JavaFX without FXML and JavaFX Concurrency

JavaFX Concurrency Example

Reference: https://examples.javacodegeeks.com/desktop-java/javafx/javafx-concurrency-example/

Introduction

- The nodes representing UI in a Scene Graph are not thread-safe.
    - Advantage: they are faster, as no synchronization is involved.
    - Disadvantage: they need to be accessed from a single thread to avoid being in an illegal state.
- JavaFX puts a restriction that a live Scene Graph must be accessed from one and only one thread, the JavaFX Application Thread.
- This restriction indirectly imposes another restriction that a UI event should not process a long-running task, as it will make the application unresponsive.
    - The user will get the impression that the application is hung.

The given Problem

- The program displays a window as shown in the GUI. It contains three controls.
  - A Label to display the progress of a task
  - A Start button to start the task
  - An Exit button to exit the application
- When you click the Start Button, a task lasting for 10 seconds is started.

- The logic for the task is in the runTask() method, which simply runs a loop ten times. Inside the loop, the task lets the current thread, which is the JavaFX Application Thread, sleep for 1 second.

The given Problem

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxConcurrencyExample1  extends Application
{
    // Create the TextArea
    TextArea textArea = new TextArea();

    // Create the Label
    Label statusLabel = new Label("Not Started...");

    // Create the Buttons
    Button startButton = new Button("Start");
    Button exitButton = new Button("Exit");
```

The given Problem

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

    @Override
    public void start(final Stage stage)
    {
        // Create the Event-Handlers for the Buttons
        startButton.setOnAction(new EventHandler <ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                runTask();
            }
        });

        exitButton.setOnAction(new EventHandler <ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                stage.close();
            }
        });

        // Create the ButtonBox
        HBox buttonBox = new HBox(5, startButton, exitButton);
```

The given Problem

```java
        // Create the VBox
        VBox root = new VBox(10, statusLabel, buttonBox, textArea);

        // Set the Style-properties of the VBox
        root.setStyle("-fx-padding: 10;" +
                "-fx-border-style: solid inside;" +
                "-fx-border-width: 2;" +
                "-fx-border-insets: 5;" +
                "-fx-border-radius: 5;" +
                "-fx-border-color: blue;");

        // Create the Scene
        Scene scene = new Scene(root,400,300);
        // Add the scene to the Stage
        stage.setScene(scene);
        // Set the title of the Stage
        stage.setTitle("A simple Concurrency Example");
        // Display the Stage
        stage.show();
    }
```

The given Problem

```java
    public void runTask()
    {
        for(int i = 1; i <= 10; i++)
        {
            try
            {
                String status = "Processing " + i + " of " + 10;
                statusLabel.setText(status);
                textArea.appendText(status+"\n");
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

The Problem

- The program has two problems. The first one:
  - Click the Start Button and immediately try to click the Exit Button.
  - Clicking the Exit Button has no effect until the task finishes. Once you click the Start Button, you cannot do anything else on the window, except to wait for 10 seconds for the task to finish.

- The second one:
  - Inside the loop in the runTask() method, the program prints the status of the task on the standard output and displays the same in the Label in the window. You don´t see the status updated in the Label.

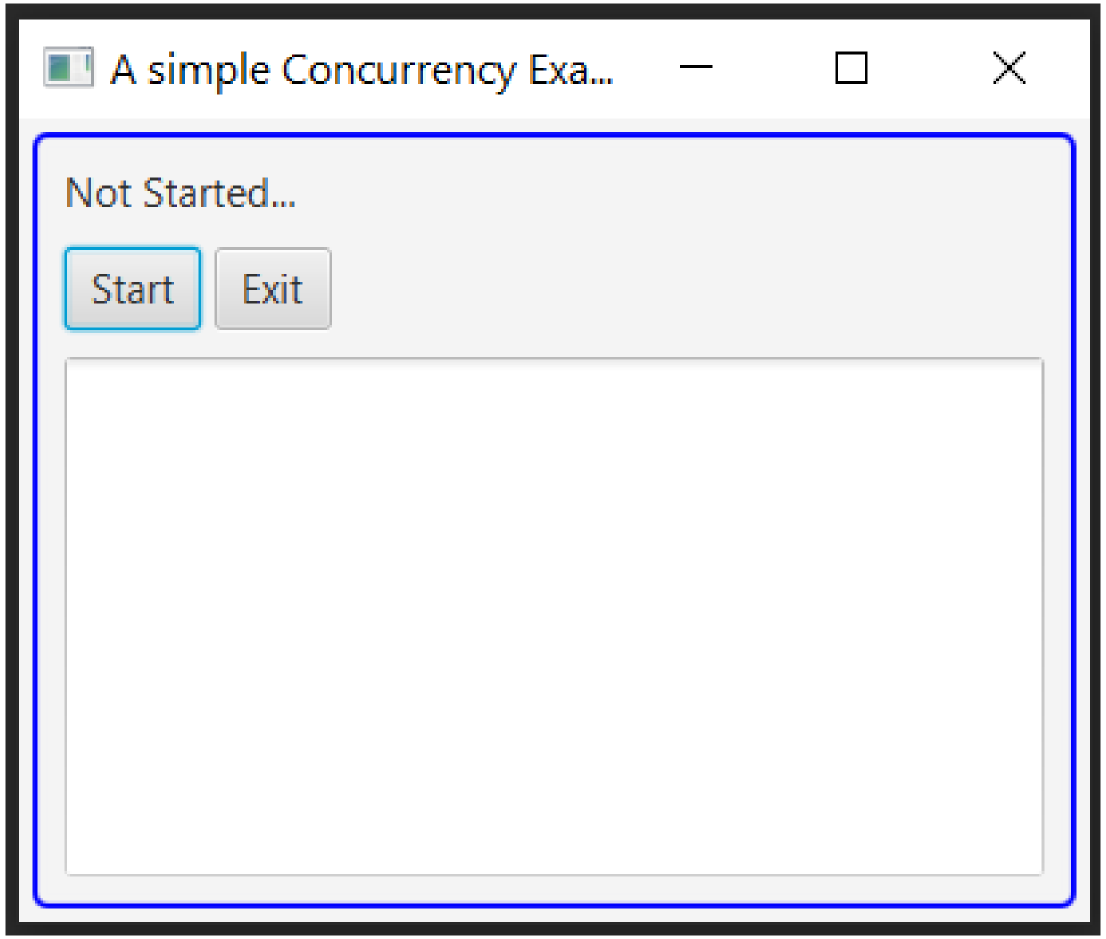
What happened?

- All UI event handlers in JavaFX run on a single thread, which is the JavaFX Application Thread.
- When the Exit Button is clicked while the task is running, an ActionEvent event for the Exit Button is generated and queued on the JavaFX Application Thread.
  - The ActionEvent handler for the Exit Button is run on the same thread after the thread is done running the runTask() method as part of the ActionEvent handler for the Start Button.
- A pulse event is generated when the Scene Graph is updated.
  - The pulse event handler is also run on the JavaFX Application Thread.
  - Inside the loop, the text property of the Label was updated ten times, which generated the pulse events.

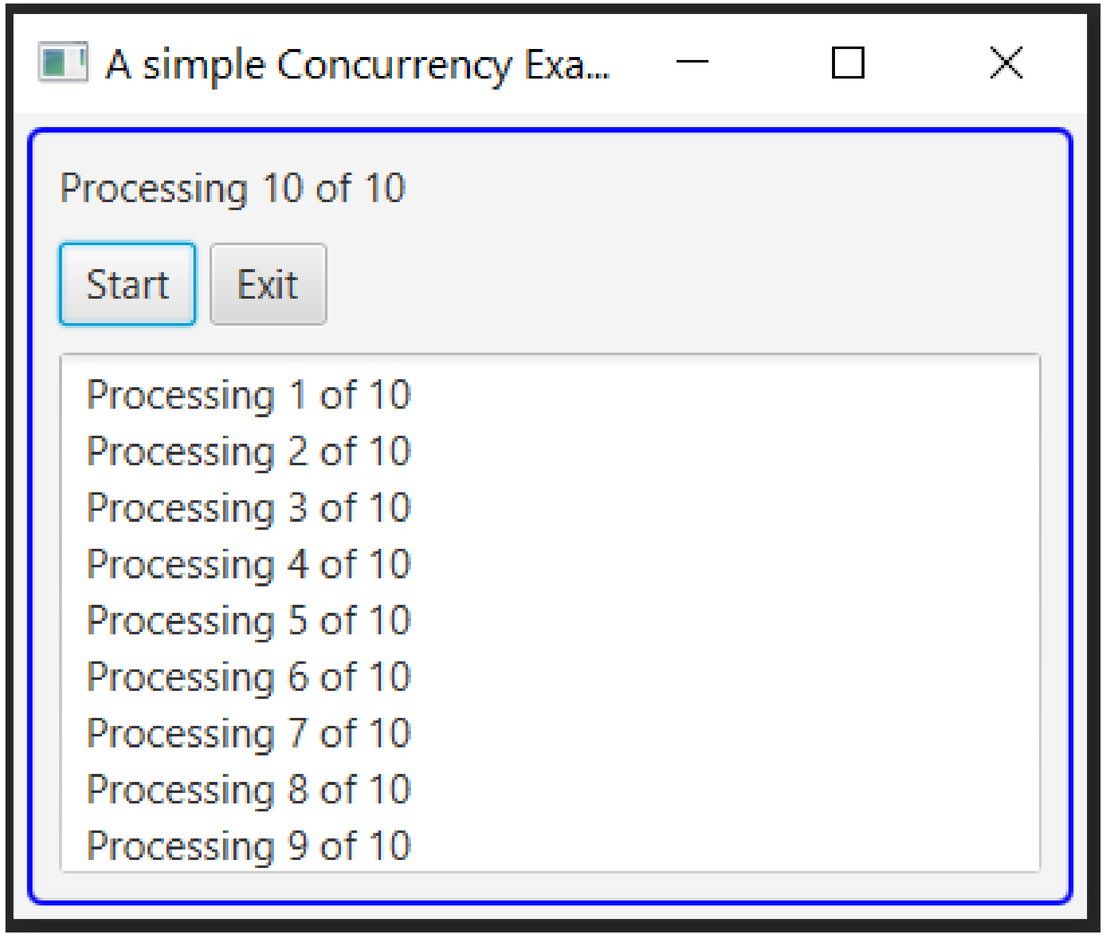
What happened?

- However, the Scene Graph was not refreshed to show the latest text for the Label
  - The JavaFX Application Thread was busy running the task and it did not run the pulse event handlers.
- Both problems arise because there is only one thread to process all UI event handlers and you ran a long-running task in the ActionEvent handler for the Start button.
- What is the solution?
  - You cannot change the single-threaded model for handling the UI events.
  - You must not run long-running tasks in the event handlers.
  - Run the long-running tasks in one or more background threads, instead of in the JavaFX Application Thread.

The GUI after clicking the Start Button



The GUI after execution



```
Processing 10 of 10
```

```
Processing 1 of 10
Processing 2 of 10
Processing 3 of 10
Processing 4 of 10
Processing 5 of 10
Processing 6 of 10
Processing 7 of 10
Processing 8 of 10
Processing 9 of 10
```

A first Solution Approach

```
    @Override
    public void start(final Stage stage)
    {
        // Create the Event-Handlers for the Buttons
        startButton.setOnAction(new EventHandler <ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                startTask();
            }
        });

        exitButton.setOnAction(new EventHandler <ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                stage.close();
            }
        });

        // Create the ButtonBox
        HBox buttonBox = new HBox(5, startButton, exitButton);
```

A first Solution Approach

```java
    public void startTask()
    {
        // Create a Runnable
        Runnable task = new Runnable()
        {
            public void run()
            {
                runTask();
            }
        };

        // Run the task in a background thread
        Thread backgroundThread = new Thread(task);
        // Terminate the running thread if the application exits
        backgroundThread.setDaemon(true);
        // Start the thread
        backgroundThread.start();
    }
```

A first Solution Approach

```java
public void runTask()
{
    for(int i = 1; i <= 10; i++)
    {
        try
        {
            String status = "Processing " + i + " of " + 10;
            statusLabel.setText(status);
            textArea.appendText(status+"\n");
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

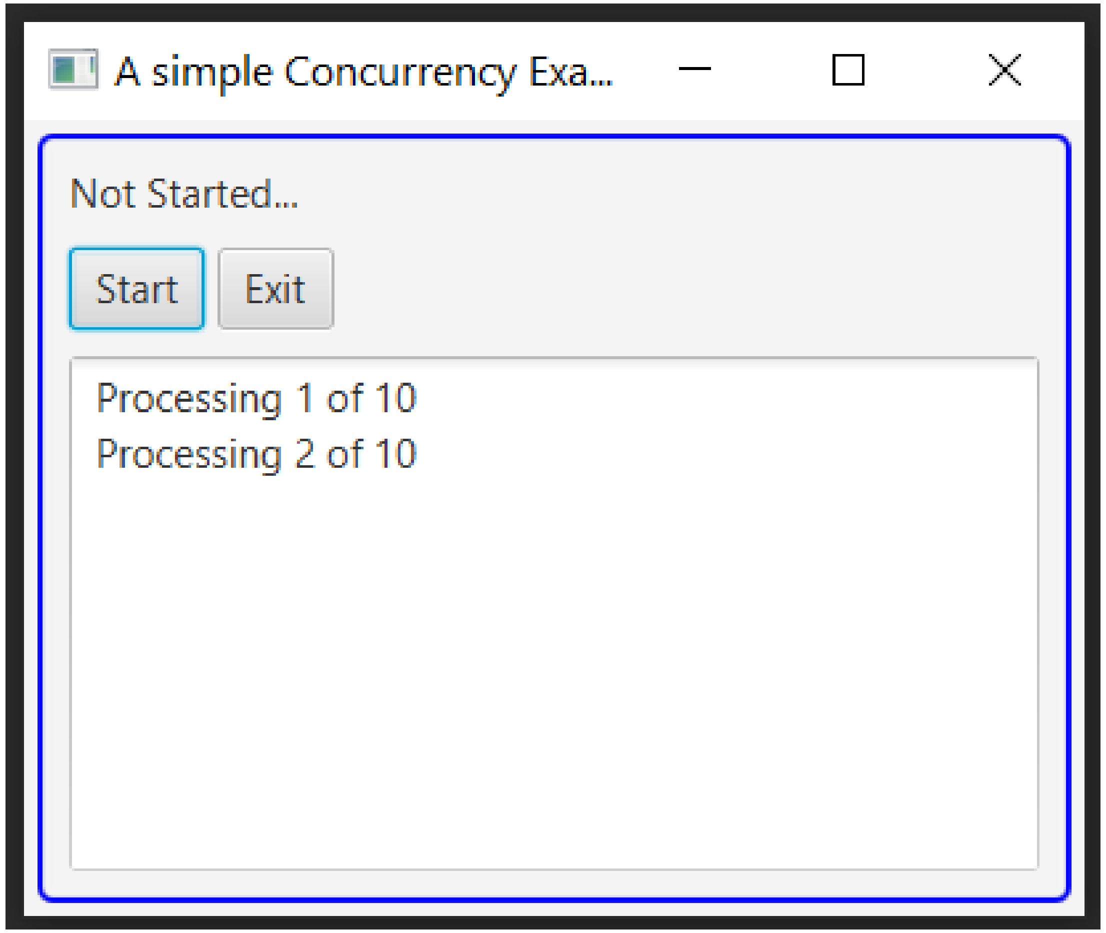
A first Solution Approach

- The above program is an incorrect attempt to provide a solution.
  - The ActionEvent handler for the Start Button calls the startTask() method, which creates a new thread and runs the runTask() method in the new thread.
- Run the program and click the Start Button.
  - A runtime exception is thrown as follows:

```
Exception in thread "Thread-4" java.lang.IllegalStateException: Not on FX application thread; currentThread = Thread-4
    at com.sun.javafx.tk.Toolkit.checkFxUserThread(Toolkit.java:236)
    at com.sun.javafx.tk.quantum.QuantumToolkit.checkFxUserThread(QuantumToolkit.java:423)
    at javafx.scene.Parent$2.onProposedChange(Parent.java:367)
    at com.sun.javafx.collections.VetoableListDecorator.setAll(VetoableListDecorator.java:113)
    at com.sun.javafx.collections.VetoableListDecorator.setAll(VetoableListDecorator.java:108)
    at com.sun.javafx.scene.control.skin.LabeledSkinBase.updateChildren(LabeledSkinBase.java:575)
    …
```

- The following statement in runTask() method generated the exception:
  - statusLabel.setText(status);

The GUI after execution



Processing 1 of 10
Processing 2 of 10

A first Solution Approach

- The JavaFX runtime checks that a live scene must be accessed from the JavaFX Application Thread.
- The runTask() method is run on a new thread, named Thread-4 as shown in the stack trace, which is not the JavaFX Application Thread.
- How do you access a live Scene Graph from a thread other than the JavaFX Application Thread?
    - The simple answer is that you cannot.
    - The complex answer is that when a thread wants to access a live Scene Graph, it needs to run the part of the code that accesses the Scene Graph in the JavaFX Application Thread.

The Solution

- The Platform class in the javafx.application package provides two static methods to work with the JavaFX Application Thread.
  - public static boolean isFxApplicationThread()
  - public static void runLater(Runnable runnable)

- The isFxApplicationThread() method returns true if the thread calling this method is the JavaFX Application Thread. Otherwise, it returns false.

- The runLater() method schedules the specified Runnable to be run on the JavaFX Application Thread at some unspecified time in future.

The Solution

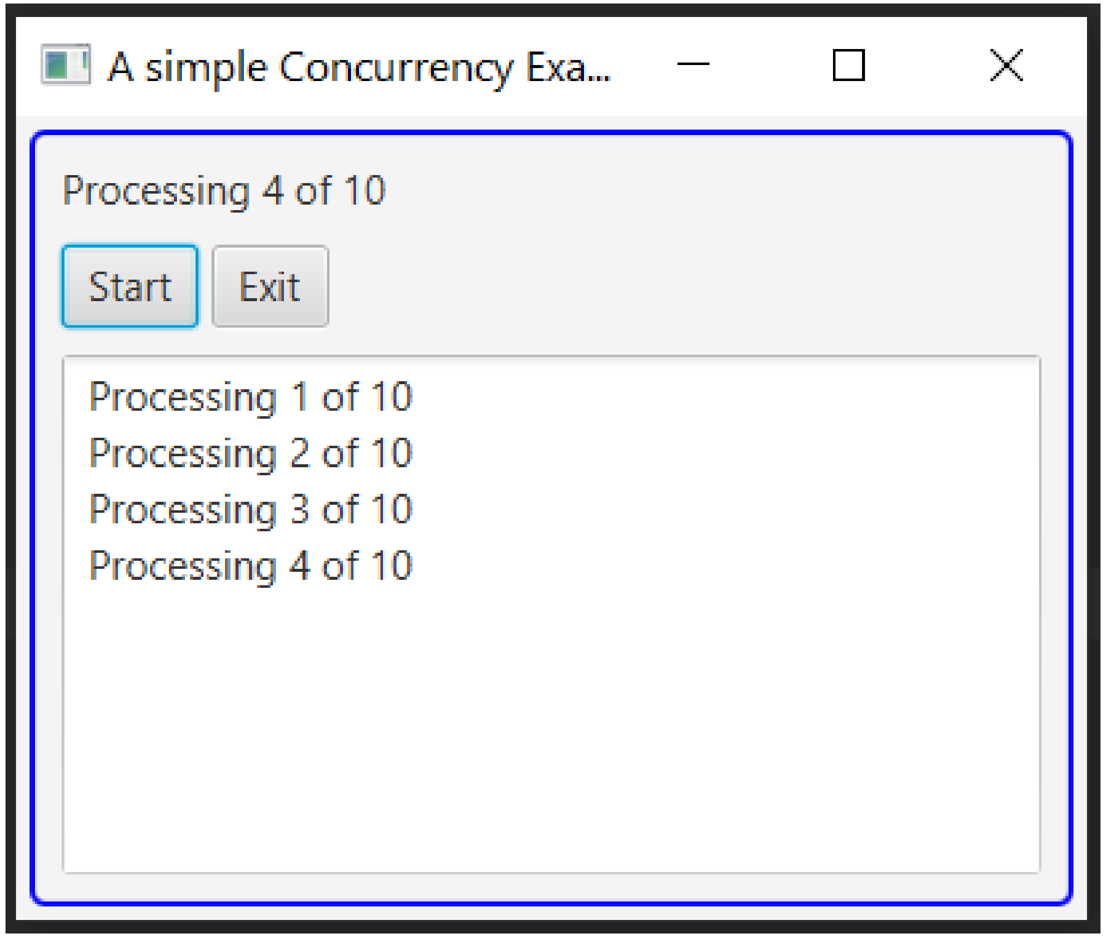
```java
    public void runTask()
    {
        for(int i = 1; i <= 10; i++)
        {
            try
            {
                // Get the Status
                final String status = "Processing " + i + " of " + 10;

                // Update the Label on the JavaFx Application Thread
                Platform.runLater(new Runnable()
                {
                    @Override
                    public void run()
                    {
                        statusLabel.setText(status);
                    }
                });

                textArea.appendText(status+"\n");

                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
```

The GUI after execution



```
Processing 4 of 10
```

```
Processing 1 of 10
Processing 2 of 10
Processing 3 of 10
Processing 4 of 10
```

The Summary

- Did you overcome the restrictions imposed by the event-dispatching threading model of the JavaFX?
- The answer is yes and no.
  - We used a trivial example to demonstrate the problem.
  - We have solved the trivial problem.
  - However, in a real world, performing a long-running task in a GUI application is not so trivial.
- The task did not return a result, nor did it have a reliable mechanism to handle errors that may occur.

The Summary

- The task in the example cannot be reliably cancelled, restarted, or scheduled to be run at a future time.
- The **JavaFX concurrency framework** has answers to all these questions.
  - The framework provides a reliable way of running a task in one or multiple background threads and publishing the status and the result of the task in a GUI application.
- More information:
  - https://examples.javacodegeeks.com/desktop-java/javafx/javafx-concurrent-framework/