latex input: mmd-article-header Title: JavaFX Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Java, programming language, syntax, GUI, desktop, application development CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

# Start and Run a JavaFX GUI Project in IntelliJ

New Project → JavaFX → Finish → `src/sample/Main` → Run

# Intro to JavaFX

From Oracle's Getting Started Tutorial

## Features

1. Look & feel customizable with CSS
2. FXML scripting language optional for UI presentation development
3. JavaFX Scene Builder for designing UIs without writing code, generates editable FXML
4. Cross-platform compatibility, with consistent maintenance ensured by Oracle
5. API designed to be friendly with Scala et al.
6. Possible to integrate with JavaScript and HTML5 via WebView
7. You can embed Swing content into JavaFX applications
8. I believe the whole system was *written in vanilla Java!* (whoa)
9. Only a subset of the whole framework is actually Public API
10. It facilitates drag-and-drop and everything else you wish was easier in JavaScript

## Libraries & APIs

3D Graphics, Camera, Canvas for drawing directly into a scene, Printing, Rich Text, Multitouch, Hi-DPI (nice displays), hardware-accelerated graphics, high- performance media engine supporting web multimedia, self-contained application deployment model, animation.

## Structure

1. The **Scene Graph** is a hierarchical tree of **Nodes** that represents all of the visual elements of the application's User Interface
2. Each Node of a Scene Graph besides the Root has a single *parent* and 0+ *children*

3. Each **Scene Node** can have --- effects, blurs, shadows, opacity, transforms, event handles (e.g. mouse & keyboard), application-specific *state*
4. **Nodes** may be --- (2D & 3D), images, media, *embedded web browser* (wow), text, UI controls, charts, groups, containers, etc.
5. "Pulses" are events, throttled to 60 frames per second, telling the scene graph to synchronize with the "Prism" renderer. They are scheduled whenever the Scene Graph changes.

## Beginner Classes

1. Example "main class" (i.e. the one that you "run")

```
public class Main extends Application {
    public static void main(String[] args) { launch(args); }
    @Override public void start(Stage primaryStage) throws Ex
        this.primaryStage = primaryStage;
        this.primaryStage.setTitle("p2p-gui");
        loadTheWindow();
    }
}
```

2. A `Stage` is a window in your OS

3. A `Scene` is a UI element that holds content inside a window/`Stage`

## Some useful classes

1. `FXCollections.observable[Collection]([collectionInstance])` --- turns your existing `Collection` into an "*observable*" one on which you can install `Listeners`

```
FXCollections.observableArrayList(/*List<T>*/ tList);
```

## Properties and Binding

From Oracle's Binding Tutorial

3. `interface Property` --- `getProp`, `setProp`, `addListener`, `removeListener`, `bind` (see § below)
    1. `class SimpleStringProperty` --- normal string methods, listeners, `fireValueChangedEvent`,

4. JavaFX represents object properties using the *JavaBeans component architecture*, which is both an API and a *design pattern*

5. **Binding** allows you to express direct relationships between variables -- changes in one object are automatically reflected in another
6. A binding *observes* its list of *dependencies* for changes, and updates itself accordingly
7. A **property** is an object field that uses simple `getProp()`/`setProp(obj)` modifier naming conventions
8. The interfaces `Observable` (which does *not* wrap a value) and `ObservableValue` (which wraps a value) *fire* change notifications
9. One then *receives* change notifications in the corresponding interfaces `InvalidationListener` (from `Observable`) and `ChangeListener` (from `ObservableValue`)
10. JavaFX bindings and properties all support **lazy evaluation** (they're only computed when the value is *requested*) *unless* you install a `ChangeListener`, which forces *eager computation* because the `ChangeListener` has to know immediately if a change has actually occurred 8. Normally, when any of a binding's dependencies are changed, it is marked `invalid`, but it is not actually recalculated until someone calls `getValue()` * You can install a listener on `invalidated()`, though note that *further* changes to an invalid property will *not* trigger the listener again

# Concurrency

From Oracle's Concurrency Docs

1. Try to use `javafx.concurrent` as opposed to the standard Java `Runnable`
2. Keep the interface responsive by *backgrounding* time-consuming tasks
3. The JavaFX Scene Graph is *not thread-safe* and hence can only be accessed & modified from the *UI thread* called "JavaFX Application thread". You must keep this thread un-clogged from long tasks
4. `interface Worker` --- provides APIs for background Worker threads to communicate with the UI
    1. `class Task` --- fully *observable*, for doing work on a background thread
    2. `class Service` --- *executes* `Tasks` (see below for more)

## Task

5. A `WorkerStateEvent` fires when a `Worker` changes **state**, both `Task` and `Service` can listen for these. The *worker states* are listed in the table below

6. `Worker` progress can be obtained via `totalWork`, `workDone`, and `progress`

7. Here's what a `Task` might look like

```
Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        int iterations;
        // ...
        return iterations;
    }
};
```

8. Beware: Don't try to modify the an active Scene Graph from `call()`---you will throw a `RuntimeException`
9. You can call `updateProgress`, `updateMessage` (this is a *property* on `Worker` potentially useful to the UI thread), and `updateTitle` (similar to `message` property) as appropriate
10. **Start** it via `task.start()` or `ExecutorService.submit(task)` (as in normal Java)
11. You're supposed to check `isCancelled()` every now and then and stop processing if `true` (as in normal Java)
12. To have a progressBar you'd call `updateProgress(...)` *inside* the `Task`, and then roughly the following

```
new ProgressBar().progressProperty().bind(task.progressPropert
```

## Service

1. `Service` facilitates interaction between background threads and the UI thread
2. So what you do is the following

```
class MyService extends Service<String> {
    @Override protected Task<String> createTask() {
        return new Task<String>() {
            @Override protected String call() {
                return "Hello, World!";
            }

            // optional...
            @Override protected void succeeded() {...}
            @Override protected void cancelled() {...}
            @Override protected void failed() {...}
        }
    }
}
```

3. Now you can *start* the service with an `Executor` or a *daemon thread*, and *restart* it automatically by using a `ScheduledService` which allows a `backoffStrategy` and a `maximumFailureCount`

| Worker state | When |
|---:|---|
| `READY` | when just starting out |
| `SCHEDULED` | after being scheduled for execution |
| `RUNNING` | while executing |
| `SUCCEEDED` | successfully completed; `value` set to *result* |
| `FAILED` | threw exception; `exception` set to exception |
| `CANCELLED` | if it gets inturrupted via `cancel()` |

## Scene Builder 2.0

From an incredible tutorial

### To hook code into the view created in scene builder

1. Say in package `view` you use Scene Builder to create a view called `PersonOverview.fxml`
2. Now *in that same package*, create the file `PersonOverviewController.java`
3. Give all fields & methods to be accessed in the FXML file the `@FXML` annotation
4. The `@FXML private void initialize(){...}` method will be automatically called after the FXML file has been loaded, at which point the FXML fields should have already been *initialized* (I think *you* are responsible for initializing the UI elements that are *not* explicitly linked into the Scene Builder)
5. In the tutorial, the controller has a field

   ```
   @FXML private TableColumn<Person, String> lastNameColumn;
   ```

   This means the Table contains instances of `Person`, and this column renders instances of `String`, so it requires that we provide a way to map given `Person` instances to `String`. We may provide this in the `initialize()` method mentioned above via a Java 8 *lambda* expression

   ```
   lastNameColumn.setCellValueFactory(
       cellData -> cellData.getValue().lastNameProperty());
   ```

   This means (probably not *entirely* correct) that whenever we obtain a new `TableItem<Person>`, we should extract the `Person` instance via `getValue()`, and then extract the `StringProperty` called `lastName` via its *getter*, and use that as the `TableCell<String>`'s `value` *property*.

6. We hook the view to the controller by selecting the proper Java class from the dropdown in SceneBuilder under $Document \rightarrow Controller \rightarrow Controllerclass$ in the left pane

7. We hook each of the UI elements into their corresponding Controller fields by selecting the element in the SceneBuilder element Hierarchy, going to $Code \rightarrow Identity \rightarrow fx : id$ in the right pane and selecting the Controller's field name

8. To hook a `Button` in the UI up to a *handler* in the `Controller`

    1. Create the handler method

        ```
        @FXML private void handleDeletePerson() {
            int selectedIndex = personTable.getSelectionModel().
            personTable.getItems().remove(selectedIndex);
        }
        ```

    2. Select the UI element in the SceneBuilder, and in the right-pane go to $Code \rightarrow Main \rightarrow OnAction$ and select the `handleDeletePerson` method from the dropdown selector

# Tables

## Cells

1. `class Control` --- a `Node` in the scene-graph which can be manipulated by the user
2. `class Labeled extends Control` --- a `Control` that has textual content associated with it (e.g. a `Button`, `Label`, or `Tooltip`)
3. `class Cell<T> extends Labeled`
    1. `T` is the type of the `Cell`'s `ObjectProperty<T> itemProperty`
    2. Used for rendering:
        1. a single "row" inside a `ListView`, `TreeView`, or `TableView`
        2. each individual "cell" inside a `TableView`
    3. Responsible for *rendering* the contained `itemProperty` (above), and sometimes *editing* it
    4. Could contain text, or another control such as a `CheckBox`, or any other UI scene `Node`, like `HBox`
    5. Extremely large data sets are actually represented using a few recycled `Cells` for efficiency

### Cell Factories

from `javafx.scene.control.Cell` docs

1. `Cell` items are *rendered* by the container's skins, e.g. by default a `ListView` will convert it to a `String` and render that as text within a `Label`
2. To set how to render your type within your `Cell` within your *thingy* (`ListView`, `TableColumn`, `TreeView`, `TableView`, or `ListView`), you must provide an implementation of the `cellFactory` callback function defined on the *thingy* (see below example)
3. A `CellFactory` gets called when creating a new `Cell`
4. Cell factories create the `Cell` *and* configure it to react properly to changes in its state

#### Their Example is Pretty Good

The main thing is the last line, though I have replaced their mess with a *lambda*. The `updateItem` method is called whenever the item in the cell changes (e.g. goes off-screen and therefore gets re-filled with a new data element, or [I think/hope] whenever the value in the item changes), so there is no need to explicitly manage bindings, though you *could* if you want to.

To format a `java.lang.Number` as a *currency*, do

```
class MoneyFormatCell extends ListCell<Number> {
    public MoneyFormatCell() {}
    @Override protected void updateItem(Number item, boolean empty)
        super.updateItem(item, empty); // REQUIRED
        setText(item == null ? "" : Util.formatMy(item));
        if (item != null)
            setTextFill(isSellected() ? Color.WHITE :
                item.doubleValue() >= 0 ? Color.BLACK : Color.RED);
    }
}

// The ListView tracks the known moneyList
ObservableList<Money> moniesList = getMoniesList();
ListView<Number> view = new ListView<>(moniesList);

// Option 1: using lambda; value needn't be filled in constructor
//           because it will be set using updateItem(T t, boolean e
view.setCellFactory(item -> return new MoneyFormatCell());

// Option 2: define it *inline* with an *anomymously overridden*
// `ListCell<Number>` factory-like piece of code like so
view.setCellFactory(new Callback<ListView<<Number>, ListCell<Number
    @Override public ListCell<Number> call(final ListView<Number> p
        return new ListCell<Number>() {
            @Override protected void updateItem(Number e, boolean em
                super.updateItem(e, empty);
                // update code goes here (e.g. setText("random.next
}};}});
```

## TreeTableView

1. A Control conceptually similar to TreeView and TableView
    1. Same TreeItem API as TreeView (see below)
    2. Same TableColumn-based approach as TreeTableView, using TreeTableColumn
2. The user can sort by multiple columns by holding the *shift* key while clicking on column headers
3. The TreeTableView automatically *observes* its root TreeItem instance
4. You can customize class TreeTableRow<T>, but "more often than not" [docs] it is easier to customize individual Cells in a row rather than the Row itself.
5. class TreeItem<T>
    1. *not* a UI Node
    2. T is the type of the value property
    3. supplies a hierarchy of values to a TreeView or TreeTableView
    4. You can override ObservableList<TreeItem<T>> getChildren()

**cellValueFactoryProperty**

```
TreeTableColumn.cellValueFactoryProperty [of type:]
    public final ObjectProperty<
        Callback<
            TreeTableColumn.CellDataFeatures<S,T>,
            OvservableValue<T>
        >
    >
```

**What does that type mean?**

1. `class ObjectProperty<T>` --- full implementation of a `Property<T>` (see above) wrapping an arbitrary object of static type `T`
2. `interface Callback<P,R>`
    1. One method: `R call(P param)`
    2. Implemented by e.g. `TreeItemPropertyValueFactory`
    3. A reusable interface for defining APIs that require a call back
3. `class TreeTableColumn.CellDataFeatures<S,T>`
    1. `S` --- `TableView` type
    2. `T` --- `TreeTableColumn` type
    3. Immutable wrapper class type to provide all necessary information for a particular `Cell`
4. `class TreeItemPropertyValueFactory<S,T>` is a convenience implementation of the `Callback` interface designed for use within the `TreeTableColumn.cellValueFactoryProperty`. I believe it means instead of

   ```
   firstNameCol.setCellValueFactory(
           p -> p.getValue().getValue().firstNameProperty());
   ```

   you would do

   ```
   firstNameCol.setCellValueFactory(
           new TreeItemPropertyValueFactory<Person,String>("firs
   ```

   It's obviously not so useful anymore, now that there are lambdas.

## Other features

1. Multimedia support via `javafx.scene.media` APIs for video (FLV) and audio (MP3, AIFF, WAV)

2. Web Browser
    1. Via "Web Component", based on Webkit, provides full browser via API, supporting HTML5, CSS, JavaScript, DOM, and SVG, Back/Forward navigation, etc.
    2. Use the `WebEngine` and a `WebView`

## Refs

1. Oracle's Hello World