COMP6700/2140 Scene Graph, Layout and Styles

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

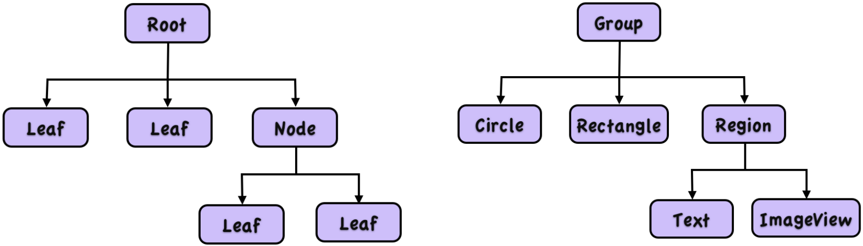
May 2017

Topics

1. Scene Graph, Scenes and Stages
2. *Node*s: shapes, regions, panes, controls
3. Controls and Events
4. *SceneGraph* Visual Editor
5. Properties
6. `fxml/css`: declarative approach to interface programming

Scene Graph of an Application

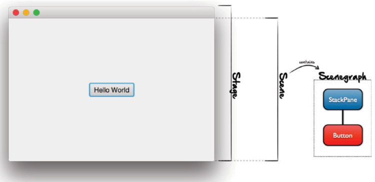
The JavaFX *Scene Graph* is the *model* of all graphical objects which exist in an application. It contains information about what objects to display, what areas of the screen need repainting, and how to render it all.



Individual objects (buttons, shapes, text *etc*) are *leaves*; groups of objects are *nodes*. The scene graph is rooted by a container, usually `javafx.scene.Group` or `javafx.scene.Region`, which is "embedded" into a `scene` object (a window). Once set, an entire scene graph is passed as a `stage` parameter to the (overridden) method `javafx.application.Application.start()` method as the starting point of execution.

"Hello World" of JavaFX Layout

The Scene Graph creation has its own "Hello World" example The full code (with some naughty additions is in HelloWorld.java):

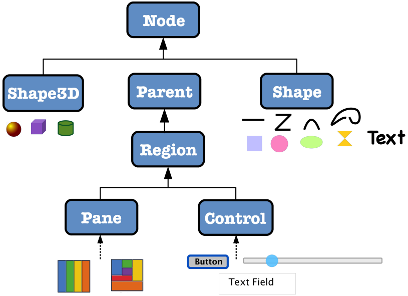


```
public void start(Stage stage) {
   Button button = new Button("Hello World");
   button.setOnAction(e -> System.out.println("Hello World"));
   StackPane pane = new StackPane();
   pane.getChildren().add(button);
   Scene scene = new Scene(myPane);
   stage.setScene(scene);
   stage.setWidth(400);
   stage.setHeight(300);
   stage.show();
}
```

UI components — Shapes, Panes, Controls

Every item in the scene graph is called a *Node*. Branch nodes are of type *Parent*, whose concrete subclasses are:

- *Group* — a container which can impose to all its children those transforms, effects, and states which applied to it
- *Region* — base class for all JavaFX Node-based UI Controls, and all layout containers; can be styled from CSS; Boxes and Pane are its children
- *Control* — base class for all user interface controls (via nodes in the scene graph which can be manipulated by the user)



Shapes

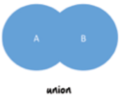
*Shape*s are basic nodes that can be shown on a scene graph. The concrete subclasses (not including 3D shapes, like Box, Cylinder, MeshView, Sphere):

- Line, Polyline, QuadCurve, CubicCurve
- (filled analogues) Rectangle, Polygon, Circle, Ellipse, Arc
- Path (can be closed and filled)
- Text (belongs to a different package `javafx.scene.text`, but is a special kind of shape)

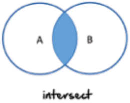
Shapes have numerous properties to determine their geometrical position, size, orientation and visual characteristics (colour, gradient, stroke type *etc*)

The API provides useful set-like binary shape operations:

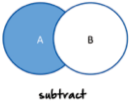
- `union`



- `intersect`



- `subtract`



Controls

These are *widgets* (in the narrow sense of the world) — nodes (instances of *Control*'s concrete subclasses or their subclasses) in the scene graph which can be manipulated by the user:

- `Button`, `RadioButton`, `ComboBox`, `ChoiceBox` and `CheckBox`,…
- `ListView`, `Pagination`
- `MenuBar`, `Menu`, `MenuItem`
- `TextField`, `PasswordField`, `Hyperlink`
- `Slider`, `ProgressBar`
- `Separator`, `SeparatorMenuItem`
- `TableView` and "associates", `TreeView`,…
- `SplitPane`, `ScrollPane`, `TabPane` and `Tab`,…
- *others*

Control nodes can register *events*, and the program should define what happens as the response to those events by setting up *callbacks*:

```
button.setOnAction(e -> <callback-action>);
```

Controls support explicit *skinning* (visual representation of user interface) to separate the functionality and appearance. css-styling can be used to define the look and feel. All controls are made of primitive shapes and panes, and can be scaled without compromising their visual qualities (sharpness and noticeable pixelating).

Layout Panes and Boxes

A set of *panes* (containers) for flexible arrangements of widgets within a scene graph:

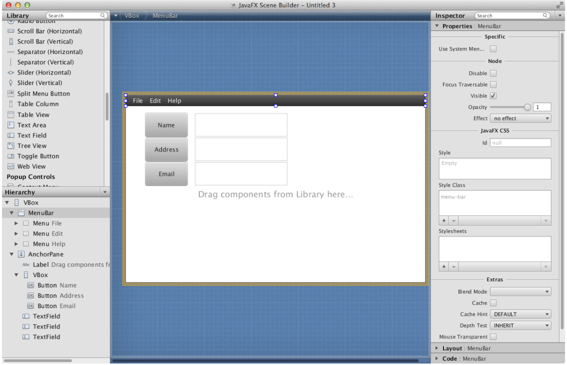
- The `BorderPane` class lays out its content nodes in top, bottom, right, left or centre
- The `HBox` class arranges its content nodes horizontally in a single row
- The `VBox` class arranges its content nodes vertically in a single column
- The `StackPane` class places its content nodes in a back-to-front single stack
- The `GridPane` class enables to create a flexible grid of rows and columns in which to lay out content nodes
- The `FlowPane` class arranges its content nodes in either a horizontal or vertical "flow", wrapping at the specified width (for horizontal) or height (for vertical) boundaries
- A few others — study `javafx.scene.layout` API package

A layout pane is parent node (in the scene graph), and it modifies the position of its children (and also their size if they are resizable). Three sizes can be specified — preferred, minimum and maximum (layout algorithms will try to optimise the size of children using those sizes). Panes store their children in `ObservableList`; the methods `Node.toFront()` and `Node.toBack()` allow to put a child in the first or last position.

One container can be nested inside another, and ultimately within a JavaFX `Application`.

Automating Layout: SceneBuilder

The assembly is automated via visual tools like *SceneBuilder* (*Netbeans* and other IDEs have plugins for visual scene assembly):



Use System Men... ☐
Disable ☐
Focus Traversable ☐
Visible ☑
Opacity 1
Effect no effect
Id null
Style Empty
Style Class menu-bar
Stylesheets
Blend Mode
Cache ☐
Cache Hint DEFAULT
Depth Test INHERIT
Mouse Transparent ☐

“Drag, drop, resize, align, link” — the layout assisted with the *SceneBuilder* tool. The created UI is saved in a .fxml-file which can be loaded in the code of a program.

Declarative Programming in JavaFX: UI Layout

The *SceneBuilder* tool saves a UI as a fxml-file. The fxml format is an XML schema in which XML-tag names match JavaFX class names, such that the content of fxml description file is mapped into the Java code when it's loaded by the *FXMLLoader* class (an optional part of any JavaFX application).

```
<StackPane prefHeight="375" prefWidth="500"
   xmlns:fx="http://javafx.com/fxml"
   fx:controller="steveonjava.Controller">
   <children>
      <ImageView fx:id="imageView">
         <image>
            <Image fx:id="image" url="http://farm1.static.flickr.com/…"/>
         </image>
      </ImageView>
      <Text fx:id="text" cache="true" text="The year's at the spring,&#10;…"/>
      <Button fx:id="button" text="Play Again" onAction="#replay"/>
   </children>
</StackPane>
```

Instead of labouring on the layout using Java's statements, use SceneBuilder to create it and load the resulting fxml-file:

```
Parent root = FXMLLoader.load(getClass().getResource("app_layout.fxml"));
Scene scene = new Scene(root);
```

Declarative Programming in JavaFX: CSS styling

Element styling: The `fxml`-file is complemented by a css-file that defines the styles ("skins") of application elements:

```
#text {
   -fx-font-family: serif;
   -fx-font-weight: bold;
   -fx-font-size: 30pt;
   -fx-fill: goldenrod;
   -fx-effect: dropshadow(three-pass-box, black, 3, .5, 0, 0);
}
#button {
   -fx-background-color: linear-gradient(darkorange, derive(darkorange, -80%));
   -fx-background-radius: 24; -fx-padding: 12;
   -fx-font-size: 16pt; -fx-font-weight: bold;
   -fx-text-fill: white;
}
```

After the UI layout description from a `fxml`-file (created by SceneBuilder) are loaded, the `CSS` definitions also can be read in:

```
scene.getStylesheets().add("app_styles.css");
```

Properties

Nodes (containers, layouts) and leaves (control elements, text, indicators like progress bar, and views like scroll views, list views and tree views) — all have *properties*: shape, size, colour, effect *etc*. When a node or a leaf element is created, its properties are set either explicitly, or by default; they can be reset as a part of transition/animation effect later during program's execution.

```
Text text = new Text("JavaFX technology is kinda cool");
text.setFont(Font.font("Serif", FontWeight.BOLD, 30));
text.setFill(Color.GOLDENROD);
DropShadow dropShadow = new DropShadow();
dropShadow.setRaduis(3);
dropShadow.setSpread(0.5);
text.setEffect(dropShadow);
text.setCache(true);
root.getChildren().add(text);
```

Remember to add a newly created element to its parent according to the scene graph (the last statement above).

Properties and JavaBeans

The concept of *properties* of a object is not a basic Java feature (other languages — for example, Python — have it as a natural part), but it is implemented in the so called *JavaBeans* model, a technique to define a Java class with additional rules on how the state of its instances and the methods one can invoke on them are defined. The name of fields and methods are tightly constrained (together they form a bean *property*). A bean can control which methods are *exported* (public) — by default, they are all public, but this can be modified at run time. Beans can also detect events (like GUI widgets).

A Java class is a JavaBean if it's defined following these specification:

- It has only one public default constructor
- Is is serialisable (implements *Serialzable* interface)
- Has *properties*; a property is a private field which can clients can
  - read/write,
  - read only,
  - write only;

  the names of `getter/setter` methods must obey a naming convention

One reason to have properties is to simplify introspection.

More on Properties

Properties (**not** the system properties used, *eg* in Assignment One) are class attributes (backed by fields) which can be read and set by `getPropName()` and `setPropName(PropType pv)`. Unlike normal `get/set` methods, the property `get()` can perform computation or data acquisition, and `set()` can trigger a change notification which may reset other properties. Unlike *Python* Java does not have a semantic support for properties ☹:

```
value = obj.property; // get-method is called under the hood
obj.property = value; // set-method is called
```

it has a simple class naming convention for the `getter/setter` pair backed by the properly named field which can be recognised by a framework (`java.bean`) and/or a tool (an IDE etc) and treated as property of thus derived name: a class with `String getText()` and `void setText(String s)` methods is recognised as one possessing the property `text`.

A similar (in general, but different in details) property based description of components is used in JavaFX. The main difference is that here, a property is an instance of an interface, not a bean. A property object can have listener attached to it; this can be used to implement callbacks when the property changes or needs to be changed (C. Hortsmann's example SliderDemo.java)

```
Label message = new Label("Hello, JavaFX!"); message.setFont(new Font(100));
Slider slider = new Slider(); slider.setValue(100);
slider.valueProperty().addListener(property
     -> message.setFont(new Font(slider.getValue())));
```

Property Binding

Sometimes (not often!) it is convenient to *bind* properties to each other, such that when one of them changes (*eg*, after a callback), another changes appropriately without an explicit callback being set on the property owner (scene, shape *etc*).

In binding two properties in value, the code is actually simpler:

```
TextArea shipping = new TextArea();
TextArea billing = new TextArea();
billing.textProperty().bindBidirectional(shipping.textProperty());
```

But when a property whose bound value has to be computed based on the value of another property, things can get ridiculous because of the "straight jacket" of property interfaces involved (which do not admit normal operations, and must be operated upon through the utility class `javafx.beans.binding.Bindings` or similar). Compare two examples to see that binding not always gives an advantage:

- MasterSlaveWithoutBinding.java
- MasterSlaveWithBinding.java

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient (not covered)
- Hortsmann's Java SE 8 for the Really Impatient, Ch. 4.
- Oracle's JavaFX Tutorial