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.

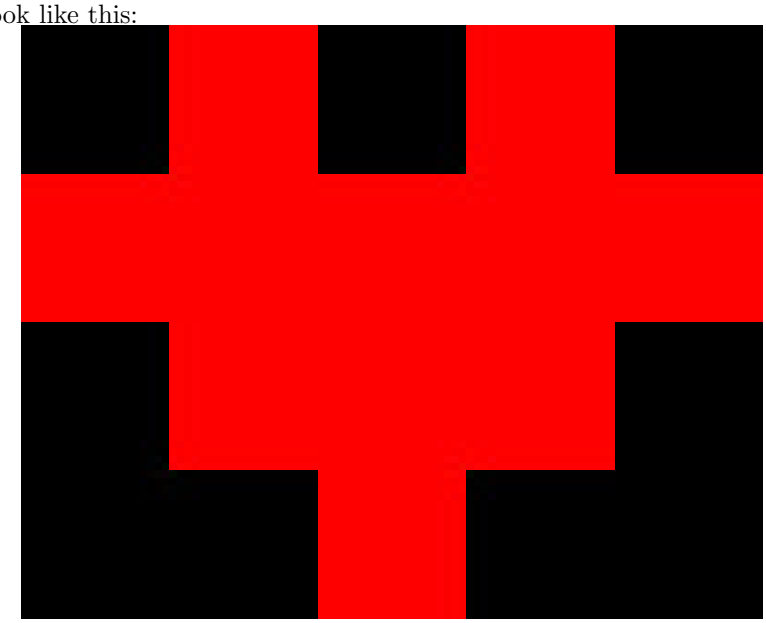
P3 format The basic structure of the file is like this:

```
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?

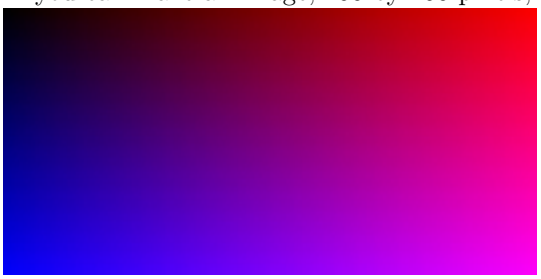
Making the heart Let's fill up the pixels in the `main` method and then write out the file.

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.