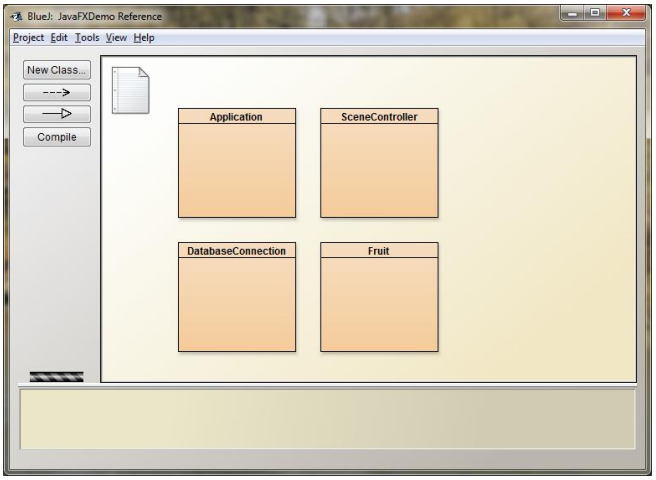
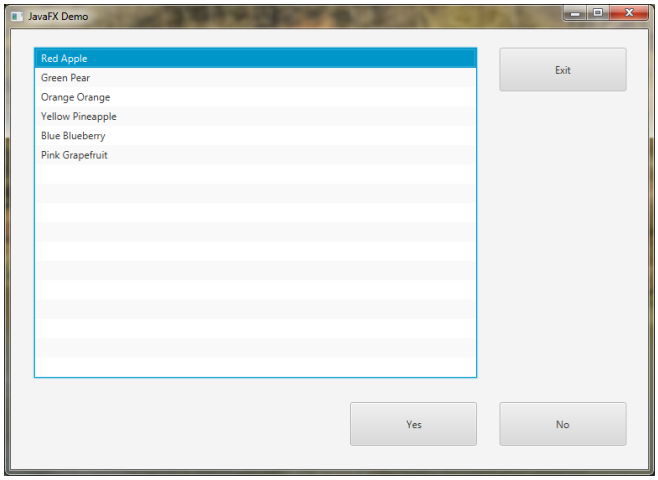
A-Level
Computer Science

&

Demonstration Application

Created using BlueJ, Scene Builder and SQLite Studio





Java and FXML Code Listing

GitHub repository for this demonstration:

`https://github.com/SteveBirtles/JavaFXDemo.git`

Application.java (Part of the Controller)

```
import javafx.application.Platform;
import javafx.embed.swing.JFXPanel;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;

/* This is the first class for our JavaFX application that contains the main method. As well as setting
 * everything in motion, it also contains a method to stop the program running. */
public class Application
{
    /* Our app needs a connection to the database, and this is all handled by the DatabaseConnection
     * class. A public static variable, called database, points to the DatabaseConnection object. */
    public static DatabaseConnection database;

    public static void main(String args[])
    {
        /* JavaFX apps run in a different processing thread to the main method so that they keep running
         * after the main method has completed. The 'start' method is invoked when this thread starts. */
        JFXPanel panel = new JFXPanel();
        Platform.runLater(() -> start());
    }

    private static void start()
    {
        try
        {
            database = new DatabaseConnection("Test.db");     // Initiate the database connection.

            /* Load the first fxml file that will create our first JavaFX scene. */
            FXMLLoader loader = new FXMLLoader(Application.class.getResource("DemoScene.fxml"));

            /* Each scene requires a Window, called a stage. The following creates and displays the stage. */
            Stage stage = new Stage();
            stage.setTitle("JavaFX Demo");
            stage.setScene(new Scene(loader.load()));
            stage.show();

            /* Loading the scene will have resulted in the creation of a scene controller. In order for this
             * controller to have direct access to the stage its scene is displayed, the stage is passed to the
             * controller's prepareStage method. */
            SceneController controller = loader.getController();
            controller.prepareStageEvents(stage);
        }
        catch (Exception ex)
        {
            System.out.println(ex.getMessage());
            terminate();
        }
    }

    /* The following method can be called from any controller class and will terminate the application. */
    public static void terminate()
    {
        System.out.println("Closing database connection and terminating application...");

        if (database != null) database.disconnect();    // Close the connection to the database (if it exists!)
        System.exit(0);                                 // Finally, terminate the entire application.
    }

}
```

SceneController.java (Part of the Controller)

```
import javafx.fxml.FXML;
import javafx.stage.Stage;
import javafx.scene.control.Button;
import javafx.scene.control.ListView;
import javafx.scene.layout.Pane;
import javafx.event.EventHandler;
import javafx.stage.WindowEvent;
import java.util.List;

public class SceneController
{
    /* The stage that the scene belongs to, required to catch stage events and test for duplicate controllers. */
    private static Stage stage;

    /* These FXML variables exactly corrispond to the controls that make up the scene, as designed in Scene
     * Builder. It is important to ensure that these match perfectly or the controls won't be interactive. */
    @FXML   private Pane backgroundPane;
    @FXML   private Button yesButton;
    @FXML   private Button noButton;
    @FXML   private Button exitButton;
    @FXML   private ListView listView;

    public SceneController()          // The constructor method, called first when the scene is loaded.
    {
        System.out.println("Initialising controllers...");

        /* Our JavaFX application should only have one initial scene. The following checks to see
         * if a scene already exists (deterimed by if the stage variable has been set) and if so
         * terminates the whole application with an error code (-1). */
        if (stage != null)
        {
            System.out.println("Error, duplicate controller - terminating application!");
            System.exit(-1);
        }

    }

    @FXML   void initialize()           // The method automatically called by JavaFX after the constructor.
    {
        /* The following assertions check to see if the JavaFX controls exists. If one of these fails, the
         * application won't work. If the control names in Scene Builder don't match the variables this fails. */
        System.out.println("Asserting controls...");
        assert backgroundPane != null : "Can't find background pane.";
        assert yesButton != null : "Can't find yes button.";
        assert noButton != null : "Can't find yes button.";
        assert exitButton != null : "Can't find exit button.";
        assert listView != null : "Can't find list box.";

        /* Next, we load the list of fruit from the database and populate the listView. */
        System.out.println("Populating scene with items from the database...");
        @SuppressWarnings("unchecked")
        List<Fruit> targetList = listView.getItems();  // Grab a reference to the listView's current item list.
        Fruit.readAll(targetList);                     // Hand over control to the fruit model to populate this list.
    }

    /* In order to catch stage events (the main example being the close (X) button being clicked) we need
     * to setup event handlers. This happens after the constructor and the initialize methods. Once this is
     * complete, the scene is fully loaded and ready to use. */
    public void prepareStageEvents(Stage stage)
    {
        System.out.println("Preparing stage events...");

        this.stage = stage;

        stage.setOnCloseRequest(new EventHandler<WindowEvent>() {
                public void handle(WindowEvent we) {
                    System.out.println("Close button was clicked!");
                    Application.terminate();
                }
            });
    }
```

```
    /* The next three methods are event handlers for clicking on the buttons.
     * The names of these methods are set in Scene Builder so they work automatically. */
    @FXML   void yesClicked()
    {
        System.out.println("Yes was clicked!");
    }

    @FXML   void noClicked()
    {
        System.out.println("No was clicked!");
    }

    @FXML   void exitClicked()
    {
        System.out.println("Exit was clicked!");
        Application.terminate();        // Call the terminate method in the main Application class.
    }

    /* This method, set in SceneBuilder to occur when the listView is clicked, establishes which
     * item in the view is currently selected (if any) and outputs it to the console. */
    @FXML   void listViewClicked()
    {
        Fruit selectedItem = (Fruit) listView.getSelectionModel().getSelectedItem();

        if (selectedItem == null)
        {
            System.out.println("Nothing selected!");
        }
        else
        {
            System.out.println(selectedItem + " (id: " + selectedItem.id + ") is selected.");
        }
    }

}
```

DemoScene.fxml (The View)

This file was generated by Scene Builder. The elements that directly link to the Java code are highlighted.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.layout.Pane?>

<Pane fx:id="backgroundPane" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
        prefHeight="524.0" prefWidth="759.0" xmlns="http://javafx.com/javafx/8"
        xmlns:fx="http://javafx.com/fxml/1" fx:controller="SceneController">

   <children>

      <Button fx:id="yesButton" layoutX="404.0" layoutY="444.0"
        onMouseClicked="#yesClicked" prefHeight="52.0" prefWidth="151.0" text="Yes" />

      <Button fx:id="noButton" layoutX="582.0" layoutY="444.0"
        onMouseClicked="#noClicked" prefHeight="52.0" prefWidth="151.0" text="No" />

      <Button fx:id="exitButton" layoutX="582.0" layoutY="22.0"
        onMouseClicked="#exitClicked" prefHeight="52.0" prefWidth="151.0" text="Exit" />

      <ListView fx:id="listView" layoutX="28.0" layoutY="22.0"
        onKeyPressed="#listViewClicked" onMouseClicked="#listViewClicked" prefHeight="393.0" prefWidth="527.0" />

   </children>

</Pane>
```

DatabaseConnection.java (Part of the Model)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;

public class DatabaseConnection {

    private Connection conection = null;

    /* This method is the constructor. When a new DatabaseConnection object is created a connection
     * to the database is established using the filename and database drive. */
    public DatabaseConnection(String dbFile)
    {
        try             // There are many things that can go wrong in establishing a database connection...
        {
            Class.forName("org.sqlite.JDBC");                               // ... a missing driver class ...
            conection = DriverManager.getConnection("jdbc:sqlite:" + dbFile); // ... or an error with the file.
            System.out.println("Database connection successfully established.");
        }
        catch (ClassNotFoundException cnfex)    // Catch any database driver error
        {
            System.out.println("Class not found exception: " + cnfex.getMessage());
        }
        catch (SQLException exception)          // Catch any database file errors.
        {
            System.out.println("Database connection error: " + exception.getMessage());
        }

    }

    /* This method is used to prepare each new query. The query isn't executed until later. */
    public PreparedStatement newStatement(String query)
    {
        PreparedStatement statement = null;
        try {
            statement = conection.prepareStatement(query);
        }
        catch (SQLException resultsexception)
        {
            System.out.println("Database statement error: " + resultsexception.getMessage());
        }
        return statement;
    }

    /* This method is used to actually execute a query that has previously been prepared. */
    public ResultSet runQuery(PreparedStatement statement)
    {
        try {
            return statement.executeQuery();
        }
        catch (SQLException queryexception)
        {
            System.out.println("Database query error: " + queryexception.getMessage());
            return null;
        }
    }

    /* Finally, this method is called when the application is terminating to close the database connection. */
    public void disconnect()
    {
        System.out.println("Disconnecting from database.");
        try {
            if (conection != null) conection.close();
        }
        catch (SQLException finalexception)
        {
            System.out.println("Database disconnection error: " + finalexception.getMessage());
        }
    }

}
```

Fruit.java (Part of the Model)

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.util.List;

/* Each table you wish to access in your database requires a model class, like this example: */
public class Fruit
{
    /* First, map each of the fields (columns) in your table to some public variables. */
    public int id;
    public String type;
    public String colour;

    /* Next, prepare a constructor that takes each of the fields as arguements. */
    public Fruit(int id, String type, String colour)
    {
        this.id = id;
        this.type = type;
        this.colour = colour;
    }

    /* A toString method is vital so that your model items can be sensibly displayed as text. */
    @Override public String toString()
    {
        return (colour + " " + type);
    }

    /* Different models will require different read and write methods. Here is an example 'loadAll' method
     * which is passed the target list object to populate. */
    public static void readAll(List<Fruit> list)
    {
        list.clear();       // Clear the target list first.

        /* Create a new prepared statement object with the desired SQL query. */
        PreparedStatement statement = Application.database.newStatement("SELECT id, fruit, colour FROM demotable");

        if (statement != null)      // Assuming the statement correctly initated...
        {
            ResultSet results = Application.database.runQuery(statement);       // ...run the query!

            if (results != null)        // If some results are returned from the query...
            {
                try {                                                           // ...add each one to the list.
                    while (results.next()) {
                        list.add( new Fruit(results.getInt("id"),
                                results.getString("fruit"),
                                results.getString("colour")) );
                    }
                }
                catch (SQLException resultsexception)       // Catch any error processing the results.
                {
                    System.out.println("Database result processing error: " + resultsexception.getMessage());
                }
            }
        }

    }

}
```