+(z_0)$ unbeschränkt $\}$ ist offen und zusammenhängend.

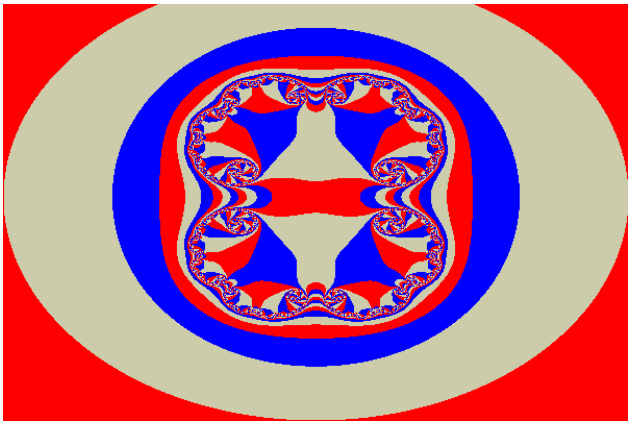
Der Rand von $\mathscr{F}_\infty$ ist das **Julia–Fraktal**.

Hat das Komplement von $\mathscr{F}_\infty$ innere Punkte, dann ist 0 einer davon. Die Parameter c, für welche der Orbit $\mathscr{O}_+(0)$ beschränkt ist, bilden die Mandelbrot-Menge, das Apfelmännchen-Fraktal.

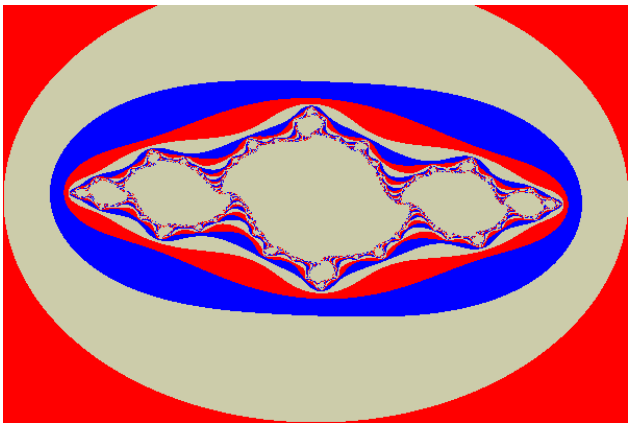
```
  static int escape(double cx, double cy, double x, double y){
   Cpx c=new Cpx(cx,cy);
   Cpx z=new Cpx(x,y);
   int count=0;

   while((count++<400) && z.abs()<2){
     z=Cpx.add(Cpx.mul(z,z),c);
   }
   return count;
 }
```



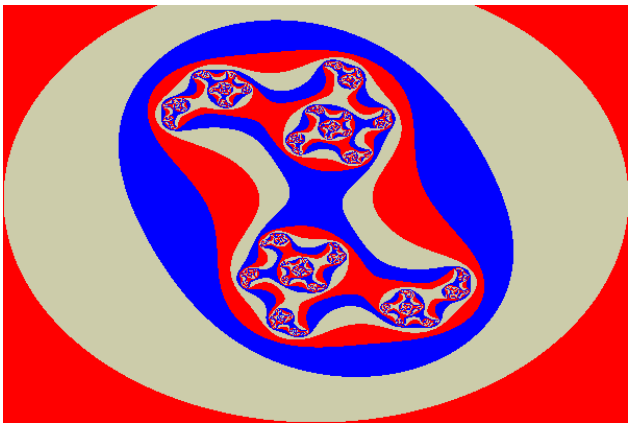
$c = -0.7 + 0.6\mathbf{i}$



$c = 0.3 + 0.0\mathbf{i}$



$c = -0.9 + 0.1\mathbf{i}$



$c = 0.5 + 0.5\mathbf{i}$