Universidad Nacional Autónoma de México

Facultad de Ingeniería

Programación Orientada a Objetos

Grupo: 04

Profesora: Ing. Guadalupe Lizeth Parrales Romay

Proyecto final

Integrantes:

Dávila Ortega Jesús Eduardo - No. Cuenta: 317199860
Díaz Hernández Marcos Bryan - No. Cuenta: 317027253
Pareja Avila Emiliano - No. Cuenta: 317081345
Vazquez Zavala Oliver Alexis - No. Cuenta: 31720226

Semestre: 2021-1
Ciudad de México 4 de Febrero de 2021

Objetivo - Justificaciones del proyecto.

La realización del presente proyecto tiene como finalidad la creación y representación gráfica, mediante el lenguaje de programación Java, de un fractal Mandelbrot y algunas de las variantes más representativas. Determinamos que este es un proyecto final íntegro, esto debido a las diversas formas que tiene para implementarse, además de que para la elaboración del mismo requerimos implementar la gran mayoría de elementos y propiedades del paradigma orientado a objetos que se estudiaron en el curso, además poder profundizar más allá de los requerimientos básicos del proyecto, ya que también empleamos la programación paralela en la elaboración del proyecto.

Introducción.

Los fractales son estructuras geométricas que se caracterizan por estar compuestas por sí mismas en distintas proporciones, uno de estos entes matemáticos el cual consideramos el más interesantes de este tipo el el fractal de Mandelbrot o también conocido como set - Mandelbrot.

El fractal de Mandelbrot es conocido por la complejidad que se ve intrínseca en este debido a que su representación se encuentra dentro de los números imaginarios y por la dificultad de la irregularidad que se manifiesta en el, ya que la representación matemática de estos no es una tarea sencilla de llevar a cabo. Sin embargo, gracias al avance tecnológico es posible delegar tareas a los procesadores de nuestras computadoras, ya que cuentan con el poder de procesamiento necesario.

Debido a lo anterior y con el objetivo del proyecto nos enfocamos en crear un programa que pudiera llevar a cabo la representación gráfica de este fractal. Por medio de las operaciones con los números complejos creamos un programa que verifica la convergencia de un conjunto de píxeles, dentro de un radio de convergencia y en base a esto poder asignarle un color al pixel, de forma que al llevar a cabo esto dentro un conjunto iterativo es posible crear un gráfico que representa el fractal de mandelbrot.

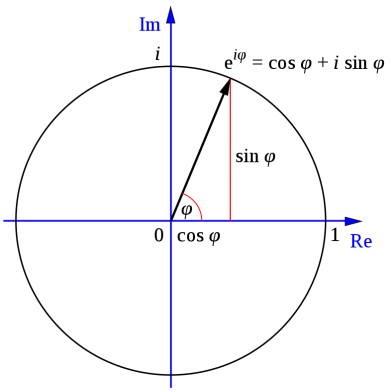
Decidimos implementar la paralelización debido a que fuimos desarrollando el programa solo con los fractales, pero al ver cómo se podía implementar en Java, comenzamos a investigar y buscar como poder paralelizar las instrucciones, lo que nos llevó a poder encontrar más clases que nos ayudaron a poder crear imágenes y designar a cada uno de los pixeles un color de acuerdo al criterio de fractal de Mandelbrot que se describe más adelante.

Como se puede ver en el código nuestro programa se compone de tres distintos fractales, Mandelbrot, Julia y Multibrot, donde los dos últimos son variantes de Mandelbrot, y debido a eso realizan operaciones similares pero no iguales, por lo que no es eficiente realizar la herencia ya que es necesario el replantear los métodos para el modelaje de cada uno de los fractales.

Para que sea posible comprender mejor el programa y por la explicación del párrafo anterior es necesario explicar la teoría detrás del conjunto de Mandelbrot, ya que cada una de las clases que modela cada uno de los fractales, utiliza el mismo concepto con sus respectivas variantes.

Antecedentes

El fractal de Mandelbrot se crea dentro de un plano complejo, por lo que para poder comprender cómo se emplea este en la implementación del programa es necesario explicarlo. El plano Argand se caracteriza por tener dos ejes perpendiculares, donde el vertical corresponde al eje imaginario y el horizontal corresponde al eje real, de tal forma que los números complejos están compuestos de una parte real y una parte imaginaria.



Es posible realizar las operaciones más simples como la multiplicación, la suma, la resta y la división en este plano, respetando los tipos de datos, es decir que los reales se simplifican con los reales, y los imaginarios de igual forma con sus correspondientes. Cada uno de los números complejos se representan o se pueden representar por medio de tres formas: polar, matricial y vectorial.

La expresión: f(z) = z^(2) + c, es una operación donde c = a + bi, es un número complejo, y por medio de las iteraciones este valor comienza recorrer el plano de imaginario, que para aplicaciones prácticas se limita a una región determinada, y a un radio que permita el determinar la convergencia.

El término c, en la iteración comienza a elevarse a la n potencia, con lo cual existen dos elementos que se deben de tomar en cuenta, que se puede repetir hasta el infinito, por ello se limita a un número determinado de iteraciones, y el segundo elemento es la multiplicación de número complejos. La cual determinará en gran porción el pixel que será coloreado, ya que la expresión permite el recorrer el plano limitado.

Descripción teórica.

El conjunto de Mandelbrot se compone por la siguiente regla iterativa:

$$z_{n+1} = z_n^2 + c$$

Con Zo = 0 y siendo $\boldsymbol{c}$ todos los puntos del plano complejo. Al ir evaluando puntos del plano complejo, notaremos que existen dos situaciones en la que nos encontremos.

- Si, por ejemplo, tomamos a ***c = 1*** veremos los siguientes términos:

$$z_0 = 0$$

$$z_1 = 0^2 + 1 = 1$$

$$z_2 = 1^2 + 1 = 2$$

$$z_3 = 2^2 + 1 = 5$$

$$z_4 = 5^2 + 1 = 26$$

$$z_5 = 26^2 + 1 = 677$$

- Como se puede apreciar, con **c = 1,** nuestra expresión crecería hasta el infinito.

Por otro lado, si consideramos el caso de **c = -1**, tendremos los siguientes primeros términos:

$$z_0 = 0$$

$$z_1 = 0^2 - 1 = -1$$

$$z_2 = (-1)^2 - 1 = 0$$

$$z_3 = 0^2 - 1 = -1$$

$$z_4 = (-1)^2 - 1 = 0$$

Donde podremos notar que el punto **c=1** del plano complejo, jamás alcanzará una magnitud mayor que **1.**

En general, **el conjunto de Mandelbrot se define como el conjunto de todos los puntos *c* del plano complejo para los que la iteración definida arriba no diverge.** De acuerdo a esta definición, podemos asegurar que el caso de **c = 1** no pertenece a este conjunto, pues la iteración en la expresión con este valor tiende al infinito, mientras que con **c = -1** aseguramos que estarán acotados los resultados de la expresión iterativa, es decir, que no divergen, por lo tanto, pertenecen al conjunto de Mandelbrot.

Para obtener más puntos del plano complejo se podrían calcular una inmensidad de valores para obtener sólo aquellos que sí pertenecen a este conjunto, una tarea imposible de realizar a mano pero que con ayuda de computadoras puede ser posible mediante una representación gráfica.
Cabe mencionar que matemáticamente se sabe que todos los puntos para los que |***c***| **> 2** son puntos que divergen, por lo que, por la misma razón, cualquier ***c*** para la que en algún punto de la iteración |Zn| > 2 también serán puntos que divergan y necesariamente no son parte del conjunto de Mandelbrot.

Así pues, el evaluar un punto exacto en esta expresión iterativa puede llevarnos a estancarnos en un bucle infinito en el cual los valores siempre permanecerán en nuestro **radio de convergencia** (menor o igual a 2), lo cual sería contraproducente pues se calcularían indefinidamente los mismos valores obtenidos. Para resolver este inconveniente es que se utiliza una variable llamada **número de iteraciones**, mediante la cual podremos limitar el número de veces que se calculará la expresión en el plano complejo para un valor ***c*** dado. Además de poder ocupar el número de iteraciones para no sobrecargar de trabajo a la computadora, se puede utilizar este mismo número para poder saber a partir de qué iteración nuestro valor en el plano complejo se sale de nuestro radio de convergencia, para así, crear una especie de **"mapa de calor"** en el que la gradiente de color represente el número de iteraciones.

Análisis - Descripción.

- import java.awt.image.BufferedImage.

La clase permite crear un buffer para poder crear el espacio de una imagen, en este caso el programa crea varias imágenes de distintos colores. El método constructor permite colocar las dimensiones de la imagen o imágenes que se van a crear, además de poder colocar el tipo de imagen, que en este caso es una imagen del tipo RGB.

```
BufferedImage img = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);
```

El método setRGB, permite asignarle un color al pixel en el que se esté trabajando durante la iteración del algoritmo, para ello se necesitan las coordenadas y el color que se le va a asignar.

```
img.setRGB(c, r, point);
```

- import java.io.File;

La clase permite el manejo de los archivos, por medio de esta se puede transmitir la información para la escritura de la imagen, y poder crear las imágenes que en conjunto los hilos realizarán

```
BufferedImage img = new BufferedImage(N, N, BufferedImage.TYPE_INT_RGB);
File archivo = new File("FractalMandelbrot"+cimg+".png");
```

- import javax.imageio.ImageIO;

La clase permite la escritura en imágenes, por medio del método write que es necesario el enviarle un objeto de BufferedImage y un File:

```
  ImageIO.write(imagen, "png", new File("Julia"+nimg+".png"));
}catch(IOException e){
```

Donde el Buffered utiliza la interfaz RenderedImage, para poder crear la imagen por medio de la asignación de los colores a los pixeles, y se utiliza el objeto File para poder crear la imagen, y por medio del nombre crear el tipo de imagen que se especifica.

Clases Implementadas

Clase Colores:

La clase colores es utilizada por las diferentes clases generadoras de fractales para poder obtener los colores deseados en el fractal.

```
package fractales;

/**
 * Esta clase permite asignar la pal
 * de los colores que tendra se da p
 */
public class Colores{
    private int maxIt;

    public Colores(int maxIt){
        this.maxIt = maxIt;
    }
```

Para esto se realizan operaciones de corrimiento de bits en un método llamado getColor y mediante if-else se regresa el formato rgb de los colores seleccionados por el usuario, este método regresa un entero.

```
        if(c==1)
            return ( (2*a<<16) );
        else if(c==2)
            return ( (255 * (i/15)) << 16 | (255 * (i/15)) );
        else if(c==3)
            return ((255 * (i/20)) << 16 | 0 | 0 );
        else if(c==4)
            return (65536 + i*256 + i/2+128);
        else if(c==5)
            return ( (0) | (2*a<<17) | (a<<4) | ((a*3)<<0) );
        return((255 * (i/10)) << 16 | (255 * (i/10)) << 8 | (255 * (i/10)) );

    }
}
```

Clase Mandelbrot:

Para poder implementar el paralelismo en el programa se decidió que la clase Mandelbrot hereda de la clase Thread, para poder trabajar con hilos.

```
package fractales;

import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;

/**
 * La clase permite la implementacion del
 * de Thread para poder implementar los hi
 */
public class Mandelbrot extends Thread {
```

Esta clase tiene atributos iniciales los cuales ya están inicializados y dentro el método mandelbrot se generan 4 hilos que harán ¼ de imagen cada uno de ellos, para ello

ocupamos un identificador entero que nos permitirá hacer que se divida la imagen en 4 y cada hilo pueda trabajar independientemente. Una vez que terminan de hacer su trabajo los hilos se destruyen, y el hilo principal, tomando una matriz con datos proporcionados por los hilos, creará la imagen y la almacenará en un archivo .png.

```
public void mandelbrot() throws InterruptedException {

    Mandelbrot thread0 = new Mandelbrot(0);
    Mandelbrot thread1 = new Mandelbrot(1);
    Mandelbrot thread2 = new Mandelbrot(2);
    Mandelbrot thread3 = new Mandelbrot(3);
    thread0.start();
    thread1.start();
    thread2.start();
    thread3.start();

    thread0.join();
    thread1.join();
    thread2.join();
    thread3.join();

    BufferedImage img = new BufferedImage(N, N, BufferedImage.TYPE_INT_RGB);
    File archivo = new File("FractalMandelbrot"+cimg+".png");
```

Para la correcta realización se hizo una sobre escritura al método run de tal manera que empezara a hacerse los cálculos para la generación del fractal pero en el método principal de esta clase se empieza a generar el fractal con los colores deseados mediante los valores generados por el método run. Este mismo mecanismo funciona con la clase Julia pero varía las operaciones que cada uno realiza.

```
public void run() {

     int begin = 0, end = 0;

    if (id == 0) {
        begin = 0;
        end = (N / 4) * 1;
    }
    else if (id == 1) {
        begin = (N / 4) * 1;
        end = (N / 4) * 2;
    }
    else if (id == 2) {
        begin = (N / 4) * 2;
        end = (N / 4) * 3;
    }
    else if (id == 3) {
        begin = (N / 4) * 3;
        end = N;
    }
```

En este segmento del código es en donde dividimos a la matriz que generará la imagen en 4 partes iguales. Al final cada hilo realizará las operaciones de cada cuarto de imagen. Y al ser operaciones totalmente independientes, estos hilos no tendrán alguna dependencia de datos que impida su desarrollo.

```
for (int i = begin; i < end; i++) {
    for (int j = 0; j < N; j++) {

        double cr = (4.0 * i - 2 * N) / N;
        double ci = (4.0 * j - 2 * N) / N;

        double zr = cr, zi = ci;

        int k = 0;
        while (k < IT && zr * zr + zi * zi < 4.0) {
            double newr = cr + zr * zr - zi * zi;
            double newi = ci + 2 * zr * zi;

            zr = newr;
            zi = newi;

            k++;
        }

        set[i][j] = k;
    }
}
```

En este segmento, es en donde realizamos las operaciones para determinar si un punto en el campo xy diverge o converge según la regla iterativa de Mandelbrot. Y como se ve al final del código, la matriz de enteros que creamos con todos los píxeles va tomando valores a partir de la variable k.

Clase Julia:

Esta clase que desarrollamos es bastante similar a la anterior, ya que también utilizamos la herencia de la clase Thread, puesto a que también para generar fractales del conjunto de Julia utilizaremos hilos que desarrollarán tareas completamente independientes. Al igual que la anterior también la inicializamos con valores por defecto, que en este caso, width será el ancho de la imagen y height el alto de la misma, zoom, será el acercamiento que haremos a la imagen y max será el número de iteraciones que realizaremos sobre este fractal, entendiendo que es un buen número de iteraciones en el cual se puede apreciar un fractal bien hecho, ya que al probar con menos o más iteraciones el fractal no se veía tan bien, además instanciamos el atributo imagen de la clase BufferedImage y pasamos como parámetros el ancho y el largo de la imagen, junto con el RGB que utilizamos.

```
public class Julia extends Thread{
  final static int ancho = 1920;
  final static int largo = 1080;
  final static int max = 300;
  private int  zoom=1, id, col, nimg;
  final static BufferedImage imagen = new BufferedImage(ancho, largo, BufferedImage.TYPE_INT_RGB);
```

Después creamos los hilos de la propia clase que se distinguen por tener un identificador entero, que nos permitirá dividir la imagen en cuatro partes, al invocar al método run de cada uno de estos hilos.

```
Julia thread0 = new Julia(0, col);
Julia thread1 = new Julia(1, col);
Julia thread2 = new Julia(2, col);
Julia thread3 = new Julia(3, col);

thread0.start();
thread1.start();
thread2.start();
thread3.start();

thread0.join();
thread1.join();
thread2.join();
thread3.join();
```

Como vemos ahora en el método run, gracias al identificador explicado anteriormente pudimos dividir la imagen en 4, generamos una paleta de colores y ahora dentro de las iteraciones sobre los renglones y columnas de píxeles, hicimos uso de la regla iterativa de mandelbrot, solo que variamos algunos valores de incremento, en este caso a la coordenada zx, en cada iteración le restamos 0.7, y a la coordenada zy le agregamos en cada iteración 0.27015. Y a diferencia del fractal anterior, ahora dentro de este método también definimos los colores de la imagen a partir de los datos que generamos, y así mejoramos aún más el rendimiento del tiempo en comparación con el fractal anterior, para poder conseguir los colores de los pixeles, utilizamos el método getColor de la clase Colores.

```
@Override
public void run(){
  int begin=0, end=0;
  if (id == 0) {
      begin = 0;
      end = (ancho / 4) * 1;
  }
  else if (id == 1) {
      begin = (ancho/ 4) * 1;
      end = (ancho/ 4) * 2;
  }
  else if (id == 2) {
      begin = (ancho/ 4) * 2;
      end = (ancho/ 4) * 3;
  }
  else if (id == 3) {
      begin = (ancho/ 4) * 3;
      end = ancho;
  }
```

```
for (int x = begin; x < end; x++) {
   Colores color = new Colores(max);
  for (int y = 0; y < largo; y++) {
    double zx = 1.5 * (x - ancho / 2) / (0.5 * zoom * ancho) ;
    double zy = (y - largo / 2) / (0.5 * zoom * largo);
    float i = max;
    while (zx * zx + zy * zy < 4 && i > 0) {
                double tmp = zx * zx - zy * zy -0.7;
                zy = 2.0 * zx * zy + 0.27015;
                zx = tmp;
                i--;
            }
    int c=0;
            if(col==7)
                c=Color.HSBtoRGB((max/ i) % 1, 1, i > 0 ? 1 : 0);
            else
                c = color.getColor((int)i, col);
            imagen.setRGB(x, y, c);
        }
 }
```

Clase Multibrot:

Esta era una aplicación un poco más compleja de un fractal, ya que involucra ahora a la potencia a la que se elevará la z en la regla iterativa de Mandelbrot, e igualmente, para tener un resultado visual agradable, definimos una imagen de 1080 x 1080 píxeles, y en este caso se iterará 100 veces el multibrot. Usamos un atributo de la clase Point2D, que nos ayudará a definir el centro de la imagen, ya que debe de estar en un rango específico, y también definimos un punto de quiebre que es el atributo pq, que es igual a 16, ya que los multibrots de potencia n, necesitan un punto de quiebre para evitar desbordamientos de imagen y que se generen fractales no deseados. El tamaño que definimos de 1.25 es para un rango de píxeles que tomaremos ya sea en un renglón o en una columna de pixeles. Y finalmente también instanciamos un atributo imagen de la clase BufferedImage, que como en los fractales anteriores nos permitirá crear una imagen dada su ancho y alto en pixeles.

```
public class Multibrot{
  final static int ancho = 1080;
  final static int alto = 1080;
  final static int max = 100;
  private int  col, nimg;
  public static Point2D centro = new Point2D.Double(-0.5, 0);
  public static double tamano = 1.25;
  public static int radioAn = ancho / alto;
  private static double pot;
  public static int pq = 16;

  final static BufferedImage imagen = new BufferedImage(ancho, alto, BufferedImage.TYPE_INT_RGB);
```

Empezamos a iterar sobre los píxeles del conjunto, y en cada una de estas iteraciones mapearemos una región pequeña de superficie en el conjunto, utilizando el método map que revisaremos más adelante. Después usamos la potencia dada por el usuario en el Main, para poder calcular valores de convergencia o divergencia, con la ya mencionada regla iterativa de Mandelbrot que en este caso serán las variables na y nb para obtenerlos hacemos uso de la clase Math que viene en java.lang, para calcular una raíz cuadrada, y en el mismo for que itera sobre los píxeles, vamos generando el conjunto de pixeles que formarán la imagen que queremos según los valores de convergencia o divergencia que hayamos obtenido.

```
for (int i = 0; i < ancho; i++) {
                for (int j = 0; j < alto; j++) {
                        double a = map(i, 0, ancho, centro.getX() - tamano * radioAn, centro.getX() + tamano * radioAn);
                        double b = map(j, 0, alto, centro.getY() - tamano, centro.getY() + tamano);

                        double ca = a;
                        double cb = b;

                        int n = 0;
                        while (n <= max) {
                                double na = Math.pow(Math.sqrt(a * a + b * b), pot) * Math.cos(pot * Math.atan2(b, a));
                                double nb = Math.pow(Math.sqrt(a * a + b * b), pot) * Math.sin(pot * Math.atan2(b, a));
                                if (Math.sqrt(a * a + b * b) > pq)
                                        break;
                                a = na + ca;
                                b = nb + cb;

                                n++;
                        }

                        int color;
                        if (n == max) {
                                color = negro;
                        } else {
                                color = (int) map(n, 0, max, 0, 255);
                        }
                        imagen.setRGB(i, j, color);
                }
        }
```

Dentro de esta clase el método map recibirá 5 parámetros, el primero el valor de la iteración o el píxel sobre el cual obtendremos una posición relativa que se calculará en el método multibrot si diverge o converge, su segundo parámetro es el límite inferior de donde se pueden tomar valores sobre un renglón o sobre una columna de pixeles en general, el tercer parámetro es el límite superior general de un renglón o una columna de pixeles, el cuarto, es a partir de dicha región de pixeles, a partir de qué píxel se empezará a mapear la superficie, y el quinto parámetro es el último píxel hasta donde se puede mapear. Y este método devolverá una posición relativa de un píxel ya sea sobre un renglón o una columna de pixeles.

```
}
public static double map(double value, double istart, double istop, double ostart, double ostop) throws NuestraExcepcion{
    if(ostart + (ostop - ostart) * ((value - istart) / (istop - istart))>ancho){
        throw new NuestraExcepcion("Linea 91");
    }
    return ostart + (ostop - ostart) * ((value - istart) / (istop - istart));
}
```

Clase NuestraExcepcion:
Esta clase está creada debido a la gran demanda de recursos que puede generar este programa al momento de ingresar parámetros muy amplios, por lo tanto se decidió limitar el programa para evitar el mal funcionamiento en ciertos equipos y esta se lanzará en el método map de la clase Multibrot, para indicar que el rango de pixeles x x y excede el rango de píxeles que puede soportar una potencia.

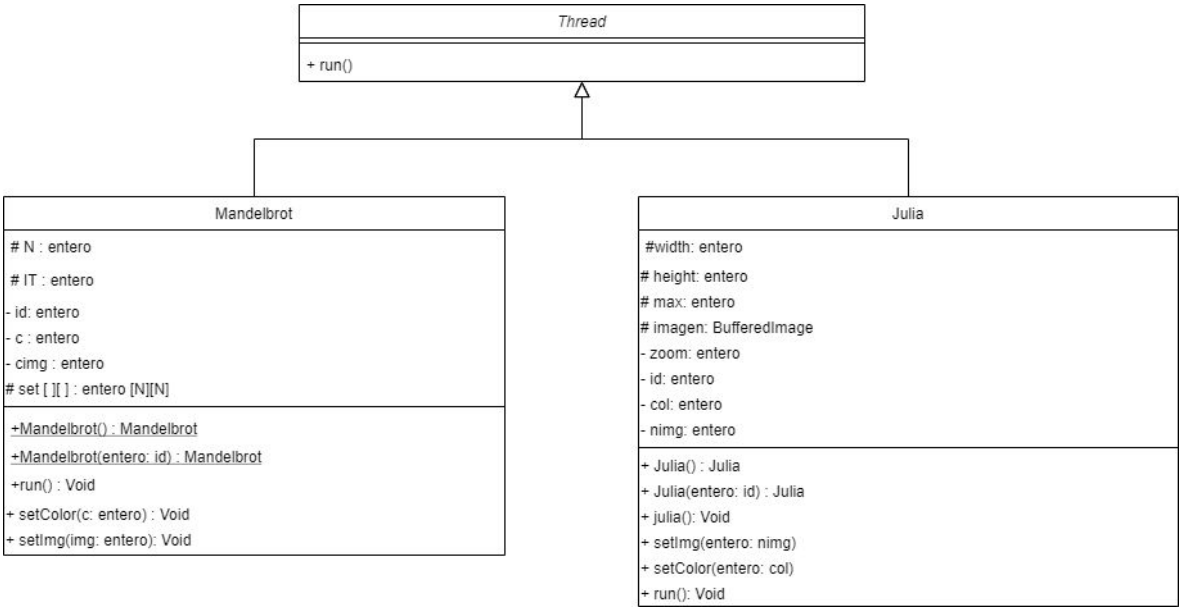
```
package fractales;
/**
 * En esta clase planteamos nuestra Excepcion en el caso de que en los calculo
 */
public class NuestraExcepcion extends Exception{
    private String error;

    NuestraExcepcion(String msg){
        super(msg);
        error="valores de 'x' y 'y' fuera del rango, debido a la potencia";
    }

    @Override
    public String toString(){
        return "Nuestra Excepcion: ["+" error "+"]";
    }
}
```

Clase NuestraExcepcion2:
Clase que funciona con cierto fractal para evitar sobrecargar la computadora debido a la complejidad de los cálculos, manda un mensaje en caso de que los parámetros sean muy grandes, esta se lanzará en el método main, en caso de que si el usuario ingrese una potencia muy elevada.

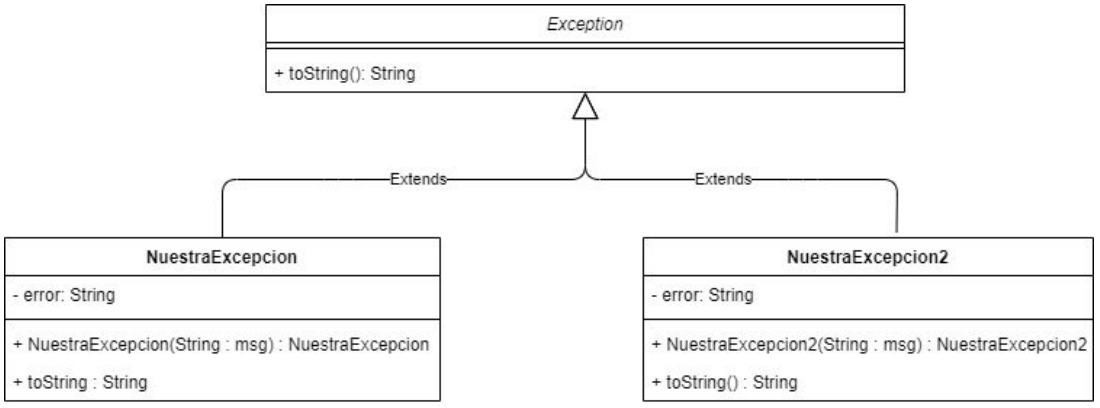
```
1  package fractales;
2
3  public class NuestraExcepcion2 extends Exception{
4      private String error;
5
6      NuestraExcepcion2(String msg){
7          super(msg);
8          error="el valor de la potencia es muy grande";
9          System.out.println(error);
10     }
11
12     @Override
13     public String toString(){
14         return "Nuestra Excepcion 2 : ["+" error "+"]";
15     }
16 }
```

Diagramas UML de las Clases Desarrolladas

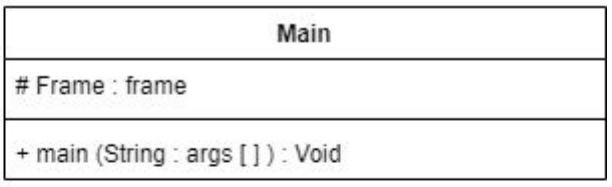
- **Clases Mandelbrot y Julia**



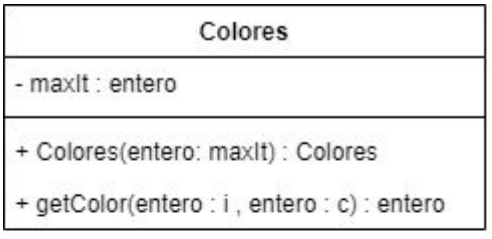
- **Clases NuestraExcepcion y NuestraExcepcion2**



- **Clase Main**



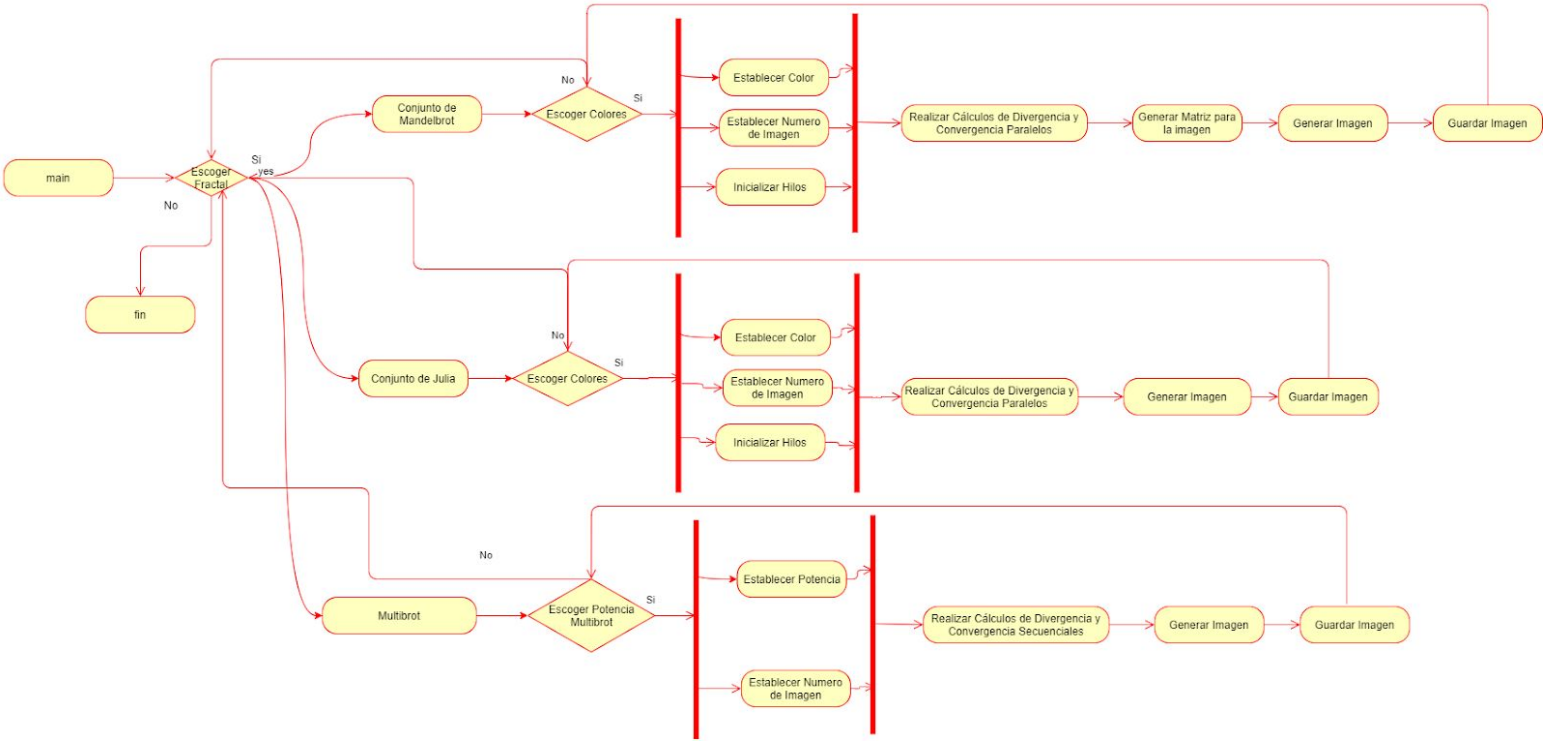
- **Clase Colores**



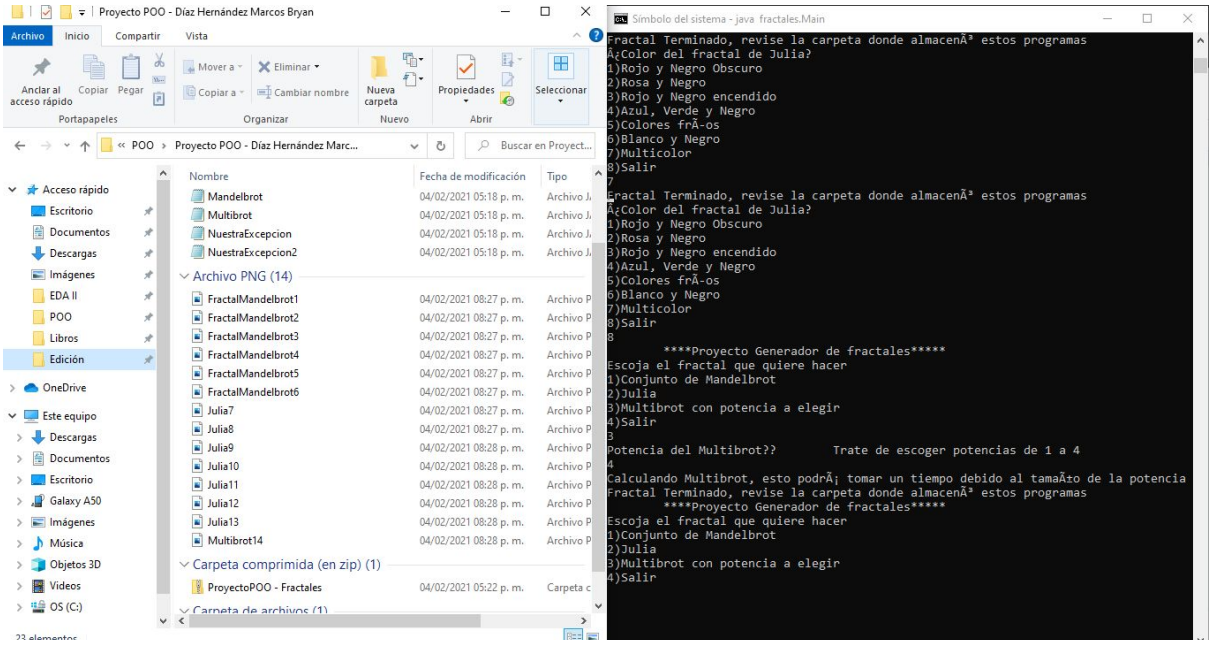
- **Clase Multibrot**

Multibrot
- col : entero - nimg : entero - pot : double # ancho : entero # alto : entero # max : entero # tamano : double # radioAn : entero # pq : entero # centro : Point2D # imagen : BufferedImage
+ multibrot() : Void + setPot(entero : pot) : Void + setImg(entero : nimg) : Void + setColor(entero : col) : Void # map(double : value, double : istart, double : istop, double : ostart, double : ostop) : double

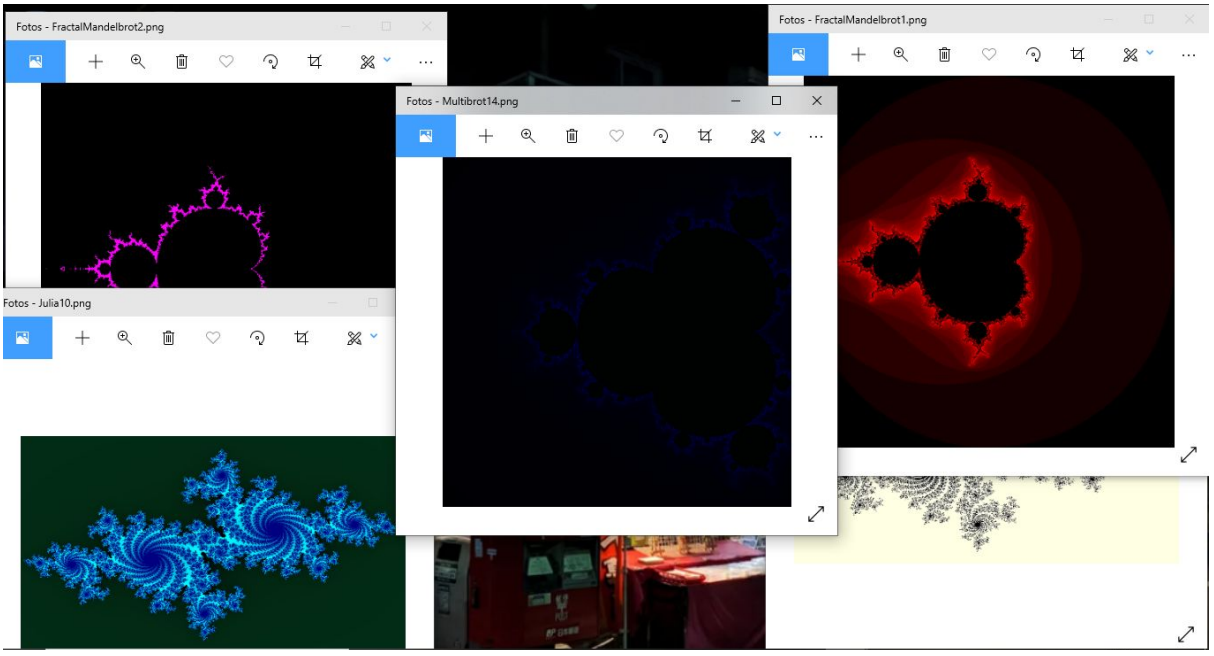
- **Diagrama de Casos de Uso**



Evidencias de implementación.



Programa en ejecución.



Fractales que se crean.

Conclusión.

La elaboración de este proyecto hemos podido implementar la gran mayoría de elementos y propiedades del paradigma orientado a objetos estudiados durante el curso desde tipos de datos abstractos, objetos, métodos y atributos, pasando por herencia, polimorfismo, encapsulamiento, entre otras, archivos y manejo de excepciones y errores, hasta llegar a implementar programación en paralelo, por ejemplo en la clase Mandelbrot creamos atributos que son propios de la clase, como el número de iteraciones que debe de realizar para que el fractal se vea bien definido, o el tamaño de la imagen que va tener. Implementamos los modificadores de acceso a ciertos atributos, ya sea haciéndolos privados o estáticos finales. Realizamos los métodos getters y setters respectivos, que son fundamentales en la Programación Orientada a Objetos, y dentro de ellas, también ocultamos variables de instancia, aplicando así el encapsulamiento, además desarrollamos la herencia de clases como en el caso de la propia clase Julia, ya que está hereda de Thread, y sobrecargamos uno de sus métodos que es el método run, hicimos manejo de excepciones y creamos excepciones propias y pudimos identificar como hacer uso de la jerarquía de estas en nuestro proyecto, para evitar usar excepciones generales. Utilizamos archivos para generar imágenes y lograr que estas se almacenarán en memoria, además, como se mencionó anteriormente, se utilizaron conceptos básicos de la programación paralela, esto para realizar 2 fractales ya que recurrimos al concepto del hilo, para hacer que cada uno de estos realizaran tareas independientes. Con todo lo anterior mencionado concluimos la realización de nuestro proyecto, en el cual aplicamos gran parte de los conceptos del paradigma orientado a objetos estudiados durante el curso, además de implementar una de las numerosas aplicaciones que este tiene y poder visualizar el potencial que este paradigma de programación posee, permitiendo mediante los objetos y la organización de datos, modelar entidades y comportamientos complejos del mundo real.

Bibliografía:

- Dejun Yan, et al. (2009). *General Mandelbrot Sets and Julia Sets Generated from Non-analytic Complex Iteration.* 25/01/2021. Dalian Nationalities University. Recuperado de: https://ieeexplore-ieee-org.pbidi.unam.mx:2443/stamp/stamp.jsp?tp=&arnumber=5362044&tag=1

- Bhanuka Manesha. (2020). *Efficient Generation of Mandelbrot Set using Message Passing Interface*. 27/01/2021. Monash University Malaysia. Recuperado de: https://arxiv.org/pdf/2007.00745.pdf

- Oracle. (2020). *Java™ Platform, Standard Edition 8 API Specification.* 25/01/2021, de Oracle. Recuperado de: https://docs.oracle.com/javase/8/docs/api/