CS100M Spring 2006 Project 5 **due Thursday 4/13 at 6pm**

Part B: Complex Numbers

You learned about complex numbers in Project 4 (and in your math classes). In this project, you will implement a `Complex` class to represent complex numbers and their operations. We can then use our `Complex` class in any Java applications that require complex numbers. We will write two such applications: drawing of a "Julia fractal" and another complex number calculator.

Throughout the project description, we will suggest code/ideas for *testing*. Frequent and incremental testing will save time overall!

Class `Complex`

First, download the skeleton file `Complex.java` from the course website. The skeleton will compile without errors. Implement the `Complex` class according to the specifications in the comments. *Do not change any provided code except for the two method stubs—dummy return statements—in the two methods whose return types are not* `void`. *Do not add additional instance variables or methods to the class.* Note the following:

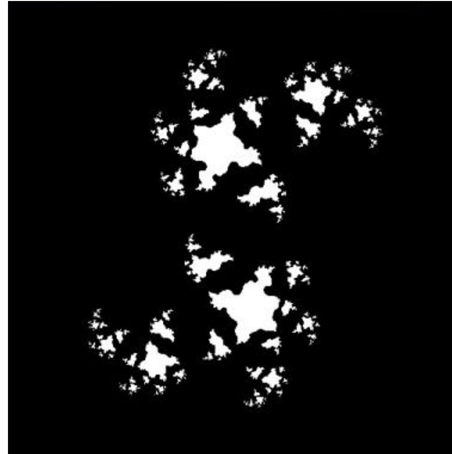
- A complex number has two attributes: a real part and an imaginary part. Two instance variables have been declared in the class to represent these attributes.

- There are two constructors (that you will implement). We say that the constructors are "overloaded." Overloading will be discussed in lecture on 4/11, so you can implement just the first one, with two `double` parameters, for now and implement the second one (with a `Complex` parameter) later.

- *Test each method after you write it!* For example, after writing the first constructor, compile your code, and in the *Interactions Pane* write code to create an object. E.g., type
  ```
  Complex n= new Complex(0, 0.5);
  ```
  This should work without error. Next you might want to implement the `toString` method to make further testing easier. For example, after you implement the `toString` method correctly, the following client code typed in the *Interactions Pane* (or in the `main` method of some class)
  ```
  Complex n= new Complex(0, 0.5);
  System.out.println(n);
  ```
  should display the text `(0 + 0.5i)`. *Continue to test one method at a time!* You're given skeleton code that compiles to begin with, so your submitted code must at least compile.

- The `add` method has been completed to serve as an example. The specification and method header read
  ```
  /* This Complex number plus Complex number z */
  public void add(Complex z)
  ```
  The return type is void, so the sum of *this* complex number (recall that an instance method lives in an object) and complex number `z` is stored back into *this* complex number, as shown in the completed method body.

Draw Julia Fractals



You will create images known as Julia fractals. These complicated jagged shapes are actually made with a relatively simple mathematical formula involving complex numbers (an iterative solution of the quadratic equation $z = z^2 + c$). We have provided the code to create a window for drawing. You will write a method to determine whether a complex number z is in the fractal.

A complex number $z = x + yi$ can be visualized as an x-y coordinate on the Cartesian plane. The image you will draw on can be thought of as the square [-1.5,1.5] $\times$ [-1.5,1.5], or in other words, it is the set of complex numbers with both real and imaginary parts in the interval [-1.5,1.5]. We would like to know whether a point (x,y), representing the complex number $z = x + yi$, is inside the fractal or not. The point is inside the fractal if the following sequence does *not* diverge:

$$f_0 = z \quad (1)$$

$$f_{n+1} = f_n^2 + c \quad (2)$$

Here c is a user-supplied complex number that determines the shape of the fractal. How do we know if the sequence diverges or not? We can approximate this by saying: if f_{20} is greater than 100, we assume the sequence diverges. (There is nothing special about the numbers 20 and 100, many other values could work fine.)

Download the skeleton file `Fractal.java`, read the first two (incomplete) methods, compile the file along with class `Complex`, and execute the program. Look for a `JFrame` called "Julia Fractal" that pops up for the graphics. It may appear in the background (behind the DrJava window or other windows) but you can drag it anywhere. You can enter values for the shape parameter but a fractal won't be drawn because a method is incomplete. Now close the frame. A message about resetting the interactions pane will appear and that is fine.

Task 1: Implement the method `isInsideFractal(Complex z)` in the file `Fractal.java`. Basically, you need to compute the sequence in equations (1) and (2) above and check to see if $f_{20} < 100$. You must follow the specifications (comments) and do not change the method header. You will need to change the stub (the given dummy return statement). You don't need arrays so don't use arrays.

Test your program before continuing with Task 2! Note that depending on your choice of the shape parameter c, the drawn fractal may not be easily visible (or it may not be drawn at all). A visible choice for testing is $c = 0.4 - 0.4i$, which generates the picture above. Try other values and see the cool pictures!

Task 2: The given code in the `main` method will get one complex number from the user and draw a fractal once. Modify the code in the `main` method so that it repeatedly prompts for another complex number and draws the corresponding fractal until the user wishes to stop. Note that you will use the same window for the graphics, so the `createWindow` method should be called just once.

Notes. The website `http://facstaff.unca.edu/mcmcclur/java/Julia/` might help to pick a good c so the fractal looks cool. By clicking on the Mandelbrot fractal (which is the set of points c such that the corresponding Julia fractal is connected) the applet will draw a Julia fractal for the value of c you clicked.

Want to save the picture of your fractal for yourself? Click on the frame, then press <Alt> + <Print Screen>. This copies the picture and then you can paste it into software such as Paint, Word, Powerpoint, etc.

Want to learn more about fractals? Note that we could try to solve the quadratic equation $z = z^2 + c$ by an iterative technique. (Of course, we know the closed form solution, so this is not that useful by itself.) This would yield the same sequence f_n as above. However, the key is the starting point of the iteration. For some points, the sequence will converge to either one of the two roots. For others, it will diverge into infinity. Yet for others (those on the boundary of the fractal) it will oscillate forever. You could color the points based on which one of these cases happens, or based on how fast does the sequence converge or diverge. Many of the colorful fractals you can find on the Internet use this simple technique. Check out `http://www.planetmike.com/fractals/`.

Complex Number Calculator: `CCalculator`

Now develop a Complex Number Calculator *using the* `Complex` *class that you have created.* You can use the Project 4 solution `SimpleComplexSol.java` as a starting point. The screen output produced by this `CCalculator` should be similar to that produced by the `SimpleComplex` program in P4. Your new complex number calculator, written in the file `CCalculator.java`, *must use* `Complex` *objects and call the methods defined in the* `Complex` *class to perform all the calculations.* You may use only the methods in the `Complex` class and in the standard library (e.g., `Math` methods, method `println` from the `System` class, etc.). Again, do not use arrays.

What will be the methods in your `CCalculator` class? The `main` method that drives the calculator program (similar to P4), the helper methods `isBinaryOp` and `isBadOperation`, and a suggested method `readComplexNumber` to prompt for the real and imaginary values to create and return a `Complex` object. The `readComplexNumber` method may look like this:

```
// =A Complex number based on user-entered real and imaginary parts
public static Complex readComplexNumber() {
  Scanner keyboard = new Scanner(System.in);
  System.out.print("Complex number x+yi?  x = ? ");
  double x_z = keyboard.nextDouble();
  System.out.print("x = " + x_z);
  System.out.print(", y = ? ");
  double y_z = keyboard.nextDouble();
  System.out.println("y = " + y_z);
  return new Complex(x_z, y_z);
}
```

Notice that the return type is `Complex` and a `Complex` object is created and returned in the last statement of the method. Calling the `readComplexNumber` method from the `main` method can help to keep `main` more concise and clear. You may write other methods in `CCalculator` in addition to those listed here. However, your code in `CCalculator` *must not repeat the functionality already provided by the class* `Complex`! Write concise code, and make sure that you revise all the comments (from the P4 solutions file) to reflect the code that you are writing for this project!

Test your program as you add functionality to the `CCalculator`! For example, after you write the `readComplexNumber` method, you will want to call it from the `main` method and print out the complex number. Remember the handy `toString` method in class `Complex`? Similarly, after you add the branch in the `main` method that deals with the add function, pause to test it before writing more code for the division function. *Incremental testing* will prevent buggy code atop buggy code and will end up saving you time!

Submission: Submit your files `Complex.java`, `Fractal.java`, and `CCalculator.java` on-line in CMS under Project 5 before the project deadline. For Java code be careful to submit the **.java** file, not the **.class** file.