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.

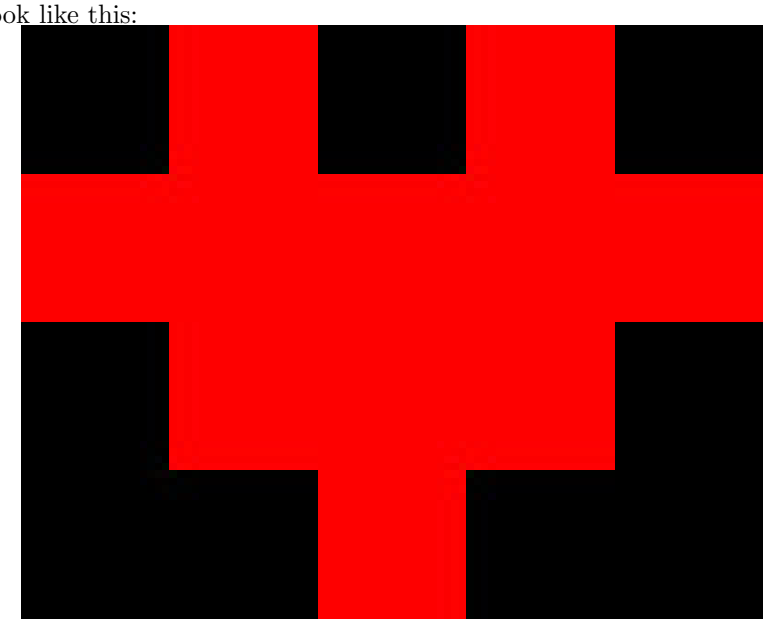
P3 format The basic structure of the file is like this:

```
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 `float`s 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?

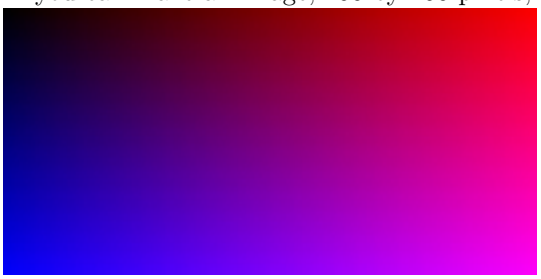
Making the heart Let's fill up the pixels in the `main` method and then write out the file.

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:

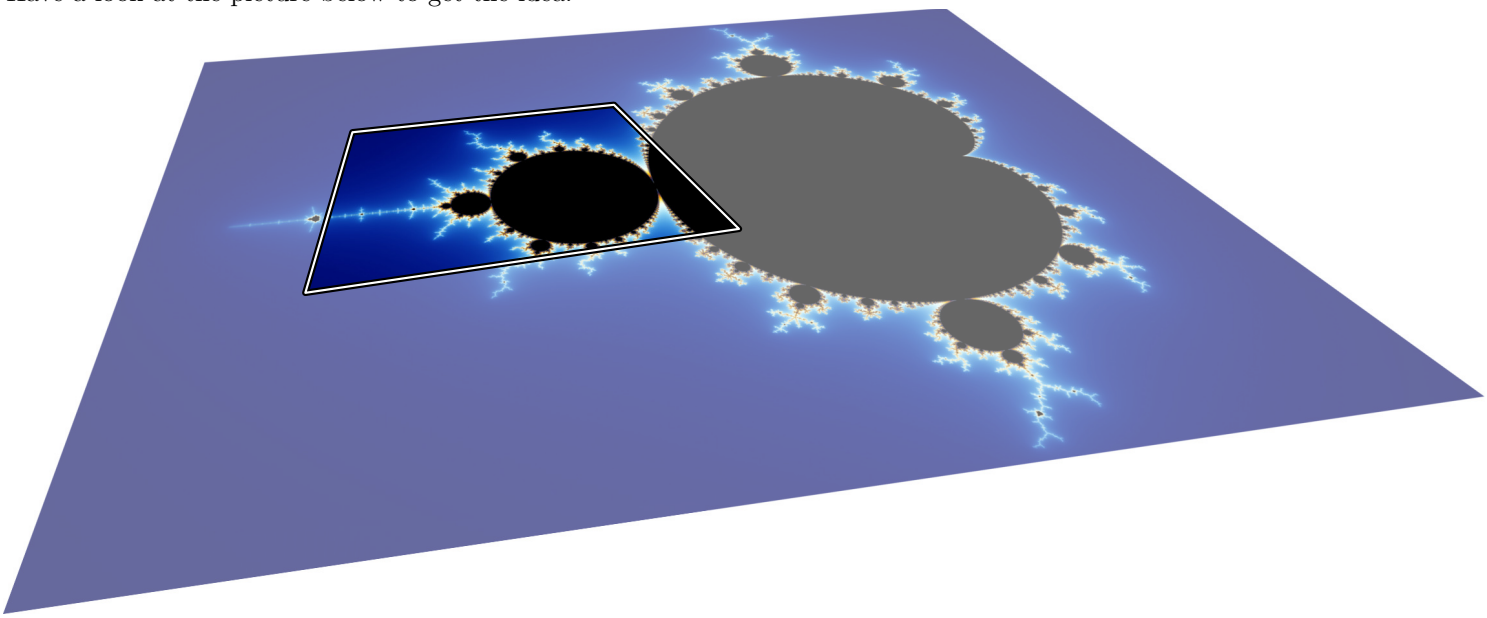
```
import java.io.*;

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.

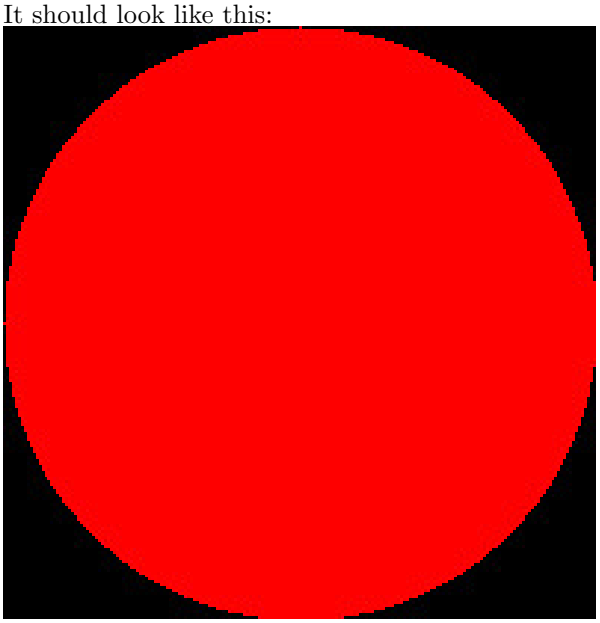
```
public class Image {
    ...
    public Image(int w, int h, ImageFn f, float lft, float top, float rgt, float bot ) {
        if(w <= 0) throw new IllegalArgumentException("width must be > 0");
        if(h <= 0) throw new IllegalArgumentException("width must be > 0");
        this.w = w;
      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

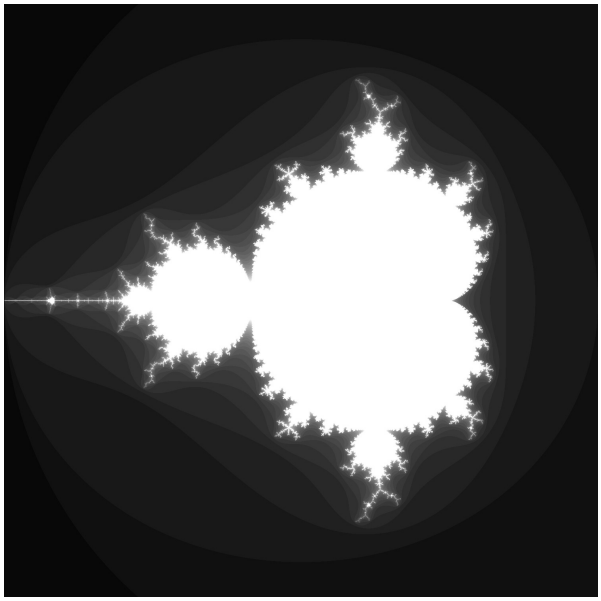
$$z_0 = 0$$

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

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.

```
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);
    }
    ...
```

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

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:

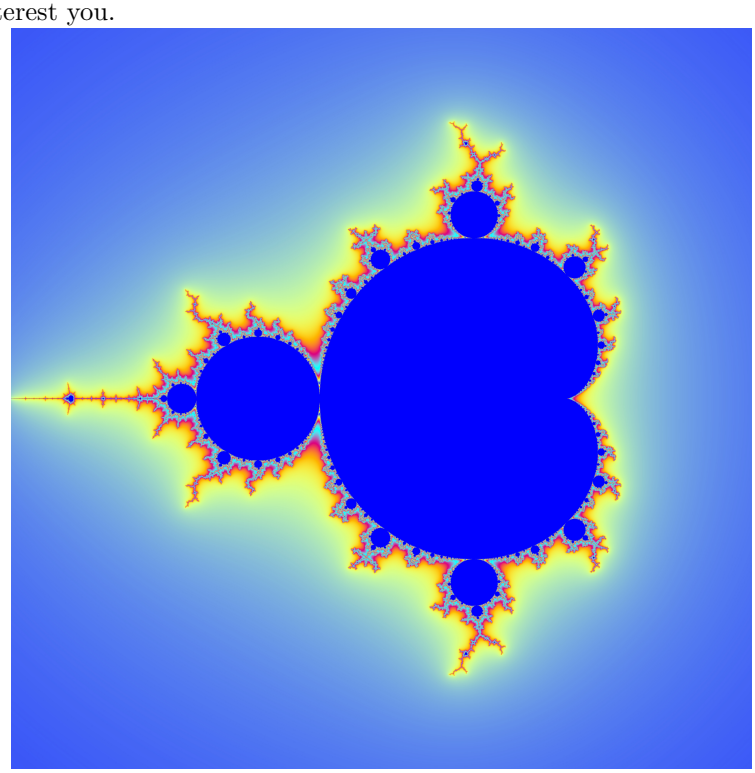
```
public class Mandelbrot {
    ...
    public void fill(Image img, float lft, float top, float rgt, float bot, ColourMap colours ) {
        img.fill(
            (x, y) -> colours.apply(escapeTime(new Complex(x, y))),
            lft, top, rgt, bot
        );
    }
}
```

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.



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).

- Build a GUI for it
- 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 `float`s so far. What are the advantages or disadvantages compared to `double`s? Certainly, you can zoom in more if you use `double`s. Should you refactor all the `float`s into `double`s? Yes, probably. Does it change the performance on your machine? What does it do to the memory consumption?

7 Brownian Trees

Coming soon...