Advanced Programming Methods

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2024

Overview

Java 8 Streams

- Creation
- Processing order
- Stream operations

Graphical user interfaces

Stream I

- java.util.Stream - a sequence of elements that supports different kinds of operations that can be performed on those elements.
- Stream operations can be:
  - *intermediate* - these return a stream, so multiple intermediate operations can be chained.
  - *terminal* - can be either void or can return a non-stream result.
- A chain of stream operations can also be referred to as an *operation pipeline*.
- All streams are created on a source, a java.util.Collection (this can be a List or a Set, but not a Map).

Stream II

```
List<String> names =
                Arrays.asList("Barbara", "James", "Brooke",
                                "Emilia", "Boris");

    names.stream()
        .filter(s -> s.startsWith("B"))
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);

// Result:
// BARBARA
// BORIS
// BROOKE
```

Stream III

- Usually a lambda parameter is passed to stream operations, a functional interface specifying the exact behaviour of the operation.
- An applied operation should be:
  - *non-interfering* - it does not modify the data source of the stream (underlying list or set).
  - *stateless* - its execution is deterministic (will always produce the same output), it does not depend on any mutable variables or states from the outer scope which might change during execution.
- Streams cannot be reused. As soon as any terminal operation is called, the stream is closed.

Creation

- Streams can be created:
  - with the stream() method, starting from Lists or Sets.

```
List<String> names =
        Arrays.asList("Barbara", "James");
names.stream()
            . // rest of operations
```

  - with the Stream.of(), starting from some object references.

```
Stream.of("Barbara", "James")
            . // rest of operations
```

Processing order

- Intermediate operations will only be executed when a terminal operation is present (lazy evaluation).
- Each intermediate operation creates a new stream, stores the provided operation/function and returns the new stream.
- The pipeline accumulates these newly created streams.
- When a terminal operation is called, traversal of streams begins and the associated functions are performed one by one.
- Elements move along the chain vertically: each element passes all operations in order and only after that the next element is processed.

Map

- map - takes a lambda expression as its only argument and changes every element according to this operation.
- Returns a new stream consisting of the results of applying the given function to the elements of this stream.
- It is an intermediate operation.

Filter

- filter - takes a lambda expression which must return a boolean value.
- According to the given lambda it is determined whether the processed element will or will not belong to the resulting Stream object.
- It is an intermediate operation.

Sorted

- sorted - returns a Stream consisting of the elements sorted according to the natural order or according to the provided *Comparator*.
- It is an intermediate operation.

Reduce I

- A reduction operation allows computing a result using all the elements present in a stream.
- reduce - aggregates the stream into a result (of a certain type).
- As parameters we can have:
    - a *BinaryOperator* - as an accumulator. For a numeric BinaryOperator the start value for the accumulation will be 0. For a string BinaryOperator the start value will be the empty string.
    - An *identity* and an *accumulator*. The identity represents the initial value of the reduction and the default result if there are no elements in the stream. The accumulator is a *BinaryOperator*.

Reduce II

- The *reduce()* operation with one parameter returns an Optional.
- *Optional* is a class used to represent if a value is absent or present (may or may not contain a non-null value).
- If a value is present, the function isPresent() will return *true* and the function get() will return the value.
- *reduce()* is a terminal operation.
- Predefined reduction operations: average(), sum(), min(), max(), count().

Match

- This is a terminal operation.
- It returns true or false, according to whether the objects in the stream match the specific criteria.
- allMatch(), anyMatch(), noneMatch()

Collect and toList (toList - from Java 16)

- Both are terminal operations.
- collect - receives elements from a steam and stores them in a collection.
- toList - receives elements from a steam and stores them to an immutable list.

JavaFX

- JavaFX is a GUI toolkit for Java.
- It integrates 2D and 3D graphics, charts, audio, video, and embedded web components (Javascript scripts, HTML5 code).
- It contains graphical user interface components for the creation of GUIs and allows managing their aspect via CSS files.
- Portability: desktop, browser, mobile devices, TV, game console.
- It ensures Swing interoperability. Swing is a part of Java Foundation Classes and it can be used to create window-based applications.

Installation and configuration with IntelliJ IDEA

- To create a new JavaFX project in IntelliJ IDEA do the following:
  1. Make sure the JavaFX plugin is enabled: see here.
  2. File → New → Project → JavaFX.
  3. Select the libraries that you want to use and click "Create".
  4. In the window on the bottom-right click "Load Maven Project". Alternatively, right click on the "pom.xml" file and choose "Add as Maven Project".
- When you run your project you should see a window with a button.
- Alternatively, if you would like to start from an existing project, follow the steps described in the document "JavaFX Tutorial.pdf". You can download JavaFX from any of the two links provided here: link.

Apache Maven

- Maven is a build automation tool used primarily for Java projects.
- It shields developers from many details and makes the build process easy.
- The Project Object Model (POM), which is an XML file that has all the information regarding project and configuration details.
- It can automatically download dependencies (libraries or JAR files).

JavaFX application I

- A JavaFX application contains one or more Stage objects, which correspond to windows.
- A JavaFX application has a primary *Stage* object which is created by the JavaFX runtime.
- Each *Stage* has a Scene attached to it.
- The Scene is needed to display things on the *Stage*.
- A *Stage* can only display one *Scene* at a time, but scenes can be changed at runtime.
- Each *Scene* can have a SceneGraph - an object graph of controls, layouts, etc.

JavaFX application II

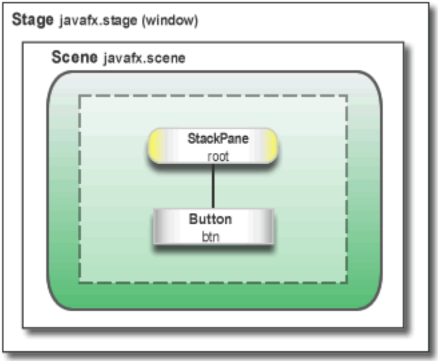

Figure: Figure source: Example of Scene Graph

JavaFX application III

- All components attached to the scene graph are called nodes. These are subclasses of the JavaFX class Node.
- There are two types of nodes:
  - Branch nodes (parent nodes) - can contain other child nodes.
  - Leaf nodes.
- *Layouts* - components which contain other components. They manages the layout of the components nested inside them.
- *Controls* - components which provide some kind of control functionality: Button, CheckBox, Label, Spinner, TableView, TextFields and many others.

JavaFX application IV

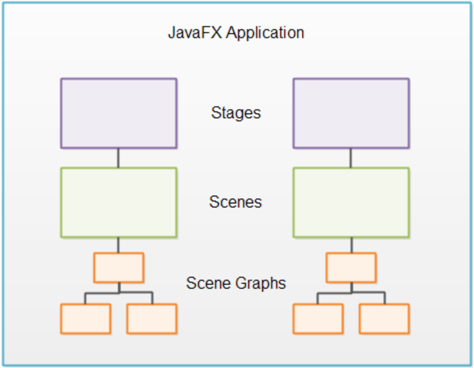

Figure: Figure source: The general structure of a JavaFX application.

JavaFX application V

- *Charts* - built-in ready-to-use chart components: BarChart, PieChart, ScatterChart and others.
- *2D and 3D Graphics* - to draw 2D and 3D graphics on the screen.
- *Audio and video* - features allowing to play audio and video in JavaFX applications. These are useful for games, streaming or educational applications.
- *WebView* - a component capable of showing web pages. It allows mixing a desktop application with a web application.

The JavaFX Application class I

- The primary launch class in our application must extend the javafx.application.Application.
- All subclasses on *Application* must implement the start(Stage stage) method.
- The *Stage* object is created by the JavaFX runtime.
- The method show() must be called on the *Stage* object to make it visible.
- A JavaFX application can be run without a main() method, but this is usually added in case any command line parameters are needed or just for clarity.
- The static method launch() in the Application class launches the application.

The JavaFX Application class II

```
public class Main extends Application {

    @Override
    public void start(Stage stage) throws Exception{
        stage.setTitle("Hello World");
        stage.show();
    }

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

Stage

- To display elements inside a Stage a new Scene must be created and set to the stage.
- The *Stage* title can be set with the setTitle method and the *Stage* position with the methods setX, setY.
- The *Stage modality* determines whether the current window will block other windows in the same application (method initModality.
- A *Stage* can be owned by another *Stage*: method initOwner. A Stage can be decorated with different styles: method initStyle.

Scene and scene graph I

- The *Scene* object is the root of the *scene graph*: it contains all the visual components.
- The *scene graph* includes all nodes which are attached to the scene.
- A scene graph can have only one *root node*.
- All other nodes will be attached to the root node in a tree-like data structure.

Scene and scene graph II

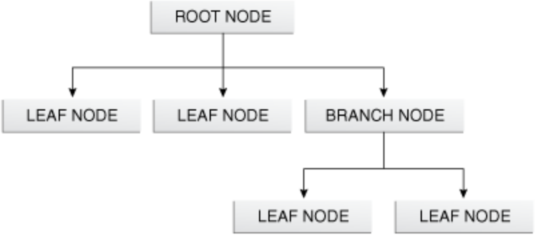

Figure: Figure source: General inheritance diagram of root, branch, and leaf nodes.

Scene and scene graph III

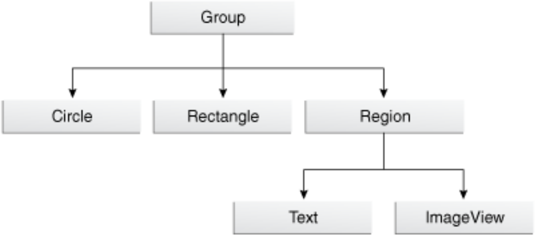

Figure: Figure source: Specific Root, Branch, and Leaf Classes.

Scene and scene graph IV

```
public void start(Stage stage) throws Exception{
        stage.setTitle("JavaFX application :)");
        stage.setX(400);
        stage.setY(400);

        Group root = new Group();
        Rectangle r = new Rectangle(25,25,100,100);
        r.setFill(Color.BLUE);
        root.getChildren().add(r);

        Circle c = new Circle(200, 200, 40);
        c.setFill(Color.RED);
        root.getChildren().add(c);

        Label label = new Label("Hello World Label!");
```

Scene and scene graph V

```
    root.getChildren().add(label);

    Scene scene = new Scene(root, 400, 300);
    stage.setScene(scene);
    stage.show();
}
```

Layouts I

- The layout container classes are called *layout panes*.
- The BorderPane provides 5 regions for node placement:

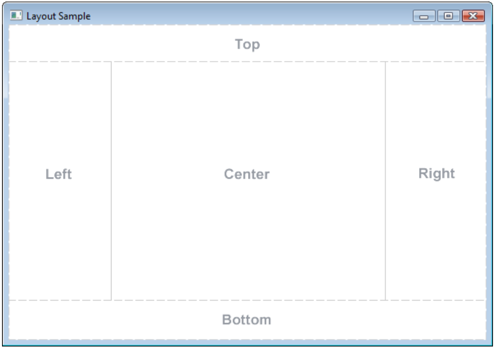

Figure: Figure source: Border Pane.

Layouts II

- HBox - nodes are arranged in a single row.
- VBox - nodes are arranged in a single column.
- StackPane - nodes are stacked, one on top of the other.

Figure: Figure source: Stack Pane.

- GridPane - allows creating grids of rows and columns in which nodes can be placed.
- FlowPane - nodes are laid out consecutively and wrapped.
- TilePane - similar to FlowPane, but each cell (or tile) has the same size.

JavaFX UI Controls

- These represent the basic elements of a GUI application.
- Each UI control is a node in the scene graph.
- UI controls can be manipulated by the user.
- The documentation for all controls can be found here.

Summary

- Streams are useful for operation pipelines.
- Chains of operations can be applied on streams, allowing us to perform many types of processing.
- Stream operations accept lambda expressions.
- JavaFX is a GUI toolkit for Java, allowing us to design applications with graphical user interfaces.
- JavaFX has been removed from JDK 11, so it must be installed and configured.
- *Next week:*
    - Graphical user interfaces - handling events.
    - FXML.