Programming Club
Fractals

Hugh Leather

10^{th} January 2018

1 Output to a file

We are going to draw our pictures into files. That means that first we need to be able to write into a file.

Task: "Hello, File!" Try to write a program to print "`Hello, File!`" to a file called `hello.txt`.

Below we'll look at all the bits you will need, step by step. At each step, try to work it out yourself first, making use of the online documentation, using these notes only if you need to.

`PrintStream` Fortunately, in Java, writing to files is pretty easy. You will probably want to look at the package documentation for the `java.io` package at some point (`https://docs.oracle.com/javase/8/docs/api/index.html?java/io/package-summary.html`).

The idea is that we create a `PrintStream`[1] which has operations like `println` that print text to a file. You have probably already seen a `PrintStream` when you use `System.out.println`. In that code, `System.out` is a `PrintStream` that prints to the standard output.

A main class. You will have to write a Java class. Let's call it `FileOutputTest`. This is going to be a public class (the public bit just means everyone can see it). In Java, any top level public class needs to go in a file of the same name with a `.java` extension. So, we'll put the class in file `FileOutputTest.java`.

This file is just going to have a `main` method which will be called when we run the program. We'll add more to it later.

```
public class FileOutputTest {
    public static void main(String[] args) {
        System.out.println("Hello, World!")
    }
}
```

For now, you should be able to compile this class with:

```
javac FileOutputTest.java
```

And then run it with:

```
java FileOutputTest
```

If all has gone well, it should print "`Hello, World!`"

Print to a file instead. To print to a file, you need to create a new `PrintStream`. You can do this by adding this to your `main` method:

```
java.io.PrintStream ps = new java.io.PrintStream("hello.txt");
```

Just a quick note: if you are using Java 10, you can just write

```
var ps = new java.io.PrintStream("hello.txt");
```

Now you can print to that stream:

```
ps.println("Hello, File!");
```

When you're finished with the file you have to close it, otherwise not all the text may get written:

```
ps.close();
```

[1] It would probably be a bit more modern to use a `PrintWriter`.

Exceptions. But, if you try to compile this, it won't work. It will complain that `FileNotFoundException` has not been handled. Java is quite picky about somethings. In this case it is saying that `new java.io.PrintStream("hello.txt")` can fail and someone had better agree to do something about it.

There are several ways we can fix this. The first is to say, hey someone else can deal with it. This means that it will be the problem of whoever calls `main`. This is okay, the `java` program will report the exception if it gets it. To do this, we say that `main` might throw this exception out. Then Java is happy again and we can compile it.

```
public static void main(String[] args) throws FileNotFoundException {
```

A better way is to catch the error do something sensible about it using a `try` and `catch`.
Even better is to use `try` with resource.
We might come back to those another time.

Imports When you use `PrintStream`, you have to either use its full name (`java.io.PrintStream`), or you can import that name so that thereafter you can just use `PrintStream` by itself. At the top of your Java file, you can write this to import the name.

```
import java.io.PrintStream;
```

But, we're probably going to use several names from that package, so we can bring them all in at once by using this import:

```
import java.io.*;
```

The final code This should be what we end up with:

```
import java.io.*;
public class FileOutputTest {
    public static void main(String[] args) throws FileNotFoundException {
        PrintStream ps = new PrintStream("hello.txt");
        ps.println("Hello, File!");
        ps.close();
    }
}
```

Compile it, run it, and check that it writes to `hello.txt`.

2 A first picture

The simplest picture format you might use is PPM. You can write out the image to a file in this format and then view it on your desktop. There several different PPM types, but the easiest for us is P3, which will allow everything to be written in text, rather than binary. It lets us build up pixels of red, green, and blue components quite simply.

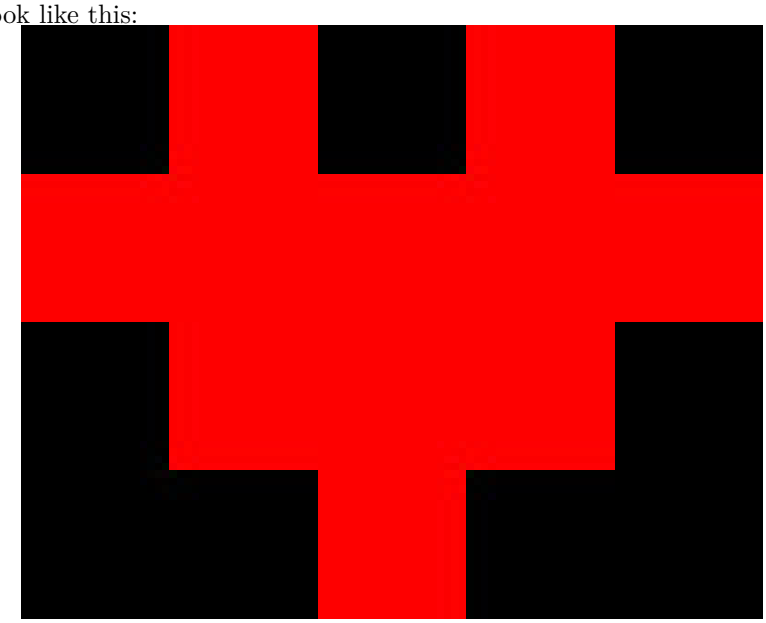
P3 format The basic structure of the file is like this:

```
P3
<width> <height>
<max>
<r> <b> <g>   <r> <b> <g>  <r> <b> <g> ...
<r> <b> <g>   <r> <b> <g>  <r> <b> <g> ...
```

All of these numbers are given as ordinary decimals. The `width` and `height` are the number of pixels wide and high the image is. `max` is the largest value any of red, green, and blue could take. I.e. fully on. A simple value for this is 255, which will work for bytes and give you 2^24 different colours. Each of `r`, `g`, and `b` are values between 0 and `max` inclusive, giving the amount of that colour for the appropriate pixel. The pixels are arranged so that the rows are done first, with the very first pixel being the top left of the image.

Between each element of the file, there has to be some whitespace. These can be any number of spaces, tabs, or newlines. **The last character of the file has to be whitespace!**

Task: Make a heart Since Valentine's day is coming soon, make a very small image, just using your text editor. It should look like this:



I admit, it's not a very good heart.

The heart file Here's what the file should look like. Remember to have some whitespace at the very end.

```
P3
5 4
255
0 0 0       255 0 0     0 0 0       255 0 0     0 0 0
255 0 0     255 0 0     255 0 0     255 0 0     255 0 0
0 0 0       255 0 0     255 0 0     255 0 0     0 0 0
0 0 0       0 0 0       255 0 0     0 0 0       0 0 0
```

Task: A Java heart Use Java to write out the heart file instead.

Add two static variables for the height and width of the picture. Add three arrays of `floats` for the red, green, and blue pixels (we'll use `0.0` for black and `1.0` for fully read, green, or blue). Add a function to convert from a `float` between `0.0` and `1.0` into an integer between 0 and 255. Add a function to write the image to a file. Write it.

Adding some variables Let's make a new class and add some variables to it. Put this code into `Heart.java`.

Note that all the variables are `static`. This makes them available from `static` methods like main.

Also, we could have used a multidimensional array for each of the colours, but this works as well. You have to convert though from your `x` and `y` coordinates into a single index.

```
import java.io.*;

public class Heart {
    static int width = 5;
    static int height = 4;

    static float[] red = new float[width * height];
    static float[] green = new float[width * height];
    static float[] blue = new float[width * height];

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

Setting a pixel Let's add a function to set the value of a pixel.

Note that we convert x and y to an index - we should probably do some bounds checking there, eh?

```
static void set(int x, int y, float r, float g, float b) {
    int i = x + y * width;
    red[i] = r;
    green[i] = g;
    blue[i] = b;
}
```

Convert a channel value to an `int` Converting channels (i.e. values of red, green, or blue pixels) into integers for the file isn't too bad.

We want `0.0` to map to `0`.

We want `1.0` to map to `255`.

We want an even spread between.

So this equation will do:[2] $colour \to colour \times 255$

```
static int colourToInt(float c) {
    return (int)(c * 255);
}
```

Now you will notice that there is a bit in there which "casts to an int". That is because Java complains if you try to convert from one type to another where you might lose information. The cast says that this really is what you want.

Writing out an image Let's add a method called `write` to dump the image to a file. It should take a file name as a string.

Note that this code might throw and exception, so we need to let anyone using it know that.

```
static void write(String fileName) throws FileNotFoundException {
    PrintStream ps = new PrintStream(fileName);
    ps.println("P3");
    ps.println(width + " " + height);
    ps.println(255);

    for(int i = 0; i < width * height; ++i) {
        ps.print(colourToInt(red[i]) + " ");
        ps.print(colourToInt(green[i]) + " ");
        ps.print(colourToInt(blue[i]) + " ");
    }

    ps.close();
}
```

[2] Actually this isn't quite right since really we want the interval $(1 - \epsilon, 1]$ to map to 255. What would ϵ be? How would you change the rest?

Making the heart Let's fill up the pixels in the `main` method and then write out the file.

Note that the array values are filled with zeros to start with, so we only have to set the red pixels.

```
public static void main(String[] args) throws FileNotFoundException {
    // Set the pixels
    set(1,0, 1,0,0);
    set(3,0, 1,0,0);

    set(0,1, 1,0,0);
    set(1,1, 1,0,0);
    set(2,1, 1,0,0);
    set(3,1, 1,0,0);
    set(4,1, 1,0,0);

    set(1,2, 1,0,0);
    set(2,2, 1,0,0);
    set(3,2, 1,0,0);

    set(2,3, 1,0,0);

    // Write out the file
    write("heart-from-java.ppm");
}
```

3 A smooth image

See if you can make an image, 200 by 100 pixels, that looks like this:

Looping over the pixels This might look pretty easy, but there's a little trap waiting for you, which we'll get to in a bit.

First of all copy the previous code into a new file and change the class name. Maybe `Smooth.java` and `Smooth` respectively. Then you'll need to change the width and height of the image:

```
import java.io.*;

public class Smooth {
    static int width = 200;
    static int height = 100;
    //...
```

Now we replace the main method which builds the image (including sending it to a different file). We'll make a couple of `for` loops. The first will iterate over the `x` axis, the second will iterate over the `y` axis.

```
public static void main(String[] args) throws FileNotFoundException {
    for(int x = 0; x < width; ++x) {
        for(int y = 0; y < height; ++y) {
            // Set the colour for (x, y) here
        }
    }
    // Write out the file
    write("smooth.ppm");

}
```

How do you set the colour for `(x, y)`? Well, you want the `x` axis to change the red channel from 0 to 1, and the `y` axis to change the blue channel from 0 to 1. So it should look like this, right?

```
set(x,y, x/width,0,y/height);
```

Try it. What happens?

Watch out for integer division Hmm, the last thing made an image of the right size, but all the pixels were black. What went wrong?

Well, if you look at the types of `x` and `width`, you will see they are both integers. Integer division rounds down, so instead of a number between `0.0` and `1.0`, you get `0`. Then that is converted from an integer version of zero to floating point version of zero - i.e. `0.0`.

What we need to do is to tell Java that we really want to do this with floating point division, not integer division.

The easy way is to tell it that one of the numbers is a `float`:

```
set(x,y, x/(float)width,0,y/(float)height);
```

Phew, now everything works.

4 An Image Object

So this code that we've written is okay. But it's a bit painful to use. Each time you need a new picture you have to make a new class file for the image. And, you have to manually copy over all the `width` and `height` fields, as well as the methods we wanted, like `set` and `write`. It is hardly ideal, is it?

I guess what we'd like is to be able to have lots of images kicking around all at once, but have them share most of the functionality. Java makes this easy by giving us objects.

With the `static` fields in the classes we've made, there is only one copy per class. With objects, there is a copy per object, and you make as many objects as you like, even inside a loop. In fact, Java expects you to use objects so much that they are the default, you have to mention if something is static specifically.

Make an Image class. Try to work out how to do this yourself!

Okay, I'll show you :-) First, let's make a new Java file for our class. We'll call it `Image.java`.

Copy in everything from the other files except the `main` method and don't use the word `static`. Non static things belong to a particular object.

In fact, wherever you had `static`, use the word `public`. This means that Java will let you use that name from outside the object. I think Volker is going to discuss both `static` and `public` in Inf1OP at some point, so I won't go into them in much detail.

```
import java.io.*;

public class Image {
    public int width;
    public int height;

    public float[] red = new float[width * height];
    public float[] green = new float[width * height];
    public float[] blue = new float[width * height];

    public void set(int x, int y, float r, float g, float b) {
        int i = x + y * width;
        red[i] = r;
        green[i] = g;
        blue[i] = b;
    }

    public int colourToInt(float c) {
        return (int)(c * 255);
    }
    public void write(String fileName) throws FileNotFoundException {
        PrintStream ps = new PrintStream(fileName);
        ps.println("P3");
        ps.println(width + " " + height);
        ps.println(255);

        for(int i = 0; i < width * height; ++i) {
            ps.print(colourToInt(red[i]) + " ");
            ps.print(colourToInt(green[i]) + " ");
            ps.print(colourToInt(blue[i]) + " ");
        }

        ps.close();
    }

}
```

Make a new image So now we we can make as many images as we want. But how do we use them?

Let's make the *smooth* image using our object class. We'll put it in class `SmoothObj`

To make a new object, use `new`. For the image, we would use:

```
Image img = new Image();
```

Now wherever previously you used a static variable or method, call `img`'s version instead.

```
import java.io.*;

public class SmoothObj {

    public static void main(String[] args) throws FileNotFoundException {
        Image img = new Image();
        img.width = 200;
        img.height = 100;

        // Set the pixels
        for(int x = 0; x < img.width; ++x) {
            for(int y = 0; y < img.height; ++y) {
                img.set(x,y, x/(float)img.width,0,y/(float)img.height);
            }
        }
        // Write out the file
        img.write("smooth.ppm");
    }
}
```

You could now have as many images as you'd like.

Eeek, this didn't work! Why not?

An image constructor When you made a new new `Image`, did you notice that there were parentheses after `Image`? This might be a bit suggestive to you that maybe you could put function parameters there. Well, yes, you can.

We currently have to tell Java after we constructed the `Image` what it's width and height are. That could be dangerous.

In fact, it was dangerous for us! You may have noticed that you got an `java.lang.ArrayIndexOutOfBoundsException` when running the previous code. The reason for this is that we didn't set the `Image`'s `width` or `height` before making the arrays for `red`, `green`, and `blue`. In java, when you don't initialise a field, it is set to `0`, if it's a number, or `null` if it's an object. So, `width` and `height` were both zero when the arrays were created. Then, when we tried to set a pixel, the index we chose 'fell off' the end of the array.

By using constructors and then not letting other people access critical bits of a class, we can make sure that things are always in a good state. A constructor is an object method with special name, the same name as the class. Also, it doesn't return anything, you can only call it during `new`, and there are some other rules about what they have to do which we won't worry about just now.

Let's make a constructor for our `Image` class that takes the `width` and `height`, and then creates the arrays for the colours.

```
public class Image {
    public int width;
    public int height;

    public float[] red;
    public float[] green;
    public float[] blue;

    public Image(int w, int h) {
        width = w;
        height = h;
        red = new float[width * height];
        green = new float[width * height];
        blue = new float[width * height];
    }
    ...
```

Great! But now the `new` call in `SmoothObj` doesn't work because we don't have a constructor which takes no arguments. Let's change that.

```
import java.io.*;

public class SmoothObj {

    public static void main(String[] args) throws FileNotFoundException {
        Image img = new Image(200, 100);

        // Set the pixels
        for(int x = 0; x < img.width; ++x) {
            for(int y = 0; y < img.height; ++y) {
                img.set(x,y, x/(float)img.width,0,y/(float)img.height);
            }
        }
        // Write out the file
        img.write("smooth.ppm");
    }
}
```

Finally secure BTW, the things we're doing here, up until we talk about Mandelbrots, are just to make things better and to show you more Java. If you want to just get on with the pretty pictures, you could skip ahead to Mandelbrots and try to make things work like that.

So, we have a constructor, but we still aren't entirely safe. What if someone changes the `width` after they've made the image? Well, then all hell could break loose. Maybe we should stop them doing that.

In Java, there is a simple way to stop that. We can tell Java that once a field is initialised then it can't ever be changed again. We do that with the `final` keyword.

```
public class Image {
    public final int width;
    public final int height;

    public final float[] red;
    public final float[] green;
    public final float[] blue;
```

Do that and see what happens if you try to change `width`, say by adding one to it.

Complaining about crazy people Okay, so we've protected against crazy people trying to mess with things in the wrong order, but we still aren't safe. What would happen if someone used a negative `width` for the `Image`? Or what if someone tries to set the pixel at `(-100,5000)`? What if they use colour values outside of the range `0.0` to `1.0`? They would probably make the universe explode (try it?).

Maybe we should check for this bad things and report something sensible to the user. What we will do is throw an exception. That means we tell Java to stop what it's doing and report a problem that someone has to fix or the stop the program if no one does. We've already seen that making a `PrintStream` can throw a `FileNotFoundException`. It is just like that, except for two things. 1) we will throw an `IllegalArgumentException` instead. 2) Java isn't as persnickety about that exception, so we don't need to add it to the list in `main`, or anywhere else. [3]

```
public Image(int w, int h) {
    if(w <= 0) throw new IllegalArgumentException("width must be > 0");
    if(h <= 0) throw new IllegalArgumentException("width must be > 0");
    //...
```

Now, if someone tries to make a badly sized image, they will get a nice error about it. Try it.

Can you make anything else in the `Image` class safe this way?

[3] Java has two types of exception. One has to be handled or the compiler won't compile your program. They other type (called a `RuntimeException`) doesn't. It's an historical thing, mostly, and not a little bit annoying.

5 Objects, Objects, Everywhere!

Objects are kind of useful in Java. They let us make lots of things easier and more natural. Let's turn our attention to colours.

Colour So far, we've been using three separate colour channels to represent our three colours. This means you can't define a constant for mauve, say, and then pass it into the `set` method on the image. You still have to put in the three colours individually. It would be much better if we had an idea of a colour object instead that encapsulated the three colour channels for us.

So, let's create a `Colour` class. What do you think it should look like?

Let's put it in `Colour.java`.

```
public class Colour {
    public final float r, g, b;

    public Colour(float r, float g, float b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Now, you might notice that there's a special name used here, `this`. It refers to the current object. You can imagine that there is an extra parameter to a non `static` method, called `this` that points to the current object. In fact, under the covers, this is exactly what Java does.

```
public Colour(Colour this, float r, float g, float b)
```

So, now, when we refer to `this.r`, it means the `r` belonging to the object itself, not the `r` passed in. This is great because it means we can have the same name appear twice without getting confused.

Oh, notice that we made the `r`, `b`, `g` fields of the `Colour` object all `final`. This means that `Colour`s are *immutable*. Immutable objects are a bit safer since they are harder to mess up. Consider if we added some `static` fields into the `Colour` class for common colours.

```
public class Colour {
    ...
    public static final Colour BLACK = new Colour(0,0,0);
    public static final Colour WHITE = new Colour(1,1,1);
    public static final Colour RED = new Colour(1,0,0);
    ...
}
```

Notice we made those fields `final` so no one can swap out `BLACK` for a different colour which would certainly confuse people trying to use `BLACK` afterwards.

Oh, also notice that this *constant* is spelled with all capital letters. This is standard Java naming practice. You should use the standard coding conventions of whatever language you are using - it tends to annoy other developers if you don't.

But, now imagine that the `r`, `b`, `g` fields of the `Colour` object weren't `final`. Now you could change `BLACK` into red, which probably wouldn't be the best idea:

```
Colour.BLACK.r = 1;
```

While we're here, let's make another constructor, this time for gray scale colours. It will only take one value, the amount of gray, and will call the other constructor to do it's work.

```
public class Colour {
    ...
    public Colour(float c) {
        this(c, c, c);
    }
    ...
}
```

Did you notice the odd little notation to call another constructor from the same class?

Can you finish off the `Colour` class? Add more standard colours. Add some error checking so people don't give crazy numbers to the constructor.

An `Image` is made of `Colours` Let's go ahead and make the `Image` class have a single array of `Colours` rather than individual channels. Actually, why don't you try that before we show you? (Don't look below til you've tried it.)

Now, we'll see that, but I'm going to make a few other changes as well which I'll explain afterwards.

```
public class Image {
    public final int w;
    public final int h;

    private final Colour[] pix;

    public Image(int w, int h) {
        if(w <= 0) throw new IllegalArgumentException("width must be > 0");
        if(h <= 0) throw new IllegalArgumentException("width must be > 0");
        this.w = w;
        this.h = h;
        pix = new Colour[w * h];
        for(int i = 0; i < w * h; ++i) pix[i] = Colour.BLACK;
    }

    public void set(int x, int y, Colour c) {
        if(x < 0 || x >= w || y < 0 || y > h) throw new IllegalArgumentException("Cannot access pixel,"+x+","+y)
        if(c == null) throw new IllegalArgumentException("Cannot set pixel to null");
        pix[x + y * w] = c;
    }
    public Colour get(int x, int y) {
        if(x < 0 || x >= w || y < 0 || y > h) throw new IllegalArgumentException("Cannot access pixel,"+x+","+y)
        return pix[x + y * w];
    }

    private int colourToInt(float c) {
        return (int)(c * 255);
    }

    public void write(PrintStream ps) {
        ps.println("P3");
        ps.println(w + " " + h);
        ps.println(255);

        for(Colour p : pix) {
            ps.print(colourToInt(p.r) + " ");
            ps.print(colourToInt(p.g) + " ");
            ps.print(colourToInt(p.b) + " ");
        }
    }

    public void write(String fileName) throws FileNotFoundException {
        PrintStream ps = new PrintStream(fileName);
        write(ps);
        ps.close();
    }
}
```

What's different here?

First, I changed the names of `width` and `height` to `w` and `h`, since we now know we can reuse names. This is a pretty pointless change really, but it's good to see where you need to use `this.w` and `this.h`.

Then you'll see that the arrays, `r`, `b`, and `g` have been replaced by a single array of `Colours`, called `pix`. In the constructor you'll see that gets initialised with `new`.

The constructor then fills in each element of `pix` with `Colour.BLACK`. That is because a newly created array of objects will have each element set to `null`. We'd much rather the elements had sensible colours to start with.

Now you might also see that the `pix` array is declared `private`, not `public`. This prevents any code except that written in the `Colour` class from messing with the array. If we don't let anyone else change the array, then no one can put `nulls` in it which might break our code.

The `set` method now takes a `Colour`, rather than individual channels.

Since we have made the `pix` field private, no one else can access it to write it. But they also can't read it. The `get` method lets people see what's in an image's pixels safely.

I also changed `colourToInt` to be `private` because it's only really for the `Image` class's internal use.

Finally, I made two versions of `write`. One works on `PrintStream`s, the other does the sensible thing with a file name. In Java, you can have two methods with the same name if they have different parameter types. Notice that the file name version calls the `PrintStream` version. You can now easily print an image to standard output as well as to files.

Filling an image I had another motivation for wanting use `Colour` objects rather than three channels. It means that we can return colours from a method. This lets us do something cool.

At the moment, you have to set pixels individually or write a couple of nested for loops, iterating over `x` and `y`. That latter form is likely to be a common thing to do but it's a bit of pain.

Suppose we had a type of object with a function that takes an `Image`, `x` and `y` and gives a `Colour`. Maybe it looks like this (put this in a file called `PixelFn.java`):

```
public interface PixelFn {
    Colour apply(Image img, int x, int y);
}
```

We could then have a method in our `Image` class which takes one of these functions and fills in the pixels:

```
public void fill(PixelFn f) {
    for( int x = 0; x < w; ++x) {
        for( int y = 0; y < h; ++y) {
            set(x, y, f.apply(this, x, y));
        }
    }
}
```

That would mean that we could fill an entire image with just one line of code if we had a suitable `PixelFn`.

But, wait a second! What is all that going on with `PixelFn`? I said it was a *class*, but it says `interface`. And, the method doesn't have an implementation, and the method isn't `public`.

An interface is a very special type of type which gets added to other classes to let you share types. We might use it like this:

```
public class Reddish implements PixelFn {
    public Colour apply(Image img, int x, int y) {
        return Colour(x/(float)img.w);
    }
}
```

This says `Reddish` is also an PixelFn, so you can use it wherever you would use an `PixelFn`.

```
public static void main(String[] args) {
    Image img = new Image(200, 100);
    img.fill(new Reddish());
    img.write(System.out);
}
```

That looks a bit better, but Java has something really cool to make this even better.

Lambda functions! Java has a pretty cool way of making these little classes which are only used to pass forward a method. They are called *lambda*s. They are definitely worth you while Googling about. Here, I'm just going to show you how we can use them, but I won't go into too much explanation (you can always ask a helper).

We do two things. The first is to declare that `PixelFn` is one of these special little classes. Java uses something called an *annotation* for this. An annotation just tags a class with some extra information. To do it for `PixelFn`, we add this:

```
@FunctionalInterface
public interface PixelFn {
    Colour apply(Image img, int x, int y);
}
```

Now, when we use it in the `fill` method, we can create an `PixelFn` with some pretty cool syntax:

```
public static void main(String[] args) {
    Image img = new Image(200, 100);
    img.fill(
        (im, x, y) -> new Colour(x/(float)im.w, y/(float)im.h, 0)
    );
    img.write(System.out);
}
```

That bit inside of `fill` implicitly makes an `PixelFn` whose `apply` method makes that `Colour`. It's really quite neat.

Now that we've got this, we can also add another constructor to `Image`, since often we'd want to fill the image as soon as we make it.

```
public class Image {
    ...
    public Image(int w, int h, PixelFn f) {
        if(w <= 0) throw new IllegalArgumentException("width must be > 0");
        if(h <= 0) throw new IllegalArgumentException("width must be > 0");
        this.w = w;
        this.h = h;
        pix = new Colour[w * h];
        fill(f);
    }
    ...
}
```

Now, that whole thing to make our smooth image becomes:

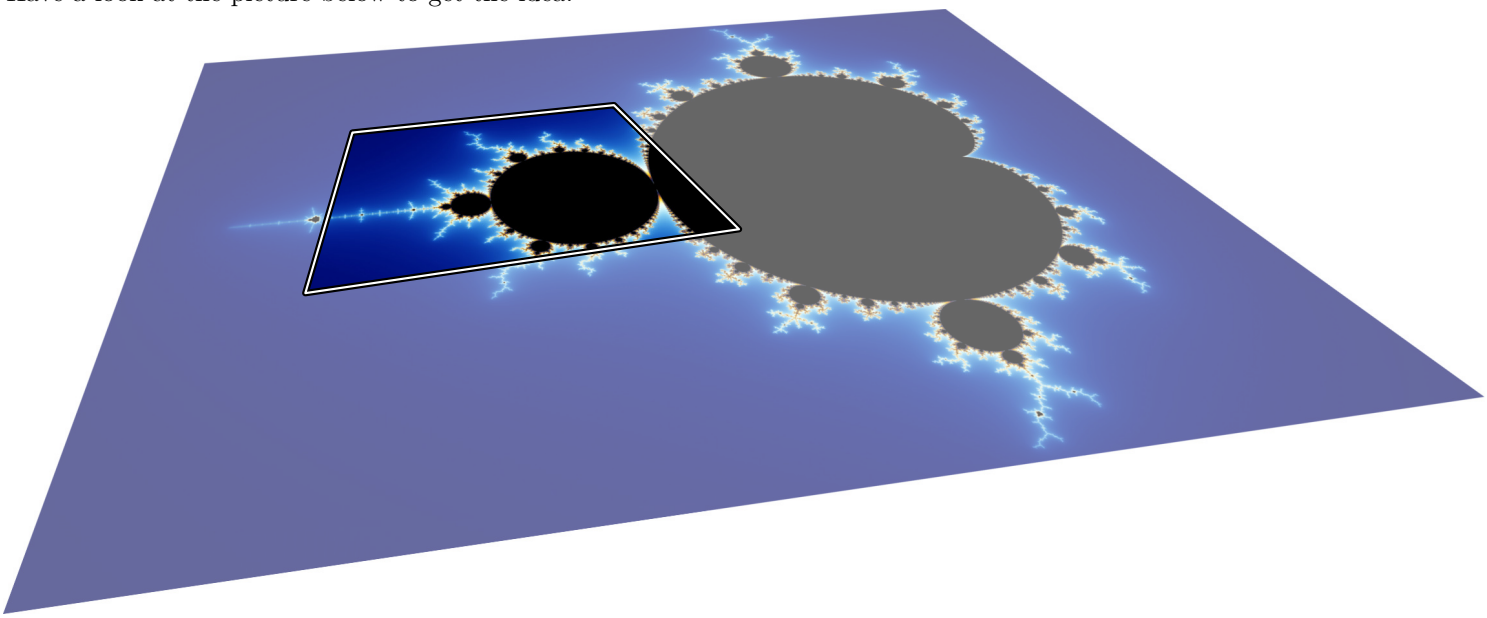
```
import java.io.*;

public class SmoothLambda {
    public static void main(String[] args) throws FileNotFoundException {
        new Image(200, 100, (im, x, y) -> new Colour(x/(float)im.w, y/(float)im.h, 0)).write("smooth.ppm");
    }
}
```

BTW, the `apply` method's name isn't important. The name comes from functional programming terminology.

Windowed fill The images we have drawn are 2d arrays of pixels. But what they represent isn't necessarily pixelated. You may have heard about Mandelbrots, for example. They define a 2 dimensional real valued space[4]. At each point in the space we can ask what colour that point is. It isn't a discreet space, like the bitmap type images we have drawn so far, made up of little squares. Instead you can look as closely or as far away as you like.

So, suppose we have an image function which takes in an `x` and `y` both of type `float` and returns a `Colour`. We want to make the `Image` bitmap be a little window onto this real valued image function. This isn't too different from the `PixelFn` we had before, except it uses floats and doesn't take the `Image` as a parameter. It shouldn't need the `Image` since it should never change itself because of the window.

We then like to fill an `Image` by giving it one of these functions and telling it the rectangle of the window we want to look at. Have a look at the picture below to get the idea.



The `Image` we fill will be the little rectangle in the middle. The number of pixels in each dimension are given by the `Image`'s width and height. We can move the rectangle around and 'see' different bits of the underlying function. Do you think you could add these capabilities to the `Image` class? Try it.

Let's start by making the `ImageFn`. Remember to put this in a file called `ImageFn.java`.

```
@FunctionalInterface
public interface ImageFn {
    Colour apply(float x, float y);
}
```

Now for the `fill` method, we need to take the `ImageFn` and the coordinates of the window on the `ImageFn`. We could (and probably should) do that by making a `Rectangle` class, but for the moment I'll just take the values for the *top*, *left*, *bottom*, and *right* of the window individually.

```
public class Image {
    ...
    public void fill(ImageFn f, float lft, float top, float rgt, float bot ) {
        for( int x = 0; x < w; ++x) {
            float fw = rgt - lft;
            float fx = lft + x * fw / w;
            for( int y = 0; y < h; ++y) {
                float fh = bot - top;
                float fy = top + y * fh / h;
                set(x, y, f.apply(fx, fy));
            }
        }
    }
    ...
}
```

The important bit is that we've mapped from the bitmap coordinates, `x` and `y`, into the `ImageFn` coordinates, `fx` and `fy`[5] Then we called the `ImageFn`, `f`, to get the colour and set the appropriate pixel in the `Image`.

[4] Actually the coordinate system is given by complex numbers.

[5] I'm beginning to think `BitMap` or something might have been a better name than `Image`. You could always change the names, of course.

We should probably make a constructor like this as well.

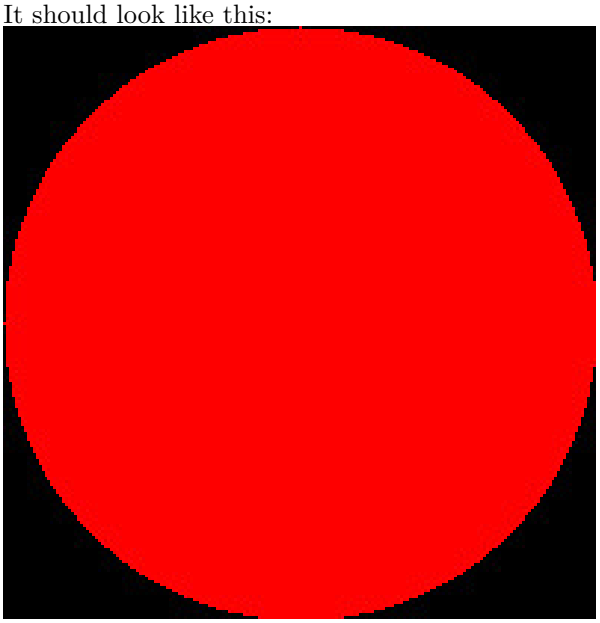
```
public class Image {
    ...
    public Image(int w, int h, ImageFn f, float lft, float top, float rgt, float bot ) {
        if(w <= 0) throw new IllegalArgumentException("width must be > 0");
        if(h <= 0) throw new IllegalArgumentException("width must be > 0");
        this.w = w;
        this.h = h;
        pix = new Colour[w * h];
        fill(f, lft, top, rgt, bot);
    }
    ...
}
```

Shall we use it? Can you make an ImageFn that represents a filled in red circle of radius one at the origin with black elsewhere? Can you then make an `Image` of that with a suitable window?

Here we go:

```
public class CircleTest {
    public static void main(String[] args) {
        new Image(
            200, 200,
            (x,y) -> x*x + y*y > 1 ? Colour.BLACK : Colour.RED,
            -1, 1, 1, -1
        ).write("circle.ppm");
    }
}
```

It should look like this:



I bet you can come up with a whole bunch of cool things to add to these classes. You should, give it a go!

But, for the moment, we will stop playing with this and move on to drawing some pretty pictures

Other things Here some ideas of things you might try.

- Anti-aliasing - allow supersampling of the `ImageFn` so that the circle doesn't get so jaggy.
- What about having an alpha channel (transparency).
- Can you make transforms from one image to another. E.g. grayscale, colour shift, scale, rotate, etc?
- Can you make methods to compose images?
- There is also a P6 version of PPM which is more compact, it uses binary. Can you use that instead?
- Can you write directly to a different file format?
- Skip this lot and look at how Java supports images.
- Can you make a GUI to show moving images?
- Can you do the same thing in HTML, using JavaScript and an HTML Canvas object?

6 Mandelbrots

We're going to make a very famous type of fractal called a Mandelbrot. The Mandelbrot set is those complex points, c, for which $f_c(z) = z^2 + c$ does not diverge when iterated. What that means is that if you view a complex number as a point on a 2 dimensional plane – the real component is the `x` axis, and the imaginary component is the `y` axis – then we can tell whether that point is in the set by repeatedly applying that equation forever and seeing if it grows without bound or not.

Well, obviously doing things for ever is a tiny bit tedious. So instead what we'll do is repeat the equation a maximum number of times and see if the complex number gets bigger than some threshold in that time. This is called the escape time algorithm.

In fact, some numbers 'escape' more quickly than others, so we can use that 'speed of escape' to assign a colour to the points which aren't in the set.

That is, for a given point, c, start with

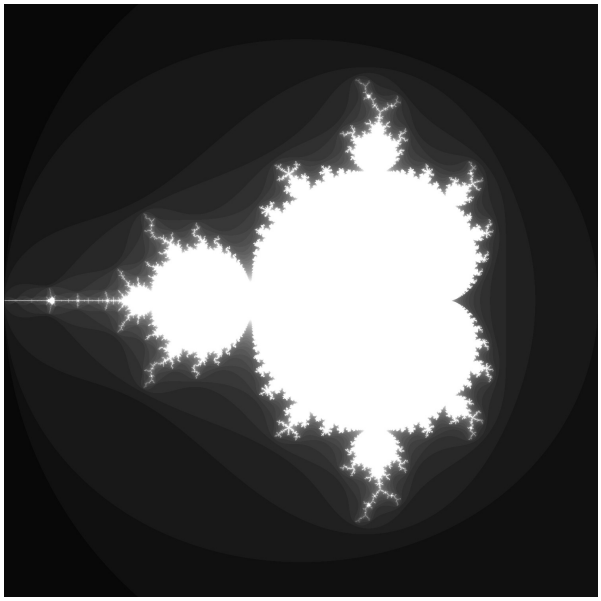
$$z_0 = 0$$

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

Now iterate and find out the first n for which $|z_n| > 2$ or $n > 32$ (the latter bit stops you going on forever). You can change the 2 and the 32 if you like.

This gives you number, n, in the interval, $[0, 32]$. You can convert this number into a colour, making 0 black, 32 white, and interpolating in between.

Now imagine your image ranges over the complex space, with the x axis ranging from -2 to 1, and the y axis ranging from -1.5i to 1.5i. You can now plot the Mandelbrot set for this part of the space. It should look like this:



Have a go at doing this yourself, then I'll walk you through my solution.

Complex numbers, a reminder It's pretty clear we're going to need complex numbers here. Unfortunately, Java doesn't provide them out of the box. There are a bunch of online implementations - maybe you could search for some. But we also should have learned enough to be able to make our own. Let's do that.

A quick reminder about what complex numbers are.

A complex number has a real and imaginary component:

$$z = a + bi$$

Where i is the imaginary square root of -1.

We can extract those components in some mathematical notation:

$$z = a + bi \rightarrow re(z) = a$$

$$z = a + bi \rightarrow im(z) = b$$

We can add and multiply complex numbers (parentheses are just make doubly clear which bits are real and imaginary):

$$z_1 = a_1 + b_1 i, z_2 = a_2 + b_2 i \rightarrow z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$$

$$z_1 = a_1 + b_1 i, z_2 = a_2 + b_2 i \rightarrow z_1 z_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i$$

This let's us square a number:

$$z = a + bi \rightarrow z^2 = (a^2 - b^2) + 2abi$$

We can also ask the magnitude of a complex number:

$$z = a + bi \rightarrow |z| = \sqrt{a^2 + b^2}$$

That should give us everything we need to implement a complex number class.

`Complex` class Let's make a file called `Complex.java` and put our class in it. Like `Colour`, we'll make it *immutable* for safety. Can you make the right fields and constructors[6]?

```
public class Complex {
    public final float re;
    public final float im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }
    public Complex(float re) {
        this(re, 0);
    }
}
```

What about the other operations? Magnitude is easy.

But let's also have a method called `magSq` which will give us the magnitude squared. Why? Well, often you can avoid doing the expensive square root operation. Consider the check we will need to do to find out if our magnitude is greater than 2. Instead we could check if the squared magnitude is greater than 4. No square root required.

```
public class Complex {
    ...
    public float magSq() {
        return re * re + im * im;
    }
    public float mag() {
        return (float)Math.sqrt(magSq());
    }
    ...
```

Did you notice that we had to cast the `double` result from `Math.sqrt` into the smaller `float`?

There are different ways we could do the add and multiply. We could have a `static` method which takes two complex numbers, or we could have an instance member which takes the other number to use. Or we can have both and let the user decide which to use. Neither are going to look as good as languages which allow operator overloading, but we can't help that in Java.

[6] Uggh, why doesn't Java allow default arguments in constructors? This is why I hate this language. One of the reasons, anyway.

```
public class Complex {
    ...
    public static Complex add(Complex a, Complex b) {
        return new Complex(a.re + b.re, a.im + b.im);
    }
    public Complex add(Complex that) {
        return add(this, that);
    }
    public static Complex mul(Complex a, Complex b) {
        return new Complex(a.re * b.re - a.im * b.im, a.re * b.im + a.im * b.re);
    }
    public Complex mul(Complex that) {
        return mul(this, that);
    }
    ...
```

Oh, we can add some constants, too.

```
public class Complex {
    ...
    public static final Complex ZERO = new Complex(0);
    public static final Complex ONE = new Complex(1);
    public static final Complex I = new Complex(0, 1);
    ...
```

The Mandelbrot escape function Now we are ready to make our escape function. Can you do it? It should take a complex point and tell you how long it takes to escape.

We can make a `Mandelbrot` class. This should take our escape threshold and maximum iterations as parameters to the constructor. Then we'll have an `escapeTime` function which does the actual calculation.

```
public class Mandelbrot {
    public final float threshold;
    public final int maxIter;

    public Mandelbrot(float threshold, int maxIter) {
        this.threshold = threshold;
        this.maxIter = maxIter;
    }

    public int escapeTime(Complex c) {
        Complex z = Complex.ZERO;
        int n = 0;
        float thresSq = threshold * threshold;
        while(n < maxIter && z.magSq() < thresSq) {
            z = z.mul(z).add(c);
            n++;
        }
        return n;
    }
}
```

Colour mapping We need to map from the escape count to a colour. There are lots of ways we might do this, but we said we'd at least start by doing it in grayscale. Can you make a class to do that for you?

Now, I'm going to make this class map floating point numbers from a range into colours. Really I'm just doing this because maybe it might be useful later.

Hmm, maybe we should make an interface for colour mapping so that we could have different colour maps?

```
@FunctionalInterface
public interface ColourMap {
    Colour apply(float x);
}
```

Now, our gray colour map is:

```
public class GrayColourMap implements ColourMap {
    public final float min;
    public final float max;

    public GrayColourMap(float min, float max) {
        this.min = min;
        this.max = max;
    }
    public Colour apply(float x) {
        float g = (x - min) / (max - min);
        return new Colour(g);
    }
}
```

Making the image We're almost there. Let's make the `Image`. Can you see how to do this?

We'll add to our `Mandelbrot` class:

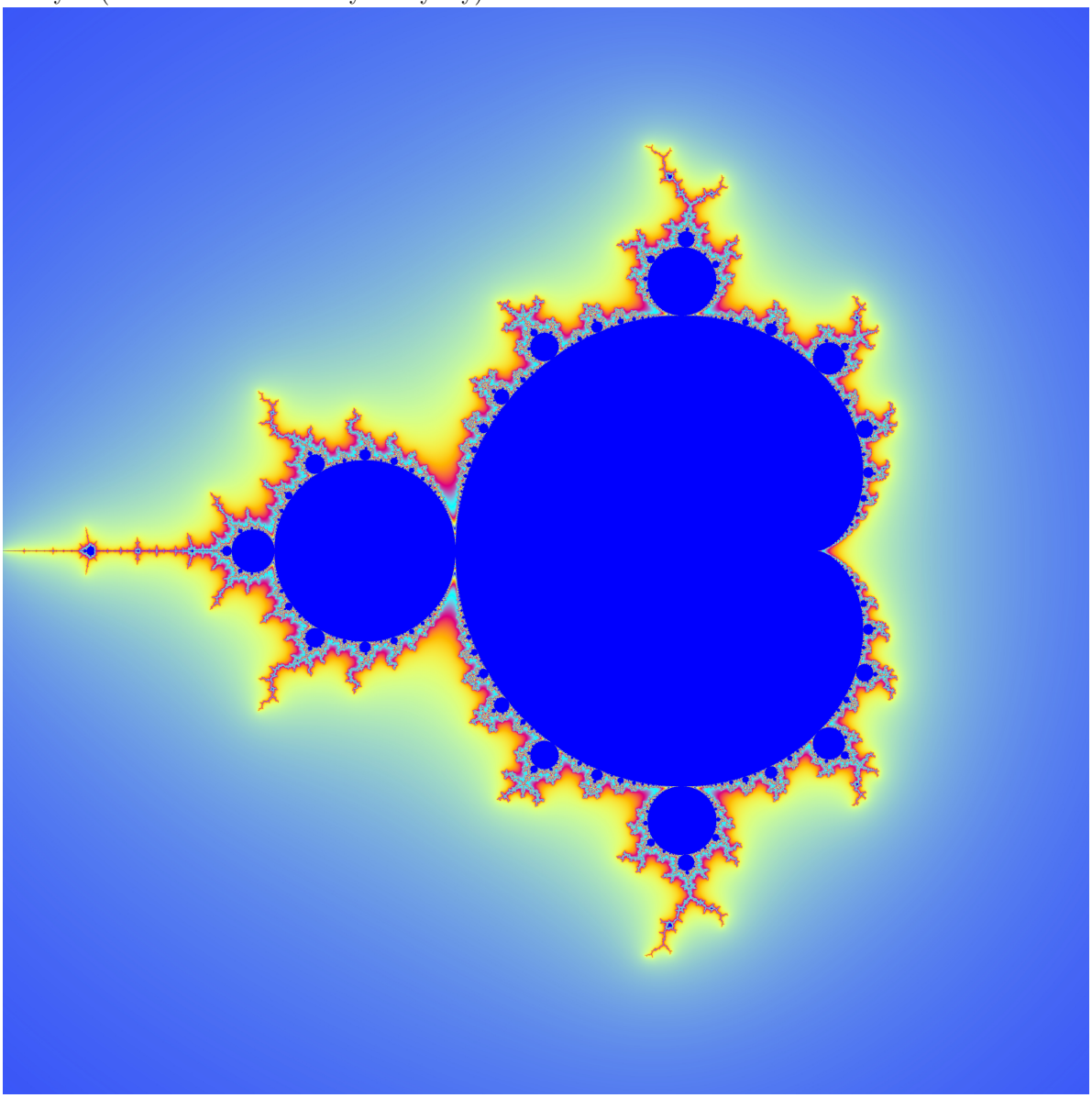
```
public class Mandelbrot {
    ...
    public void fill(Image img, float lft, float top, float rgt, float bot, ColourMap colours ) {
        img.fill(
            (x, y) -> colours.apply(escapeTime(new Complex(x, y))),
            lft, top, rgt, bot
        );
    }
}
```

Now we can put a `main` method to create the image, fill it, and write it to a file:

```
public class Mandelbrot {
    ...
    public static void main(String[] args) throws FileNotFoundException {
        Mandelbrot mandelbrot = new Mandelbrot(2, 32);
        ColourMap colours = new GrayColourMap(0,32);
        Image img = new Image(400, 400);
        mandelbrot.fill(
            img,
            -2, 1.5f, 1, -1.5f,
            colours
        );
        img.write("mandel.ppm");
    }
}
```

This should give us the image we were aiming for earlier.

We could use a different colour map to get funkier colours. Here's one I did with some different colours, a higher max iteration count and a finer resolution, oh and I used a continuous colouring. That last bit is in the list of other things below that might interest you (but I solve it later for you anyway).



Change the parameters and have a zoom about.

Other things Here some ideas of things you might try (or move on to the next fractal).

- It is really slow for big images. What solutions have people come up with? Try some?
- Try histogram colouring
- Try continuous colouring
- Do Multibrots - $z_n = z_{n-1}^d + c$, for some d
- Try contour or border mapping
- How do you go beyond 32 or 64 bit precision?
- Try showing Julia sets

Here's a question for you. We've used `floats` so far. What are the advantages or disadvantages compared to `doubles`? Certainly, you can zoom in more if you use `doubles`. Should you refactor all the `floats` into `doubles`? Yes, probably. Does it change the performance on your machine? What does it do to the memory consumption?

7 A JavaFX Gui

Okay, I thought we could try something a bit different. Shall we make a very simple GUI for our Mandelbrot pictures? Yes? Good.

Right. Well, the first thing I should admit to you is that I haven't made a GUI for anything in Java for about 15 years. I have just been looking at what's there and it has all changed, almost beyond all recognition. So, we're going to have work this out together. And, when I show you some code for how to do things, realise that you can probably do better yourself. If you do write some better code, send me an email and I can update this project.

The next thing I'm going to tell you is that a bunch of the stuff we wrote before is going to be wasted because Java already did them for you. Oh well, hopefully you learned something while doing it, though, right? For example, there is already a very good `Color` class, and there is already a pretty good `Image` class. We're going to throw ours away and use Java's. We'll make a new project (if you're using an IDE like Eclipse), or a new directory for our code (if you're doing it old school).

Java still doesn't have a `Complex` class, so we can copy that over. But, while you're at it, make this version use only `doubles`, not floats.

We're going to be using *JavaFX*, which is Java's new graphics and GUI APIs. You will be able to find lots of tutorials and things online as well as the raw API documentation. I'm only going to show you the bits we need for our simple user interface, but you should definitely check out other resources.

Here's a bit for you. From now on, we'll use JavaFX's `Color` class. You can find it at: `https://docs.oracle.com/javase/9/docs/api/javafx/scene/paint/Color.html`. Have a look at, see the similarities and differences between it and the simpler version we made. For the most part, it's similar enough that we don't need to discuss it too much.

There's also an `Image` class, here: `https://docs.oracle.com/javase/9/docs/api/javafx/scene/image/Image.html`. It's a bit different from ours. The basic class let's you load an image of various types from a file and scale it, etc. But, we're not very interested in that. How do you read and write pixels?

For reading you ask the `Image` for a `PixelReader` using the `getPixelReader()` method. A `PixelReader` can then get you the `Color` for a pixel. The `PixelReader` interface is given here: `https://docs.oracle.com/javase/9/docs/api/javafx/scene/image/PixelReader.html`.

But there doesn't seem to be anyway to set the pixels in an `Image`, right? Right. There isn't. In fact, we need a different class in the same package, `WritableImage` – `https://docs.oracle.com/javase/9/docs/api/javafx/scene/image/WritableImage.html`. A `WritableImage` is actually a subclass of `Image`, which means that everything you can do with an `Image`, you can do to a `WritableImage`. A `WritableImage` has a method, `getPixelWriter()`, which gives you a `PixelWriter`, `https://docs.oracle.com/javase/9/docs/api/javafx/scene/image/PixelWriter.html`, that lets you actually set the pixels. Why is it this complicated?

A first JavaFX image So, let's create an `Image` programmatically. We'll make an image that is 200 by 100 pixels and we'll set the pixels in the way we did for the smooth image before.

```
import javafx.scene.image.WritableImage;
import javafx.scene.image.PixelWriter;
import javafx.scene.paint.Color;

public class ImageTest {
    public static void main(String[] args) {
        WritableImage img = new WritableImage(200, 100);
        PixelWriter pw = img.getPixelWriter();

        // Set the pixels
        for(int x = 0; x < img.getWidth(); ++x) {
            for(int y = 0; y < img.getHeight(); ++y) {
                Color c = Colour.color(x/img.getWidth(), 0, y/img.getHeight());
                img.setColor(x, y, c);
            }
        }
    }
}
```

BTW, note that there's something a bit weird going on with the width and height of `Images`. You create a `WritableImage` using integer width and height. But when you ask for the width and height afterwards using `getWidth()` and `getHeight()`, the answers are `doubles`!

Writing a JavaFX image to disk Great. But how on earth do we view it? Before we go on to making a GUI, we'll work out how to write an image to a file. This is going to be a little bit painful because the `Image` doesn't support it natively. We're actually going to convert the `Image` into an older API, and then use that to write out the image to disk.

We'll make a class to hold this, and any other, image utilities. Let's call it `ImageUtil`.

What we do is use `SwingFXUtils`, `https://docs.oracle.com/javase/9/docs/api/javafx/embed/swing/SwingFXUtils.html`, to convert the image to a `BufferedImage`, `https://docs.oracle.com/javase/9/docs/api/java/awt/image/BufferedImage.html`, then use `ImageIO`, `https://docs.oracle.com/javase/9/docs/api/javax/imageio/ImageIO.html`, to write that image out. Uggh, what a palaver! Hopefully, though, once we have the code for it once we won't ever have to worry about it again, we can just use our `ImageUtil` class.

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javafx.embed.swing.SwingFXUtils;
import javafx.scene.image.Image;
import javax.imageio.ImageIO;

public class ImageUtil {
    public static void write(Image img, File file) throws IOException {
        BufferedImage swingImg = SwingFXUtils.fromFXImage(img, null);
        String type;
        String fileName = file.getName();
        int lastDot = fileName.lastIndexOf(".");
        if(lastDot == -1) type = "png";
        else type = fileName.substring(lastDot + 1);
        ImageIO.write(swingImg, type, file);
    }
}
```

Note that here I get the type of the image from the extension of the file name, or uses "png". I haven't tried other types. You might consider how to make this more robust.

Now we should be able to update our `ImageTest` class to write out the image for us:

```
import java.io.*;
...
public class ImageTest {
    ...
    public static void main(String[] args) throws IOException {
        ...
        ImageUtil.write(img, new File("smooth.png"));
    }
}
```

Filling an image Maybe we quite liked our previous ability to fill an image just from a function. Let's replicate that ability here.

First, we'll make two versions of colour functions. One for mapping a 1-d input to a colour and one for mapping a 2-d input to a colour.

Here's the one dimensional version:

```
import javafx.scene.paint.Color;

@FunctionalInterface
public interface ColorMap1d {
    public Color apply(double x);
}
```

Here's the two dimensional version:

```
import javafx.scene.paint.Color;

@FunctionalInterface
public interface ColorMap2d {
    public Color apply(double x, double y);
}
```

Now, we can make a `fill` method in `ImageUtil`. Oh, and we can make use of the `Rectangle2D` class that JavaFX gives us, `https://docs.oracle.com/javase/9/docs/api/javafx/geometry/Rectangle2D.html`.

```
...
import javafx.geometry.Rectangle2D;

public class ImageUtil {
    public static void fill(WritableImage img, ColorMap2d f, Rectangle2D port) {
        PixelWriter pw = img.getPixelWriter();

        double iw = img.getWidth();
        double ih = img.getHeight();

        double fminx = port.getMinX();
        double fminy = port.getMinY();
        double fw = port.getWidth();
        double fh = port.getHeight();

        for(int ix = 0; ix < iw; ++ix) {
            double fx = fminx + ix * fw / iw;

            for(int iy = 0; iy < ih; ++iy) {
                double fy = fminy + iy * fh / ih;

                pw.setColor(ix, iy, f.apply(fx, fy));
            }
        }
    }
    ...
}
```

The code is basically the same as the code that we had before, just converted to use our new classes.

Let's add a convenience method to `ImageUtil` to create a new image directly from a `ColorMap2d`, rather than needing us to supply it.

```
...
public class ImageUtil {
    public static WritableImage image(int w, int h, ColorMap2d f, Rectangle2D port) {
        WritableImage img = new WritableImage(w, h);
        ImageUtil.fill(img, f, port);
        return img;
    }
    ...
}
```

And now the `main` method of our `ImageTest` class can be replaced with:

```
import javafx.geometry.Rectangle2D;
...
public class ImageTest {
    ...
    public static void main(String[] args) throws IOException {
        WritableImage img = ImageUtil.image(
            200, 100,
            (x, y) -> Color.color(x, 0, y),
            new Rectangle2D(0, 0, 1, 1)
        );
        ImageUtil.write(img, new File("smooth.png"));
    }
}
```

Great. We're pretty much back to where we started. Why didn't we just do this from the start?

An image in a window We are at the point where we can put an image in a window.

Let's make a class called `SoothApp`. Now, I'm about as new to JavaFX as you are, so my explanation of this might be a bit hokey.

There is an `Application` class, `https://docs.oracle.com/javase/9/docs/api/javafx/application/Application.html`, which manages the lifecycle of the application. Specifically for us, this means we have a method we can fill in that the system will call when the application starts. That method is called, wait for it ..., `start`. This takes a `Stage` object, `https://docs.oracle.com/javase/9/docs/api/javafx/stage/Stage.html`, which you can think of as the window (it is literally a subclass of `Window`, `https://docs.oracle.com/javase/9/docs/api/javafx/stage/Window.html`). The `Stage` needs a `Scene`, `https://docs.oracle.com/javase/9/docs/api/javafx/scene/Scene.html`, which describes what's in the window. The `Scene` in turn needs a set of things to show inside it. These things are given different default layouts. Probably the simplest is `Pane`, `https://docs.oracle.com/javase/9/docs/api/javafx/scene/layout/Pane.html`. Into a `Pane` we can put a `Node` that knows how to render our `Image`. This `Node` is called an `ImageView`, `https://docs.oracle.com/javase/9/docs/api/javafx/scene/image/ImageView.html`.

Finally, you `launch` the `Application` from `main`.

That all sounds a bit complicated. It might not look so bad once you see the code:

```
import javafx.application.*;
import javafx.geometry.Rectangle2D;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.image.*;
import javafx.scene.paint.*;
import javafx.stage.*;

public class SmoothApp extends Application {
    public void start(Stage stage) {
        WritableImage img = ImageUtil.image(
            200, 100,
            (x, y) -> Color.color(x, 0, y),
            new Rectangle2D(0, 0, 1, 1)
        );

        ImageView iv = new ImageView(img);
        Pane root = new Pane(iv);
        Scene scene = new Scene(root, img.getWidth(), img.getHeight());

        stage.setTitle("Smooth!");
        stage.setScene(scene);
        stage.setResizable(false);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

A Mandelbrot GUI Can you make a Mandelbrot show up in a GUI?

Okay, I'll show you a way. We're going to make a bunch of classes, some of which are very close to things you've done before so won't get discussed much.

First up, let's have some better colours than just gray. Here's a multi color one that cycles through a bunch of colours:

```
public class MultiColorMap1d implements ColorMap1d {
    public final double min;
    public final double max;
    public MultiColorMap1d (double min, double max) {
        this.min = min;
        this.max = max;
    }
    public Color apply(double x) {
        double s = Math.PI * (x - min) / (max - min);
        return Color.color(
            Math.abs(Math.sin(s*40)),
            Math.abs(Math.sin(s*60)),
            Math.abs(Math.cos(s*40))
        );
    }
}
```

Then let's make our `Mandelbrot` class, much like before. But, I've made it all use `double`s and also I've added a `smoothEscape` function (which you can find out more about here: `https://en.wikipedia.org/wiki/Mandelbrot_set#Continuous_(smooth)_coloring`, I stole their implementation).

```
class Mandelbrot {
    public final double threshold;
    public final int maxIter;

    public Mandelbrot(double threshold, int maxIter) {
        this.threshold = threshold;
        this.maxIter = maxIter;
    }

    public int escapeTime(Complex c) {
        Complex z = Complex.ZERO;
        int n = 0;
        double thresSq = threshold * threshold;
        while(n < maxIter && z.magSq() < thresSq) {
            z = z.mul(z).add(c);
            n++;
        }
        return n;
    }
    public double escapeSmooth(Complex c) {
        Complex z = Complex.ZERO;
        int n = 0;
        double thresSq = threshold * threshold;
        while(n < maxIter && z.magSq() < thresSq) {
            z = z.mul(z).add(c);
            n++;
        }
        double iter = n;
        double log2 = Math.log(2);
        if(iter < maxIter) {
            // sqrt of inner term removed using log simplification rules.
            double log_zn = Math.log(z.magSq( )) / 2;
            double nu = Math.log(log_zn / log2) / log2;
            // Rearranging the potential function.
            // Dividing log_zn by log(2) instead of log(N = 1<<8)
            // because we want the entire palette to range from the
            // center to radius 2, NOT our bailout radius.
            iter = iter + 1 - nu;
        }
        return iter;
    }
}
```

Now I'm going to make a `ColorMap2d` from that by making a class that derives from `Mandelbrot`.

This class is a `Mandelbrot` and also a `ColorMap2d`. That is declared on the line which says `extends` and `implements`. You `extend` the *single* super class, but you can `implement` as many *interfaces* as you like.

We'll also see here that the constructor calls `super()`. Rather like before where we could call a different constructor in our class using `this()`, we can call a constructor from the parent class using `super()`.

```
public class MandelbrotSmoothMap extends Mandelbrot implements ColorMap2d {
    public final ColorMap1d map;

    public MandelbrotSmoothMap(double threshold, int maxIter, ColorMap1d map) {
        super(threshold, maxIter);
        this.map = map;
    }

    public Color apply(double x, double y) {
        return map.apply(escapeSmooth(new Complex(x, y)));
    }
}
```

Finally, our `Application` class is quite straightforward.

```
public class MandelbrotApp extends Application {
    public void start(Stage stage) {
        int maxIter = 1000;
        WritableImage img = ImageUtil.image(
            500, 500,
            new MandelbrotSmoothMap(1 << 16, maxIter, new MultiColorMap1d(0, maxIter)),
            new Rectangle2D(-2, -1.5, 3, 3)
        );

        ImageView iv = new ImageView(img);
        Pane root = new Pane(iv);
        Scene scene = new Scene(root, img.getWidth(), img.getHeight());

        stage.setTitle("Mandelbrot");
        stage.setScene(scene);
        stage.setResizable(false);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

If all has gone well, you should get a window with your pretty picture inside it!

Hopefully, you are now thinking of a million things you would like to explore with this. Please, go do those things!

Zooming in I'll do one more thing with this Mandelbrot. We'll zoom in when the user double clicks.

First, let's see how we can trap mouse clicks on the image. JavaFX makes this super easy. We add an `EventHandler` for `MouseEvents` to the `onMouseClicked` event handler list. Umm, that's much easier than it sounds:

```
...
public class MandelbrotApp extends Application {
    public void start(Stage stage) {
        ...
        ImageView iv = new ImageView(img);
        iv.setOnMouseClicked(e -> System.out.println("Clicked at x=" + e.getX() + " y = " + e.getY());
        ...
    }
    ...
}
```

Hmm, maybe you can work out how zoom in now? If you write to the image inside the event handler, it will update the image. Give it a try!

Okay, I'll do it, too. I'll make a `zoom` function to work out how the `port` should change. It's fairly simple. It takes the current port, the dimensions of the image, and the click point. It returns the new port.

```
...
public class MandelbrotApp extends Application {
    ...
    private Rectangle2D zoom(Rectangle2D port, double iw, double ih, double ex, double ey) {
        double w = port.getWidth();
        double h = port.getHeight();

        // Where was clicked in the image space
        double x = port.getMinX() + w * ex / iw;
        double y = port.getMinY() + h * ey / ih;

        // Make the port smaller
        w *= 0.5;
        h *= 0.5;

        // Center around the clicked point
        return new Rectangle2D(x - w / 2, y - h / 2, w, h);
    }
    ...
}
```

Not too bad. Why did I say it was private? Well, just because I don't really want any other classes messing with it.

Now we just need to make the click handler zoom and fill, right?

```
iv.setOnMouseClicked(
    e -> {
        port = zoom(port, img.getWidth(), img.getHeight(), e.getX(), e.getY());
        ImageUtil.fill(img, map, port);
    }
);
```

Umm, ooops. Something badly wrong happened there. Can you work out what?

It isn't the braces or multiple statements in the lambda function. Those are fine.

To understand, we have to remember what happens to the lambda function. It is really just giving the implementation of the `handle` method in a `EventHandler` class (`https://docs.oracle.com/javase/9/docs/api/javafx/event/EventHandler.html`) that is implicitly created. The lambda is syntactic sugar. It as if we wrote something like:

```
class ZoomClickHandler implements EventHandler<MouseEvent> {
    Rectangle2D port;
    WritableImage img;
    ColorMap2d map;

    public ZoomClickHandler(Rectangle2D port, WritableImage img, ColorMap2d map) {
        this.port = port;
        this.img = img;
        this.map = map;
    }
    public void        handle(MouseEvent e) {
        port = zoom(port, img.getWidth(), img.getHeight(), e.getX(), e.getY());
        ImageUtil.fill(img, map, port);
    }
}
```

And then added the event handler:

```
iv.setOnMouseClicked(new ZoomClickHandler(port, img, map));
```

But look what happens. The local variables have been copied. When we assign to `port`, it affects the copy, not the original. Java assumes this is because we messed up (though actually it would be fine here, Java just doesn't know that). So it insists that the only local variables from the parent function you are allowed to access must be marked final so you know they can't change. Yes, Java thinks we are idiots.

But we can't mark `port` as final because we need to change it. Uggh.

As a dirty hack, we could just throw a bunch of stuff up a level into the object's fields so they aren't local anymore.

```
public class MandelbrotApp extends Application {
    private int maxIter = 1000;
    private ColorMap2d map = new MandelbrotSmoothMap(1 << 16, maxIter, new MultiColorMap1d(0, maxIter));
    private Rectangle2D port = new Rectangle2D(-2, -1.5, 3, 3);
    private WritableImage img = ImageUtil.image(500, 500, map, port);
```

Or you could make a proper zoom click handler. Or do it properly some other way. Up to you.

But now you should have zoomable Mandelbrot! What else can you add to it?

Other things So far we have built about the worst GUI in the history of GUIs. Here are a few of the millions of things you might do next:

- Make it so you can pan, zoom out, reset, select a rectangle, etc.
- Make the window resizable
- Have a menu bar to allow other options, which might include:
  - Saving the current view to disk
  - Changing the current fractal (did you try Julia sets?)
  - Changing the colour map
- Make the zooming smooth – overlay the old image with the new one, scale them and change transparency so they animate well.
- Have a thumbnail of the whole fractal showing where you are
- Save and restore the current state
- Do the same thing but as an Android app
- Do the same thing using HTML and JavaScript

8 Brownian Trees

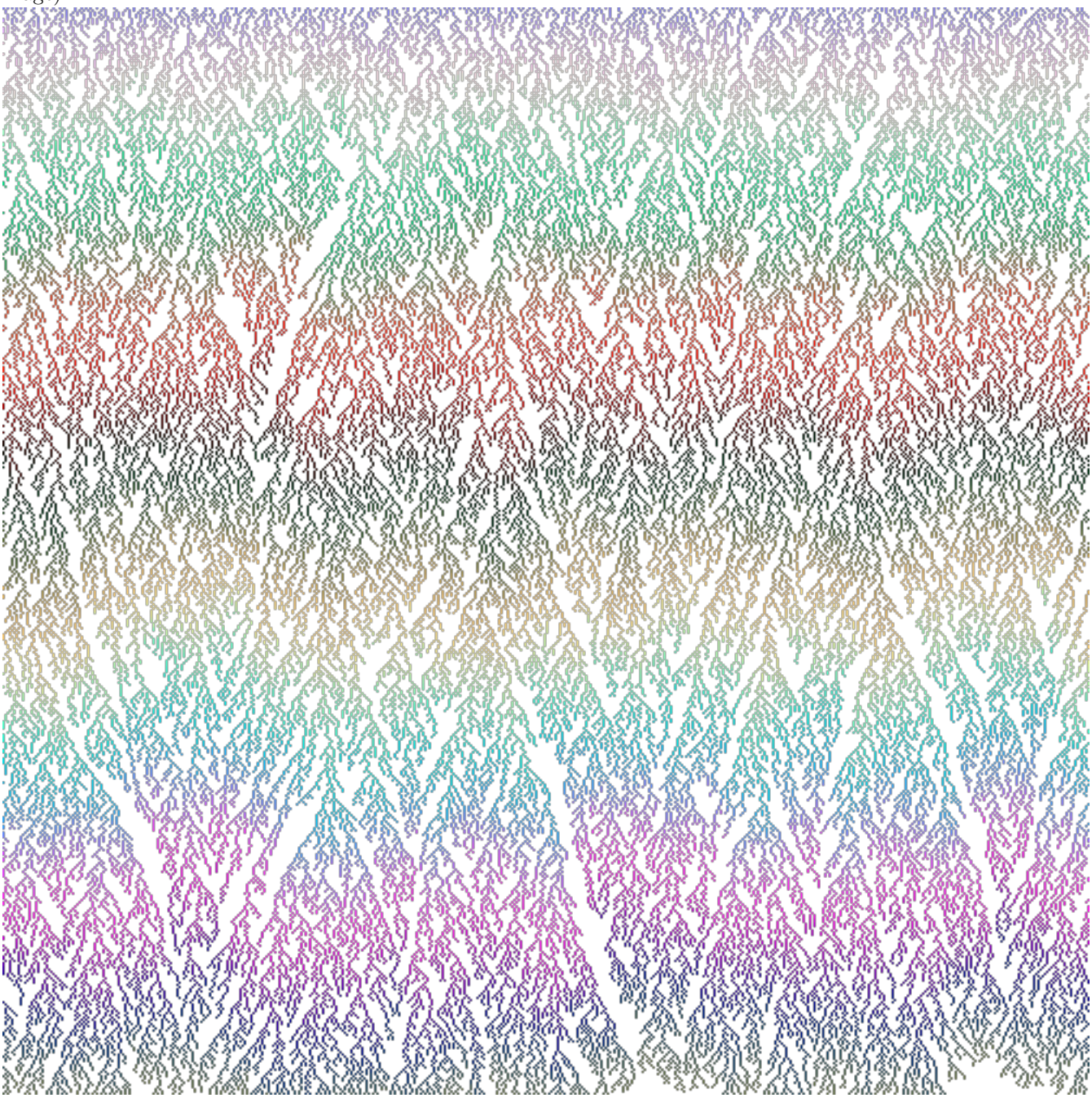
Brownian trees model a bunch of particles jittering around at random and sticking to anything they hit. We're going to do a particularly simple version and then you can maybe try different types yourself.

Choose a random x value and "drop a flake of snow" from the top of the image, straight down from the x axis, until there is a white pixel just below it, or one to left and below, or to the right and below, or until it hits the bottom. When it stops, leave it there in white. Repeat.

Rather than dropping the snow a square at time, you can make it more efficient by remembering the highest value for each x coordinate and using that directly to work out where a bit of snow should stop.

Finish when you can't drop snow more than one square.

The kind of thing we expect to make should look like this (note these trees are growing down and are coloured according to their age):



Have a go at doing this before you look at my solution.

A simple class for Brownian trees Have you noticed how for the most part I have separated out bits of code to handle the bits that are important to them? For instance, the `Mandelbrot` class doesn't have anything to do with graphics, it just handles the escape calculations. This kind of separation is a good idea. It keeps things simple.

Let's do the same for our Brownian trees. We'll make a class that does the important bits of Brownian trees but doesn't know anything about graphics, windows etc.

We'll call the class `BrownianTree`.

The main operation it's going to have is to drop a new particle. When we do this, we want to know where it lands. This will require us to return both the `x` and the `y` coordinates. There are many different ways we could do this. You could return a two element array of integers (yuck!). You could see if there are some existing classes in the Java library that will do the job.

There are some.

You could use a `javafx.util.Pair`, `https://docs.oracle.com/javase/9/docs/api/javafx/util/Pair.html`. This lets us have any types in the pairs. But it wraps the integers as objects which is a bit heavy weight, and it doesn't look like a Point, which is kind of what we want to return.

You could make a new class to hold a pair of integers. But let's not bother for the moment.

You could use a `javafx.geometry.Point2D`, `https://docs.oracle.com/javase/9/docs/api/javafx/geometry/Point2D.html`. This returns `double`s not `int`s. That's not ideal, it would be great to already have an integer point class, but this will do.

So, those bits are going to look like this:

```
import javafx.geometry.Point2D;

public class BrownianTree {
    public Point2D drop() {
        // Something here
    }
}
```

How do we work out which point when we drop? Well, we are going to need a random number generator and the current heights in all the columns of the image. Java has a `java.util.Random` for the generator, and we can use an array of `int`s for the heights. Let's make fields for them and constructors, too.

```
import java.util.Random;
...

public class BrownianTree {
    public final Random random;
    public final int width;
    private final int[] heights;

    public BrownianTree(int width, Random random) {
        this.random = random;
        this.width = width;
        this.heights = new int[width];
    }
    public BrownianTree(int width) {
        this(width, new Random());
    }
    ...
}
```

Now, once we've chosen a column to drop a particle into, we have to look at the current height of the column and the two columns adjacent. We're going to use the `max` of those three columns' heights. This sounds nice and easy. We could just put `y` as:

```
y = max(heights[x-1], heights[x], heights[x+1]);
```

But, no. There are some problems with this. First, Java only has a binary `max`[7], so we have to have some nested `max`es.

```
y = max(heights[x-1], max(heights[x], heights[x+1]));
```

[7] Actually, you can `max` a stream, but we'll ignore that at the moment.

More problematically, we are running the risk of disappearing off the end of the array. This will cause an `ArrayIndexOutOfBoundsExc` We don't want that.

We could have special code the handle the two cases when `x = 0` and `x = width - 1`.

```
if(x == 0)
    y = max(heights[x], heights[x+1]);
else if(x == width - 1)
    y = max(heights[x-1], max(heights[x]);
else
    y = max(heights[x-1], max(heights[x], heights[x+1]));
```

But that is *ugly*.

We could restrict our points so that x can only be chosen from the range $[1, width - 2]$. Then the two end columns act as *sentinels*. You can make the internal columns a two more than the ones asked for and do the appropriate conversions. Sentinels are a common technique to avoid special casing. It is worth thinking about.

Finally, we could wrap the column. So if you ask for column `-1`, we give you column `width-1`. This is what I'll do here.

Now, you might think you can just use the modulo operator `%` for this. But there's an annoying problem. When you mod a negative number you get a negative number! But there's a quick fix. Add the `width` to `x` first before taking the mod with `width`. We'll put this in a `private` method called `h`.

Then we just need to work out the new height, remember that we've changed it and return the point where the particle lands.

```
import static java.lang.Math.max;
...
public class BrownianTree {
    ...
    private int h(int x) {
        return heights[(x + width) % width];
    }
    public Point2D drop() {
        int x = random.nextInt(width);
        int y = max(h(x - 1), max(h(x), h(x + 1)));
        heights[x] = y + 1;
        return new Point2D(x, y);
    }
}
```

I'm going to add another thing. It will be useful later to know the range that the heights can fall into. I'm going to ask Java to compute these for me. A `Stream` can do this and we can make a `Steam` for an array using the `Arrays` class. Let's not worry about it too much. I'll show you and you can check out the Java docs to see what these classes offer.

```
import java.util.Arrays;
import java.util.IntSummaryStatistics;

public class BrownianTree {
    ...
    public IntSummaryStatistics getHeightStats() {
        return Arrays.stream(heights).summaryStatistics();
    }
}
```

Now, actually, I added the thing above just to introduce you to a different library. Probably, we will mostly be interested in finding out the maximum height. We could use the summary statistics, or we could use the `Arrays.stream(heights).max().getAsInt(` But we can make this faster by incrementing a maximum height field every time it increases.

```
...
public class BrownianTree {
    private int maxHeight;
    ...
    public Point2D drop() {
        ...
        heights[x] = y + 1;
        if(heights[x] > maxHeight) maxHeight = heights[x];
        ...
    }
    public int getMaxHeight() {
        return maxHeight;
    }
    ...
}
```

Filling an `Image` with a `BrownianTree` Filling an image using a `BrownianTree` should now be pretty easy for you. Give it a go before seeing what I did. You'll probably do it better!

We're going to make a class that starts looking very much like `SmoothApp` and then add things to it. Here it is:

```
import javafx.application.*;
import javafx.geometry.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.image.*;
import javafx.scene.paint.*;
import javafx.stage.*;

public class BrownianTreeApp extends Application {
    public void start(Stage stage) {
        WritableImage img = new WritableImage(1000, 1000);
        ImageView iv = new ImageView(img);
        Pane root = new Pane(iv);
        Scene scene = new Scene(root, img.getWidth(), img.getHeight());

        stage.setTitle("Brownian Tree!");
        stage.setScene(scene);
        stage.setResizable(false);
        stage.show();

        // Here's the tree bit.  Pretty simple really.
        BrownianTree tree = new BrownianTree((int)img.getWidth());
        PixelWriter pw = img.getPixelWriter();
        while(tree.getMaxHeight() < img.getHeight()) {
            Point2D p = tree.drop();
            pw.setColor((int)p.getX(), (int)p.getY(), Color.BLACK);
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Animating the tree Fine. But boring. Did we learn anything new? Not so much.

Hey, what if we animate the drawing of the tree so we can see it being built? Let's make a class called `BrownianTreeAnimator`. We won't make it `public`, we'll put it in the same file as `BrownianTreeApp`. This is just because I want a new class for clarity, but it doesn't deserve to be a top level class[8].

Now the `BrownianTreeAnimator` will derive from `AnimationTimer`, `https://docs.oracle.com/javase/9/docs/api/javafx/animation/AnimationTimer.html`. It has one method to fill in, `handle`. This method is called repeated by the animation loop. Each time it is called it tells us what the time is in nanoseconds. We can use this method to `drop` a new particle into the image.

The `BrownianTreeAnimator` will need to know the image we want to draw in, so we'll pass that into the constructor. It also might as well make the `BrownianTree`, and we could pull out the `PixelWriter` from the image to start with.

Also, we're going to replace the code in `BrownianTreeApp.start()` with a call to `start` up our new animator.

Here's what it looks like:

```
...
import javafx.animation.*;
...
public class BrownianTreeApp extends Application {
    public void start(Stage stage) {
        ...
        new BrownianTreeAnimator(img).start();
    }
    ...
}

class BrownianTreeAnimator extends AnimationTimer {
    private WritableImage img;
    private BrownianTree tree;
    private PixelWriter pw;

    public BrownianTreeAnimator(WritableImage img) {
        this.img = img;
        this.pw = pw = img.getPixelWriter();
        this.tree = new BrownianTree((int)img.getWidth());
    }
    public void handle(long now) {
        if(tree.getMaxHeight() < img.getHeight()) {
            Point2D p = tree.drop();
            pw.setColor((int)p.getX(), (int)p.getY(), Color.BLACK);
        } else {
            stop();
        }
    }
}
```

[8] Actually, I'm really just doing this to show you it can be done.

Speeding up the tree animation Umm, did you die of old age yet? What happened? Our animation is super, super slow.

There is a lot of overhead to the animation. Each frame just adds one particle and we have a lot of particles to add. Maybe we should add more than one each time? How about a whole width's worth? Can you do it?

We need a relatively minor change to the `handle` method. It should do the old body in a loop of *width* times.

```
...
class BrownianTreeAnimator extends AnimationTimer {
    ...
    public void handle(long now) {
        for(int i = 0; i < img.getWidth(); ++i) {
            if(tree.getMaxHeight() < img.getHeight()) {
                Point2D p = tree.drop();
                pw.setColor((int)p.getX(), (int)p.getY(), Color.BLACK);
            } else {
                stop();
            }
        }
    }
}
```

That's a little more fun, right?

Continuously scrolling the tree But, we stop at the end of the image which is a bit tedious. Why can't we just go on forever, scrolling the image as it needs more space? Well, we can.

If I tell you that a `PixelWriter` can set multiple pixels at once from a `PixelReader` - even if that `PixelReader` comes from the same image, can you see then how to do it? Why not have a go before looking at what I did?

Okay, so the idea is that we are going to drop a whole load of particles and see how much we need to scroll by. Then we scroll. Then we draw in those particles. We need to do it in that order, since if draw first they'll be outside the image.

To work out how much to scroll by, and how to translate the height of a dropped particle to a y coordinate on the ever scrolling image, we maintain a `heightOffset` field that we update every time we scroll a bit.

```
...
class BrownianTreeAnimator extends AnimationTimer {
    ...
    // How much have we scrolled by?
    private int heightOffset;
    ...
    public void handle(long now) {
        int w = (int)img.getWidth();
        int h = (int)img.getHeight();

        // Drop the particles first and remember them to draw later.
        Point2D[] ps = new Point2D[w];
        for(int i = 0; i < w; ++i) ps[i] = tree.drop();
        int maxHeight = tree.getMaxHeight();

        // Do we need to scroll?
        int heightDiff = maxHeight - h - heightOffset;
        if(heightDiff > 0) {
            // Scroll the image by heightDiff pixels.
            pw.setPixels(0, 0, w, h - heightDiff, img.getPixelReader(), 0, heightDiff);
            // Remember we have scrolled by a little more.
            heightOffset += heightDiff;
        }

        // Paint the particles, remembering to account for the scroll amount.
        for(Point2D p : ps) {
            int y = (int)p.getY() - heightOffset;
            // It is possible, but really, really unlikely,
            // that lots of particles could all land on the same column.
            // Theoretically, we could then have particles appearing far below
            // the `tip' of the tree.  Potentially, these could be so far back
            // that they are off the image.  It is really, really unlikely, but
            // we should probably check for it.  This `if' only paints the particle
            // if it is valid to do so.
            if(y > 0) {
                pw.setColor((int)p.getX(), y, Color.BLACK);
            }
        }
    }
}
```

Hey, that kind of works. But there's something funky going on with black smudges appearing up the image. What's that about?

Well, we aren't clearing the rectangle we scrolled away from. We need to blank it with the background color first (white?). Let's bring scrolling out into its own method to make it clearer.

```
...
class BrownianTreeAnimator extends AnimationTimer {
    ...
    public void handle(long now) {
        ...
        if(heightDiff > 0) scroll(heightDiff);
        ...
    }
    private void scroll(int diff) {
        int w = (int)img.getWidth();
        int h = (int)img.getHeight();

        pw.setPixels(0, 0, w, h - diff, img.getPixelReader(), 0, diff);
        heightOffset += diff;

        // Blank the old pixels.
        for(int x = 0; x < w; x++) {
            for(int y = h - diff; y < h; y++) {
                pw.setColor(x, y, Color.WHITE);
            }
        }
    }
    ...
```

Age based colours Finally, let's make the colours change based on the time the particle was placed. We can do this by having an `age` field in `BrownianTreeAnimator` which gets updated every time we add a particle. We can then use one of our `ColorMap1d`s to choose colours for the particles. Can you do it?

I'll show how I did it. It should be fairly self explanatory by now.

```
...
public class BrownianTreeApp extends Application {
    public void start(Stage stage) {
        ...
        new BrownianTreeAnimator(img, new AgeColorMap1d()).start();
    }
    ...
}

class BrownianTreeAnimator extends AnimationTimer {
    ...
    private ColorMap1d ageMap;
    private int age;

    public BrownianTreeAnimator(WritableImage img, ColorMap1d ageMap) {
        ...
        this.ageMap = ageMap;
    }

    public void handle(long now) {
        ...
        if(y > 0) {
            pw.setColor((int)p.getX(), y, ageMap.apply(age));
        }
        age++;
        ...
    }
    ...
}

class AgeColorMap1d implements ColorMap1d {
    public Color apply(double x) {
        return Color.color(
            (1+Math.sin(x*0.0001))/2,
            (1+Math.sin(x*0.0002))/2,
            (1+Math.cos(x*0.000003))/2
        );
    }
}
```

Other things

- This isn't the standard way of doing Brownian trees. This is more like 'Brownian snow'. The normal method has some fixed seed points, then particles randomly jitter around until they stick to something (initially to a seed point, then to other stuck points). Try doing it that way. What happens if the chance of sticking is not 100%? There are lots of variations.
- Make the GUI less terrible.
- Try doing the same thing on Android.
- Try doing it in JavaScript and HTML.

9 Fractal Landscapes

Coming soon...