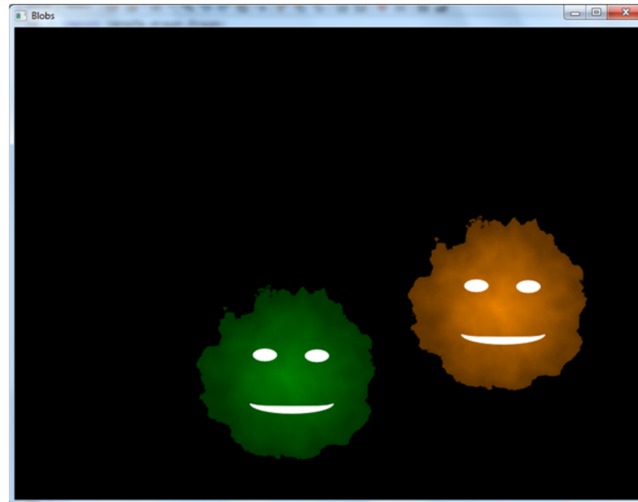
THE JAVAFX CANVAS – PART I: USING IMAGES

This tutorial will create the application 5. The aim is to use images and paint them on to the canvas. We will also look at how to generate a random number and practise collision detection once again. If you haven't completed the basic creating a game tutorials (part 1 here), then go back and do so.



STARTING OFF: CREATE THE BASIC APPLICATION

Create a basic JavaFX application, call it **CanvasPractice**. Set the window size to be 800x600 as a minimum. Title it 'Blobs'. Use a `Group` as the root layout and add a `Canvas` to it.

I achieved the black background like this:

```
Scene scene = new Scene(g, width, height, Color.BLACK);
```

This saves having to draw a rectangle on the canvas and leaving transparent patches when using the clearRect() method. You may notice that I have created two global variables for height and width instead of hard coding the values in. This is because I want to avoid magic numbers appearing throughout my code. We will need to use the window height and width in several places in this application

THE BLOB CLASS

Add a new class to your project and call it **Blob**.

Use the techniques you have learnt before to create global variables as shown below. These have been declared as private so they are hidden inside any Blob object.

```
private int x, y, dx, dy, windowWidth, windowHeight,imgWidth, imgHeight;
private Image img;
```

Following from this create **public** methods called getX(), getY() and getImage() that return x, y, and img. We will also need to create methods that return the height and width of the image. This can be done using the `getWidth()` and `getHeight()` methods that belong to `Image`.

```
public double getHeight(){
    return img.getHeight();
}
```

Now it is time to develop the constructor for a Blob. This will need to know the file path for the blob image along with the height and width of the application window. In order to vary the Blobs, we can use a random number for the speed (dx, dy) and the starting x and y co-ordinates.

The `nextInt()` method of the Random class will generate a number within the specified range. nextInt(10) will generate a number from 0-9. As we need to specify a speed in both directions, we add 1 so that there is a minimum velocity of 1. Remember that x and y start from the top left corner. For the Blob's we need to make sure that it starts on screen so we subtract the height and width of the image from the height and width of the image to set its initial x and y.

```
Blob(InputStream is, int w, int h){
    windowWidth = w;
    windowHeight = h;
    img = new Image(is);
    imgWidth = (int)img.getWidth();
    imgHeight = (int)img.getHeight();

    Random rand = new Random();
    dy=-(rand.nextInt(10)+1);
    dx=-(rand.nextInt(10)+1);

    y=rand.nextInt(windowHeight-imgHeight);
    x=rand.nextInt(windowWidth-imgWidth);
}
```

You may have realised that there is nothing here that stops two blobs being initialised on top of each other. You can develop this further by adding a `place()` method. This could then be called repeatedly from the main method until the two Blobs are not overlapping.

Creating and Painting Blobs

In the CanvasPractice class I have added two global Blobs.

```
private Blob blob1;
private Blob blob2;
```

After we show the primaryStage, we can start to manipulate the window. We need to instantiate blob1 and blob2, initialising them using the `InputStream` corresponding to the image file, width and height of the window. We also need to get the `GraphicsContext` so we can draw to the canvas. As the canvas will be updated periodically, we can use an `AnimationTimer`. Although it is not really necessary in this simple application, I have created a paint() method that handles the updating of the canvas. This method needs to receive the GraphicsContext.

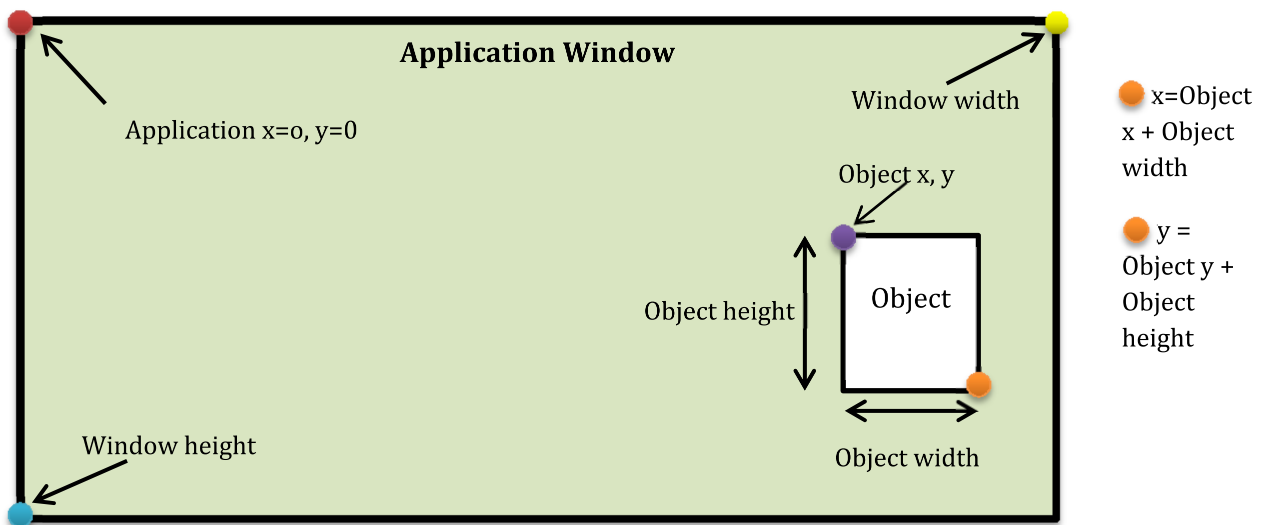
```
blob1 = new Blob(getClass().getResourceAsStream("blob1.png"), width, height);
blob2 = new Blob(getClass().getResourceAsStream("blob2.png"), width, height);

GraphicsContext gc = canvas.getGraphicsContext2D();

new AnimationTimer(){
   @Override
   public void handle(long now){
       paint(gc);
   }
}.start();
```

This is my paint method:

```
private void paint(GraphicsContext gc){
    gc.clearRect(0,0,width, height);
    gc.drawImage(blob1.getImage(), blob1.getX(), blob1.getY());
    gc.drawImage(blob2.getImage(), blob2.getX(), blob2.getY());
}
```

Moving Blobs

Remember that the x and y of an object are measured from the top left corner. The diagram below illustrates what happens when an object is moving around the window; we need to make sure that the point marked 🟠 does not go off screen. This means when moving across a layout we need to pay attention to the right-hand and bottom-edges.



Remember we are using **dx** to set a new value for x. If the right edge of the object touches the rightmost extreme of our movement boundary (in this case the right edge of the window) then we need to reverse the sign of dx. **dy** is manipulated in a similar manner using the bottom edges of the object and window. This means that my method for controlling the x movement looks somewhat like:

```
IF objectX+objectWidth > windowWidth AND dx>0 THEN
      dx = - dx
ELSE IF object < 0 THEN
      dx = -dx
END IF
x = x+dx
```

Here is my setX() method:

```
private void setX(){
    if((x+imgWidth>windowWidth)&&(dx>0)){
        dx=-dx;
    }
    else if((x<0)&&(dx<0)){
        dx=-dx;
    }
    x = x+dx;
}
```

I have created a similar private method for setY. The final touch is to create a public move() method that itself calls setX() and setY(). This can be called for each blob from the `AnimationTimer` in the `CanvasPractice` class.

```
new AnimationTimer(){
   @Override
   public void handle(long now){

       blob1.move();
       blob2.move();
       paint(gc);
   }
}.start();
```

Running the application now should allow you to see bouncing blobs on screen.

Collision Detection: Bouncing Blobs off each other

The collision detection here works in a similar manner to the [earlier tutorial]. The major different is that we are working here with an `Image` and not an `ImageView`.

In order to make the collision detection work via the `intersects()` method, we need to create a `Rectangle` object that represents the current bounds of a blob. We can then see if this rectangle as an intersection with a rectangle that represents the bounds of the other blob. If they have collided then we need to negate dx and dy for each object to reverse the trajectory.

A `Rectangle` is created by specifying the z, y width and height during initialisation. To keep my animation timer tidy I have created a collisionDetect() method. A public crash() method in my blob class reverses the sign of dx and dy.

```
private void collisionDetect(){
    Rectangle r1 = new Rectangle(blob1.getX(), blob1.getY(), blob1.getWidth(), blob1.getHeight());
    Rectangle r2 = new Rectangle(blob2.getX(), blob2.getY(), blob2.getWidth(), blob2.getHeight());
    if(r1.getBoundsInParent().intersects(r2.getBoundsInParent())){
       System.out.println("CRash");
       blob1.crash();
       blob2.crash();
    }
}
```

The collision detection is not particularly great on the corners due to the difference between a rectangle and the circular shape of the Blob. You can experiment with using `Shape` to create better quality collision detection.

Finally, these last two tutorials have made use of the canvas instead of manipulating a Layout. This is particularly relevant if you want to improve the performance of your game. A grid based game could make use of the layouts. A game that makes use of a great deal of animation would benefit from performance increases associated with using the canvas. Finally, when developing for Android you would benefit from making use of the canvas or SurfaceView; both will add greater capability, flexibility and performance benefits.