6th homework assignment; OPRPP1

Napravite prazan Maven projekt: u Eclipsovom workspace direktoriju napravite direktorij `hw06-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.jmbag0000000000:hw06-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema biblioteci `junit`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka. Potrebne biblioteke koje se spominju smjestite u direktorij `lib` unutar projekta (napravite ga), i zatim podesite `pom.xml` tako da koristi te biblioteke. Pri tome ćete JAR-arhive koje sadrže bytekod importati u Vaš lokalni maven-repozitorij; uputa o točnim koordinatama dana je kroz daljnji tekst. Direktorij `lib` NE uploadate u okviru predaje Vaše zadaće.

Problem 1.

Radimo vrlo jednostavnu biblioteku koja nudi potporu za rad s dvije vrste različitih matematičkih objekata: kompleksni brojevi te polinomi.

Napravite paket `hr.fer.zemris.math` i u njega smjestite modele kompleksnih brojeva i polinoma.

Radimo model kompleksnog broja, te dva modela polinoma koji su zadani nad kompleksnim brojevima, i čiji su koeficijenti kompleksni brojevi. Sva tri modela moraju stvarati neizmjenjive (*read-only*) objekte.

Krenimo redom. Napravite razred `Complex` koji modelira kompleksni broj, prema predlošku u nastavku.

```
public class Complex {

	...

	public static final Complex ZERO = new Complex(0,0);
	public static final Complex ONE = new Complex(1,0);
	public static final Complex ONE_NEG = new Complex(-1,0);
	public static final Complex IM = new Complex(0,1);
	public static final Complex IM_NEG = new Complex(0,-1);

	public Complex() {...}

	public Complex(double re, double im) {...}

	// returns module of complex number
	public double module() {...}

	// returns this*c
	public Complex multiply(Complex c) {...}

	// returns this/c
	public Complex divide(Complex c) {...}

	// returns this+c
	public Complex add(Complex c) {...}

	// returns this-c
	public Complex sub(Complex c) {...}

	// returns -this
	public Complex negate() {...}
```

```
    // returns this^n, n is non-negative integer
    public Complex power(int n) {...}

    // returns n-th root of this, n is positive integer
    public List<Complex> root(int n) {...}

    @Override
    public String toString() {...}
}
```

Napravite razred `ComplexRootedPolynomial` koji modelira polinom nad kompleksim brojevima, prema predlošku u nastavku. Radi se o polinomu *f(z)* oblika $z_0*(z-z_1)*(z-z_2)*...*(z-z_n)$, gdje su z_1 do z_n njegove nultočke a z_0 konstanta (sve njih zadaje korisnik kroz konstruktor). Primjetite, radi se o polinomu *n*-tog stupnja (kada biste izmnožili zagrade). Svi z_i zadaju se kao kompleksni brojevi, a i sam *z* je kompleksan broj. Metoda *apply* prima neki konkretan *z* i računa koju vrijednost ima polinom u toj točki.

```
public class ComplexRootedPolynomial {

	// ...

	// constructor
	public ComplexRootedPolynomial(Complex constant, Complex ... roots) {...}

	// computes polynomial value at given point z
	public Complex apply(Complex z) {...}

	// converts this representation to ComplexPolynomial type
	public ComplexPolynomial toComplexPolynom() {...}

	@Override
	public String toString() {...}

	// finds index of closest root for given complex number z that is within
	// treshold; if there is no such root, returns -1
	// first root has index 0, second index 1, etc
	public int indexOfClosestRootFor(Complex z, double treshold) {...}
}
```

Napravite razred `ComplexPolynomial` koji modelira polinom nad kompleksim brojevima, prema predlošku u nastavku. Radi se o polinomu *f(z)* oblika $z_n*z^n+z^{n-1}*z_{n-1}+...+z_2*z^2+z_1*z+z_0$, gdje su z_0 do z_n koeficijenti koji pišu uz odgovarajuće potencije od *z* (i zadaje ih korisnik kroz konstruktor). Primjetite, radi se o polinomu *n*-tog stupnja (što još zovemo red – engl. *polinom order*). Svi koeficijenti zadaju se kao kompleksni brojevi, a i sam *z* je kompleksan broj. Metoda *apply* prima neki konkretan *z* i računa koju vrijednost ima polinom u toj točki. Redoslijed faktora predanih u konstruktoru s lijeva na desno se tumači kao z_0, z_1, z_2, ...

```
public class ComplexPolynomial {

	// ...

	// constructor
	public ComplexPolynomial(Complex ...factors) {...}

	// returns order of this polynom; eg. For (7+2i)z^3+2z^2+5z+1 returns 3
	public short order() {...}

	// computes a new polynomial this*p
	public ComplexPolynomial multiply(ComplexPolynomial p) {...}

	// computes first derivative of this polynomial; for example, for
	//  (7+2i)z^3+2z^2+5z+1 returns (21+6i)z^2+4z+5
	public ComplexPolynomial derive() {...}

	// computes polynomial value at given point z
	public Complex apply(Complex z) {...}

	@Override
	public String toString() {...}
}
```

Evo u nastavku jednostavnog primjera.

```
ComplexRootedPolynomial crp = new ComplexRootedPolynomial(
      new Complex(2,0), Complex.ONE, Complex.ONE_NEG, Complex.IM, Complex.IM_NEG
);
ComplexPolynomial cp = crp.toComplexPolynom();
System.out.println(crp);
System.out.println(cp);
System.out.println(cp.derive());
```

Daje okvirni ispis:

```
(2.0+i0.0)*(z-(1.0+i0.0))*(z-(-1.0+i0.0))*(z-(0.0+i1.0))*(z-(0.0-i1.0))
(2.0+i0.0)*z^4+(0.0+i0.0)*z^3+(0.0+i0.0)*z^2+(0.0+i0.0)*z^1+(-2.0+i0.0)
(8.0+i0.0)*z^3+(0.0+i0.0)*z^2+(0.0+i0.0)*z^1+(0.0+i0.0)
```

Problem 2.

We will consider another kind of fractal images: fractals derived from Newton-Raphson iteration. As you are surely aware, for about three-hundred years we know that each function that is *k*-times differentiable around a given point x_0 can be approximated by a *k*-th order Taylor-polynomial:

$$f(x_0+\varepsilon)=f(x_0)+f'(x_0)\varepsilon+\frac{1}{2!}f''(x_0)\varepsilon^2+\frac{1}{3!}f'''(x_0)\varepsilon^3+\ldots$$

So let x_1 be that point somewhere around the x_0:

$$x_1=x_0+\varepsilon$$

Substituting it into previously given formula we obtain:

$$f(x_1)=f(x_0)+f'(x_0)(x_1-x_0)+\frac{1}{2!}f''(x_0)(x_1-x_0)^2+\frac{1}{3!}f'''(x_0)(x_1-x_0)^3+\ldots$$

For approximation of function *f* we will restrict our self on linear approximation, so we can write:

$$f(x_1)\approx f(x_0)+f'(x_0)(x_1-x_0)$$

Now, let us assume that we are interested in finding x_1 for which our function is equal to zero, i.e. we are looking for x_1 for which $f(x_1)=0$. Plugging this into above approximation, we obtain:

$$0=f(x_0)+f'(x_0)(x_1-x_0)$$

and from there:

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)}$$

However, since we used the approximation of *f*, it is quite possible that $f(x_1)$ is not actually equal to zero; however, we hope that $f(x_1)$ will be closer to zero than it was $f(x_0)$. So, if that is true, we can iteratively apply this expression to obtain better and better values for *x* for which $f(x)=0$. So, we will use iterative expression:

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

which is known as Newton-Raphson iteration.

For this homework we will consider complex polynomial functions. For example, lets consider the complex polynomial whose roots are +1, -1, *i* and -*i*:

$$f(z)=(z-1)(z+1)(z-i)(z+i)=z^4-1$$

After deriving we obtain:

$$f'(z)=4z^3$$

It is easy to see that our function f becomes 0 for four distinct complex numbers *z*. However, we will pretend that we don't know those roots. Instead, we will start from some initial complex point *c* and plug it into our iterative expression:

$$z_{n+1}=z_n-\frac{f(z_n)}{f'(z_n)}=x_n-\frac{z^4-1}{4z^3} \quad \text{with} \quad z_0=c \quad .$$

We will generate iterations until we reach a predefined number of iterations (for example 16) or until module $|z_{n+1}-z_n|$ becomes adequately small (for example, convergence threshold 1E-3). Once stopped, we will find the closest function root for final point z_n, and color the point *c* based on index of that root (let root indexes start from 1). However, if we stopped on a z_n that is further than predefined threshold from all roots, we will color the point *c* with a color associated with index 0.

convergencethreshold

For example, if the function roots are +1, -1, *i* and -*i*, if acceptable root-distance is 0.002, if convergence threshold equals 0.001 and if we stopped iterating after z_7=-0.9995+*i*0 because z7 was closer to z_6=-0.9991+*i*0 then convergence threshold, we will determine that z_7 is closest to second function root (first is +1, second is -1, third is +i, fourth is -i) and that z_7 is within predetermined root-distance (0.002) to -1, so we will color pixel *c* based on color associated with index 2. Since `ComplexRootedPolynomial.indexOfClosestRootFor` returns 0-based indexes, in pseudocode below we make coloring based on the returned index value incremented by 1.

We will proceed just as with Mandelbrot fractal:

height = ymax
width = xmax

```
for(y in ymin to ymax) {
  for(x in xmin to xmax) {
    c = map_to_complex_plain(x, y, xmin, xmax, ymin, ymax, remin, remax, immin, immax);
    zn = c;
    iter = 0;
    iterate {
      znold = zn;
      zn = zn - f(zn)/f'(zn);
      iter++;
    } while(|zn-znold|>convergenceTreshold && iter<maxIter);
    index = findClosestRootIndex(zn, rootTreshold);
    data[offset++]=index+1;
  }
}
```

We use *data[]* array same way as we did for Mandelbrot fractal and the GUI component will handle the rest; the only difference here is that content of *data[]* array does not represent the speed of divergence but instead

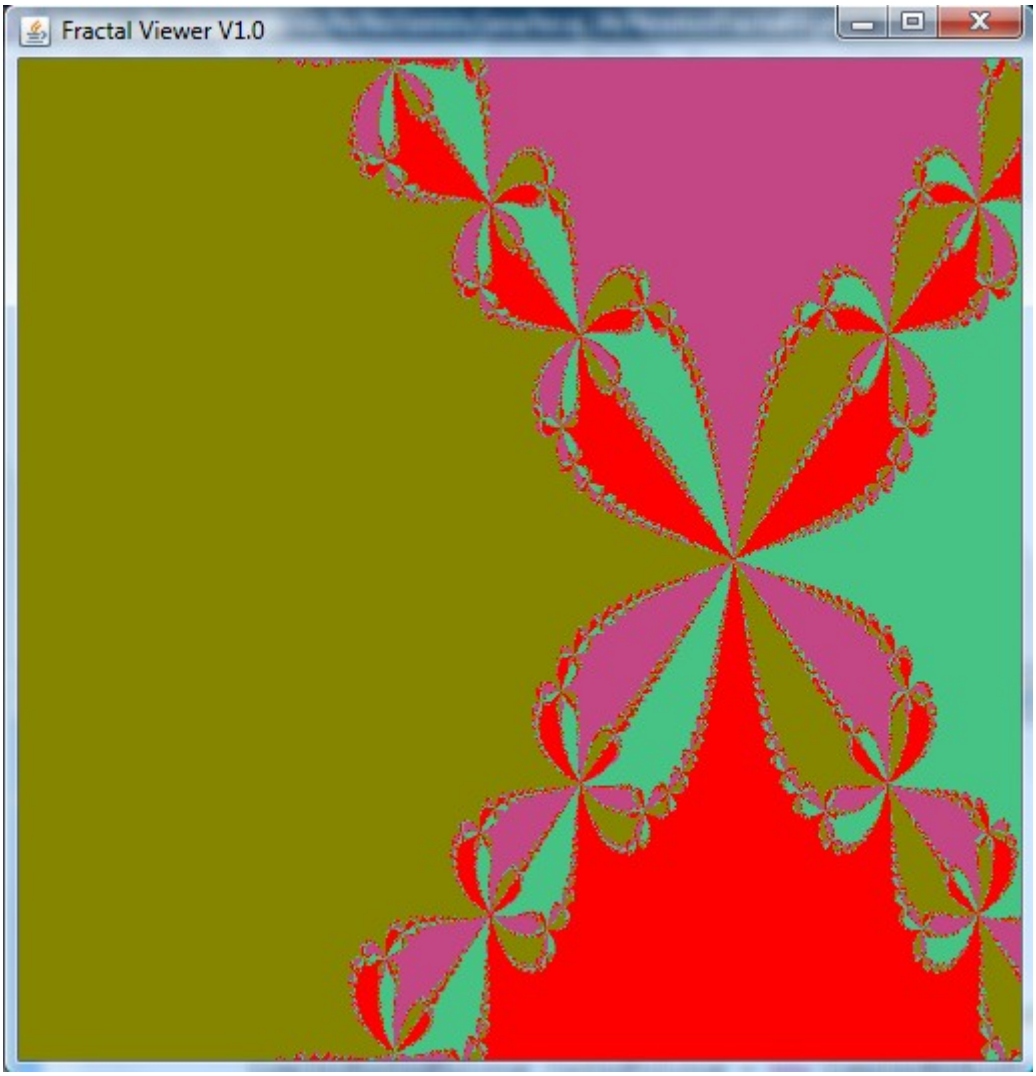
holds the indexes of roots in which observed complex point *c* has converged or 0 if no convergence to a root occurred. Another difference is that the upper limit to *data[i]* is number of roots, so we won't call observer with:

```
observer.acceptResult(data, (short)(m), requestNo);
```

but instead with:

```
observer.acceptResult(data, (short)(polynom.order()+1), requestNo);
```

If you completed this correct, for our first example with roots +1, -1, +i and -i you will get the following picture:



In order to solve this, install in your local maven repository jar `fractal-viewer-1.0.jar` under `hr.fer.zemris.java.fractals:fractal-viewer:1.0`, and add it as dependency to your `pom.xml`. Javadoc is available as separate jar.

More verbose introduction to fractals based on Newton-Raphson iteration can be found at:
http://www.chiark.greenend.org.uk/~sgtatham/newton/

Details

Given the classes you developed in problem 1, the core of iteration loop can be written as:

```
Complex numerator = polynomial.apply(zn);
Complex denominator = derived.apply(zn);
Complex znold = zn;
Complex fraction = numerator.divide(denominator);
Complex zn = zn.sub(fraction);
module = znold.sub(zn).module();
```

Write a main program `hr.fer.zemris.java.fractals.Newton`. The program must ask user to enter roots as given below (observe the syntax used), and then it must start fractal viewer and display the fractal. In order to run this successfully, you will have to add classpath configuration argument in command line when starting `java` (or you can run it through maven).

```
C:\somepath> java hr.fer.zemris.java.fractals.Newton
Welcome to Newton-Raphson iteration-based fractal viewer.
Please enter at least two roots, one root per line. Enter 'done' when done.
Root 1> 1
Root 2> -1 + i0
Root 3> i
Root 4> 0 -i1
Root 5> done
Image of fractal will appear shortly. Thank you.
```

(user inputs are shown in red)

General syntax for complex numbers is of form *a+ib* or *a-ib* where parts that are zero can be dropped, but not both (empty string is not legal complex number); for example, zero can be given as 0, i0, 0+i0, 0-i0. If there is '*i*' present but no *b* is given, you must assume that *b=1*.

In program `Newton`, your implementation must be sequential (non-multithreaded; see program `FraktalSlijednoProsireno` from lectures).

Then write new program `NewtonParallel`, which can be started from command line with following syntax:

```
java hr.fer.zemris.java.fractals.NewtonParallel --workers=2 --tracks=10
```

The number of threads to use for paralellization is controlled with --workers=N; alternatively, shorter form can be used: -w N. If this parameter is not given, use the number of available processors on the computer (see: Runtime.getRuntime() for number of available processors).

The number of tracks (i.e. jobs) to be used is specified with --tracks=K (or shorter: -t K). Minimal acceptable K is 1. If user specifies K which is larger than the number of rows in picture, “silently” use number of rows for number of jobs. If this parameter is not present, use 4 * `numberOfAvailableProcessors` jobs.

Please note, all parameters are optional (since default values are defined); each parameter can be specified using two different syntax; the order in which parameters appear is not important. Finally, it is error to specified the same parameter more than once (even if the value is the same).

Each time image generation is started, write effective number of threads and number of jobs.

treba napravit više konstruktora -> sa i bez neke od vrijednosti i s obe vrijednosti
-> ako ništa nije zadano - u onaj sa obje vrijednosti poslati default

The implementation of `IFractalProducer` that you will supply in this case must use parallelization based on `BlockingQueue`; see `FraktalParalelno2` from lectures for template (reuse this code and modify where appropriate).

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else), except the libraries mentioned in this homework which are available on Ferko. You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

There are no mandatory junit tests in this homework.
You are encouraged to write them.

When your **complete** homework is done, pack it in zip archive with name `hw06-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted.