Mat 2170

Algorithms & Methods

Spring 2014

Student Responsibilities

- Reading: Textbook, Chapter 5, 6.2
- Prelab & Lab
- Attendance
- **Lab 8, Exercise 4, Julia Set**:
  when publishing, select the 800 by 800 window size.

Applying Programming Tools to Problems

- Suppose you have learned how to use a hammer, sander, drill, and saw, and to apply polyurethane finishes to wood — by building a small bird house.

- Now, suppose further that you have been given the task of building a large roll–top desk. And that you will be judged on the sturdiness, usefulness, and elegance of this desk.

- How would you begin? Some sort of **plan** is needed.

- In programming, this plan is called an **algorithm**.

Algorithms

- **Algorithms** express the **logic** of a solution strategy: **the steps necessary to accomplish a task**.

- A programming problem should be broken down into logical sub–problems by finding a **general algorithm** — one that **outlines** your **overall** solution strategy.

- The algorithms for these sub–problems are then further refined into **specific algorithms** until they are easily implementable.

- **Specific algorithms** are implemented as **methods**

  General algorithms provide the **order and way** in which methods will be used to solve the problem.

Methods

- Methods are important in programming because

  1. they are the **building blocks** of a solution
  2. they allow for easier **re–use** of key blocks of code
  3. they give **meaningful names** to logical blocks of code
  4. their **interfaces** describe exactly the values needed and returned
  5. they allow us to more easily **solve large problems**, and to **test** our solutions for correctness

- Algorithms for solving a particular problem can vary widely in their **efficiency** — it makes sense to **think carefully** when developing an algorithm because making a bad choice can be extremely costly.

Where to Get Information on Classes and Methods

- read the textbook and slides; come to lecture (and stay awake!)
- **javadoc**: `jtf.acm.org/javadoc/student/index.html`
- using **netbeans** to inspect the **acmLibrary** files
- search the internet for information **(**`java.sun.com`**)**

It helps to know whether the class is part of **acmLibrary** or another library, such as **java.awt**

The GPoint Class

- The **acm.graphics** package provides the class **GPoint** which allows us to combine **two double** values into a single encapsulated unit.

- A **GPoint** can represent a point in the graphics window, a point in the Euclidean plane, or just a couple of related values.

- The primary **advantage** of encapsulating two individual values into a composite object is that the object can then **be passed** from one method to another as a single entity, via the **return** statement.

GPoint Constructor and Methods

`new GPoint(`x`, `y`)`
Creates a new `GPoint` object containing the coordinate values x and y.

`object``.`**getX**`()`
returns the x component of a `GPoint`

`object``.`**getY**`()`
returns the y component of a `GPoint`

`object``.`**setLocation**`(`x`, `y`)`
Changes the coordinates of the object to the point (x, y)

`object``.`**translate**`(`dx`, `dy`)`
Modifies the `GPoint` object by adding dx to its x coordinate and dy to its y coordinate.

GPoint Instantiation Examples


```
GPoint p = new GPoint(x, y);

GPoint WorldCenter = new GPoint(0.0, 0.0);

GPoint JuliaTerm = new GPoint(-0.9, 0.12);

GPoint Coord = new GPoint(1.0, 0.0);
```

Methods are able to return a `GPoint` object, for example:

```
return p;          // as created above, or
return new GPoint(Re, Im);
```

Accessing GPoint Coordinates

Given the method header:

```
public Color JuliaColor(GPoint p)
```

we can gain access to argument **p**'s `x` and `y` values by:

```
GPoint Z = new GPoint(p.getX(), p.getY());
```

Fractals: Julia Sets

- To every ordered pair of real values, (a, b), we can associate a **function** of two variables — referred to as the **Julia Map** for (a, b), and denoted by $F_{(a,b)}$

- $F_{(a,b)}$ is described by the formula:

$$F_{(a,b)}(x, y) = (\ x^2 - y^2 + a,\ \ 2xy + b\ ).$$

- Note: when $F_{(a,b)}$ is given a pair of coordinates, it produces another pair:

$$F_{(a,b)}(x, y) = (x', y')$$

Generating Sequences of Points

- For example, if $a = -1$ and $b = 0$:

$$\begin{aligned} F_{(-1,0)}(x, y) &= (x^2 - y^2 + a, 2xy + b) \\ &= (x^2 - y^2 + (-1), 2xy + 0) \\ &= (x^2 - y^2 - 1, 2xy) \end{aligned}$$

- We can start with a point $P_0 = (x_0, y_0)$ and compute the following sequence of coordinate pairs:

$$P_1 = F_{(a,b)}(P_0) = (x_1, y_1),$$
$$P_2 = F_{(a,b)}(P_1) = (x_2, y_2),$$
$$P_3 = F_{(a,b)}(P_2) = (x_3, y_3),$$
$$P_4 = F_{(a,b)}(P_3) = (x_4, y_4), \text{ etc.},$$

Julia Map Iterations Examples

The beginning sequences for $F_{(-1,0)}$ and three different initial points:

Iterations of $F_{(-1,0)}(x_0, y_0)$			
n	$F^n_{(-1,0)}(0.5, 0.5)$	$F^n_{(-1,0)}(0.5, 0.0)$	$F^n_{(-1,0)}(1.0, 0.0)$
0	(0.5, 0.5)	(0.5, 0.0)	(1.0,0.0)
1	(-1.0, 0.5)	(0.75, 0.0)	(0.0,0.0)
2	(-0.25,-1)	(0.438,0.0)	(-1.0,0.0)
3	(-1.938,0.5)	(-0.809, 0.0)	(0.0,0.0)
4	(2.504,-1.938)	(-0.346, 0.0)	(-1.0,0.0)
5	(1.516,-9.703)	(-0.880, 0.0)	(0.0,0.0)
6	**(-92.844,-29.411)**	**(-0.225, 0.0)**	**(-1.0,0.0)**

- Starting at (0.5, 0.5), by the sixth iteration the current point is out in the fourth quadrant of the plane, quite a distance (relatively) from the origin. Successive iterations will move it away even faster.

- On the other hand, starting at each of the other two sample points leads to sequences that stay pretty close to the origin.

- We observe **two qualitatively different types of behavior**. The sequence of points P_0, P_1, P_2, $P3$, . . . either:

    1. starts to **get farther and farther away** from the origin, or
    2. the sequence **stays pretty close** to the origin

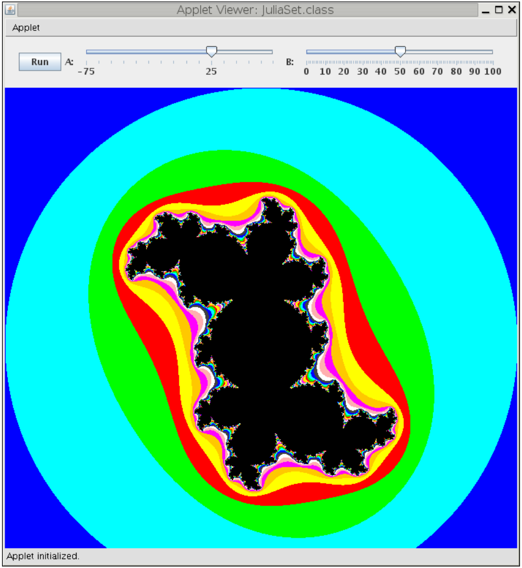
- The **Julia set** for (a, b), denoted by $J_{(a,b)}$, is the **collection of all points** in the plane from which you can start and **never get too far away** from the origin by repeated iterations of $F_{(a,b)}$.

- These sets turn out to be bizarre **fractal sets**. Different choices of (a, b) often give rise to quite exotic sets $J_{(a,b)}$.

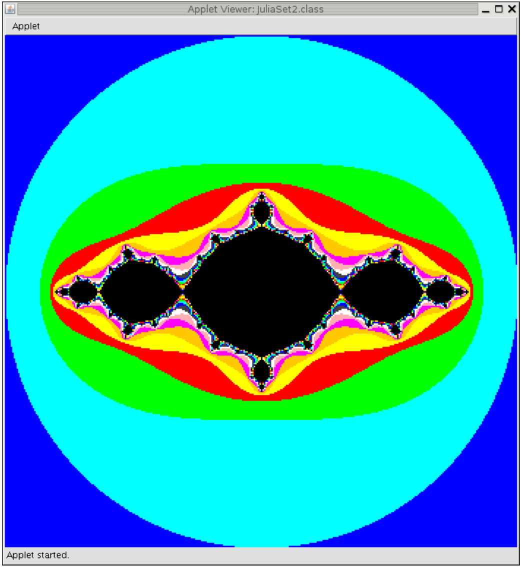
- One way to picture these is to color the points in the plane according to **how many iterations** it takes, starting from that point, to get outside a **threshold circle** (we use a radius of 2).

- The points that **don't get out** within a certain, preset number of iterations are the ones that are in the Julia set and they are colored **black**.

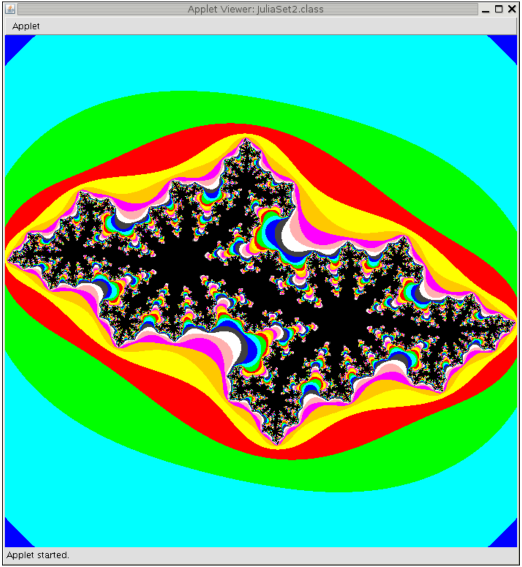
Julia Set - As Given — Note Orientation



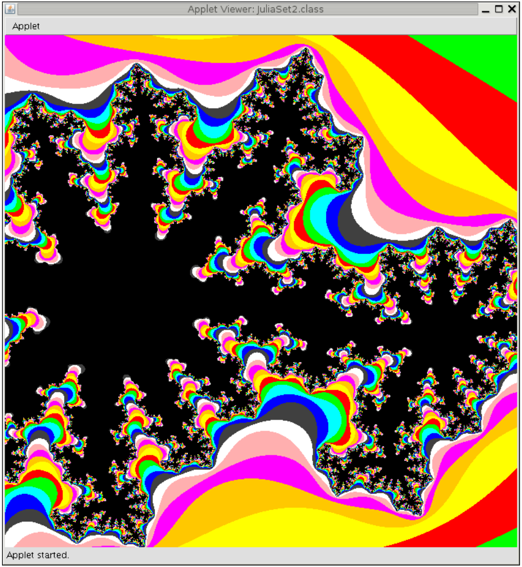
Julia Set - Modified a, b



Julia Set - Modified a, b



Julia Set - Modified World Size & Center



Lab 8 Exercise #4: Julia Set

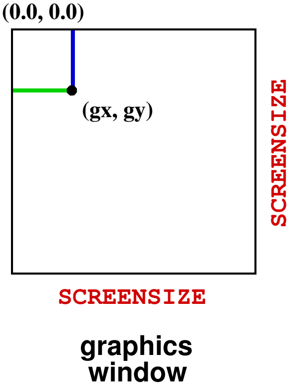
- A grid of tiny **square blocks** (much like the checkerboards we've already seen) are to be drawn in a **graphics window**.

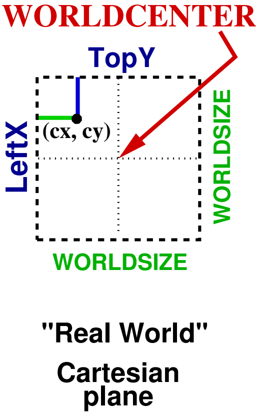
- These blocks **represent** a grid of **points** in a square **region of the Cartesian plane**.

- Each block is to be colored according to how the Julia set coloring algorithm would color the corresponding points in the plane.

(The Julia coloring code is provided for you.)

Translating between systems





Overview of the Program

- Tile the window as you would for a very large checkerboard.
- Each block's color depends on the Julia Color of its corresponding point in the Cartesian plan.
- So the main idea in this problem: **map each block** from the **graphics window** to its **corresponding point** in the "**world**" **region** that we are representing, then use the Julia color of that point for the color of the block.
- Do not leave a black border on the blocks — the picture is much brighter and easier to see if each block is pure color.

The Big Picture

For each block in the window grid:

1. find the upper left corner position (**BlockCorner()**)
2. find its corresponding point in the world region (**ScreenToWorld()**)
3. find the color this point (and hence the block) should get (**JuliaColor()**)
4. draw the block

BlockCorner()

1. Determines the coordinates of the block located at `row` and `column` in the graphics window.
2. It is **passed** two `int` parameters representing the current **row** and **column**.
3. It **returns** a `GPoint` representing the **location** of the block.
4. You are to complete this method.

ScreenToWorld()

1. Given a point in the window, determine the corresponding point in the "world."
2. It is **passed** a `GPoint` representing a **point in the graphics window**.
3. It **returns** a `GPoint` representing the coordinates of the **corresponding point in the world region**.
4. You are to complete this method.

Provided Methods

1. A skeleton and the method **JuliaColor()** are provided for you.
2. The method **Norm()** is used by **JuliaColor()**, and the method **NextPoint()** is just the Julia map mentioned earlier.
3. Do not modify these methods.

JuliaSet Constants

```
// Size of graphics window
public static final double SCREENSIZE = 700;

// Size of the real world (Euclidean plane) region
public static final double WORLDSIZE = 4.0;

// Center of the world region
public static final GPoint WORLDCENTER = new GPoint(0.0, 0.0);

// Number of rows and columns in screen grid
public static final int GRIDSIZE = 350;
```

```
// size of squares
public static final double BLOCKSIZE = SCREENSIZE / GRIDSIZE;


// Number of colors used for the display
public static final int MAXCOLORS = 11;


// Maximum number of iterations before a
//           number is declared in the Julia set
public static final int MAXITERATIONS = 40;


// Distance beyond which a point will not return
public static final double THRESHOLD = 2.0;
```

To Be Completed for Lab 8


```
// the coordinates of the world point corresponding
// to the screen position p

 public GPoint ScreenToWorld(GPoint p)
  {
    // Replace this code
    return new GPoint(0.0, 0.0);
  }



// position of block at row, col

  public GPoint BlockCorner(int row, int col)
  {
    // Replace this code
    return new GPoint(0.0, 0.0);
  }
```

To Be Completed


```
public class JuliaSet extends DualSliderProgram {

  public void init() {
    setSize(700, 795);
    super.init();
    setRangeA(-75, 75);
    setRangeB(0, 100);
  }

  public void run() {
    // the a and b of F_a,b(x,y)
    double a = getA() / 100.0;
    double b = getB() / 100.0;
    GPoint JuliaTerm = new GPoint(a, b);

   // add code to draw Julia blocks here

    }
  }
```