GIUSEPPE TURINI

CS-102 COMPUTING AND ALGORITHMS 2
LESSON 07
GRAPHICAL USER INTERFACES WITH JAVAFX

Kettering UNIVERSITY

JAVA ILLUMINATED (5TH EDITION), **CHAP 4, 165-194**, AND **CHAP 12, 781-952**.

HIGHLIGHTS

The Model-View-Controller Software Architectural Pattern
The Event-Driven Programming Paradigm
The Scene Graph Data Structure

JavaFX Overview

- JavaFX Overview and Key Features
- JavaFX Architecture
- JavaFX Application Examples: Hello World, and Login Form

JavaFX User Interface Components and Application Logic

- Working with the JavaFX Scene Graph: Overview and API
- JavaFX Event Handling: Events, Event Filters and Event Handlers
- Java Lambda Expressions and GUI Event Handlers
- JavaFX UI Components: Labels, Buttons, TableViews, Menus, and FileChoosers
- JavaFX Collections: Observable Lists and Maps, and Listeners

STUDY GUIDE

Study Material

- This slides.
- "Java Illuminated (5th Ed.)", chap. 4, pp. 165-194.
- "Java Illuminated (5th Ed.)", chap. 12, pp. 781-951.

Selected Exercises

- None.

Additional Resources

- Wikipedia JavaFX: en.wikipedia.org/wiki/javafx
- Oracle JavaFX Overview (Release 8): docs.oracle.com/javase/8/javafx
- Oracle JavaFX 8 API: docs.oracle.com/javase/8/javafx/api

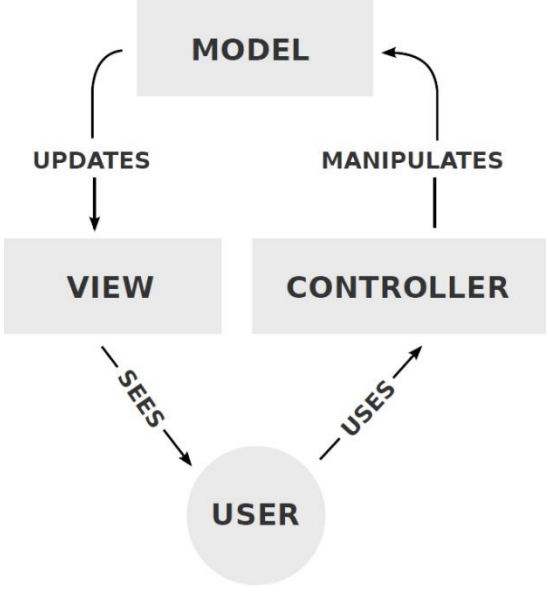
MODEL-VIEW-CONTROLLER 1

Model–View–Controller (MVC): a SW architectural pattern for implementing **user interfaces (UIs)**.

The MVC divides a software application into 3 interconnected parts:

- the **model**,
- the **view**, and
- the **controller**,

to separate internal representations of information from the ways that information is presented to or accepted from the user.



See: en.wikipedia.org/wiki/model-view-controller

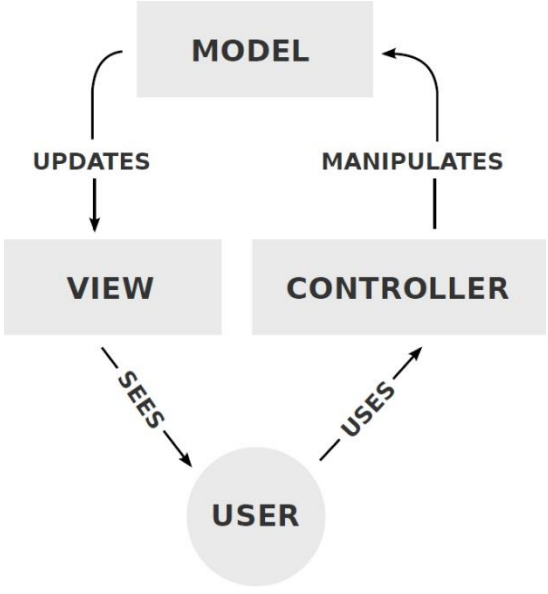
MODEL-VIEW-CONTROLLER 2

Model: it is the central component of the MVC, it captures the behavior of the application in terms of its problem domain, independent of the UI.
The model directly manages the data, logic, and rules of the application.

View: a view can be any output representation of data (charts, diagrams etc.), and multiple views of the same data are possible.

Controller: accepts input and converts it to commands for the model or view.

See: en.wikipedia.org/wiki/model-view-controller



MODEL-VIEW-CONTROLLER 3

The MVC design also defines the **interactions between the 3 MVC components:**

- **controller sends commands to model** to update the model;
- **controller sends commands to view** to change the model presentation;
- **model stores data retrieved by controller and displayed in the view;**
- **whenever there is a change to the data it is updated by the controller;**
- **view requests information from the model** to show model data to the user.

See: en.wikipedia.org/wiki/model-view-controller

EVENT-DRIVEN PROGRAMMING 1

Event-Driven Programming: is a programming paradigm in which the flow of the program is driven by events (user actions, sensor outputs) to have an interactive app.

Event-driven programming is the main paradigm for GUIs and interactive apps.

In an event-driven app:

- **A main loop listens for events, and triggers a callback function when an event is detected.**
- **When an event occurs, the main loop collects data about the event and fires it, dispatching the event to the event handler** (SW that will deal with it).
- **A program can ignore events**, and there may be libraries to dispatch events to multiple handlers that may be programmed to listen for a particular event.

See: en.wikipedia.org/wiki/event-driven_programming

EVENT-DRIVEN PROGRAMMING 2

Event: an action detected by the program that may be handled by the program.

Typical **sources of events** include:

- the user (**keystrokes**, **mouse clicks**),
- hardware devices (**timers**) etc.

Any program can trigger its own custom set of events as well (e.g. to communicate the completion of a task).

Events are typically used in UIs (mouse clicks, window-resizing, keyboard presses, messages from other programs, etc.), and programs written for many windowing environments consist predominantly of event handlers.

See: en.wikipedia.org/wiki/event_(computing)

EVENT-DRIVEN PROGRAMMING 3

Event Handler (aka Listener): is a callback function handling events in a program.

Event Dispatcher: It processes events as soon as they are created. It typically manages the **associations between events and event handlers**, and may queue event handlers or events for later processing.

Callback: a function passed as an argument to other code, which is expected to call back (i.e. execute) the function at some time. The function call may be immediate (**synchronous callback**), or delayed (**asynchronous callback**). In all cases, the intention is to specify a function as an entity that is similar to a variable.

See: en.wikipedia.org/wiki/event_(computing)
See: en.wikipedia.org/wiki/callback_(computer_programming)

EVENT-DRIVEN PROGRAMMING 4

THE DELEGATE EVENT MODEL

A common MVC variant is the delegate event model, which is based on 3 entities:

- **a control** (the event source);
- **consumers** (aka listeners) that receive the events from the control (source);
- **interfaces** describe the protocol by which every event is to be communicated.

Furthermore, the model requires that:

- every listener must implement the interface for the event it wants to listen to;
- every listener must register with the source to listen to some particular event;
- when a source generates an event, it communicates it to registered listeners.

See: en.wikipedia.org/wiki/event_(computing)

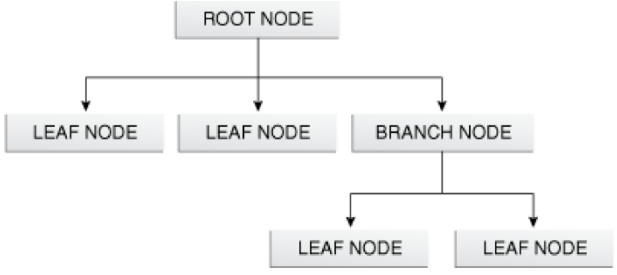
THE SCENE GRAPH

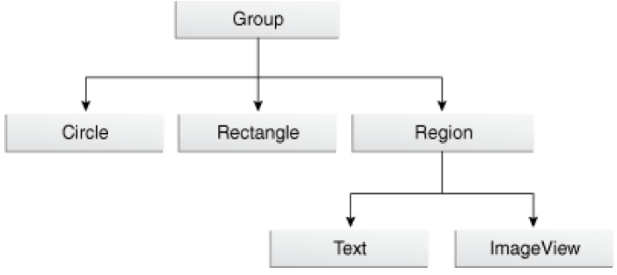
Scene Graph: a general data structure used by computer graphics applications and video games, arranging logical and spatial representations of a 2D/3D scene.

A scene graph is a collection of nodes in a graph or tree structure.

A node may have many children but usually only a single parent. An operation performed on a parent, auto-propagates to all of its descendants (e.g. rotation).

See: en.wikipedia.org/wiki/scene_graph





JAVAFX OVERVIEW 1

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy cross-platform rich client applications. It is available in both Java SE Runtime Environment (JRE) and Java Development Kit (JDK).

JAVAFX APPLICATIONS

- JavaFX is written as a Java API, so **JavaFX code can reference any Java APIs**.
- The graphical style of JavaFX applications can be customized:
  - **Cascading Style Sheets (CSS)** separate look/style from implementation;
  - or, you can develop the UI in an **FXML** script;
  - or, you can design the UI visually using the **JavaFX Scene Builder**;
  - or, you can implement the entire UI in Java.

JAVAFX OVERVIEW 2

JAVAFX KEY FEATURES A

Java APIs: JavaFX is a Java library that consists of classes and interfaces that are written in Java code. The APIs are designed to be a friendly alternative to Java Virtual Machine (Java VM) languages, such as JRuby and Scala.

FXML and Scene Builder: FXML is an XML-based declarative markup language for constructing a JavaFX application user interface. A designer can code in FXML or use JavaFX Scene Builder to interactively design the GUI.

Printing API: The **javafx.print** package has been added in Java SE 8 release and provides the public classes for the JavaFX Printing API.

Hi-DPI Support: JavaFX 8 now supports Hi-DPI displays.

JAVAFX OVERVIEW 3

JAVAFX KEY FEATURES B

WebView: A web component that uses WebKitHTML technology to make it possible to embed web pages within a JavaFX application. JavaScript running in WebView can call Java APIs, and Java APIs can call JavaScript running in WebView. Support for additional HTML5 features, including Web Sockets, Web Workers, and Web Fonts, and printing capabilities have been added in JavaFX 8.

Swing Interoperability: Existing Swing applications can be updated with JavaFX features, such as rich graphics media playback and embedded Web content. The **SwingNode** class, wich enables you to embed Swing content into JavaFX applications, has been added in JavaFX 8.

Rich Text Support: JavaFX 8 brings enhanced text support to JavaFX.

JAVAFX OVERVIEW 4

JAVAFX KEY FEATURES C

Built-in UI Controls and CSS: JavaFX provides all the major UI controls that are required to develop a full-feature application. Components can be skinned with standard Web technologies such as CSS.

3D Graphics Features: JavaFX 8 includes 3D graphics capabilities through new API classes such as **Shape3D**, **Material**, **Camera** etc.

Canvas API: The Canvas API enables drawing directly within an area of the JavaFX scene that consists of one graphical element (node).

Multitouch Support: JavaFX provides support for multitouch operations, based on the capabilities of the underlying platform.

JAVAFX OVERVIEW 5

JAVAFX KEY FEATURES D

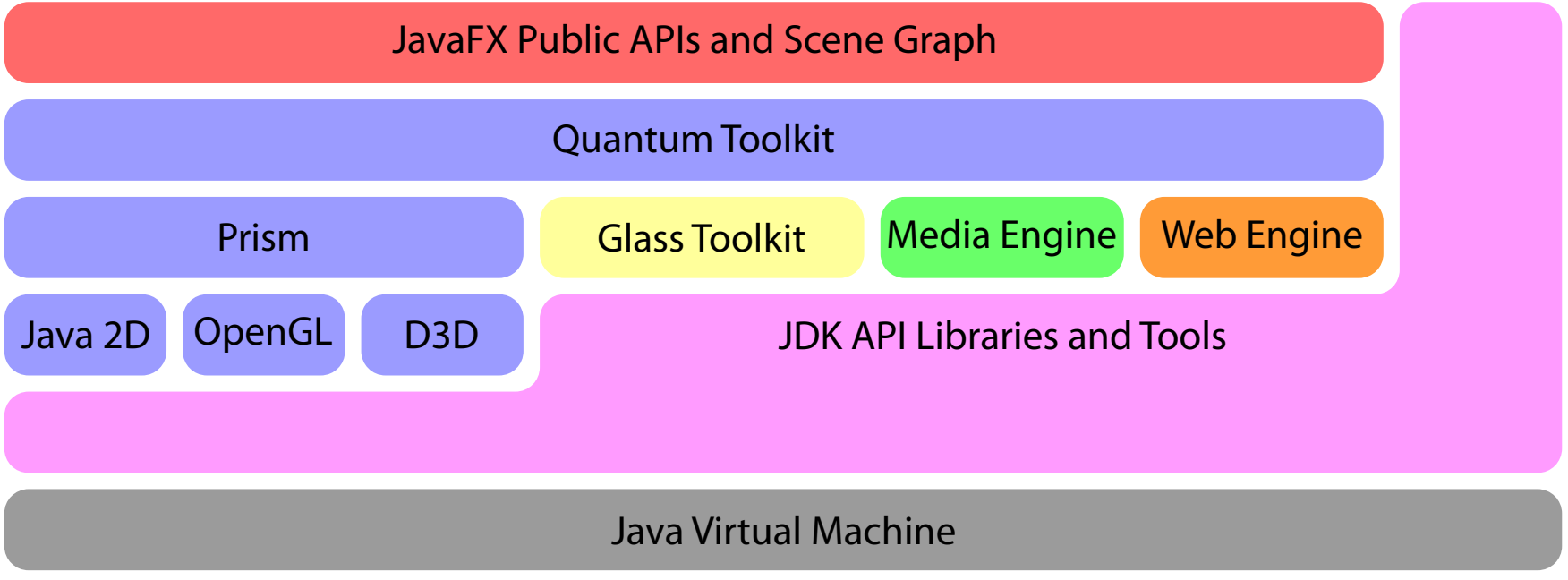
Hardware-accelerated Graphics Pipeline: JavaFX graphics are based on the graphics rendering pipeline (Prism). Rendering is performed through Prism when it is used with a supported Graphics Processing Unit (GPU). Otherwise, Prism defaults to the software rendering stack.

High-performance Media Engine: The media pipeline (based on GStreamer) supports the playback of web multimedia content.

Self-contained Application Deployment Model: Self-contained application packages have all of the application resources and a private copy of the Java and JavaFX runtimes. They are distributed as native installable packages and provide same installation and launch experience as native applications for that operating system.

JAVAFX ARCHITECTURE 1

The following is a high level description of the **JavaFX architecture** and ecosystem.



JAVAFX ARCHITECTURE 2

JAVAFX ARCHITECTURE: SCENE GRAPH

The JavaFX scene graph (shown in **red**) is the starting point for constructing a JavaFX application. It is a hierarchical tree of nodes that represents all of the visual elements of the application's user interface. It can handle input and can be rendered.
A single element in a scene graph is called a **node**. Each node has an ID, style class, and bounding volume. **With the exception of the root node of a scene graph, each node in a scene graph has a single parent and zero or more children.** It can also have: effects (blurs, shadows etc.), opacity, transforms, event handlers (mouse, key etc.), and an application-specific state.

The **javafx.scene** API allows the creation/specification of several types of content:

- **Nodes:** shapes, images, media, text, UI controls, etc.
- **State:** transforms (positioning and orientation nodes), visual effects etc.
- **Effects:** blurs, shadows, color adjustments etc.

JAVAFX ARCHITECTURE 3

JAVAFX ARCHITECTURE: GRAPHICS SYSTEM

The JavaFX graphics system (shown in **blue**) is an implementation detail beneath the JavaFX scene graph layer. It supports both 2D and 3D scene graphs, and it provides software rendering when the graphics hardware on a system is insufficient to support hardware accelerated rendering.

Two graphics accelerated pipelines are implemented on the JavaFX platform:

- **Prism:** processes render jobs and it can run on both hardware and software renderers. It includes 3D graphics via DirectX or OpenGL depending on the OS.

- **Quantum Toolkit:** ties Prism and Glass together and makes them available to the JavaFX layer above them in the stack. It also manages the threading rules related to rendering versus events handling.

JAVAFX ARCHITECTURE 4

JAVAFX ARCHITECTURE: GLASS WINDOWING TOOLKIT

The Glass Windowing Toolkit (shown in **yellow**) is the lowest level in the JavaFX graphics stack. Its main responsibility is to provide native operating services, such as managing the windows, timers, and surfaces. It serves as the platform-dependent layer that connects the JavaFX platform to the native operating system.

Glass is also responsible for managing the event queue. Unlike the Abstract Window Toolkit (AWT), which manages its own event queue, the Glass toolkit uses the native operating system's event queue functionality to schedule thread usage.

JAVAFX ARCHITECTURE 5

JAVAFX ARCHITECTURE: THREADS

The JavaFX system runs 2 or more of the following threads at any given time:

- **JavaFX Application Thread:** this is the primary thread used by JavaFX application developers. Any "live" scene, which is a scene that is part of a window, must be accessed from this thread.

- **Prism Render Thread:** this thread handles the rendering separately from the event dispatcher. It allows frame N to be rendered while frame N+1 is being processed.

- **Media Thread:** this thread runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.

JAVAFX ARCHITECTURE 6

JAVAFX ARCHITECTURE: PULSE

A pulse is an event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements on the scene graph with Prism. A pulse is throttled at 60 frames-per-second (fps) maximum and it is fired whenever animations are running on the scene graph. Even when an animation is not running, a pulse is scheduled when something in the scene graph is changed. For example, if a position of a button is changed, a pulse is scheduled.

The Glass Windowing Toolkit is responsible for executing the pulse events. It uses the high-resolution native timers to make the execution.

JAVAFX ARCHITECTURE: MEDIA AND IMAGES

JavaFX media functionality is available through the **javafx.scene.media** APIs. JavaFX supports both visual and audio media. Support is provided for MP3, AIFF, and WAV audio files and FLV video files.

JavaFX media functionality is provided as 3 separate components:

- the **Media** object represents a media file,
- the **MediaPlayer** plays a media file, and
- a **MediaView** is a node that displays the media.

JAVAFX ARCHITECTURE: WEB COMPONENTS

The Web component is a JavaFX UI control, based on WebKit, that provides a Web viewer and full browsing functionality through its API. This Web Engine component (shown in **orange**) is based on WebKit, which is an open-source web browser engine that supports HTML5, CSS, JavaScript, DOM, and SVG. It enables developers to implement the following features in their Java applications:

- render HTML content from local or remote URL,
- support history and “back” / “forward” / “reload” navigation commands,
- apply effects to the web content and edit the HTML content,
- execute JavaScript commands, and handle events.

This embedded browser component is composed of the following classes:

- **WebEngine** provides basic web page browsing capabilities,
- **WebView** (an extension of the Node class) encapsulates a WebEngine object, and incorporates HTML content into an application’s scene.

JAVAFX ARCHITECTURE: CSS

JavaFX Cascading Style Sheets (CSS) provides the ability to apply customized styling to the user interface of a JavaFX application without changing any of that application's source code. CSS can be applied to any node in the JavaFX scene graph and are applied to the nodes asynchronously. JavaFX CSS styles can also be easily assigned to the scene at runtime, allowing an application's appearance to dynamically change.

JavaFX CSS is based on the W3C CSS version 2.1 specifications, with some additions from current work on version 3.

JAVAFX ARCHITECTURE: UI CONTROLS

The JavaFX UI controls available through the JavaFX API are built by using nodes in the scene graph. They can take full advantage of the visually rich features of the JavaFX platform and are portable across different platforms. JavaFX CSS allows for theming and skinning of the UI controls. The controls reside in the **javafx.scene.control** package.

JAVAFX ARCHITECTURE 11

JAVAFX ARCHITECTURE: LAYOUTS

Layout containers or panes can be used to allow for flexible and dynamic arrangements of the UI controls within a scene graph of a JavaFX application.

The JavaFX Layout API has container classes to automate common layout modes:

- **BorderPane:** lays out content nodes in: top, bottom, right, left, or center area.
- **HBox:** arranges its content nodes horizontally in a single row.
- **VBox:** arranges its content nodes vertically in a single column.
- **StackPane:** places its content nodes in a back-to-front single stack.
- **GridPane:** uses a flexible grid of rows and columns to lay out content nodes.
- **FlowPane:** arranges its content nodes in either horizontal or vertical "flow".
- **TilePane:** places its content nodes in uniformly sized layout cells (or tiles).
- **AnchorPane:** creates anchor nodes to the layout top, bottom, left, or center.

JAVAFX ARCHITECTURE 12

JAVAFX ARCHITECTURE: 2D AND 3D TRANSFORMATIONS

Each node in the JavaFX scene graph can be transformed in the XY coordinate using the following **javafx.scene.transform** classes:

- **translate:** move a node from one place to another in XYZ space.
- **scale:** resize a node to appear larger/smaller depending on the scaling factor.
- **shear:** rotate one axis so the coordinates of a node are shifted.
- **rotate:** rotate a node about a specified pivot point of the scene.
- **affine:** perform a linear mapping from 2D/3D points to other 2D/3D points.

JAVAFX ARCHITECTURE: VISUAL EFFECTS

The JavaFX effects are primarily image pixel-based and, hence, they take the set of nodes that are in the scene graph, render it as an image, and apply the specified effects to it. See the **javafx.scene.effect** package.

JAVAFX HELLO WORLD 1

HELLO WORLD A

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

// Note: main class for a JavaFX app extends the javafx.application.Application class.
public class HelloWorld extends Application {
```

JAVAFX HELLO WORLD 2

HELLO WORLD B

```
// The start method is the main entry point for all JavaFX applications.
// Stage: The A JavaFX app defines the UI container by means of a stage and a scene.
//        The JavaFX Stage class is the top-level JavaFX container.
@Override
public void start( Stage primaryStage ) {
   Button btn = new Button(); // Button control node.
   btn.setText( "Say 'Hello World'" ); // Button setup: button text.

   // Event handler for button control node: print message when button is pressed.
   // setOnAction: Sets the value of the property onAction.
   // onAction: the button's action, which is invoked whenever the button is fired.
   btn.setOnAction( new EventHandler<ActionEvent>() {
      // handle: invoked when a specific event (see handler registration) happens.
      @Override
      public void handle( ActionEvent e ) { System.out.println("Hello World!"); } } );
```

JAVAFX HELLO WORLD 3

HELLO WORLD C

```
        StackPane root = new StackPane(); // Scene graph root node: resizeable layout node.
        root.getChildren().add( btn ); // Set button control node as child of root node.
        // Scene: A JavaFX app defines the UI container by means of a stage and a scene.
        //        The JavaFX Scene class is the container for all content (scene graph).
        Scene scene = new Scene( root, 500, 250 ); // Scene: root, window width and height.
        primaryStage.setTitle( "Hello World!" ); // Stage setup: window title.
        primaryStage.setScene(scene); // Stage setup: scene graph for the content.
        primaryStage.show(); // Show the window via the stage.
    }

    // The main method is not required for JavaFX applications.
    // However, it is useful if JavaFX tools are not fully integrated in your IDE.
    public static void main( String[] args ) {
        launch( args ); // Launch this standalone JavaFX application.
    }

}
```

JAVAFX LOGIN FORM 1

LOGIN FORM A

```
import javafx.application.Application;
import javafx.event.ActionEvent; import javafx.event.EventHandler;
import static javafx.geometry.HPos.RIGHT; // Static import allows unqualified access.
import javafx.geometry.Insets; import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;
```

JAVAFX LOGIN FORM 2

LOGIN FORM B

```
// Note: main class for a JavaFX app extends the javafx.application.Application class.
public class LoginForm extends Application {

    // The start method is the main entry point for all JavaFX applications.
    // Stage: The A JavaFX app defines the UI container by means of a stage and a scene.
    //        The JavaFX Stage class is the top-level JavaFX container.
    @Override
    public void start( Stage primaryStage ) {
        primaryStage.setTitle( "JavaFX Welcome" );
        GridPane grid = new GridPane(); // Lay out children nodes in a grid.
        grid.setAlignment( Pos.CENTER ); // Set grid alignment to centered.
        grid.setHgap( 10 ); grid.setVgap( 10 ); // Set H/V gaps between cols/rows.
        grid.setPadding( new Insets( 25, 25, 25, 25 ) ); // Padding around content area.
        // ...
        Text scenetitle = new Text( "Welcome" );
        scenetitle.setFont( Font.font( "Tahoma", FontWeight.NORMAL, 20 ) );
        grid.add( scenetitle, 0, 0, 2, 1 ); // Adds child to grid at pos and with span.
```

JAVAFX LOGIN FORM 3

LOGIN FORM C

```
// ...
Label userName = new Label( "User Name:" ); grid.add( userName, 0, 1 );
// ...
TextField userTextField = new TextField(); grid.add( userTextField, 1, 1 );
// ...
Label pw = new Label( "Password:" ); grid.add( pw, 0, 2 );
// ...
PasswordField pwBox = new PasswordField(); grid.add( pwBox, 1, 2 );
// ...
Button btn = new Button( "Sign in" );
HBox hbBtn = new HBox( 10 ); // Lay out children nodes in a single row.
hbBtn.setAlignment( Pos.BOTTOM_RIGHT );
hbBtn.getChildren().add( btn );
grid.add( hbBtn, 1, 4 );
```

JAVAFX LOGIN FORM 4

LOGIN FORM D

```
// ...
final Text actiontarget = new Text();
grid.add( actiontarget, 0, 6 );
grid.setColumnSpan( actiontarget, 2 );
grid.setHalignment( actiontarget, RIGHT );
actiontarget.setId( "actiontarget" ); // Set node ID for scene graph searches.

// ...
btn.setOnAction( new EventHandler<ActionEvent>() {
    // ...
    @Override
    public void handle( ActionEvent e ) {
        actiontarget.setFill( Color.FIREBRICK );
        actiontarget.setText( "Signed in: " +
                              userTextField.getText() + " " +
                              pwBox.getText() ); } } );
```

JAVAFX LOGIN FORM 5

LOGIN FORM E

```
        // ...
        Scene scene = new Scene( grid, 300, 275 );
        primaryStage.setScene( scene );
        primaryStage.show();
    }

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

}
```

JAVAFX SCENE GRAPH 1

JAVAFX SCENE GRAPH: OVERVIEW A

The JavaFX scene graph is a **retained mode API**, meaning that:

JavaFX scene graph maintains an internal model of all GUI objects in your app.

At any given time, it knows:

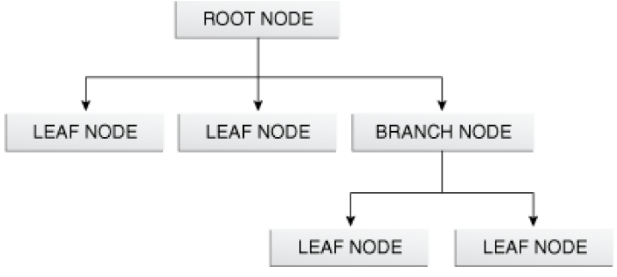
- what objects to display,
- what areas of the screen need repainting, and
- how to render it all in the most efficient manner.

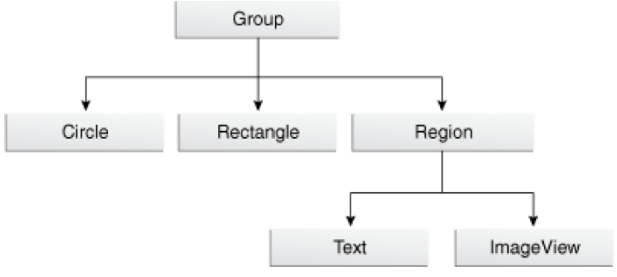
Instead of invoking primitive drawing methods directly, you instead use the scene graph API and let the system automatically handle the rendering details.
This approach significantly reduces the amount of code that is needed in your app.

JAVAFX SCENE GRAPH 2

JAVAFX SCENE GRAPH: OVERVIEW B

Individual elements in the JavaFX scene graph are known as nodes. Each node is classified as either a **branch node** (it can have children), or a **leaf node** (it cannot have children). The topmost node is called the **root node** (it has no parent).





Example: A **Group** object acts as the root node. The **Circle** and **Rectangle** objects are leaf nodes (they cannot have children). The **Region** object (which defines an area of the screen with children than can be styled using CSS) is a branch node that contains 2 more leaf nodes (**Text** and **ImageView**).

JAVAFX SCENE GRAPH 3

JAVAFX SCENE GRAPH: EXPLORING THE API

The **javafx.scene** package defines several classes, but 3 are most important:

- **Node:** the **abstract base class** for all scene graph nodes.
- **Parent:** the **abstract base class** for all branch nodes (directly extends **Node**).
- **Scene:** the **base container class** for all content in the scene graph.

These base classes define functionality that will be inherited by subclasses, including: paint order, visibility, composition of transformations, support for CSS styling, etc.

You will also find various **branch node classes** that inherit directly from the **Parent** class, such as: **Control**, **Group**, **Region**, and **WebView**.

The **leaf node classes** are defined throughout a number of additional packages, such as: **javafx.scene.shape** and **javafx.scene.text**.

JAVAFX EVENT HANDLING 1

JAVAFX EVENT HANDLING: EVENT A

A Java FX event is an instance (or a subclass) of the **javafx.event.Event** class. JavaFX provides several events, including: **DragEvent**, **KeyEvent**, **MouseEvent**, **ScrollEvent**, etc.; and you can define your own events by extending the **Event** class.
Every event includes (at least) the information described below:

- **Event Type:** type of event that occurred.
- **Source:** origin of the event, with respect to the location of the event in the event dispatch chain. Source changes as the event is passed along the chain.
- **Target:** node on which the action occurred and the end node in the event dispatch chain. The target does not change, but if an event filter consumes the event during the event capturing phase, the target will not receive the event.

JAVAFX EVENT HANDLING 2

JAVAFX EVENT HANDLING: EVENT B

Event subclasses provide additional information that is specific to the type of event.

Example: The **MouseEvent** class includes information such as:

- which button was pushed,
- the number of times the button was pushed, and
- the position of the mouse.

JAVAFX EVENT HANDLING 3

JAVAFX EVENT HANDLING: EVENT TYPES A

An event type is an instance of the **EventType** class. Event types further classify the events of a single event class.

Example: The **KeyEvent** class contains these event types: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**.

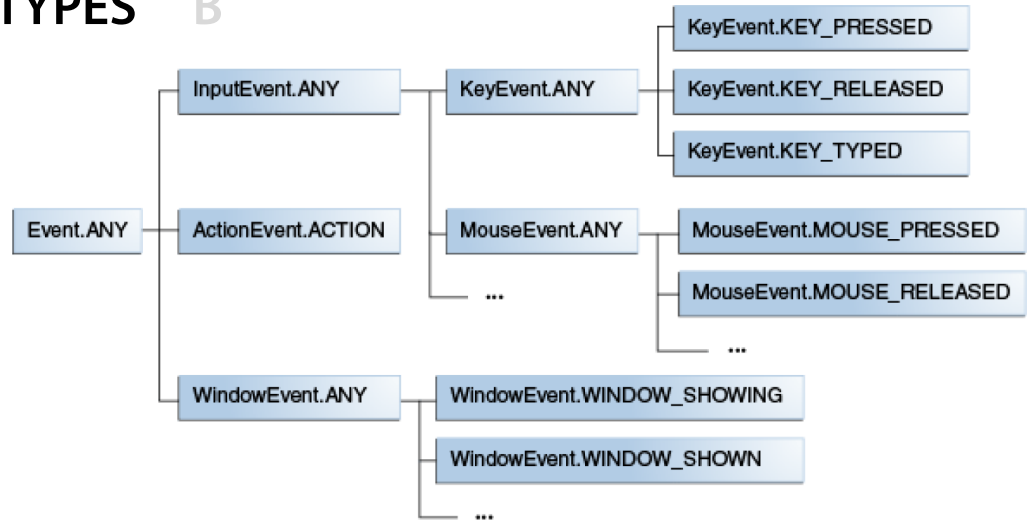
Event types are **hierarchical**. Every event type has a **name** and a **super type**.

Example: The name of the event for a key being pressed is **KEY_PRESSED**, and the super type is **KeyEvent.ANY**. The super type of the top-level event type is **null**.

JAVAFX EVENT HANDLING 4

JAVAFX EVENT HANDLING: EVENT TYPES B

Figure: A subset of the event type hierarchy:



The topmost event type in the hierarchy is **Event.ROOT** (same as **Event.ANY**). In the subtypes, the event type **ANY** is used to mean any event type in the **Event** class.

Example: To provide the same response to any type of key event, use **KeyEvent.ANY** as the event type for the event filter or event handler. To respond only when a key is released, use the **KeyEvent.KEY_RELEASED** event type etc.

JAVAFX EVENT HANDLING 5

JAVAFX EVENT HANDLING: EVENT TARGETS

The target of an event can be an instance of any class that implements the **EventTarget** interface.

The implementation of the **buildEventDispatchChain** creates the event dispatch chain that the event must travel to reach the target.

The **Window**, **Scene**, and **Node** classes implement the **EventTarget** interface and subclasses of those classes inherit the implementation.

So, most elements in your UI have their dispatch chain defined, so you can focus on responding to events and not be concerned with creating the event dispatch chain.

JAVAFX EVENT HANDLING 6

JAVAFX EVENT HANDLING: EVENT DELIVERY PROCESS

The event delivery process contains the following steps:

1. Target Selection,
2. Route Construction,
3. Event Capturing,
4. Event Bubbling.

JAVAFX EVENT HANDLING 7

JAVAFX EVENT HANDLING: 1 - TARGET SELECTION

When an action occurs, the system finds the target node based on internal rules:

1. for **key events**: the target is the node that has focus;
2. for **mouse events**: the target is the node at the location of the cursor (for synthesized mouse events, the touch point is considered the cursor location);
3. for **gesture events**, **swipe events**, and **touch events** (touch screen)…

If more than 1 node is located at the cursor, the target is the topmost node!

When a mouse button is pressed and the target is selected, all subsequent mouse events are delivered to the same target until the button is released.

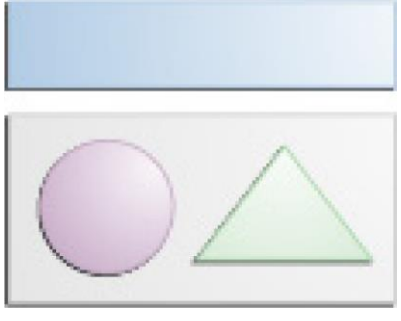
Similarly for gesture events, from the gesture start to the completion of the gesture, gesture events are delivered to the target identified at the beginning of the gesture.

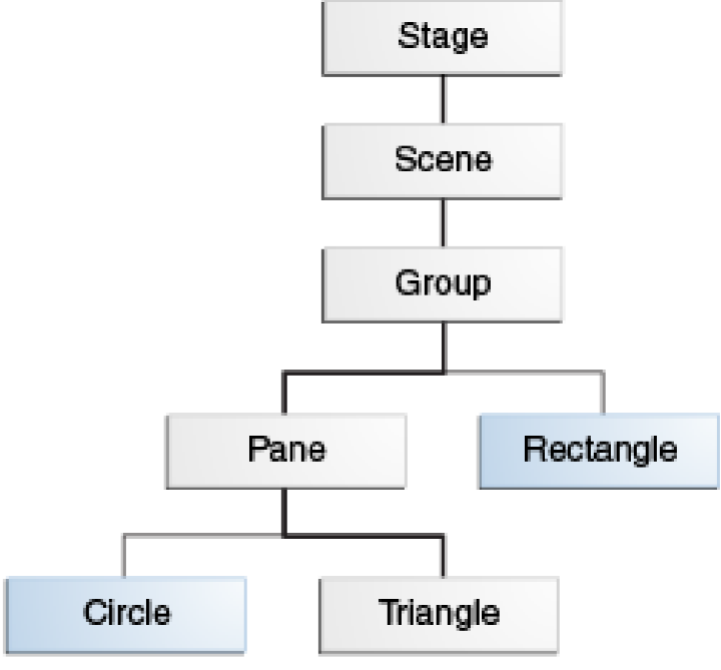
JAVAFX EVENT HANDLING 8

JAVAFX EVENT HANDLING: 2 - ROUTE CONSTRUCTION

The initial event route is determined by the event dispatch chain created by the **buildEventDispatchChain()** function of the selected event target.

Example: If a user clicks the triangle (left image), the initial route will be the path of gray nodes (right image).





JAVAFX EVENT HANDLING 9

JAVAFX EVENT HANDLING: 3 - EVENT CAPTURING

In the event capturing phase, the event is dispatched by the root node of your application and passed down the event dispatch chain to the target node.

If any node in the chain has
an event filter registered for the type of event that occurred,
that filter is called.
When the filter completes,
the event is passed to the next node down the chain.

Note: If no filter is registered for a node, event is passed to the node down the chain.
Note: If no filter consumes the event, the event target eventually receives the event.

JAVAFX EVENT HANDLING: 4 - EVENT BUBBLING

After the event target is reached and all registered filters have processed the event, the event returns along the dispatch chain from the target to the root node.

If any node in the chain has
a handler registered for the type of event encountered,
that handler is called.
When the handler completes,
the event is returned to the next node up the chain.

Note: If no handler is registered for a node, event is passed to the node up the chain.
Note: If no handler consumes the event, the root node eventually receives the event.

JAVAFX EVENT HANDLING 11

JAVAFX EVENT HANDLING: EVENT FILTERS VS EVENT HANDLERS

Event handling is provided by event filters and event handlers, which are implementations of the **EventHandler** interface.

If you want an application to be notified when an event occurs, register a filter or a handler for the event.

The primary difference between a filter and a handler is when each one is executed.

See: docs.oracle.com/javase/8/javafx/api/javafx/event/eventhandler

JAVAFX EVENT HANDLING: EVENT FILTERS

An **event filter** is executed during the **event capturing** phase. An event filter for a parent node can provide common event processing for multiple child nodes and if desired, consume the event to prevent the child node from receiving the event.

Filters that are registered for the event type that occurred are executed as the event passes through the node that registered the filter. A node can register multiple filters.

The order in which each filter is called is based on the hierarchy of event types. Filters for a specific event type are executed before filters for generic event types. The order in which two filters at the same level are executed is not specified.

JAVAFX EVENT HANDLING 13

JAVAFX EVENT HANDLING: EVENT HANDLERS

An **event handler** is executed during the **event bubbling**. If an handler for a child node does not consume the event, an handler for a parent node can process the event after the child node, providing common processing for multiple child nodes.

Handlers registered for the event type are executed as the event returns through the node that registered the handler. A node can register multiple handlers.

The order in which each handler is called is based on the hierarchy of event types. Handlers for a specific event type are executed before handlers for generic event types. The order in which two handlers at the same level are executed is not specified.

See: docs.oracle.com/javase/8/javafx/api/javafx/event/eventhandler

JAVAFX EVENT HANDLING: CONSUMING OF AN EVENT

Events can be consumed by an event filter or an event handler at any point in the event dispatch chain by calling the consume() method. This method signals that event processing is complete and traversal of the event dispatch chain ends.

Consuming the event in an event filter prevents any child node on the event dispatch chain from acting on the event.
Consuming the event in an event handler stops any further processing of the event by parent handlers on the event dispatch chain.

Note: However, if the node that consumes the event has more than one filter or handler registered for the event, the peer filters or handlers are still executed.
Note: Default handlers for JavaFX UI controls typically consume most of input events.

JAVA LAMBDA EXPRESSIONS 1

JAVA LAMBDA EXPRESSION

A **lambda expression** (aka anonymous function, or function literal) is a function definition not bound to an identifier.
These functions are often:

- passed as **arguments to higher-order functions**, or
- used to create the **result of a higher-order function** that returns a function.

If a function is used few times, a lambda expression may be **syntactically lighter**.

To process GUI events you typically create event handlers, which usually involves implementing a particular interface. **Often, event handler interfaces are functional interfaces with only 1 method, that can be specified using lambda expressions.**

See: docs.oracle.com/javase/tutorial/java/javaoo/lambdaexpressions

JAVA LAMBDA EXPRESSIONS 2

JAVA LAMBDA EXPRESSION: SYNTAX IN JAVA 8

A lambda expression consists of the following:

- A **comma-separated list of formal parameters** in parentheses **()**.
  Note: You can omit parameter types and parentheses if using only 1 parameter.
- The **arrow token**: **->**.
- A **body**: a single expression or a statement block.
  Note: If a single expression, its value is returned, otherwise use a **return**.

Note: Lambda expressions are like anonymous methods (methods without a name).

```
( MouseEvent e ) -> { label3.setScaleX( 1.5 ); label3.setScaleY( 1.5 ); } // Lambda exp.

new EventHandler<MouseEvent>() { // Standard event handler.
   @Override public void handle( MouseEvent e ) {
      label3.setScaleX( 1.5 ); label3.setScaleY( 1.5 ); } }
```

JAVAFX USER INTERFACE CONTROLS 1

JAVAFX UI CONTROLS: LIST OF BUILT-IN UI CONTROLS AVAILABLE

- **Label**.
- **Button**, Radio Button, and Toggle Button.
- Checkbox, Choice Box, and Combo Box.
- **Text Field**, **Password Field** (see Login Form example).
- Scroll Bar, Scroll Pane, Slider, Progress Bar, and Progress Indicator.
- List View, **Table View**, Tree View, and Tree Table View.
- Separator, Hyperlink, and Tooltip.
- HTML Editor, Titled Pane and Accordion, and Pagination Control.
- **Menu**.
- Color Picker, and Date Picker.
- **File Chooser**.

JAVAFX USER INTERFACE CONTROLS 2

JAVAFX UI CONTROLS: LABEL

The JavaFX **Label** class (**javafx.scene.control** package) is used to display text. Using a label you can: wrap text to fit the specific space, add an image, or apply visual effects.
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/label

LABEL SAMPLE A

```
import javafx.scene.Group; import javafx.application.Application;
import javafx.scene.Scene; import javafx.scene.control.Label;
import javafx.scene.image.Image; import javafx.scene.image.ImageView;
import javafx.scene.input.MouseEvent; import javafx.scene.layout.HBox;
import javafx.scene.paint.Color; import javafx.scene.text.Font;
import javafx.scene.text.TextAlignment; import javafx.stage.Stage;
import javafx.event.EventHandler;

// Main class for a JavaFX application needs to extend javafx.application.Application.
public class LabelSample extends Application {
```

JAVAFX USER INTERFACE CONTROLS 3

LABEL SAMPLE B

```
private final Label label3 = new Label( "A label that needs to be wrapped" );

// Main method to enable JavaFX support in all IDEs.
public static void main( String[] args ) { launch( args ); }

// Main entry point for JavaFX applications.
@Override
public void start( Stage stage ) {
   Scene scene = new Scene( new Group() ); // Scene created with a Group node as root.
   Image image = new Image( "labels.jpg" ); // Create an icon for label1.
   // Configure label1.
   Label label1 = new Label( "Search" );
   label1.setGraphic( new ImageView( image ) );
   label1.setFont( new Font( "Arial", 30 ) );
   label1.setTextFill( Color.web( "#0076a3" ) ); // Color as hex web value.
   label1.setTextAlignment( TextAlignment.JUSTIFY );
```

JAVAFX USER INTERFACE CONTROLS 4

LABEL SAMPLE C

```
// Configure label2.
Label label2 = new Label( "Values" );
label2.setFont( Font.font( "Cambria", 32 ) );
label2.setRotate( 270 ); label2.setTranslateY( 50 );
// Configure label3.
label3.setWrapText( true ); label3.setTranslateY( 50 ); label3.setPrefWidth( 100 );
// LAMBDA EXPRESSION (JAVA 8).
// label3.setOnMouseEntered(
//    ( MouseEvent e ) -> { label3.setScaleX( 1.5 ); label3.setScaleY( 1.5 ); } );
label3.setOnMouseEntered(
   new EventHandler<MouseEvent>() {
      @Override public void handle( MouseEvent e ) {
         label3.setScaleX( 1.5 ); label3.setScaleY( 1.5 ); } } );
label3.setOnMouseExited(
   new EventHandler<MouseEvent>() {
      @Override public void handle( MouseEvent e ) {
         label3.setScaleX( 1 ); label3.setScaleY( 1 ); } } );
```

JAVAFX USER INTERFACE CONTROLS 5

LABEL SAMPLE D

```
        // Configure layout.
        HBox hbox = new HBox();
        hbox.setSpacing( 10 );
        hbox.getChildren().add( label1 );
        hbox.getChildren().add( label2 );
        hbox.getChildren().add( label3 );
        // Add the horizontal box layout as child of the scene graph root.
        ( (Group) scene.getRoot() ).getChildren().add( hbox );
        // Configure primary stage.
        stage.setTitle( "Label Sample" );
        stage.setWidth( 420 );
        stage.setHeight( 180 );
        stage.setScene( scene );
        stage.show();
    }
}
```

JAVAFX USER INTERFACE CONTROLS 6

JAVAFX UI CONTROLS: BUTTON

The JavaFX **Button** enables to process an action when a user clicks a button, and it is an extension of the **Labeled** class. A **Button** can display text, an image, or both.
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/button
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/labeled

BUTTON SAMPLE A

```
import javafx.application.Application; import javafx.event.ActionEvent;
import javafx.geometry.Pos; import javafx.scene.Group; import javafx.scene.Scene;
import javafx.scene.control.Button; import javafx.scene.control.Label;
import javafx.scene.effect.DropShadow; import javafx.scene.image.Image;
import javafx.scene.image.ImageView; import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox; import javafx.scene.layout.VBox;
import javafx.scene.paint.Color; import javafx.scene.text.Font;
import javafx.stage.Stage;
```

JAVAFX USER INTERFACE CONTROLS 7

BUTTON SAMPLE 8

```
public class ButtonSample extends Application {

   private static final Color color = Color.web( "#FF00FF" ); // Magenta.
   Button button3 = new Button( "Decline" );
   DropShadow shadow = new DropShadow(); // Visual effect for shadowing.
   Label label = new Label();

   // Main method to enable JavaFX support in all IDEs.
   public static void main( String[] args ) { launch( args ); }

   // Main entry point for JavaFX applications.
   @Override
   public void start( Stage stage ) {
      Scene scene = new Scene( new Group() );
      scene.getStylesheets().add( "ControlStyle.css" );
```

JAVAFX USER INTERFACE CONTROLS 8

BUTTON SAMPLE C

```
// Configure the primary stage (part 1).
stage.setTitle( "Button Sample" );
stage.setWidth( 300 );
stage.setHeight( 190 );
// Configure label.
label.setFont( Font.font( "Times New Roman", 22 ) );
label.setTextFill( color );
// Load icon images.
Image imageDecline = new Image( "not.png" );
Image imageAccept = new Image( "ok.png" );
// Configure button1.
Button button1 = new Button( "Accept", new ImageView( imageAccept ) );
button1.getStyleClass().add( "button1" );
button1.setOnAction( ( ActionEvent e ) -> { label.setText( "Accepted" ); } );
// Configure button2.
Button button2 = new Button( "Accept" );
button2.setOnAction( ( ActionEvent e ) -> { label.setText( "Accepted" ); } );
```

JAVAFX USER INTERFACE CONTROLS 9

BUTTON SAMPLE D

```
// Configure button3.
button3.setOnAction( ( ActionEvent e ) -> { label.setText( "Declined" ); } );
button3.addEventHandler( MouseEvent.MOUSE_ENTERED,
                         ( MouseEvent e ) -> { button3.setEffect( shadow ); } );
button3.addEventHandler( MouseEvent.MOUSE_EXITED,
                         ( MouseEvent e ) -> { button3.setEffect( null ); } );
// Configure a horizontal box layout.
HBox hbox1 = new HBox();
hbox1.getChildren().add( button2 );
hbox1.getChildren().add( button3 );
hbox1.getChildren().add( label );
hbox1.setSpacing( 10 );
hbox1.setAlignment( Pos.BOTTOM_CENTER );
// Configure button4.
Button button4 = new Button();
button4.setGraphic( new ImageView( imageAccept ) );
button4.setOnAction( ( ActionEvent e ) -> { label.setText( "Accepted" ); } );
```

BUTTON SAMPLE E

```
// Configure button 5.
Button button5 = new Button();
button5.setGraphic( new ImageView( imageDecline ) );
button5.setOnAction( ( ActionEvent e ) -> { label.setText( "Declined" ); } );
// Configure a horizontal box layout.
HBox hbox2 = new HBox();
hbox2.getChildren().add( button4 );
hbox2.getChildren().add( button5 );
hbox2.setSpacing( 25 );
// Configure a vertical box layout.
VBox vbox = new VBox();
vbox.setLayoutX( 20 );
vbox.setLayoutY( 20 );
vbox.getChildren().add( button1 );
vbox.getChildren().add( hbox1 );
vbox.getChildren().add( hbox2 );
vbox.setSpacing( 10 );
```

BUTTON SAMPLE F

```
        // Set the vertical box layout as the scene graph root.
        ( (Group) scene.getRoot() ).getChildren().add( vbox );
        // Configure the primary stage (part 2).
        stage.setScene( scene );
        stage.show();
    }
}
```

CONTROL STYLE CSS

```
// CSS file for the ButtonSample Java class
.button1 { -fx-font: 22 arial; -fx-base: #b6e7c9; }
```

See: en.wikipedia.org/wiki/cascading_style_sheets

JAVAFX UI CONTROLS: TABLE VIEW

The most important classes for **creating tables in JavaFX** applications are:

- **TableView**,
- **TableColumn**, and
- **TableCell**.

You can populate a table by implementing the **data model** (e.g. a **Person** class) and by applying a **cell factory** (e.g. a renderer for the cell). The table classes provide built-in capabilities to sort data in columns and to resize columns when necessary.

See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/tableview
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/tablecolumn
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/tablecell

JAVAFX UI CONTROLS: HOW A TABLE VIEW CELL IS RENDERED A

Each table cell will receive an object (e.g. an instance of **Person**). To do the rendering, the table cell will need a **CellValueFactory** and a **CellFactory** (see **TableColumn**).
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/tablecolumn

The table cell must know which part of **Person** it needs to display.
Example: For all table cells in the "birthday" column, the part of **Person** to display is the **birthday** (i.e. an instance of **LocalDate**) data field. This is our "birthday" column:

```
private TableColumn<Person, LocalDate> birthdayColumn;
```

Later during initialization, we will set the **CellValueFactory**:

```
birthdayColumn.setCellValueFactory( cellData -> cellData.getValue().birthdayProperty() );
```

JAVAFX UI CONTROLS: HOW A TABLE VIEW CELL IS RENDERED B

Once the table cell has the value to be displayed, it must know how to display it.

Example: The birthday value will be formatted/colored depending on the month.

A **CellFactory** is a renderer of the table cell from the table cell item.

Note: Set this property to enable cell editing or to customize cell visualization.
Note: The next **CellFactory** includes: lambda expression, generics, and anonymous inner classes. However, now we focus only on the **updateItem()** function: which is called whenever the table cell is rendered, receiving the birthdate to be rendered.

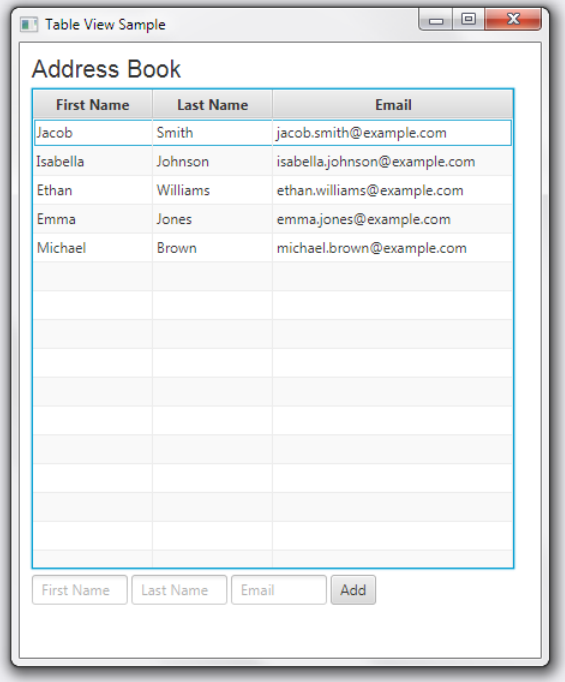
See: docs.oracle.com/javase/8/javafx/api/javafx/scene/control/tablecolumn

JAVAFX UI CONTROLS: HOW A TABLE VIEW CELL IS RENDERED C

```
DateTimeFormatter myDateFormatter = DateTimeFormatter.ofPattern( "dd.MM.yyyy" );
// Custom rendering of the table cell.
birthdayColumn.setCellFactory(
   column -> {
      return new TableCell< Person, BirthDate>() {
         // updateItem: called whenever cell should be rendered.
         @Override protected void updateItem( BirthDate item, boolean empty ) {
            super.updateItem( item, empty );
            if( item == null || empty) { setText( null ); setStyle( "" ); }
            else { // Format birth date.
               setText( myDateFormatter.format( item ) );
               // Note: place custom styling here!
            }
         } // End updateItem().
      };
   } // End lambda expression.
); // End setCellFactory().
```

JAVAFX USER INTERFACE CONTROLS 16

TABLE VIEW SAMPLE



JAVAFX USER INTERFACE CONTROLS 17

TABLE VIEW SAMPLE A

```
import javafx.application.Application; import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections; import javafx.collections.ObservableList;
import javafx.event.ActionEvent; import javafx.geometry.Insets;
import javafx.scene.Group; import javafx.scene.Scene;
import javafx.scene.control.Button; import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableColumn.CellEditEvent;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.HBox; import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

// ...
public class TableViewSample extends Application {
```

JAVAFX USER INTERFACE CONTROLS 18

TABLE VIEW SAMPLE B

```
private final TableView<Person> table = new TableView<>(); // ...
// ObservableList: a list that allows listeners to track changes when they occur.
private final ObservableList<Person> data = FXCollections.observableArrayList(
   new Person( "Jacob", "Smith", "jacob.smith@example.com" ),
   new Person( "Isabella", "Johnson", "isabella.johnson@example.com" ),
   new Person( "Ethan", "Williams", "ethan.williams@example.com" ),
   new Person( "Emma", "Jones", "emma.jones@example.com" ),
   new Person( "Michael", "Brown", "michael.brown@example.com" ) );
private final HBox hb = new HBox(); // ...

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

TABLE VIEW SAMPLE C

```
// ...
@Override
public void start( Stage stage ) {
   Scene scene = new Scene( new Group() );
   stage.setTitle( "Table View Sample" );
   stage.setWidth( 450 ); stage.setHeight( 550 );
   final Label label = new Label( "Address Book" );
   label.setFont( new Font( "Arial", 20 ) );
   table.setEditable( true );
   TableColumn< Person, String > firstNameCol = new TableColumn<>( "First Name" );
   firstNameCol.setMinWidth( 100 );
   firstNameCol.setCellValueFactory( new PropertyValueFactory<>( "firstName" ) );
   firstNameCol.setCellFactory( TextFieldTableCell.<Person>forTableColumn() );
   firstNameCol.setOnEditCommit(
      ( CellEditEvent< Person, String > t ) -> {
         ( (Person) t.getTableView().getItems().get(
            t.getTablePosition().getRow() ) ).setFirstName( t.getNewValue() ); } );
```

JAVAFX USER INTERFACE CONTROLS 20

TABLE VIEW SAMPLE D

```
TableColumn< Person, String > lastNameCol = new TableColumn<>( "Last Name" );
lastNameCol.setMinWidth( 100 );
lastNameCol.setCellValueFactory( new PropertyValueFactory<>( "lastName" ) );
lastNameCol.setCellFactory( TextFieldTableCell.<Person>forTableColumn() );
lastNameCol.setOnEditCommit(
   ( CellEditEvent< Person, String > t ) -> {
      ( (Person) t.getTableView().getItems().get(
         t.getTablePosition().getRow() ) ).setLastName( t.getNewValue() ); } );
TableColumn< Person, String > emailCol = new TableColumn<>( "Email" );
emailCol.setMinWidth( 200 );
emailCol.setCellValueFactory( new PropertyValueFactory<>( "email" ) );
emailCol.setCellFactory( TextFieldTableCell.<Person>forTableColumn() );
emailCol.setOnEditCommit(
   ( CellEditEvent< Person, String > t ) -> {
      ( (Person) t.getTableView().getItems().get(
         t.getTablePosition().getRow() ) ).setEmail( t.getNewValue() ); } );
```

TABLE VIEW SAMPLE E

```
table.setItems( data );
table.getColumns().addAll( firstNameCol, lastNameCol, emailCol );
final TextField addFirstName = new TextField();
addFirstName.setPromptText( "First Name" );
addFirstName.setMaxWidth( firstNameCol.getPrefWidth() );
final TextField addLastName = new TextField();
addLastName.setMaxWidth( lastNameCol.getPrefWidth() );
addLastName.setPromptText( "Last Name" );
final TextField addEmail = new TextField();
addEmail.setMaxWidth( emailCol.getPrefWidth() );
addEmail.setPromptText( "Email" );
```

TABLE VIEW SAMPLE F

```
        final Button addButton = new Button( "Add" );
        addButton.setOnAction(
            ( ActionEvent e ) -> {
                data.add( new Person( addFirstName.getText(),
                                      addLastName.getText(),
                                      addEmail.getText() ) );
                addFirstName.clear(); addLastName.clear(); addEmail.clear(); } );
        hb.getChildren().addAll( addFirstName, addLastName, addEmail, addButton );
        hb.setSpacing( 3 );
        final VBox vbox = new VBox();
        vbox.setSpacing( 5 ); vbox.setPadding( new Insets( 10, 0, 0, 10 ) );
        vbox.getChildren().addAll( label, table, hb );
        ( (Group) scene.getRoot() ).getChildren().addAll( vbox );
        stage.setScene( scene );
        stage.show();
    }
```

TABLE VIEW SAMPLE G

```
// Data model.
public static class Person {
   private final SimpleStringProperty firstName;
   private final SimpleStringProperty lastName;
   private final SimpleStringProperty email;
   // Constructor.
   private Person( String fName, String lName, String email ) {
      this.firstName = new SimpleStringProperty( fName );
      this.lastName = new SimpleStringProperty( lName );
      this.email = new SimpleStringProperty( email ); }
   // Accessors and modifiers.
   public String getFirstName() { return firstName.get(); }
   public void setFirstName( String fName ) { firstName.set( fName ); }
   public String getLastName() { return lastName.get(); }
   public void setLastName( String fName ) { lastName.set( fName ); }
   public String getEmail() { return email.get(); }
   public void setEmail( String fName ) { email.set( fName ); } } }
```

JAVAFX UI CONTROLS: MENU A

A menu is a list of actionable items that can be displayed upon request.
You can use the following JavaFX classes to build menus in your JavaFX app:

- **MenuBar**
- **MenuItem**
  - **Menu**
  - **CheckMenuItem**
  - **RadioMenuItem**
  - **Menu**
  - **CustomMenuItem** (**SeparatorMenuItem**)
- **ContextMenu**

Menus are usually organized in a **menu bar** located at the top of the window.

Note: If your UI is too small for a menu bar, you can use **context menus**.

JAVAFX USER INTERFACE CONTROLS 25

JAVAFX UI CONTROLS: MENU B

When a menu is **visible**, users can select 1 menu item at time. After a user clicks an item, the menu returns to **hidden** mode.

Menus in a menu bar are typically grouped into categories. The coding pattern is:

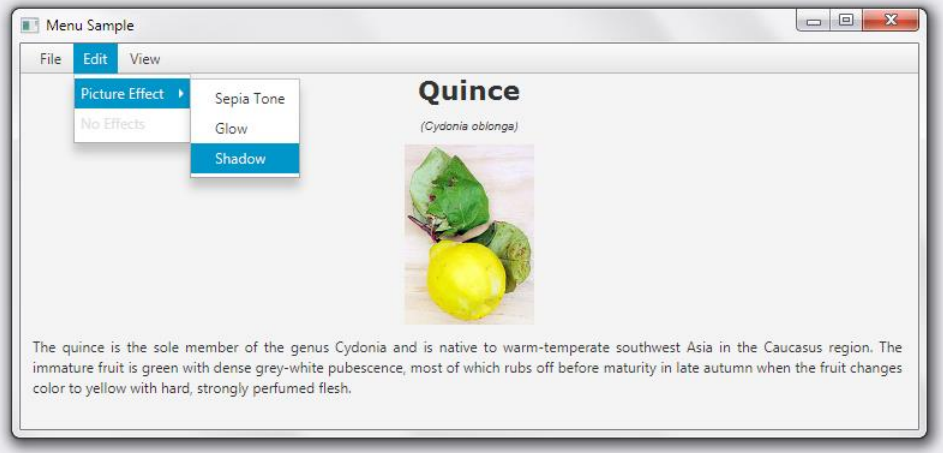
1. to declare a **menu bar**,
2. to define the **category menus**, and
3. to populate the category menus with **menu items**.

Use the following menu item classes when building menus in your JavaFX app:

- **MenuItem:** to create one actionable option.
- **Menu:** to create a submenu.
- **RadioButtonItem:** to create a mutually exclusive selection.
- **CheckMenuItem:** to create an option that can be selected/unselected.
- **SeparatorMenuItem:** to separate menu items within one category.

JAVAFX USER INTERFACE CONTROLS 26

MENU SAMPLE



JAVAFX USER INTERFACE CONTROLS 27

MENU SAMPLE A

```
import java.util.AbstractMap.SimpleEntry; import java.util.Map.Entry;
import javafx.application.Application; import javafx.beans.value.ObservableValue;
import javafx.event.ActionEvent;
import javafx.geometry.Insets; import javafx.geometry.Pos;
import javafx.scene.Node; import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.effect.DropShadow; import javafx.scene.effect.Effect;
import javafx.scene.effect.Glow; import javafx.scene.effect.SepiaTone;
import javafx.scene.image.Image; import javafx.scene.image.ImageView;
import javafx.scene.input.*;
import javafx.scene.layout.VBox; import javafx.scene.paint.Color;
import javafx.scene.text.Font; import javafx.scene.text.TextAlignment;
import javafx.stage.Stage;

// ...
public class MenuSample extends Application {
```

MENU SAMPLE B

```
// ...
final PageData[] pages = new PageData[] { new PageData("Apple",
                                                       "Bla bla bla...",
                                                       "Malus Domestica"),
                                          new PageData("Hawthorn",
                                                       "Bla bla bla...",
                                                       "Crataegus Monogyna"),
                                          new PageData("Ivy",
                                                       "Bla bla bla...",
                                                       "Parthenocissus Tricuspidata"),
                                          new PageData("Quince",
                                                       "Bla bla bla...",
                                                       "Cydonia Oblonga") };

// ...
final String[] viewOptions = new String[] { "Title", "Binomial", "Image", "Desc." };
```

MENU SAMPLE C

```
// An array of map entries (key-value pairs) for visual effects to affect the picture.
final Entry< String, Effect >[] effects = new Entry[] {
                                    new SimpleEntry<>( "Sepia Tone", new SepiaTone() ),
                                    new SimpleEntry<>( "Glow", new Glow() ),
                                    new SimpleEntry<>( "Shadow", new DropShadow() ) };

final ImageView pic = new ImageView(); // ...
final Label name = new Label(); // ...
final Label binName = new Label(); // ...
final Label description = new Label(); // ...
private int currentIndex = -1; // ...

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

JAVAFX USER INTERFACE CONTROLS 30

MENU SAMPLE D

```
// ...
@Override
public void start( Stage stage ) {
   stage.setTitle( "Menu Sample" );
   // Create a scene graph with a vertical box layout as the root node.
   Scene scene = new Scene( new VBox(), 400, 350 );
   scene.setFill( Color.OLDLACE );
   // Set fonts for image title and subtitle.
   name.setFont( new Font( "Verdana Bold", 22 ) );
   binName.setFont( new Font( "Arial Italic", 10 ) );
   // Configure picture settings.
   pic.setFitHeight( 150 );
   pic.setPreserveRatio( true );
   // Configure description settings.
   description.setWrapText( true );
   description.setTextAlignment( TextAlignment.JUSTIFY );
```

MENU SAMPLE E

```
// Randomly select another picture updating: title, subtitle, and description.
shuffle();
// Create a new menu bar.
MenuBar menuBar = new MenuBar();
// GRAPHICAL ELEMENTS
// Arrange title, subtitle, picture, and description in a vertical box.
final VBox vbox = new VBox();
vbox.setAlignment( Pos.CENTER );
vbox.setSpacing( 10 );
vbox.setPadding( new Insets( 0, 10, 0, 10 ) );
vbox.getChildren().addAll( name, binName, pic, description );
// MENU FILE
// Create a new menu "File" including items: "Shuffle", "Clear", and "Exit".
Menu menuFile = new Menu( "File" );
MenuItem add = new MenuItem( "Shuffle",
                             new ImageView( new Image("n.png",16,16,true,true) ) );
```

MENU SAMPLE F

```
add.setOnAction(
   ( ActionEvent t ) -> { shuffle(); vbox.setVisible( true ); } ); // Show content.
MenuItem clear = new MenuItem( "Clear" );
clear.setAccelerator( KeyCombination.keyCombination( "Ctrl+X" ) );
clear.setOnAction(
   ( ActionEvent t ) -> { vbox.setVisible( false ); } ); // Hide content.
MenuItem exit = new MenuItem( "Exit" );
exit.setOnAction( ( ActionEvent t ) -> { System.exit( 0 ); } ); // Exit app.
// Add all items to menu "File".
menuFile.getItems().addAll( add, clear, new SeparatorMenuItem(), exit );
// MENU EDIT
// Create menu "Edit" including: "Picture Effect" (submenu), and "No Effects".
Menu menuEdit = new Menu( "Edit" );
Menu menuEffect = new Menu( "Picture Effect" );
// Note: only a single Toggle within a ToggleGroup may be selected at any one time.
final ToggleGroup groupEffect = new ToggleGroup();
```

MENU SAMPLE G

```
// Create a radio menu item for each visual effect in "effects".
for( Entry< String, Effect > effect : effects ) {
   RadioMenuItem itemEffect = new RadioMenuItem( effect.getKey() );
   itemEffect.setUserData( effect.getValue() );
   itemEffect.setToggleGroup( groupEffect );
   menuEffect.getItems().add( itemEffect ); }
// Create and initialize the menu item "No Effects".
final MenuItem noEffects = new MenuItem( "No Effects" );
noEffects.setDisable( true );
// If "No Effects" is clicked disable any visual effect in the toggle group.
noEffects.setOnAction(
   ( ActionEvent t ) -> {
      pic.setEffect( null );
      groupEffect.getSelectedToggle().setSelected( false );
      noEffects.setDisable( true ); } );
```

MENU SAMPLE H

```
// Enable "No Effects" as soon as a visual effect is activated in toggle group.
groupEffect.selectedToggleProperty().addListener(
   ( ObservableValue<? extends Toggle > ov, Toggle oldTgl, Toggle newTgl ) -> {
      if( groupEffect.getSelectedToggle() != null ) {
         Effect effect = (Effect) groupEffect.getSelectedToggle().getUserData();
         pic.setEffect( effect );
         noEffects.setDisable( false ); }
      else { noEffects.setDisable( true ); } } );
// Add submenu "Picture Effects" and menu item "No Effects" to menu "Edit".
menuEdit.getItems().addAll( menuEffect, noEffects );
// MENU VIEW - Create menu "View" with: "Title", "Binomial", "Image", and "Desc.".
Menu menuView = new Menu( "View" );
// Configure check menu items to link their selection to nodes visibility.
CheckMenuItem titleView = createMenuItem( "Title", name );
CheckMenuItem binNameView = createMenuItem( "Binomial", binName );
CheckMenuItem picView = createMenuItem( "Image", pic );
CheckMenuItem descriptionView = createMenuItem( "Desc.", description );
```

MENU SAMPLE

```
// Add all the menu items to the menu "View".
menuView.getItems().addAll( titleView, binNameView, picView, descriptionView );
// Add all the menus to the menu bar.
menuBar.getMenus().addAll( menuFile, menuEdit, menuView );
// CONTEXT MENU
// Create a new context menu including only one menu item: "Copy Image".
final ContextMenu cm = new ContextMenu();
MenuItem cmItem1 = new MenuItem( "Copy Image" );
cmItem1.setOnAction(
   ( ActionEvent e ) -> {
      // Clipboard class represents OS clipboard to place data during: copy etc.
      Clipboard clipboard = Clipboard.getSystemClipboard();
      // ClipboardContent is a data container for Clipboard data.
      ClipboardContent content = new ClipboardContent();
      content.putImage( pic.getImage() );
      clipboard.setContent( content ); } );
cm.getItems().add( cmItem1 ); // Add "Copy Image" menu item to the context menu.
```

MENU SAMPLE J

```
    // Enable context menu only on the picture!
    pic.addEventHandler( MouseEvent.MOUSE_CLICKED,
                         ( MouseEvent e ) -> {
                            if( e.getButton() == MouseButton.SECONDARY ) {
                               cm.show( pic, e.getScreenX(), e.getScreenY() ); } } );
    // Add menu bar and content (arranged in vertical box) to scene graph root node.
    ( (VBox) scene.getRoot() ).getChildren().addAll( menuBar, vbox );
    stage.setScene( scene );
    stage.show();
}
```

MENU SAMPLE K

```
// Randomly select another picture updating: title, subtitle, and description.
private void shuffle() {
   int i = currentIndex;
   while( i == currentIndex ) { i = (int) ( Math.random() * pages.length ); }
   pic.setImage( pages[i].image );
   name.setText( pages[i].name );
   binName.setText( "(" + pages[i].binNames + ")" );
   description.setText( pages[i].description );
   currentIndex = i;
}
```

JAVAFX USER INTERFACE CONTROLS 38

MENU SAMPLE L

```
// Configure a check menu item to link its selection to a scene graph node visibility.
private static CheckMenuItem createMenuItem( String title, final Node node ) {
   CheckMenuItem cmi = new CheckMenuItem( title );
   cmi.setSelected( true );
   cmi.selectedProperty().addListener(
      ( ObservableValue<? extends Boolean > ov, Boolean oldVal, Boolean newVal ) -> {
         node.setVisible( newVal ); } );
   return cmi;
}
```

MENU SAMPLE M

```
   // Class to store together the data of a page.
   private class PageData {
      public String name;
      public String description;
      public String binNames;
      public Image image;
      public PageData( String name, String description, String binNames ) {
         this.name = name;
         this.description = description;
         this.binNames = binNames;
         image = new Image( name + ".jpg" );
      }
   }
}
```

JAVAFX UI CONTROLS: FILE CHOOSER

The **FileChooser** class is located in the **javafx.stage** package along with the other basic root graphical elements, such as **Stage**, **Window**, and **Popup**.

A **FileChooser** object can be used to invoke a **file open dialog** for selecting either a single file or multiple files, and to enable a **file save dialog**.

In addition to opening and filtering files, the **FileChooser** allows a user to specify a file name (i.e. full path) for a file to be saved by the application. The **showSaveDialog** method of the **FileChooser** class opens a save dialog window, and then returns the file chosen by the user (or **null** if no selection has been performed).

See: docs.oracle.com/javase/8/javafx/api/javafx/stage/filechooser

JAVAFX USER INTERFACE CONTROLS 41

FILE CHOOSER SAMPLE

JAVAFX USER INTERFACE CONTROLS 42

FILE CHOOSER SAMPLE A

```
import java.awt.Desktop;
import java.io.File; import java.io.IOException;
import java.util.List; import java.util.logging.Level; import java.util.logging.Logger;
import javafx.application.Application; import javafx.event.ActionEvent;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button; import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane; import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

// ...
public final class FileChooserSample extends Application {

   private final Desktop desktop = Desktop.getDesktop(); // ...
```

FILE CHOOSER SAMPLE B

```
@Override
public void start( final Stage stage ) {
   stage.setTitle( "File Chooser Sample" );
   final FileChooser fileChooser = new FileChooser();
   final Button openButton = new Button( "Open a Picture..." );
   final Button openMultipleButton = new Button( "Open Pictures..." );
   openButton.setOnAction(
      ( final ActionEvent e ) -> {
         configureFileChooser( fileChooser );
         File file = fileChooser.showOpenDialog( stage );
         if( file != null ) { openFile( file ); } } );
   openMultipleButton.setOnAction(
      ( final ActionEvent e ) -> {
         configureFileChooser( fileChooser );
         List<File> list = fileChooser.showOpenMultipleDialog( stage );
         if( list != null ) {
            list.stream().forEach( ( file ) -> { openFile( file ); } ); } } );
```

FILE CHOOSER SAMPLE C

```
    final GridPane inputGridPane = new GridPane();
    GridPane.setConstraints( openButton, 0, 1 );
    GridPane.setConstraints( openMultipleButton, 1, 1 );
    inputGridPane.setHgap( 6 );
    inputGridPane.setVgap( 6 );
    inputGridPane.getChildren().addAll( openButton, openMultipleButton );
    final Pane rootGroup = new VBox( 12 );
    rootGroup.getChildren().addAll( inputGridPane );
    rootGroup.setPadding( new Insets( 12, 12, 12, 12 ) );
    stage.setScene( new Scene( rootGroup ) );
    stage.show();
}

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

FILE CHOOSER SAMPLE D

```
    private static void configureFileChooser( final FileChooser fileChooser ) {
        fileChooser.setTitle( "View Pictures" );
        fileChooser.setInitialDirectory( new File( System.getProperty( "user.home" ) ) );
        fileChooser.getExtensionFilters().addAll(
            new FileChooser.ExtensionFilter( "All Images", "*.*" ),
            new FileChooser.ExtensionFilter( "JPG", "*.jpg" ),
            new FileChooser.ExtensionFilter( "PNG", "*.png" ) );
    }

    private void openFile( File file ) {
        try { desktop.open( file ); }
        catch( IOException e ) {
            Logger.getLogger(FileChooserSample.class.getName()).log(Level.SEVERE,null,e); }
    }

}
```

MENU SAMPLE MOD: ADD NEW SAVE IMAGE ACTION TO CONTEXT MENU

```
// ...
MenuItem cmItem2 = new MenuItem( "Save Image" );
cmItem2.setOnAction(
   ( ActionEvent e ) -> {
      FileChooser fileChooser1 = new FileChooser();
      fileChooser1.setTitle( "Save Image" );
      System.out.println( pic.getId() );
      File file = fileChooser1.showSaveDialog( stage );
      if( file != null ) {
         try { ImageIO.write( SwingFXUtils.fromFXImage( pic.getImage(), null ),
                              "png",
                              file ); }
         catch(IOException exc) { System.out.println( exc.getMessage() ); } } } );
//cm.getItems().add( cmItem1 ); // Add "Copy Image" menu item to the context menu.
cm.getItems().addAll( cmItem1, cmItem2 ); // Add "Copy Image" and "Save Image“...
```

JAVAFX COLLECTIONS 1

JAVAFX COLLECTIONS: MAIN INTERFACES AND CLASSES

Collections in JavaFX are defined by the **javafx.collections** package, including:

Interfaces:
ObservableList: a list that enables listeners to track list changes.
ListChangeListener: tracks changes to ObservableList.
ObservableMap: a map that enables listeners to track map changes.
MapChangeListener: tracks changes to ObservableMap.

Classes:
FXCollections: utility class with static methods for collections.
ListChangeListener.Change: a change made to an ObservableList.
MapChangeListener.Change: a change made to an ObservableMap.

See: docs.oracle.com/javase/8/javafx/api/javafx/collections/package-summary

JAVAFX COLLECTIONS 2

JAVAFX COLLECTIONS: USING OBSERVABLE LISTS AND MAPS

The **ObservableList** and **ObservableMap** interfaces both extend **javafx.beans.Observable** (and **java.util.List** or **java.util.Map**, respectively) to provide a list/map supporting **observability**. These interfaces include methods for adding/removing **listeners** (**ListChangeListener** and **MapChangeListener**.

Once an **ObservableList** has been created, you can register a **ListChangeListener** that will receive notification whenever a change is made on the **ObservableList**.

```
observableList.addListener(
   new ListChangeListener() {
      @Override
      public void onChanged( ListChangeListener.Change change ) {
          System.out.println( "Detected a change!" ); } } );
```

JAVAFX COLLECTIONS 3

JAVAFX COLLECTIONS: TRACKING COLLECTION CHANGES

When using a **ListChangeListener** or **MapChangeListener**, the **onChanged** method contains an object storing info about the change:

- an instance of **ListChangeListener.Change** or **MapChangeListener.Change**.

Note: Remember that a **ListChangeListener.Change** object can store multiple changes; but a **MapChangeListener.Change** objects can only store 1 change.

```
observableList.addListener( new ListChangeListener() {
      @Override public void onChanged( ListChangeListener.Change change ) {
         while( change.next() ) {
            System.out.println( "Added? " + change.wasAdded() );
            System.out.println( "Removed? " + change.wasRemoved() );
            System.out.println( "Replaced? " + change.wasReplaced() );
            System.out.println( "Permutated? " + change.wasPermutated() ); } } } );
```