Introducing JavaFX

10

Outcomes:

By the end of this chapter you should be able to:

- *briefly describe the history of graphics programming in Java*;
- *explain the structure and life cycle of a* ***JavaFX*** *application*;
- *produce 2D graphical shapes in* ***JavaFX***;
- *build an interactive graphics application in* ***JavaFX*** *using common components such as buttons, textfields and labels*;
- *program a* ***JavaFX*** *control to listen for events using a* ***lambda expression***;
- *make use of a variety of different* ***JavaFX*** *containers*;
- *create borders, fonts and colours*;
- *format decimal numbers so that they appear in an appropriate form in a graphics application*.

10.1 Introduction

At last it is time to learn about graphics programming. In this chapter you will start to move away from that rather uninteresting text screen you have been using and build attractive windows programs for input and output.

In order to do this you are going to be using the Java graphics package known as **JavaFX**. This package provides all the graphics tools and components that you need to produce the sort of graphical interfaces that we have all become used to in modern day applications.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_10

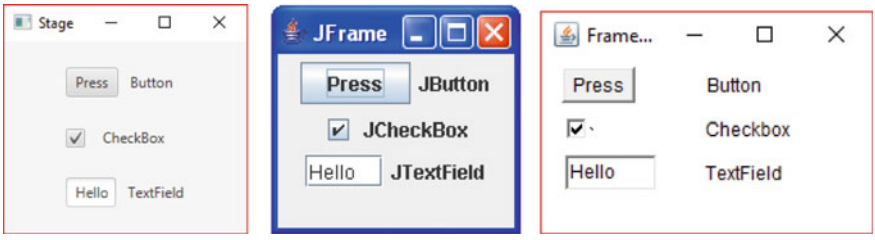
10.2 A Brief History of Java Graphics

First we will give you a little bit of history. In the earliest versions of Java, graphical programming was achieved exclusively by making use of a package known as the **Abstract Window Toolkit (AWT)**. The idea with AWT was to provide a system of graphics in which any component that we create is associated with the corresponding component in the native operating system. So with AWT, if we were to create a graphics component (such as a button or text field for example) the component would be provided by the operating system—Windows™ or macOS™ for example—so that your button or text field would look exactly like the one you were used to in the particular operating system. Components that rely on the native operating system make extensive use of the system's resources and are therefore described as **heavyweight** components.

Because it used a lot of resources, and because of functional differences between operating systems, AWT was not entirely successful, and this system was replaced by a package called **Swing**. Swing classes are for the most part written in Java, and because they do not rely on the system components they are known as **lightweight** components. Unlike AWT, Swing components look the same regardless of the operating system the program is running on.

From around the year 2000–2014, Swing was the main platform for producing Java Graphics. However, its look and feel became rather old-fashioned compared to today's graphics that run across multiple devices, and with the release of Java 8 in 2014 came the latest version of a new technology known as JavaFX, together with the announcement that Swing will not be developed further (although it will continue to be packaged with Java).

In Fig. 10.1 you can see some examples of common graphics components using the three different technologies.



(a) JavaFX (b) Swing (c) AWT (Windows)

Fig. 10.1 Some typical JavaFX components compared with equivalent Swing and AWT components

10.3 JavaFX: An Overview

In this book we will be using JavaFX exclusively for our graphics applications. So first, some terminology. Firstly, you need to know that a JavaFX program is referred to as an **application**. Your JavaFX class will extend the `Application` class, for which you need the following import statement:

```
import javafx.application.Application
```

The top-level window in which the application runs is called a **stage**—normally this will be a window such as you see in Fig. 10.1a—but if, as you can do with many JavaFX applications, you run the program in full-screen mode, then the screen becomes the stage. Some applications can be made to run in a browser, in which case the browser is the stage. The contents of the stage—the graphic itself—is called a **scene**, and is often referred to as a **scene graphic**. The items that make up the scene are referred to as **nodes**. They are very commonly the kind of components that allow interaction with the user, such as buttons, text fields, labels and check boxes, which are often referred to collectively as **controls**. They can also be 2D or 3D graphics shapes. But nodes can also be **containers**. Containers are components that hold other nodes, and each container arranges the nodes in a particular way—for example, vertically, horizontally, in a grid, or stacked one on top of the other. Normally, we wouldn't see the container, but it is perfectly possible to put a border around it if we want to. Importantly, containers can contain other containers, so we can develop a hierarchy in our scene. We normally place a single top level node in our scene, and this is referred to as the **root** node. We use the terms **parent** and **children** for the containing and contained nodes respectively.

Figure 10.2 should make this clear. Here we have a sample scene in which the root node is a `VBox`—this is a container that arranges its child nodes vertically. We have given it a black border so that you can see it. The `VBox` has three children—a `TextField`, a `Label` and an `HBox`, around which we have again put a border. As you can probably guess, an `HBox` is similar to a `VBox`, but arranges its child nodes horizontally. In this case it has three child nodes which are `Buttons`. All of these components will become familiar to you as you proceed through this chapter—in particular you will see how we have made extensive use of the `VBox` and `HBox` to construct our scene graphics.

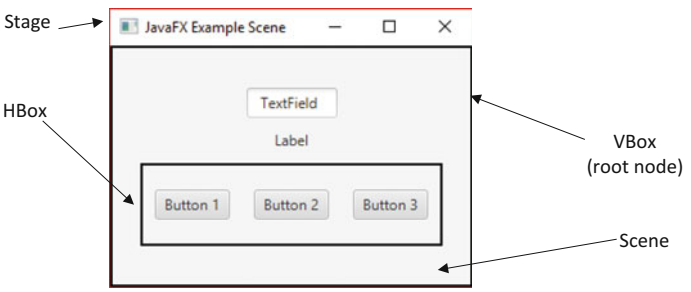


Fig. 10.2 A hierarchical scene

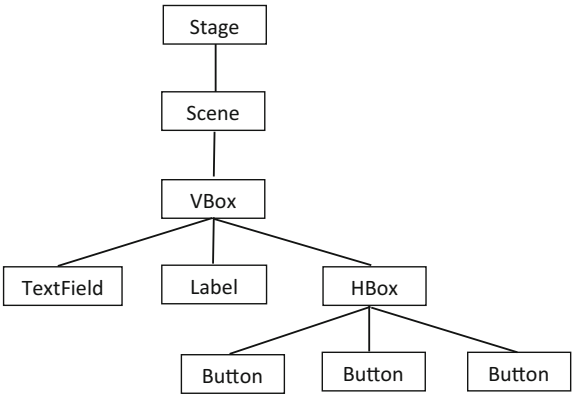


Fig. 10.3 The hierarchical structure of the scene in Fig. 10.2

To help you understand the way that a scene is constructed in a hierarchal way, we have shown you in Fig. 10.3 the hierarchy that makes up the scene graphic in the above example.

When a JavaFX application begins, there are three methods that are called in order. These are:

```
void init()
abstract void start(Stage stage)
void stop()
```

The first of these, `init`, is where we would place any routines that need to be carried out before the application itself starts, while the `stop` method is where we would place any code that we would want to be executed after the application finishes. We will not be concerning ourselves with these two methods in this chapter. What we will be concerning ourselves with, however, is the very important `start` method. As you can see, it is an **`abstract`** method and therefore has to be coded. It is in this method that the code for our application is placed. You can, of course, break this up by adding some helper methods, but it is with this method that the application itself begins.

So how do we launch a JavaFX application? Surprisingly it is not always via a `main` method. If the application is run from a command line, as described in the first chapter, then it doesn't actually require a `main` method to launch it. Neither does it need a `main` method if it is deployed as a `.jar` file, which is something you will learn about in Chap. 19. But if you run your program within an IDE, as most of you will be doing at first, then we do require a `main` method, and for that reason we have chosen to include such a method with each application that we develop here. You will see that the main method takes the following form:

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

As you can see the main method calls the application's `launch` method, and passes to it any arguments received by the `main` method itself.

The `launch` method is a **`static`** method, and we can use it to launch a JavaFX application from another program. It is overloaded to accept the name of the compiled `.class` file as its first parameter. If we wanted a program called, say, `LaunchApplication` to run an application called `MyApp`, we would do it like this:

```
import javafx.application.Application;

class LaunchApplication
{
    public static void main(String[] args)
    {
        Application.launch(MyApp.class, args);
    }
}
```

So now that you know how a JavaFX application is structured, and you are aware of the sequence in which its methods are called, we can go on to develop our first graphics application.

10.4 2D Graphics: The *SmileyFace* Class

Our first graphics application is going to create a smiley face, as shown in Fig. 10.4.

Fig. 10.4 The Smiley Face application

Although it is a rather simple application, in that there is no user interaction, it nonetheless introduces many new concepts. In particular it shows you how to create a scene, to add items to the scene, and to add the scene to a stage. It also introduces you to 2D graphics, which enables you to draw two-dimensional shapes such as circles, lines, ellipses, rectangles and arcs. Here we draw circles for the face and eyes, and an arc for the mouth. You will also see how to create text which you can configure using different colours and fonts.

As explained in the previous section, we have included a `main` method which launches the application. The complete code is shown below:

SmileyFace

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;


public class SmileyFace extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure the main circle for the face
        Circle face = new Circle(125, 125, 80);
        face.setFill(Color.YELLOW);
        face.setStroke(Color.RED);

        // create and configure the circle for the right eye
        Circle rightEye = new Circle(86, 100, 10);
        rightEye.setFill(Color.YELLOW);
        rightEye.setStroke(Color.BLUE);

        // create and configure the circle for the left eye
        Circle leftEye = new Circle(162, 100, 10);
        leftEye.setFill(Color.YELLOW);
        leftEye.setStroke(Color.BLUE);

        // create and configure a smiling mouth
        Arc mouth = new Arc(125, 150, 45, 35, 0, -180);
        mouth.setFill(Color.YELLOW);
        mouth.setStroke(Color.BLUE);
        mouth.setType(ArcType.OPEN);

        // create and configure the text
        Text caption = new Text(80, 240, "Smiley Face");
        caption.setFill(Color.BLUE);
        caption.setFont(Font.font("Verdana", 15));


        // create a group that holds all the features
        Group root = new Group(face, rightEye, leftEye, mouth, caption);

        // create and configure a new scene
        Scene scene = new Scene(root, 250, 275, Color.YELLOW);
```

```
        // add the scene to the stage, then set the title
        stage.setScene(scene);
        stage.setTitle("Smiley Face");

        // show the stage
        stage.show();
    }

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

}
```

There are a number of new concepts here. First, let's take a look at the **`import`** clauses, which show you that all of our classes come from a package called `javafx`, which as you can see has many subpackages including `scene` and `stage`.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
```

Now look at the class header:

```
public class SmileyFace extends Application
```

As we explained, all JavaFX programs run as an application, and we therefore have to extend the `Application` class. `Application` requires you to code the `start` method, which we talked about in the previous section. Let's take a look at it now, starting with the header:

```
public void start(Stage stage)
```

When `start` is called, it is automatically sent an object of the `Stage` class, which will be the main container for our graphic.

The first thing we do within the `start` method is to create and configure the main circle for the face:

```
Circle face = new Circle(125, 125, 80);
face.setFill(Color.YELLOW);
face.setStroke(Color.RED);
```

The `Circle` class, which resides in a the `javafx.scene.shape` library, has a number of constructors (which you can look up on the Oracle™ site). The constructor we are using here takes three parameters of type **`double`**.

The first two of these parameters represent, respectively, the *x* and *y* positions of the centre of the circle (with respect to the top left hand corner of the parent node), measured in pixels. The third parameter represents the radius of the circle, also in pixels. You will see later that we have chosen our initial scene to be 250 × 275 pixels, so that our circle with its centre at (125, 125) will be horizontally centred, but will leave enough vertical room for a caption.

We have used two other methods of `Circle`, namely `setFill` and `setStroke`, to set the fill colour and line colour of the circle. To each of these we have passed a pre-defined attribute of the `Color` class (note the American spelling), which resides in `javafx.scene.paint`. In Sect. 10.10 you will find how to create your own colours if you want to—but the paint library provides a great many colours that you can use, and which you can look up—or which you can choose from the list of suggestions that your IDE will make after pressing the full stop.

In a similar manner we draw the right eye and the left eye:

```
Circle rightEye = new Circle(86, 100, 10);
rightEye.setFill(Color.YELLOW);
rightEye.setStroke(Color.BLUE);

Circle leftEye = new Circle(162, 100, 10);
leftEye.setFill(Color.YELLOW);
leftEye.setStroke(Color.BLUE);
```

You might be wondering how we decided upon the exact position in which to draw these circles. In theory it is possible to calculate exactly where you want everything to be on a graphic—but often it is easier (and actually quite good fun) simply to make an estimate and see how it looks, then change the values until you are happy. That's what we did here. We strongly recommend that once you have got the application up and running, you play about with the different values to explore what they do. This is the best way to become familiar with all of the graphics objects.

Now we come to the smiling mouth, which is a little more complicated.

```
Arc mouth = new Arc(125, 150, 45, 35, 0, -180);
mouth.setFill(Color.YELLOW);
mouth.setStroke(Color.BLUE);
mouth.setType(ArcType.OPEN);
```

Creating an object of the `Arc` class draws part (or all) of an ellipse. The constructor we have used is specified on the Oracle™ website like this:

```
Arc(double centreX, double centreY, double radiusX, double radiusY,
                                                    double startAngle, double length)
```

The names of the parameters mostly speak for themselves. The first two represent the position of the centre of the ellipse. The next two are the horizontal and vertical radii respectively. `startAngle` represents the angle at which we start drawing the arc. The only confusing name is the last one, `length`, which represents the size of the angle through which the arc is drawn. Figure 10.5 should make it clear.

In our case we have chosen the radii to give us an arc of an ellipse which is somewhat wider than it is high. We have chosen a start angle of 0° and you will notice that the value of the final angle (the `length` parameter) is set to −180. The negative sign indicates that this angle is formed by moving from the start angle in an a clockwise direction (so that the mouth is smiling). A positive sign indicates an anticlockwise direction (as, for example, in Fig. 10.5).

The next lines of code set the fill colour and line colour (referred to as the stroke colour) of the mouth. The final line of code selects the type of arc we want, which in this case is `ArcType.OPEN`. Two other types exist (`ArcType.CHORD` and `ArcType.ROUND`), and these are demonstrated in Sect. 10.6.

The next thing we do is to add a caption:

```
Text caption = new Text(80, 240, "Smiley Face");
caption.setFill(Color.BLUE);
caption.setFont(Font.font("Verdana", 15));
```

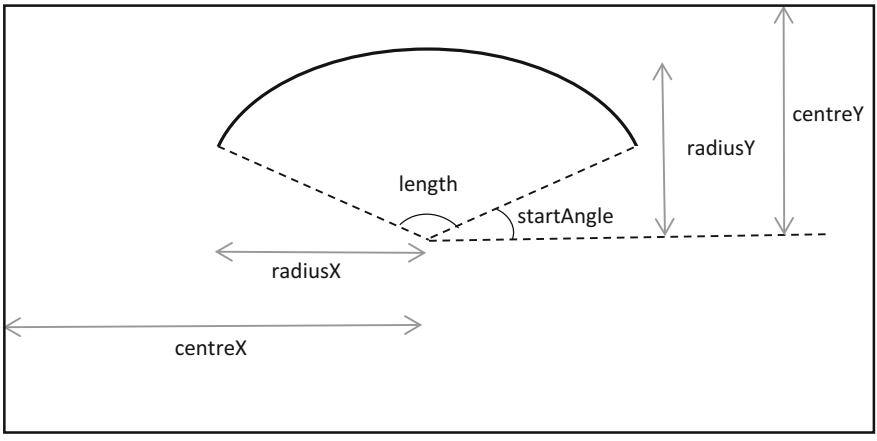


Fig. 10.5 The Arc class

For this purpose we are creating an instance of the `Text` class, which resides in `javafx.scene.text`. The constructor takes three parameters—two **`double`**s and a `String`. The first two are used to position the text (they are the co-ordinates of the beginning of the `String`), and the final one holds the value of the text itself.

We have gone on to set the colour, using the `setFill` method, and then we have set the font, with the `setFont` method. There will be more in Sect. 10.10 on how to create your own fonts, but for now you can just look at the syntax to see how we select the name and size of the font—in this case "Verdana", 15 points.

Now that we have defined all of our features we want them to stay together as a group. We can do this with the `Group` class from the `javafx.scene` package. This class acts like an invisible container—it is very useful when we have already defined the position of our shapes (as we have done here), so we don't have to worry any further about how they will be laid out within the container:

```
Group root = new Group(face, rightEye, leftEye, mouth, caption);
```

We are using the convention of naming the first node that we add to our scene `root`, as it is the root node. In this case it is the only node. We have created our new scene like this:

```
Scene scene = new Scene(root, 250, 275, Color.YELLOW);
```

Here we have chosen to use the constructor that allows us to set the size (width and height) of the initial scene, together with the background colour. If you don't set these values initially, the `Scene` class (and other graphics components) have many `set`-methods such as `setMinWidth` and `setMaxHeight` that you can code later.

Now all that remains to complete the `start` method is to add the scene to the stage, set the title, and finally make the stage visible, which we do by calling its `show` method:

```
stage.setScene(scene);
stage.setTitle("Smiley Face");
stage.show();
```

As we mentioned before, we have included a `main` method (and will continue to do so) in order that you can run the application in any environment.

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

10.5 Event-Handling in JavaFX: The *ChangingFace* Class

The `SmileyFace` class that we developed in the last section was "passive" and didn't involve any interaction with the user. In practice of course, graphics applications will normally require input from the user in the form of clicking a button, entering text and so on.

Controls such as buttons need to respond when the user performs some action such as clicking a mouse. The way this works is that in response to the user performing such an action, the control generates an `Event` object. This object is sent to an `EventHandler` which we attach to a particular control and supply it with the instructions for what to do when the action occurs. There are many actions that the user could perform, such as pressing a key, dragging a mouse and so on, but in this chapter we will concern ourselves only with a simple mouse-click on a button.

Our first application will modify our `SmileyFace` class and turn it into a `ChangingFace` class that can change its mood so it can be sad as well as happy. We are going to add a couple of buttons, as shown in Figs. 10.6 and 10.7.

You can see that we have now changed our title and caption from "Smiley Face" to "Changing Face"—because when we have finished we will be able to click on the *Frown* button and get the face to look like the one you see in Fig. 10.7. Clicking the *Smile* button will get the face to smile again.

The code for our class is shown below. There are quite a lot of new concepts and techniques here, so we will discuss it in detail once you have had a look at it.

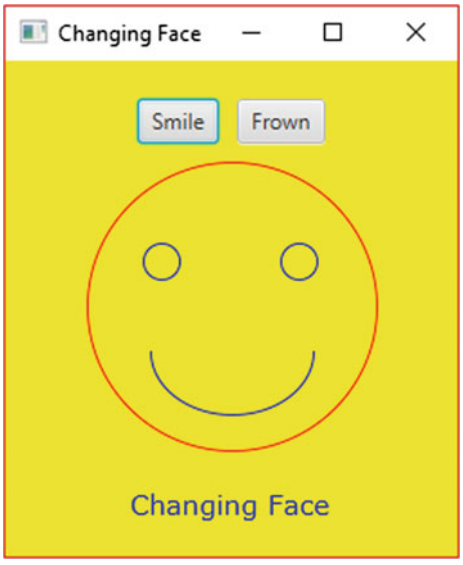


Fig. 10.6 The *ChangingFace* class (still smiling)

ChangingFace

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Background;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.geometry.Pos;

public class ChangingFace extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure the main circle for the face
        Circle face = new Circle(125, 125, 80);
        face.setFill(Color.YELLOW);
        face.setStroke(Color.RED);

        // create and configure the circle for the right eye
        Circle rightEye = new Circle(86, 100, 10);
        rightEye.setFill(Color.YELLOW);
        rightEye.setStroke(Color.BLUE);

        // create and configure the circle for the left eye
        Circle leftEye = new Circle(162, 100, 10);
        leftEye.setFill(Color.YELLOW);
        leftEye.setStroke(Color.BLUE);

        // create and configure a smiling mouth (this is how it will start)
        Arc mouth = new Arc(125, 150, 45, 35, 0, -180);
        mouth.setFill(Color.YELLOW);
        mouth.setStroke(Color.BLUE);
        mouth.setType(ArcType.OPEN);

        // create and configure the text
        Text caption = new Text(68, 240, "Changing Face");
        caption.setFill(Color.BLUE);
        caption.setFont(Font.font ("Verdana", 15));

        // create a group that holds all the features
        Group group = new Group(face, rightEye, leftEye, mouth,  caption);

        // create a button that will make the face smile
        Button smileButton = new Button("Smile");

        // create a button that will make the face frown
        Button frownButton = new Button("Frown");


        // create and configure a horizontal container to hold the buttons
        HBox buttonBox = new HBox(10);
        buttonBox.setAlignment(Pos.CENTER);

        //add the buttons to the horizontal container
        buttonBox.getChildren().addAll(smileButton, frownButton);

        // create and configure a vertical container to hold the button box and the face group
        VBox root = new VBox(10);
```

```
        root.setBackground(Background.EMPTY);
        root.setAlignment(Pos.CENTER);

        //add the button box and the face group to the vertical container
        root.getChildren().addAll(buttonBox, group);

        // create and configure a new scene
        Scene scene = new Scene(root, 250, 275, Color.YELLOW);

        // supply the code that is executed when the smile button is pressed
        smileButton.setOnAction(e -> mouth.setLength(-180));

        // supply the code that is executed when the frown button is pressed
        frownButton.setOnAction(e -> mouth.setLength(180));

        // add the scene to the stage, then set the title
        stage.setScene(scene);
        stage.setTitle("Changing Face");

        // show the stage
        stage.show();
    }

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

}
```

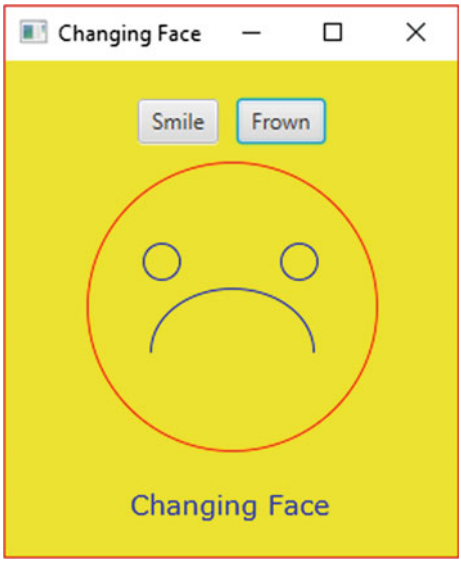


Fig. 10.7 The *ChangingFace* class (frowning)

We have proceeded as before when it comes to creating the face. Once we have done this we have created two instances of the `Button` class. A button is an extremely common feature of graphics programming, and the `Button` class, along with many other similar components is to be found in `javafx.scene.control`. Here is the code for the buttons:

```
Button smileButton = new Button("Smile");

Button frownButton = new Button("Frown");
```

You can see that we have used a version of the constructor that allows us to set the text that appears on the button. You can also use the `setText` method of `Button` for this purpose.

Having created our two buttons, we now go on to create a container to hold them:

```
HBox buttonBox = new HBox(10);
buttonBox.setAlignment(Pos.CENTER);
```

We have created an instance of an `HBox`. As we mentioned earleir, this is a container that arranges the contained nodes horizontally. The constructor we have used takes a parameter that sets the distance between the items, in this case 10 pixels. We have then gone on to use its `setAlignment` method, into which we send a pre-defined constant, an attribute of the `Pos` class which is found in the package `javafx.geometry`. The constant we have chosen is `Pos.CENTER` in order to centre the components that the `HBox` contains. There are a number of other options, and these are demonstrated in Sect. 10.9.1.

Having created our `HBox`, we need to add our buttons to it. We do this by calling a method of `HBox`, called `getChildren`, which returns a list of all the child nodes. This list has two methods for adding the nodes: the `add` method will add a single item, and `addAll` a list of items. As we need to add two buttons, we have used the latter:

```
buttonBox.getChildren().addAll(smileButton, frownButton);
```

So now have an `HBox` containing our buttons and a `Group` containing the shapes that make up our face. We need to organize these so that the face is placed vertically below the buttons, so we use a `VBox` which lines the items up vertically, just as the `HBox` does horizontally. We have given the name `root` to this instance of `VBox`, because this will be the root node that we add to our scene.

```
VBox root = new VBox(10);
root.setBackground(Background.EMPTY);
root.setAlignment(Pos.CENTER);
root.getChildren().addAll(buttonBox, group);
```

You will notice that we have done something else here, which is to add an empty background to the `VBox`—this is so that the yellow colour of the scene isn't hidden.

Now we can add the `HBox` containing the buttons, and the group containing the face, to the `VBox`. We then add this `VBox` to the scene graphic:

```
root.getChildren().addAll(buttonBox, group);
Scene scene = new Scene(root, 250, 275, Color.YELLOW);
```

We are almost ready to take the final step of adding the scene to the stage.

Almost but not quite! There is one really vital thing we have to do, which is to enable the buttons to respond when they are pressed, and to provide the code that tells the buttons what to do when this happens. At the beginning of this section we explained that a control can be programmed to generate an `Event` object in response to some action taken by the user. The type of `Event` that we are interested in here is called an `ActionEvent`, which is the one that handles a simple mouse-click. We need to add an `EventHandler` to each button, which means it will generate the `ActionEvent` as soon as the mouse is clicked. Effectively we are programming our button to "listen out" for a mouse-click.

`EventHandler` has a method called `handle`, and it is the code for this method that we need to supply in order that the button knows what to do when the mouse is clicked.

Now, you might think that all this sounds rather complicated—but we are in luck! Java 8 has furnished us with two things that mean our code for doing all this stuff is very simple. The code for doing this for each button is shown below. It contains some new syntax, which we will explain once you have had a look at it:

```
smileButton.setOnAction(e -> mouth.setLength(-180));
frownButton.setOnAction(e -> mouth.setLength(180));
```

You can see that a `Button` has a method called `setOnAction`. This an example of what is known as a **convenience method**, a feature of JavaFX. It certainly is convenient because it means that all we have to do in order to add an `EventHandler` is to supply the code it needs for its `handle` method. Most controls have these convenience methods, all starting with `setOn-`. Other examples that you will come across in the second semester are `setOnMouseMoved` and `setOnKeyTyped`, as well as many others.

You can see that the code (which looks rather unfamiliar because of the `->` notation) is sent directly into the method. This is the other thing that we have been able to do as a result of innovations in Java 8. The code that you see is called a **lambda expression**. Lambda expressions allow us to simply send in some code to a method as an argument, just as we would with a value of a primitive type like **`int`**, or an object of a class like `String`.

We will look in detail at lambda expressions in Chap. 13. For now we will just tell you what you need to know in order to code the `setOnAction` method.

In each case we are using the `setLength` method of `Arc` to redraw the mouth. As we have seen, giving this a negative value draws it clockwise, so that the mouth smiles, and giving it a positive value makes the mouth frown. So the instructions for the `smileButton` and `frownButton` respectively are `mouth.setLength(-180)` and `mouth.setLength(180)`.

These instructions are the ones we have to supply to the `handle` method of the `EventHandler`. As we have said, lambda expressions enable us to supply this code by passing it as an argument to the `setOnAction` method. You can see from the above that our lambda expressions look like this:

```
e -> mouth.setLength(-180) //smile
e -> mouth.setLength(180) //frown
```

There are two parts to a lambda expression, one on either side of the `->` symbol. The code goes on the right of the symbol. On the left we give names to the parameters that the method (in this case `handle`) expects to receive. The `handle` method receives an `ActionEvent`, and we have given this the name `e`. Even though we are not, in this case, going to use this variable, we nonetheless have to give a name.

There is a lot more to lambda expressions. But for now you are only going to use them in connection with a `setOnAction` method, so this is all you need for the moment. One thing we should add, however, is that if there is more than one instruction to our code we have to enclose the code in curly brackets. For example, if we wanted to paint the mouth violet when it smiles, our lambda expression would look like this:

```
smileButton.setOnAction(e -> {
                                mouth.setLength(-180);
                                mouth.setStroke(Color.VIOLET);
                            }
                        );
```

Finally we can add the scene to the stage, set the title and make the stage visible:

```
stage.setScene(scene);
stage.setTitle("Changing Face");

stage.show();
```

10.6 Some More 2D Shapes

Before we move away from 2D graphics, we will draw your attention to a some more shapes that will increase your repertoire.

We have shown a few examples in Fig. 10.8.

You can see that we have experimented with different colours, and with using the `setFill` and `setStroke` methods.

The rectangle that you see in the top left-hand corner was created with the following code:

```
Rectangle rectangle = new Rectangle(50, 50, 50, 100);
```

In this constructor, the first two parameters (all of which are of type **`double`**) represent the *x* and *y* co-ordinates of the top left hand corner, and the next two represent the width and height of the rectangle respectively.

The line underneath was created using the following constructor:

```
Line line = new Line(50, 180, 80, 250);
```

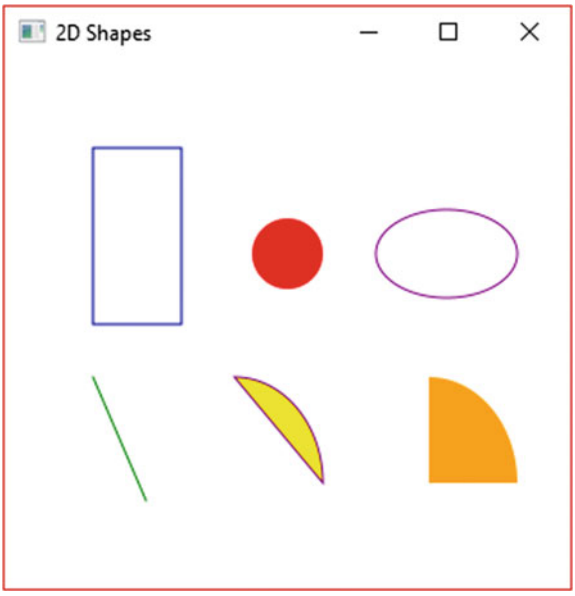


Fig. 10.8 Some more 2D shapes

The first two parameters are the *x* and *y* co-ordinates of the start position, and the last two are the co-ordinates of the end position.

The ellipse that you see in the top right hand corner was drawn simply by creating an `Arc` and drawing the line through an angle of 360°. In our previous examples you saw the effect of choosing `ArcType.OPEN` for our arc type. The two arcs you see on the bottom row show the effect of choosing `ArcType.CHORD` and `ArcType.ROUND` respectively.

All of the above shapes reside in `javafx.scene.shape`. You can check out the many other constructors and methods of these and other shapes on the Oracle™ website.

10.7 An Interactive Graphics Class

Most common applications involve controls (buttons, check boxes, text fields and so on) rather than graphical shapes. The next class—which we have called `PushMe`—is going to have controls that allow the user to input information via a graphics screen. The program isn't all that sophisticated, but it introduces the basic elements that you need to build interactive graphics classes.

This application allows the user to enter some text and then, by clicking on a button, to see the text that was entered displayed below the button. You can see what it looks like in Fig. 10.9.

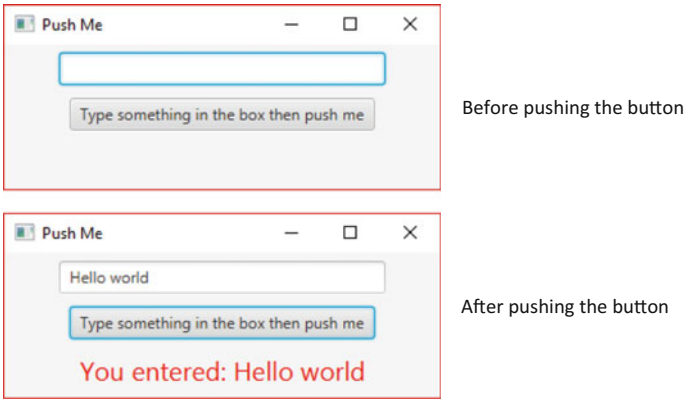


Fig. 10.9 The *PushMe* class

As usual we will show you the code first and discuss it afterwards:

PushMe

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class PushMe extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure a text field for user entry
        TextField pushMeTextField = new TextField();
        pushMeTextField.setMaxWidth(250);

        // create and configure a label to display the output
        Label pushMeLabel= new Label();
        pushMeLabel.setTextFill(Color.RED);
        pushMeLabel.setFont(Font.font("Arial", 20));

        // create and configure a label which will cause the text to be displayed
        Button pushMeButton = new Button();
        pushMeButton.setText("Type something in the box then push me");
        pushMeButton.setOnAction(e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText()));

        // create and configure a VBox to hold our components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);

        //add the components to the VBox
        root.getChildren().addAll(pushMeTextField, pushMeButton, pushMeLabel);


        // create a new scene
        Scene scene = new Scene(root, 350, 150);

        //add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Push Me");
        stage.show();
    }

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

}
```

The box into which we type our text is called a `TextField`. This allows us to type in one line of text:

```
TextField pushMeTextField = new TextField();
pushMeTextField.setMaxWidth(250);
```

You can see that we have set the maximum width of our `TextField` to 250—if we had not done this, it would simply have filled the width of its parent container. You might want to explore a similar class, `TextArea`, that allows you to add several rows of text—you will see an example of this in the next section.

When the button is pressed, the text entered will be displayed underneath the button on a `Label`. As its name suggests, its purpose is simply to display some chosen text. We have created and configured it with the following lines of code:

```
Label pushMeLabel= new Label();
pushMeLabel.setTextFill(Color.RED);
pushMeLabel.setFont(Font.font("Arial", 20));
```

Next we have the code for the `Button`:

```
Button pushMeButton = new Button();
pushMeButton.setText("Type something in the box then push me");
pushMeButton.setOnAction(e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText()));
```

We have already seen how to create and code a button, so this should be familiar to you. Look carefully a the lambda expression, which is explained in Fig. 10.10.

Having done all this, we create and configure a `VBox`, add the three components, and then add the `VBox` to the scene.

```
VBox root = new VBox();
root.setSpacing(10);
root.setAlignment(Pos.CENTER);

root.getChildren().addAll(pushMeTextField, pushMeButton, pushMeLabel);

Scene scene = new Scene(root, 350, 150);
```

Finally we add the scene to the stage, then add a title and make it visible.

```
stage.setScene(scene);
stage.setTitle("Push Me");
stage.show();
```

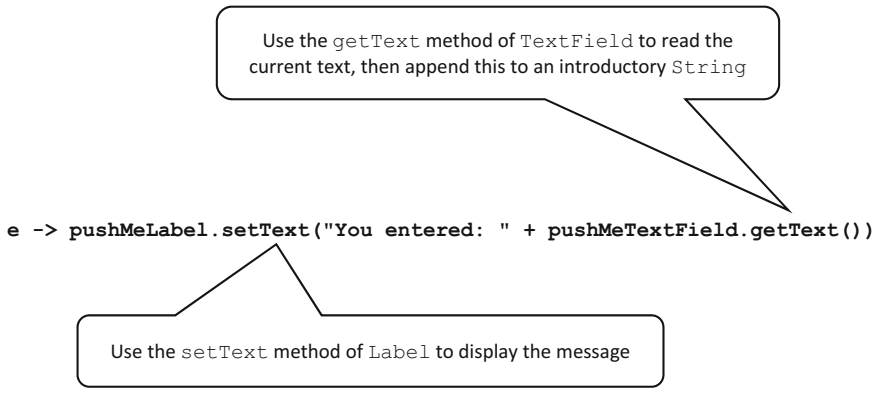


Fig. 10.10 The lambda expression explained

10.8 A Graphical User Interface (GUI) for the *Oblong* Class

Up till now, when we wanted to write programs that utilize our classes, we have written text-based programs. In most cases, however, you will be wanting to a create graphical user interface (GUI) for your classes. Let's do this for the `Oblong` class that we developed in Chap. 8. The sort of interface we are talking about is shown in Fig. 10.11.

Here is the code for the GUI:

***Oblong*GUI**

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;


public class OblongGUI extends Application
{
    // create an object of the Oblong class as an attribute
    private Oblong testOblong = new Oblong(0, 0);

    @Override
    public void start(Stage stage)
    {
        // create and configure text fields for input
        TextField lengthField = new TextField();
        lengthField.setMaxWidth(50);

        TextField heightField = new TextField();
        heightField.setMaxWidth(50);


        // create and configure a non-editable text area to display the results
        TextArea display = new TextArea();
        display.setEditable(false);
        display.setMinSize(210,50);
        display.setMaxSize(210,50);


        // create and configure Labels for the text fields
        Label lengthLabel = new Label("Length");
        lengthLabel.setTextFill(Color.RED);
        lengthLabel.setFont(Font.font("Arial", 20));

        Label heightLabel= new Label("Height");
        heightLabel.setTextFill(Color.RED);
        heightLabel.setFont(Font.font("Arial", 20));


        // create and configure a button to perform the calculations
        Button calculateButton = new Button();
        calculateButton.setText("Calculate");
        calculateButton.setOnAction( e ->
                    {
                        // check that fields are not empty
                        if(lengthField.getText().isEmpty() || heightField.getText().isEmpty())
                        {
                               display.setText("Length and height must be entered");
                        }
                        else
                        {
                            //convert text input to doubles and set the length and height of the Oblong
                            testOblong.setLength(Double.parseDouble(lengthField.getText()));
                            testOblong.setHeight(Double.parseDouble(heightField.getText()));

                            // use the methods of Oblong to calculate the area and perimeter
                            display.setText("The area is: " + testOblong.calculateArea()
                                                            + "\n" + "The perimeter is: "
                                                               + testOblong.calculatePerimeter());
                         }
                    }
                                );

        // create and configure an HBox for the labels and text inputs
        HBox inputComponents = new HBox(10);
```

```
        inputComponents.setAlignment(Pos.CENTER);
        inputComponents.getChildren().addAll(lengthLabel, lengthField, heightLabel, heightField);

        // create and configure a vertical container to hold all the components
        VBox root = new VBox(25);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(inputComponents, calculateButton, display);

        // create a new scene and add it to the stage
        Scene scene = new Scene(root, 350, 250);
        stage.setScene(scene);
        stage.setTitle("Oblong GUI");
        stage.show();
    }

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

}
```

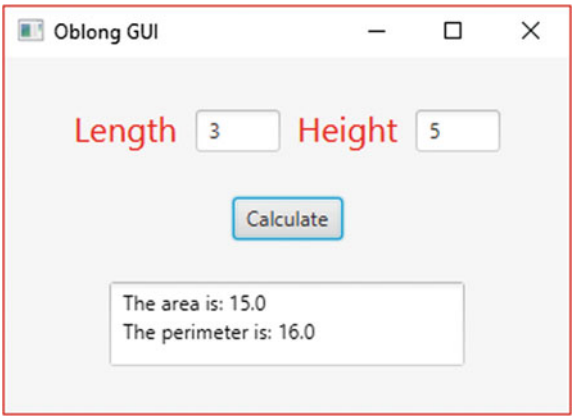


Fig. 10.11 A GUI for the *Oblong* class

In order to connect a GUI to a class, we create an object of that class within the GUI class—and as you can see that is what we have done here. We have declared an attribute, `testOblong`, which we have initialized as a new `Oblong` with a length and height of zero (since the user hasn't entered anything yet):

```
private Oblong testOblong = new Oblong(0,0);
```

After this we declare the graphics components; the only one of these that you have not yet come across is the `TextArea`, which is the large text area that you see in Fig. 10.8, where the area and perimeter of the oblong will be displayed. As you can see, it is a useful component for entering and displaying text, although this time we are using it only to display text, not to enter it. We have declared and configured it as follows:

```
TextArea display = new TextArea();
display.setEditable(false);
display.setMinSize(210,50);
display.setMaxSize(210,50);
```

We have prevented the possibility of entering text by the use of the `setEditable` method, and we have given it a fixed size by calling both the `setMinSize` and `setMaxsize` methods.

The only other thing we need to draw your attention to is the lambda expression that we have sent into the `setOnAction` method of the calculate button.

```
calculateButton.setOnAction( e ->
          {
               // check that fields are not empty
               if(lengthField.getText().isEmpty() || heightField.getText().isEmpty())
               {
                    display.setText("Length and height must be entered");
               }
               else
               {
                    //convert text input to doubles and set the length and height of the Oblong
                    testOblong.setLength(Double.parseDouble(lengthField.getText()));
                    testOblong.setHeight(Double.parseDouble(heightField.getText()));

                    // use the methods of Oblong to calculate the area and perimeter
                     display.setText("The area is: " + testOblong.calculateArea()
                                                          + "\n" + "The perimeter is: "
                                                               + testOblong.calculatePerimeter());
               }
          }
                    );
```

The first thing that we do here is to check that something has actually been entered. We do this by reading the `String` that is currently in the `TextField` by calling its `getText` method, and then calling the `isEmpty` method of `String`.

If either one of the fields is empty then an error message is displayed; otherwise we continue with the task. We could have, if we had wanted to, done some more input validation—for example we could have checked that zeros or negative numbers hadn't been entered. Or, if we wanted to be very strict about the definition of "oblong", we could have checked that the two adjacent sides were not equal. These are left as exercises at the end of the chapter.

If there is no error, we use the `setLength` and `setHeight` methods of `Oblong` to set the length and the height of `testOblong` to the values entered. However, these methods expect to receive **`double`**s—but the `getText` method of `TextField`, which we use to see what has been entered, returns a `String`!

We must therefore perform a conversion. To do this we use the `parseDouble` method of the `Double` class—one of the wrapper classes you learnt about in Chap. 8. `parseDouble` takes a `String` and converts it to a **`double`**:

```
testOblong.setLength(Double.parseDouble(lengthField.getText()));
testOblong.setHeight(Double.parseDouble(heightField.getText()));
```

It might have occurred to you that if the `String` did not contain a number, then this would cause a problem. We will show you how we deal with this sort of error in Chap. 14.

You should note that had we wanted to convert the `String`s to **`int`**s, we would have used the `parseInt` method of the `Integer` class.

Incidentally, if you want to do this the other way round and convert a **`double`** or an **`int`** to a `String` you can do so simply by concatenating it onto an empty `String`, as shown in the examples below:

```
String s = "" + 3;
```

or:

```
String s = "" + 3.12;
```

If you take a look at the rest of the code you will see that we have arranged our items by using an `HBox` to hold the labels and fields for input so they are lined up horizontally, and then used a `VBox` to line this up vertically with the button and the display area. This is something you should be getting used to by now, so we can move on to the next section where we explain more about how to use these boxes, as well as other containers, each of which lays out the components in a different way.

10.9 Containers and Layouts

You have already seen how much we can achieve just with an `HBox` and a `VBox`—we have found these containers to be very versatile, and for simple applications you can do an enormous amount just with these two containers. So we start this section telling you a bit more about what we can do with these, and then we go on to show you some other containers with different layout policies.

10.9.1 More About *`HBox`* and *`VBox`*

In Fig. 10.12 you can see twelve `VBox`es (although they could have been `HBox`es because each one contains only one component), organized in four groups of three. We have drawn a border around each one and coloured the background (we will show you how to do this in a moment). Each box contains a `Button`—the presence of the border shows the effect of setting the alignment to different values.

You will recall that the only alignment we have seen so far is `Pos.CENTER`, but there are 11 others that we can choose from. Most are self-explanatory—but the three on the right need a little explanation. `Pos.BASELINE_LEFT`, `Pos.BASELINE_CENTER` and `Pos.BASELINE_RIGHT` place the component in the

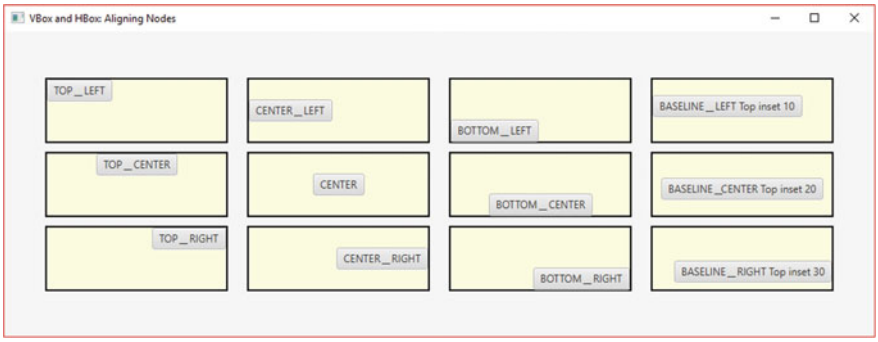


Fig. 10.12 Aligning nodes

lowest position available, and are most relevant to text fields where we want the text to appear at the bottom of a window—as in a chat application, for example. In our diagram we have set our boxes to have a top inset, and the buttons are then positioned accordingly.

To set some insets on a component we use the `setPadding` method, with a statement such as this:

```
box.setPadding(new Insets(10, 20, 10, 20));
```

The parameters to the `Insets` constructor are all **`double`**s, and define the insets for the top, right, bottom and left insets respectively. A single parameter would set all four insets to the same value.

Here `box` could be any component such as a `VBox` or `HBox`, or a `Button`, `Label` or `TextField` for example, as all these inherit the `setPadding` method from a higher level class.

We also promised to show you how to create a border and background colour. To get the black borders that you see in the diagram we did the following:

```
box.setBorder(new Border(new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
                                          new CornerRadii(0),new BorderWidths(2))));
```

This does seem to be a rather complicated process, but if you study it you can easily see what's going on. The `setBorder` method requires an object of the class `Border`. To create this we have to send the constructor an object of `Border-Stroke`, which requires four arguments. The argument names should speak for themselves, except perhaps for `ConerRadii`, which determines the roundness of the corners; in this example a value of zero produces square corners. To achieve the background colour we did this:

```
box.setBackground(new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                                    new CornerRadii(0), new Insets(0))));
```

Again, although it looks complicated at first sight, it isn't hard to work out what is actually going on.

All of these border and background classes reside in `javafx.scene.layout`.

We will do some more work on borders and colours in Sect. 10.10.

10.9.2 GridPane

A `GridPane` is a very useful container. As its name suggests, it lays out the components in a matrix of rows and columns, as shown in Fig. 10.13.

The following lines of code would create a `GridPane` object, and configure it to position the components in the centre of each cell, and to leave a 10 pixel vertical gap (the gap between rows) and a 5 pixel horizontal gap (the gap between columns).

```
GridPane pane = new GridPane();
pane.setAlignment(Pos.CENTER);
pane.setVgap(10);
pane.setHgap(5);
```

The really good thing about a `GridPane` is its flexibility—it is sized dynamically as you insert the components, as are the individual cells. For example we could insert a `Button`, `myButton`, to the above `GridPane` object as follows:

```
pane.add(myButton, columnIndex, rowIndex);
```

`columnIndex` and `rowIndex` are **`int`**s—we start counting from zero, so the following line of code would add the button in column 4, row 6:

Fig. 10.13 Using a *GridPane*

```
pane.add(myButton, 3, 5);
```

Because of its flexibility, `GridPane` can be very versatile and will allow you to create quite complex presentations—the best thing you can do, as usual, is to try some experiments of your own.

10.9.3 StackPane

A `StackPane`, as its name suggests, stacks components on top of each other. In Fig. 10.14 we have created three different coloured rectangles, each one smaller than the previous one, and added them to the `StackPane` from largest to smallest.

The components are added as before by calling the `getChildren` method inherited by `StackPane`. Here we have chosen to align them in the centre of the pane. You can see that there is a lot of potential here for drawing interesting shapes, and as before you should conduct your own experiments.

There is another very useful way in which we can utilize this container, by creating several items each the same size and stacking them one on top of the other. We can then choose which one is visible, so that the stack behaves like a pack of cards. In this way we can create a series of screens which could, for example, be forms that need to be filled in progressively. We will show you an example of this in Chap. 17.

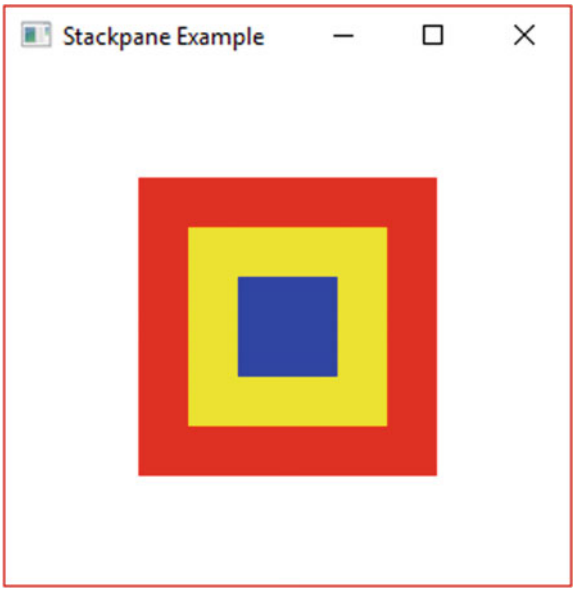


Fig. 10.14 Using a *StackPane*

10.9.4 *FlowPane* and *BorderPane*

As we explained in Sect. 10.2, prior to the existence of JavaFX the principal package for producing graphics in Java was Swing. With Swing, the way that a container arranged its components (its *layout policy*) was determined by associating a particular *layout manager* with it. Two of the most common of these were `FlowLayout` and `BorderLayout`. As you have already seen, JavaFX works by providing different containers, each with its own layout policy—and with the existence of `VBox` and `HBox`, flow layout and border layout policies are not as useful as they were in the days of Swing. However, two containers that produce similar results to these are nonetheless provided in JavaFX—these are `FlowPane` and `BorderPane`.

A `FlowPane` operates by arranging the items in a row, in the order that they were added, starting a new row when necessary. If the window is resized, the components move about accordingly, as shown in Fig. 10.15.

As with the `VBox` and `HBox`, nodes are added to a `FlowPane` object by calling the `getChildren` method and then using `add` or `addAll`.

The `BorderPane` operates by dividing the window into five regions called *Top*, *Bottom*, *Left*, *Right*, *Center*, as shown in Fig. 10.16.

If we use a `BorderPane` the components don't get moved around when the window is resized, as you can see from Fig. 10.17.

To add an item called `myButton`, for example, to the top region of a `BorderPane` named `pane`, we would do the following:

```
pane.setTop(myButton);
```

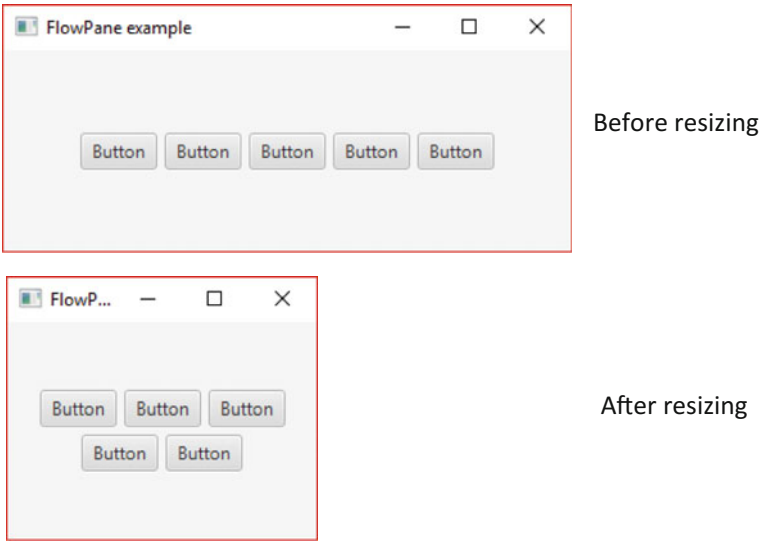


Fig. 10.15 The effect of resizing when using *FlowPane*

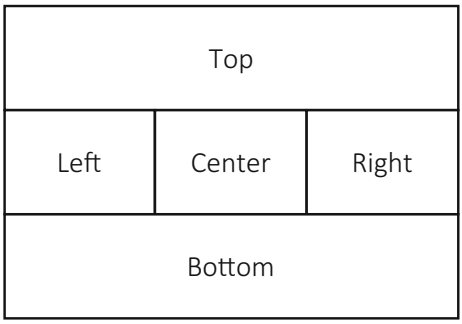


Fig. 10.16 The *BorderPane* layout strategy

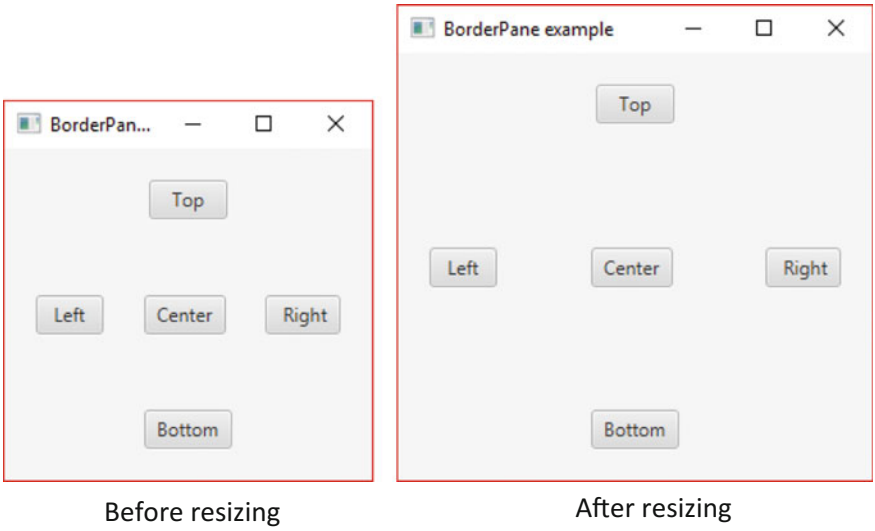


Fig. 10.17 The effect of resizing when using *BorderPane*

Similarly, for the other regions `BorderPane` has methods `setBottom`, `setLeft`, `setRight` and `setCenter`.

10.10 Borders, Fonts and Colours

You have already seen examples of how we can place borders around components and how we can determine the colour of text and background, and define different fonts. In this section we will give you a few more pointers as to how to enhance your applications with these features.

10.10.1 Borders

In Fig. 10.18 we see six buttons all with different border styles.

You will recall from Sect. 10.9.1 that in order to place a border round a component such as a button we use the `setBorder` method—this requires an object of the class `Border`, which in turn is created with an object of the class `BorderStroke`.

So, for example, for the first button (`button1`) we created the following `BorderStroke`:

```
BorderStroke stroke1
    = new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID, new CornerRadii(0), new BorderWidths(6)
```

We then applied this to our button:

```
button1.setBorder(new Border(stroke1));
```

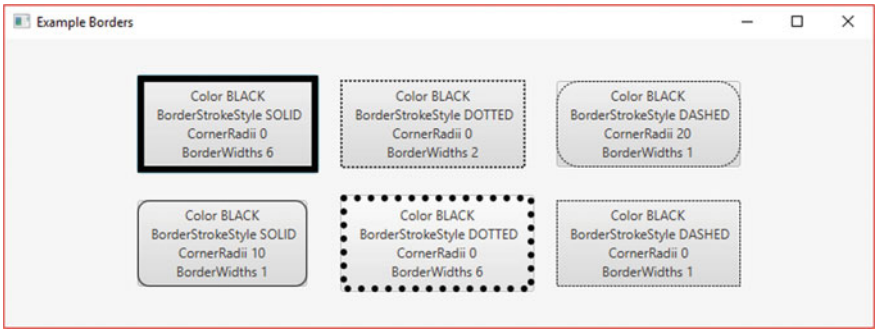


Fig. 10.18 Examples of border styles

Fig. 10.19 Titled panes

One further effect is to create a titled border as shown in Fig. 10.19.

To do this we use a `TitledPane`, which is actually a control class, and resides in the package `javafx.scene.control`. The pane comes with a downward arrow that allows you to collapse the pane, as you see in the second box in Fig. 10.19. If you want a title only for decorative purposes you can turn this feature off as in the first box. In each case a `VBox` was added to the pane; other nodes could then be added to the `VBox`.

The code for creating the first pane is as follows:

```
VBox box1 = new VBox();
box1.setMinSize(100, 75);
TitledPane firstPane = new TitledPane("Pane 1", box1);
firstPane.setCollapsible(false);
```

As you can see, the last line turns off the collapsible feature.

10.10.2 Fonts

Figure 10.20 shows some different font examples.

In our example we have applied our fonts to various `Text` objects. To achieve this we used the `setFont` method of `Text`—controls such as `Button` also have a `setFont` method.

There are two ways of doing this. Firstly we can use the `font` method of `Font`. In our first example, assuming the `Text` object is called `caption1`, we would have written:

```
Font font1 = Font.font ("Verdana", FontWeight.BOLD, FontPosture.ITALIC, 12);
caption1.setFont(font1);
```

Fig. 10.20 Font examples

There are a number of different versions of the `font` method which you can look up on the Oracle™ website.

The other way of doing this is to create a new font with one of two constructors. The first requires only the font size (a **`double`**) and uses the default system font. The second requires the name of the font and the size. Our fifth example in Fig. 10.20 was achieved like this:

```
Font font5 = new Font("Calibri", 40);
```

Underlining is done by using the `setUnderline` method of a component.

10.10.3 Colours

You have seen how the JavaFX `Color` class has a great many predefined colours. However we can, if we wish, create our own colours. Those of you who have studied some elementary physics will know that there are three primary colours, red, green and blue[1]; all other colours can be obtained by mixing these in different proportions.

Mixing red, green and blue in equal intensity produces white light; the colour we know as black is in fact the absence of all three. Mixing equal amounts of red and green (and no blue) produces yellow light; red and blue produce a mauvish colour called magenta; and mixing blue and green produces cyan, a sort of turquoise.

The `Color` class has a method called `rgb` which allows us to specify the intensity of each of the primary colours, red, green, blue respectively. The intensity for each colour can range from a minimum of zero to a maximum of 255. So there are 256 possible intensities for each primary colour, and the total number of different colours available to us is therefore $256 \times 256 \times 256$, or 16,777,216.

As an example, you could create the following colour:

```
Color color1 = Color.rgb(200, 100, 50);
```

You could then, for example, set the text colour of a button, `button1`, like this:

```
button1.setTextFill(color1);
```

This particular combination produces a kind of orange—you should experiment with different values.

[1]Don't confuse this with the mixing of coloured paints, where the rules are different. In the case of mixing coloured lights (as on a computer monitor) we are dealing with reflection of light—in the case of paints we are dealing with absorption, so the primary colours, and the rules for mixing, are different. For paints the primary colours are red, blue and yellow.

10.11 Number Formatting

We are now going to tell you about a technique that is not specifically related to JavaFX, but is something you will often want to use in your graphics applications. We frequently need our numerical output to appear in a suitable format—for example with no more than two numbers after the decimal point—or perhaps with *exactly* two numbers after the decimal point. In order to do this we make use of the `DecimalFormat` class that resides in the `java.text` package. We would need the following import statement:

```
import java.text.DecimalFormat;
```

Once you have access to this class you can create `DecimalFormat` objects in your program. These objects can then be used to format decimal numbers for you. The `DecimalFormat` constructor has one parameter, the format string. This string instructs the object on how to format a given decimal number. Some of the important elements of such a string are given in Table 10.1.

In the example in the next section we are going to create the following `DecimalFormat` object:

```
DecimalFormat df = new DecimalFormat("0.0#");
```

Here the `DecimalFormat` object, `df`, is being informed on how to format any decimal numbers that may be given to it, as shown in see Fig. 10.21.

Having created a `DecimalFormat` object, we could then create a `String`, `s`, from a **`double`**, `d`, as follows:

```
String s = df.format(d);
```

Table 10.1 Special `DecimalFormat` characters

Character	Meaning
.	Insert a decimal point
,	Insert a comma
0	Display a single digit
#	Display a single digit or empty if no digit present

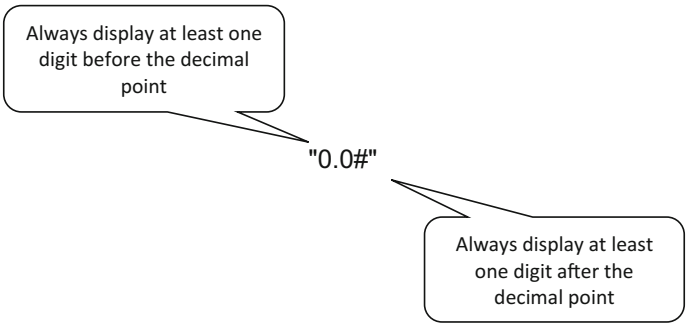


Fig. 10.21 A format String used with the *DecimalFormat* class

The program below shows some examples:

NumberFormatExample

```
import java.text.DecimalFormat;

public class NumberFormatExample
{
   public static void main(String[] args)
   {
      double number = 4376.7863;

      DecimalFormat df1 = new DecimalFormat("###,##0.0#");
      DecimalFormat df2 = new DecimalFormat("###000.00");
      DecimalFormat df3 = new DecimalFormat("00.0");
      DecimalFormat df4 = new DecimalFormat("000000.00000");
      DecimalFormat df5 = new DecimalFormat("000,000.00####");

      System.out.println(df1.format(number));
      System.out.println(df2.format(number));
      System.out.println(df3.format(number));
      System.out.println(df4.format(number));
      System.out.println(df5.format(number));
   }
}
```

The output from the above program is as follows:

```
4,376.79
4376.79
4376.8
004376.78630
004,376.7863
```

In the next chapter you will see how a similar technique can be used to output numbers as a particular currency.

10.12 A Metric Converter

Our final example in this chapter is a practical application that pulls together everything that we've learnt so far about JavaFX. Most of the world uses the metric system; however, if you are in the United Kingdom as we are, then you will still be only halfway there—sometimes using kilograms and kilometres, sometimes pounds and miles. Of course if you are in the USA (and you are not a scientist or an engineer) you will still be using the old imperial values for everything. Some might say it's time that the UK and the USA caught up with the rest of the world! But until that happens this little program, which converts back and forth from metric to imperial, is going to be very handy.

We will be building a `MetricConverter` class. Figure 10.22 shows what we are going to achieve.

The application will have a `VBox` at its root—this `VBox` will hold three `HBox`es, one for each row that you see in Fig. 10.22. Each row requires two `Button`s which will perform the calculations in either direction; these two `Button`s will be held together in a VBox.

The code for the `MetricConverter` class is now presented; it looks quite long, but most of it is just more of what you already know. Take a look at the code and then we will draw your attention to a few points.

MetricConverter

```
import java.text.DecimalFormat;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class MetricConverter extends Application
{
    @Override
    public void start(Stage stage)
    {
       DecimalFormat df = new DecimalFormat("0.0#"); // see discussion in previous section

       // first the components for converting back and forth from inches to centimetres

       // create input fields, and labels to show the units
       TextField cmText = new TextField();
       Label cmLabel = new Label("Cm");
       TextField inchText = new TextField();
       Label inchLabel = new Label("Inches");

       // create buttons to perform the calculations
       Button cmToInchButton = new Button(" ===> ");
       Button inchToCmButton = new Button(" <=== ");
```

```
        // create a VBox to hold the buttons
        VBox inchCmButtons = new VBox();
        inchCmButtons.getChildren().addAll(cmToInchButton, inchToCmButton);

        // create an HBox to hold all the items for the first row
        HBox inchCmPanel = new HBox(10); // compound container
        inchCmPanel.getChildren().addAll(cmText, cmLabel, inchCmButtons, inchText, inchLabel);
        inchCmPanel.setAlignment(Pos.CENTER);

        // next the components for converting back and forth from miles to kilometres

        // create input fields, and labels to show the units
        TextField kmText = new TextField();
        Label kmLabel = new Label("Km");
        TextField mileText = new TextField();
        Label mileLabel = new Label("Miles  "); // extra spaces make all labels the same length

        // create buttons to perform the calculations
        Button kmToMileButton = new Button(" ===> ");
        Button mileToKmButton = new Button(" <=== ");

        // create a VBox to hold the buttons
        VBox mileKmButtons = new VBox();
        mileKmButtons.getChildren().addAll(kmToMileButton, mileToKmButton);

        // create an HBox to hold all the items for the second row
        HBox mileKmPanel = new HBox(10);
        mileKmPanel.getChildren().addAll(kmText, kmLabel, mileKmButtons, mileText, mileLabel);
        mileKmPanel.setAlignment(Pos.CENTER);

        // finally the components for converting back and forth from pounds to kilograms

        // create input fields, and labels to show the units
        TextField kgText = new TextField();
        Label kgLabel = new Label("Kg "); // extra spaces make all labels the same length
        TextField poundText = new TextField();
        Label poundLabel = new Label("Lb        "); // extra spaces make all labels the same length

        // create buttons to perform the calculations
        Button kgToPoundButton = new Button(" ===> ");
        Button poundToKgButton = new Button(" <=== ");

        // create a VBox to hold the buttons
        VBox poundKgButtons = new VBox();
        poundKgButtons.getChildren().addAll(kgToPoundButton, poundToKgButton);

        // create an HBox to hold all the items for the third row
        HBox poundKgPanel = new HBox(10);
        poundKgPanel.getChildren().addAll(kgText, kgLabel, poundKgButtons, poundText, poundLabel);
        poundKgPanel.setAlignment(Pos.CENTER);

        // create a VBox to hold all three rows
        VBox root = new VBox(10);
        root.getChildren().addAll(inchCmPanel, mileKmPanel, poundKgPanel);
        root.setAlignment(Pos.CENTER);

        // write the code for the buttons
        cmToInchButton.setOnAction( e -> {
                                            String s = new String(cmText.getText());
                                            double d = Double.parseDouble(s);
                                            d = d / 2.54;
                                            s = df.format(d);
                                            inchText.setText(s);
                                        }
                                  );

        inchToCmButton.setOnAction( e -> {
                                            String s = new String(inchText.getText());
```

```
                                        double d = Double.parseDouble(s);
                                        d = d * 2.54;
                                        s = df.format(d);
                                        cmText.setText(s);
                                    }
                                );

        kmToMileButton.setOnAction( e -> {
                                        String s = new String(kmText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d / 1.609;
                                        s = df.format(d);
                                        mileText.setText(s);
                                    }
                                );

        mileToKmButton.setOnAction( e -> {
                                        String s = new String(mileText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d * 1.609;
                                        s = df.format(d);
                                        kmText.setText(s);
                                    }
                                );

        kgToPoundButton.setOnAction( e -> {
                                        String s = new String(kgText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d * 2.2;
                                        s = df.format(d);
                                        poundText.setText(s);
                                     }
                                );

        poundToKgButton.setOnAction( e -> {
                                        String s = new String(poundText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d / 2.2;
                                        s = df.format(d);
                                        kgText.setText(s);
                                     }
                                );

        // create a new scene
        Scene scene = new Scene(root);

        // add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Metric Converter");
        stage.setWidth(500);
        stage.setHeight(250);
        stage.show();
    }

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

As we have said, there is nothing here that you haven't seen before. But do study the code for the buttons. The code for each one is very similar, differing only in the particular calculation that needs to be made. Look carefully at how we have made use to the `DecimalFormat` class that we described in the previous section. We declared an object of this class at the beginning of the `start` method, and applied it to each of the buttons. The particular format we chose will always display at least one digit after the decimal point once the calculation has been made; you might want to try other formats.

We will be returning to graphics and JavaFX in the second semester. For the time being you already have a very big repertoire with which to build graphical applications. Please do explore the classes available, and try your own programs—everything

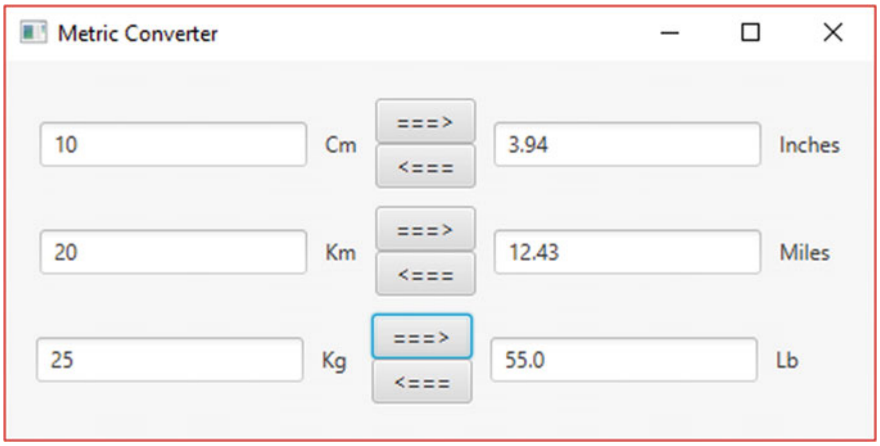


Fig. 10.22 The metric converter

you need is on the Oracle™ site. The more you try things out, the more you will learn and the more you will become familiar with what is available. And of course the more fun you will have!

10.13 Self-test Questions

1. Briefly describe the history of graphics programming in Java.
2. What is the name of the three methods that are called when a JavaFX application is launched? what is the purpose of each?
3. Which containers have been used in the following two scene graphics?

(a)

(b)

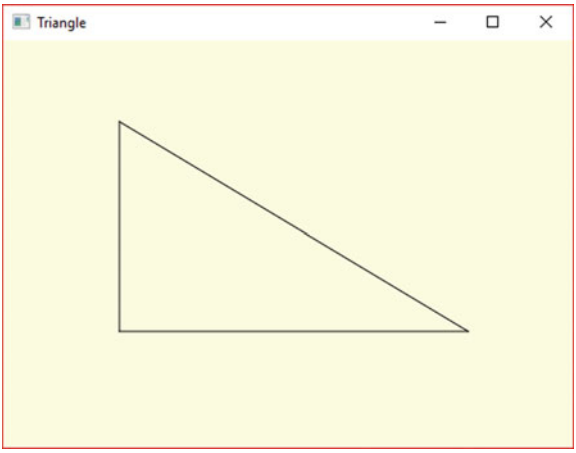
4. Describe how the following containers lay out the nodes that they contain:

   (a) a `VBox`; (b) an `HBox` (c) a `GridPane` (d) a `StackPane`

   (e) a `FlowPane` (f) a `BorderPane`

10.14 Programming Exercises

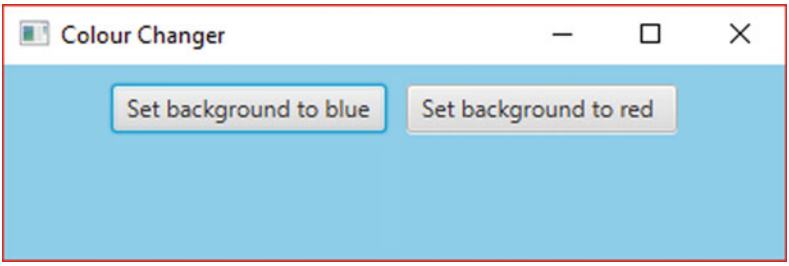
1. Implement a few of the programs that we have developed in this chapter, and experiment with different settings in order to change some the features—for example size, colour, position and so on.
2. Consider some changes or additions you could make to the `PushMe` class. For example, pushing the button could display your text in upper case—or it could say how many letters it contains. Maybe you could add some extra buttons.
3. The application shown below produces a triangle:



   See if you can write the code to produce this triangle using three lines. We suggest the following vertices:

   (100, 70) (100, 250) (400, 250).

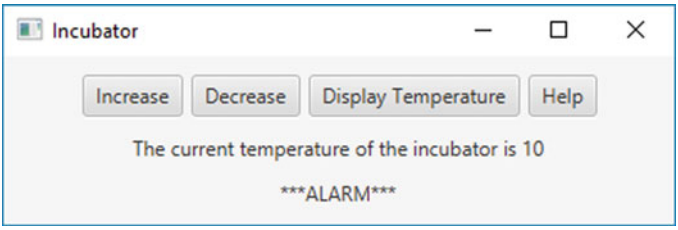
4. Below you see an application called `ColourChanger` which produces the following graphic in which two buttons can be used to change the background colour:



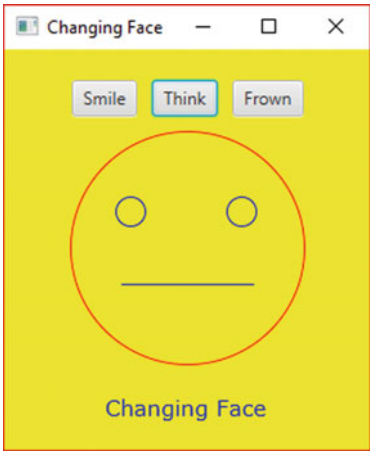
Write the code for this application.

5. Add some additional features to the `MetricConverter`—for example Celsius to Fahrenheit or litres to pints.

6. Look back at the final version of the `Incubator` class that you wrote in programming Exercise 5 of Chap. 8. Now you can create a graphical user interface for it, instead of a text menu. A suggested interface is shown below:



7. Below is a variation on the `ChangingFace` class, which has three possible moods!



Rewrite the original code to produce this new design.

Hint: *The easiest way to achieve the "thinking" mouth is to set the* `radius` *attribute of* `Arc` *to zero.*
A more difficult approach would be to draw a line, but then you would have to create three different mouths, and each time check which was the current mouth, remove that, and add the mouth you require. It is perfectly possible to do this because the list of nodes returned by the `getChildren` *method has methods named* `contains` *and* `remove` *as well as the* `add` *and* `addAll` *methods that you are used to.*

8. Look back at the `OblongGUI` that we developed in Sect. 10.8. Modify the code so that as well as checking that the values have been entered, it also checks that the values entered are not zero, and that the length and height are not equal.