# JavaFX

Rui S. Moreira

Some useful links:
https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm

## Graphical User Interfaces

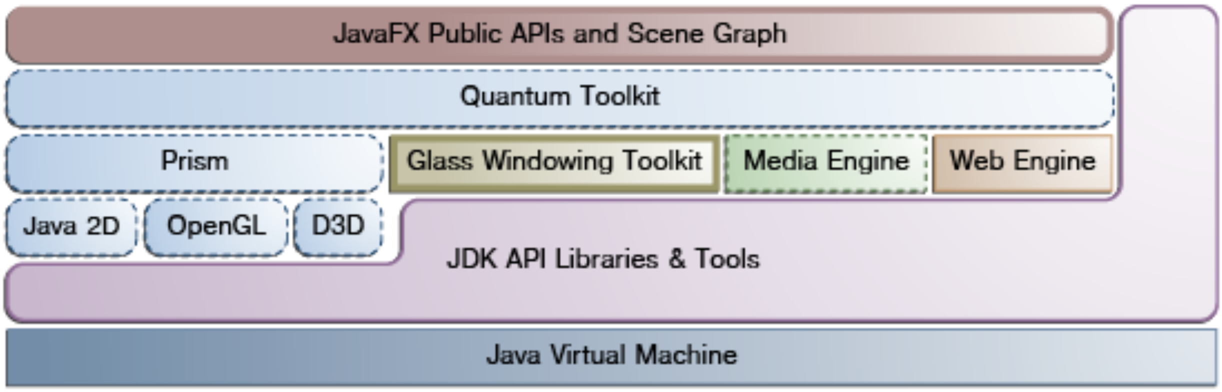
- Java provides a good integration with interactive web-based applications - Applets, JSP, Servlets
- The Java Foundation Classes (JFC) include packages (cf. **AWT** & **Swing** – J2SE) with built-in components and functionality for assembling GUIs
- More recently **JavaFX** is becoming the major ritch frontend dev framework chosen by the Java developers community

# JavaFX

- Set of Java libraries for rich client apps
- Consistent behavior across platforms
- Development separated into:
  - Declarative definition of GUI layout (XML)
  - Programmatic definition of GUI behavior (Java)



# JavaFX Architecture




**Scene Graph**:
- **Starting** point for constructing a JavaFX application;
- Hierarchical **tree** of **nodes** that represent all GUI visual elements;
- **Handle input** and can be rendered;
- JavaFX scene **graph** (unlike Swing and AWT) includes graphics primitives (e.g. draw rectangles, text), in addition to controls, layout containers, images and media.

# Scene Graph

- **Node** (an element in a **Scene**):
  - has an ID, style class, and bounding volume
  - has a single parent and zero or more children (except root)
  - Can have:
    - Effects (e.g. blurs and shadows)
    - Opacity
    - Transforms
    - Event handlers (e.g. mouse, key and input method)
    - An application-specific state



# javafx.scene API

- Allows managing:
  - **Nodes**:
    - 2-D and 3-D shapes, images, media, embedded web browser, text, UI controls, charts, groups, and containers
  - **State**:
    - transforms (positioning and orientation of nodes), visual effects, among other visual state
  - **Effects**:
    - change appearance of scene graph nodes, e.g., blurs, shadows, and color adjustments

# Graphics System

- Layer beneath JavaFX scene graph
  - **Prism**
    - Processes render jobs (rendering/rasterization of JavaFX scenes) on Hw or Sw
      - e.g. Direct X (Win), OpenGL (Mac/Linux), Hw GPUs, etc.
  - **Quantum Toolkit**
    - Ties Prism and Glass Windowing Toolkit together;
    - Manages threading rules related to rendering vs event handling



# Glass Windowing Toolkit

- Provide native operating services
  - e.g. managing windows, timers, and surfaces
- Platform-dependent layer connecting JavaFX to native OS
  - Manage event queue (using native OS event queue to schedule thread usage):
    - JavaFX application thread
    - Prism render thread
    - Media thread

# Media and Images (javafx.scene.media)

- Support for visual and audio media
  - e.g. MP3, AIFF and WAV audio and FLV video
  - Media object represents a media file
  - MediaPlayer plays a media file
  - MediaView is a node to display media



# Web Component

- JavaFX UI control
  - WebKit-based (open source web browser engine) supporting HTML5, CSS, JavaScript, DOM and SVG
  - Provides Web viewer and browsing functionality:
    - **WebEngine**:
      - provides basic web page browsing capability
    - **WebView** (extension of Node class):
      - encapsulates a WebEngine object
      - incorporates HTML content into an apps scene
      - provides fields and methods to apply effects and transformations
  - Integrates with JavaScript functions

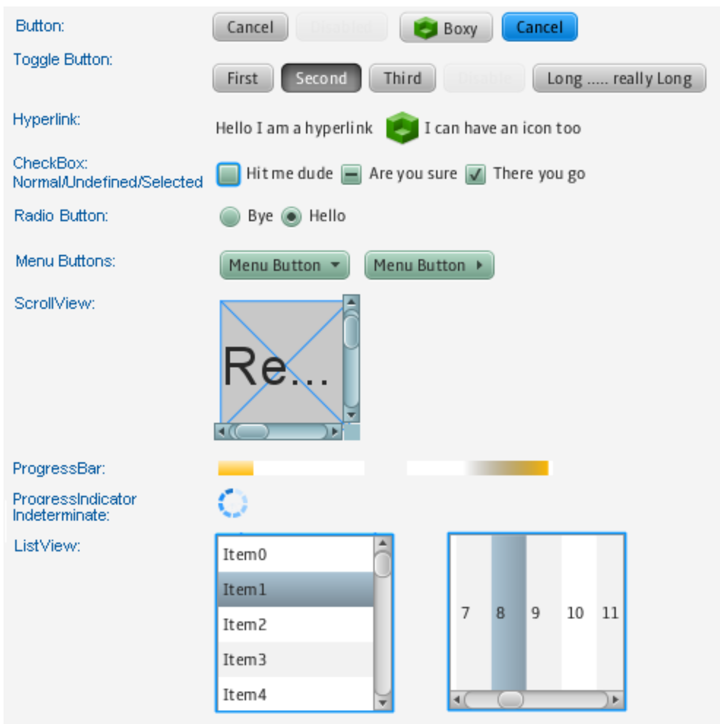
# JavaFX Cascading Style Sheets (CSS)

- Apply customized styling to GUI scene nodes without changing any of app source code
- Asynchronously adapt app appearance
- JavaFX property names are prefixed with "-fx-" vendor extension



# UI Controls

- Built by using nodes in the **scene graph**
- Portable across different platforms
- CSS allows theming and skinning of UI controls.



## Layout

- Layout containers or panes
  - Flexible and dynamic arrangements of UI controls within a scene graph
  - Includes **container** classes (can be nested):
    - BorderPane (top, bottom, right, left, or center region)
    - HBox (horizontally single row)
    - VBox (vertically single column)
    - StackPane (back-to-front single stack)
    - GridPane (grid of rows and columns)
    - FlowPane (horizontal or vertical “flow” with wrap boundaries)
    - TilePane (uniformly sized layout cells or tiles)
    - AnchorPane (anchor nodes to top, bottom, left or center)



## 2-D and 3-D Transformations

- Each **Scene** node can be **transformed**:
  - translate (move node along x, y, z planes)
  - scale (resize node according scaling factor)
  - shear (rotate x-axis or y-axis… coordinates of node shifted by specified multipliers)
  - rotate (rotate a node about a pivot point of scene)
  - affine (linear mapping from 2-D/3-D coordinates to other 2-D/3-D coordinates)
    - used with Translate, Scale, Rotate, or Shear transform

## Visual Effects

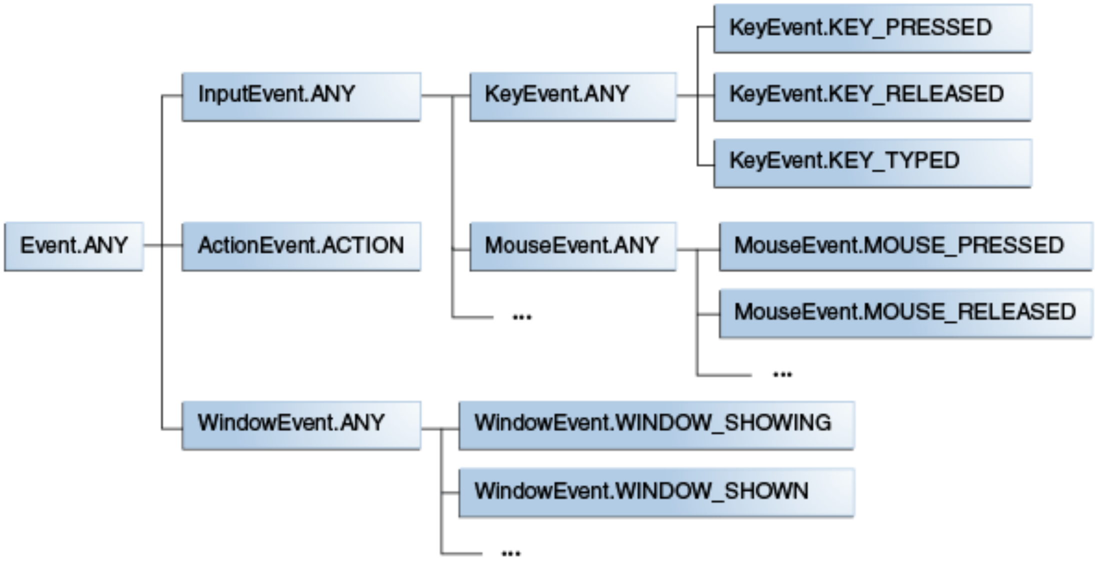
- Image pixel-based effects
  - Take set of nodes in scene graph, render it as image, and apply the specified effects
- Visual effects classes:
  - Drop Shadow (renders shadow behind content)
  - Reflection (renders reflected version of the content below actual content)
  - Lighting (simulates a light source shining on a given content)



## Processing Events

- Events notify app of **actions** taken by **user**
  - e.g. `DragEvent`, `KeyEvent`, `MouseEvent`, `ScrollEvent`
- Enable app to respond to event
  - Capturing an event from source (e.g. button)
  - Routing the event to its target (handling)
- Event **Target**
  - instance of class implementing `EventTarget` interface
  - implementation of `buildEventDispatchChain`
    - creates event **dispatch chain** that event must travel to reach **target**

# Event Types Hierarchy






# Event Targets

- **`Window`**, **`Scene`** and **`Node`** classes
  - implement `EventTarget` interface
  - Subclasses of these classes inherit implementation
    - Hence, such controls are event targets through inheritance
  - Controls not inheriting these classes must implement `EventTarget`
    - `MenuBar` control is a target through inheritance
    - `MenuItem` element (of menu bar) must implement `EventTarget` to receive events
- Most elements in GUI have their dispatch chain defined
  - programmers focus on responding to events
  - not concerned with creating event dispatch chain

# Event Delivery Process

- The event delivery process contains the following steps:

1. Target selection
2. Route construction
3. Event capturing (descendant chain)
4. Event bubbling (ascendant chain)



# Target Selection

- The target node is selected according internal rules:
  - **Key events**: target is node that has focus
  - **Mouse events**: target is node under cursor
  - **Synthesized mouse events**: touch point is the location of cursor
  - **Continuous gesture events** (touch screen): target is node at center point of all touches at the beginning of the gesture
  - **Indirect gesture events** (e.g. trackpad): target is node at location of cursor
  - **Swipe events** (touch screen): target is node at center of path of all fingers
  - **Indirect swipe events**: target is node at the location of cursor
  - **Touch events**: default target for each touch point is the node at first press

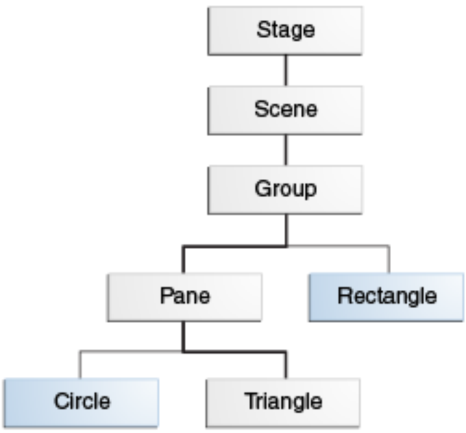
  - Different targets can be specified using ungrab() or grab(node) methods for a touch point in an event filter or event handler
  - When more than one node is located at cursor/touch then topmost node is considered the target

# Target Selection

- If click on triangle (on top of gray rectangle) then **target** is the triangle
  - When mouse button pressed and target is selected then
  - all subsequent mouse events are delivered to same target until button is released
- Similarly for gesture events...
  - from start of gesture to its completion, the events are delivered to same target



# Route Construction



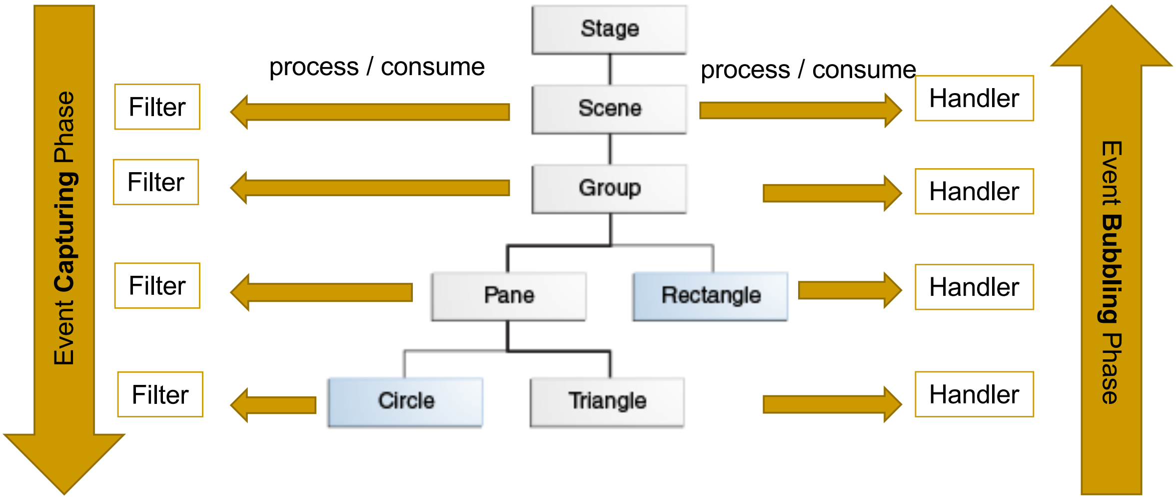

- Initial event route determined by event **dispatch chain**
  - created by implementation of `buildEventDispatchChain()` on selected event target
    - e.g. when user clicks triangle then initial route crosses gray nodes (see figure)
- When `Scene` graph node is selected as event target then
  - initial event route set in default impl of Node `buildEventDispatchChain()` sets path from stage to itself
- Route can be modified by event **filters** and event **handlers** along the route
  - each node may **process** and **consume** event

# Event Routes (Dispatching Chain)

Dispatch Chain has 2 phases: **Capturing** and **Bubbling**

**Filters:** receive events on descendant capturing phase

**Handlers:** receive events on ascendant bubbling phase




NB: each Node can have multiple attached Filters and/or Handlers



# Event Capturing Phase - Filters

- Event is dispatched by **root node** and passed **down** the event dispatch chain to **target node**
  - In **capturing phase** event travels from **root** to **target** (see previous figure)
    - e.g. event travels from `Stage` to `Triangle` nodes
- When node in chain has **registered filter** for type of event occurred
  - that filter is called/executed
  - after completion, event passed down chain (next node)
  - When no filter registered, event is passed to next node until reaches event target (that eventually processes event)

# Event Bubbling Phase - Handlers

- After reaching event target and all registered filters have processed the **event** it **returns** along **reverse path**
  - In **bubbling phase** the event travels from **target** to **root**
    - e.g. event travels from `Triangle` to `Stage` nodes
- When a node in chain has **registered handler** for the type of event encountered
  - that handler is called/executed
  - after completion, event returned up chain (next node)
  - when no handler registered, event is returned to next node up until root (that eventually processes event)



# Event Handling

- Both event **filters** and **handlers** implement the `EventHandler` interface:
  - Event **Filters**:
    - executed during event **capturing phase** (**filters** registered for **typed event** are executed when event passes **down** chain)
  - Event **Handlers**:
    - executed during event **bubbling phase** (**handlers** registered for **typed event** are executed when event passes **up** chain)

# Event Filters

- Registering a **filter** for a **parent** node
  - can provide **common event processing** for multiple child nodes
  - can **consume** event to **prevent child node** from **receiving** event

- Register **more than one filter** for each node
  - order of filters is based on hierarchy of event types.
    - filters for **specific event types** are executed before filters for **generic event types**
    - e.g. filter for `MouseEvent.MOUSE_PRESSED` called before filter for `InputEvent.ANY` event
    - order in which two filters at same level are executed is not specified



# Event Handlers

- Registering **handler** for a **parent** node
  - can act on the event **after** a **child** node processes it
  - can provide **common event processing** for multiple child nodes

- Register **more than one handler** for each node
  - order of handlers based on hierarchy of event types
    - handlers for specific event types executed before handlers for generic event types
    - e.g. handler for `KeyEvent.KEY_TYPED` called before handler for `InputEvent.ANY` event.
    - order in which 2 handlers at same level are executed is not specified (except registered by **Convenience Methods** - **executed last**)

# Consuming Events

- Events can be **consumed** by filter/handler at any point in the event dispatch chain by calling `evt.consume()` method
  - `consume` method signals that **processing** of event **ended**
  - hence, **traversal** of event dispatch chain also **ends**
- **Consuming** an event on filter/handler **prevents further processing** down/up dispatch chain
- When node consuming event has more than one registered filter/handler for the event
  - All peer filters or handlers are executed
  - e.g. suppose `Pane` node has 2 filters (`KeyEvent.KEY_PRESSED` and `InputEvent.ANY`) which consume the event
    - **Both** filters will be **executed and consume** the event
    - But `Triangle` node will not receive event (no further propagated down chain)
- NB: default handlers for UI controls typically consume input events



| User Action | Event Type | Generating Class |
|---|---|---|
| Key pressed | KeyEvent | Node, Scene |
| Mouse moved or button pressed | MouseEvent | Node, Scene |
| Full mouse press-drag-release | MouseDragEvent | Node, Scene |
| Set, change, remove or commit input method (e.g. for foreign language) | InputMethodEvent | Node, Scene |
| Drag and drop action performed | DragEvent | Node, Scene |
| Object scrolled | ScrollEvent | Node, Scene |
| Rotation gesture performed on object | RotateEvent | Node, Scene |
| Swipe gesture performed on object | SwipeEvent | Node, Scene |
| Object touched | TouchEvent | Node, Scene |
| Zoom gesture performed on object | ZoomEvent | Node, Scene |
| Context menu requested | ContextMenuEvent | Node, Scene |
| Button pressed, combo box shown/hidden, menu item selected | ActionEvent | ButtonBase, ComboBoxBase, ContextMenu,MenuItem, TextField |
| Item edition in list, table or tree | ListView.EditEvent TableColumn.CellEditEvent TreeView.EditEvent | ListView TableColumn TreeView |
| Media player error | MediaErrorEvent | MediaView |
| Menu either shown or hidden | Event | Menu |
| Popup window hidden | Event | PopupWindow |
| Tab selected or closed | Event | Tab |
| Window closed, shown or hidden | WindowEvent | Window |

# Convenience Methods - Handlers

- Create and register event **handlers** to catch/handle
  - mouse, keyboard, action, drag-and-drop, window events, etc.

- Some JavaFX classes define event **handler properties**
  - **setter methods** for event handler properties are **convenience** methods for registering event handlers
  - setting an event handler property to a user-defined handler automatically registers it to receive the corresponding event



# Using Convenience Methods

- Most convenience methods defined in `Node` class
  - Available to all of its subclasses
- Other classes also contain convenience methods
  - `setOnEvent-type(EventHandler<? super event-class> value)`
    - `Event-type` is type of event that handler processes, e.g.
      - **`setOnKeyTyped`**`()` for `KEY_TYPED` events
      - **`setOnMouseClicked`**`()` for `MOUSE_CLICKED` events
    - `Event-class` is the class that defines the event type, e.g.
      - `KeyEvent` for events related to keyboard input
      - `MouseEvent` for events related to mouse input
    - `<? super event-class>` generic of `event-class` (or its superclass) parameterizing the event handler, e.g.
      - `InputEvent` could parameterize both keyboard or mouse event-handlers

# Convenience for Key and Action Events

- Registering an event handler for key-typed events
  - Generated when key is pressed and released:

```
txt.setOnKeyTyped(EventHandler<? super KeyEvent> value)
```

- e.g. create and register **event handler** for action-typed event
  - defining handler as **anonymous class** within call to convenience method
  - event handler must implement `handle()` for processing event

```
btn.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                System.out.println(”btn pressed");
            }
        });
```



# Convenience for Mouse Events

```
Circle c = new Circle(radius, Color.RED);

c.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse entered");
    }
});

c.setOnMouseExited(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse exited");
    }
});

c.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse pressed");
    }
});
```

# Convenience for Keyboard Events

```
TextField txt = (new TextField()).setPromptText("Write here");

txt.setOnKeyPressed(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
        System.out.println("Key Pressed: " + ke.getText());
    }
});

txt.setOnKeyReleased(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
        System.out.println("Key Released: " + ke.getText());
    }
});


// Remove event handler registered by convenience method (pass null)
txt.setOnKeyPressed(null);
txt.setOnKeyReleased(null);
```



# Working with Event filters

- **Filters** used to process events generated by
  - keyboard actions, mouse actions, scroll actions, etc.
- **Single filter** can be used for
  - more than one node
  - more than one event type
- Event **filters** enables
  - parent node to provide **common processing for its child** nodes
  - **intercept** event and **prevent child nodes** from acting on event

# Register and Remove Event Filter

- An event **filter** implements `EventHandler` interface
  - code `handle()` executed when event received by node
  - Register filter with `addEventFilter()` on a node, e.g.

```
//Register event filter for single node event type
node.addEventFilter(MouseEvent.MOUSE_CLICKED,
    new EventHandler<MouseEvent>(){
        public void handle(MouseEvent e) { ... };
    });
OR using lambda function...
node.addEventFilter(MouseEvent.MOUSE_CLICKED,
    (MouseEvent e)->{ ... }
    );
```



# Register Filter

```
// Define an event filter
EventHandler filter = new EventHandler<InputEvent>() {
    public void handle(InputEvent evt) {
        System.out.println("Filtering out event " + evt.getEventType());
        evt.consume();
    }
};

// Register same filter for 2 different nodes
node1.addEventFilter(MouseEvent.MOUSE_PRESSED, filter);
node2.addEventFilter(MouseEvent.MOUSE_PRESSED, filter);
// Register filter for another event type
node1.addEventFilter(KeyEvent.KEY_PRESSED, filter);

/* NB: an event filter defined for one type of event can also be used for
any subtypes of that event */
// Remove an event filter
node1.removeEventFilter(MouseEvent.MOUSE_PRESSED, filter);
```

# Using Event Filters

- Event **filters** on a parent or branch node of event dispatch chain:
  - called during event **capturing phase** of **event handling**
  - filter performs actions such as
    - **overriding** event response
    - **blocking** an event from reaching its child destination



# Working with Event Handlers

- **Handlers** used to process events generated by
  - keyboard actions, mouse actions, scroll actions, etc.

- **Single handler** can be used for
  - more than one node
  - more than one event type

- Event **handler** enables
  - child node **consume** which **prevents** up chain (parent) processing
  - provide parent **common** event **processing** for multiple child nodes

# Register and Remove Event Handler

- Event handler implements `EventHandler` interface
  - Code handle() executed when event received by node
  - Registered handler with `addEventHandler()` on a node, e.g.

```
// Register event handler for node and event type
node.addEventHandler(DragEvent.DRAG_ENTERED,
    new EventHandler<DragEvent>() {
        public void handle(DragEvent e) { ... };
    });
OR using lambda fubction...
node.addEventHandler(DragEvent.DRAG_ENTERED,
    (DragEvent e)->{ ... }
    );
```



# Register Handler

```
// Define an event handler
EventHandler handler = new EventHandler(<InputEvent>() {
    public void handle(InputEvent evt) {
        System.out.println("Handling event " + evt.getEventType());
        evt.consume();
    }
};

// Register same handler for 2 different nodes
node1.addEventHandler(DragEvent.DRAG_EXITED, handler);
node2.addEventHandler(DragEvent.DRAG_EXITED, handler);
// Register handler for another event type
node1.addEventHandler(MouseEvent.MOUSE_DRAGGED, handler);

/* NB: an event handler defined for one type of event can also be used for
any subtypes of that event */
// Remove an event handler
node1.removeEventHandler(DragEvent.DRAG_EXITED, handler);
```

## Using Event Handlers

- Handlers on leaf or branch node of event dispatch chain
  - called during event **bubbling** phase of event handling
  - handler performs actions such as
    - defining **default response** on a parent for all child nodes
    - **blocking** an event from reaching its parent (consumed before)



## Events from Touch-Enabled Devices

- Also possible to process events generated by different types of gestures or touches
  - Recognized by gesture-enabled or touch-enabled devices
    - e.g. touch events, zoom events, rotate events and swipe events
- Type of event is determined by touch or gesture types
- Touch and gesture events are processed as other events
- Convenience registering methods also available

# JavaFX Properties

- JavaFX defines APIs and naming conventions for several attributes of GUI Components (similarly to JavaBeans)
  - Such attributes are known as **Properties**, which allow any scene to update automatically whenever the GUI component's data changes
  - Properties are **observable** objects that may be writeable/read-only
    - Properties typically extend interfaces: `ReadOnlyProperty<T>` and/or `WriteableValue<T>`
  - There are many types of Property objects:
    - `StringProperty`, `SimpleDoubleProperty`, `ReadOnlyDoubleProperty`, etc.

- Each JavaFX **property** wraps a inner Java object (e.g. `String`, `Double`, etc.) and adds functionality for **listening** and **binding**
  - The `StackPane` and `TextField` components have properties, e.g.
    - `StackPane` has `width` and `height` properties of type `ReadOnlyDoubleProperty`
    - `TextField` has `text` property of type `StringProperty`



# JavaFX Properties & Observable

- Possible to get a `Property` through specific methods
  - Given `StackPane rootPane` it is possible to get `withProperty()`

```
ReadOnlyDoubleProperty widthProp = rootPane.widthProperty();
```

  - Given `TextField txt` it is possible to get `textProperty()`

```
StringProperty txtProp = txt.textProperty();
```

- JavaFX properties have methods to facilitate:
  - **Listening** for changes to the property's value
  - Linking properties together (**binding**) properties so that they update automatically

- Each **Property** implements two interfaces:
  - The `Observable` interface allows adding a **InvalidationListener** that fires when the property invalidates (marked potentially changed without forcing recalculation)
  - The `ObservableValue<Number>` interface allows adding a **ChangeListener**, which fires when property actually changed

# JavaFX Properties & Observable

- Hence, the `addListener()` method allows adding either an `InvalidationListener` or `ChangeListener`, e.g.
  - Add `ChangeListener` **anonymous** class object to receive changes from `widthProp` property

```
widthProp.addListener( new ChangeListener(){
 @Override
 public void changed(ObservableValue<? extends Number> observableValue,
                     Number oldV, Number newV) {
    sout("widthProperty changed ” + oldV.doubleValue() + "-” +
         newV.doubleValue());
 }
});
```

  - Since **`ObservableValue`** is a **functional interface**, we can use a **lambda** function instead (since Java 8)

```
widthProp.addListener( (observableValue, oldV, newV) -> {
   sout("widthProperty changed ” + oldV.doubleValue() + "-” +
         newV.doubleValue());
});
```



# JavaFX Properties & Bindings

- JavaFX provides also functionalities for **binding** values through `Bindings` objects
  - A `Bindings` enforces a link between 2+ objects/properties
  - A relationship in which one object is dependent on 1+ other objects
  - An `Observable` object/event can be used to update the value of another object
  - Bindings can be unidirectional/bidirectional
- Bindings can be created either from properties with `High-Level Fluent API`, via `Bindings` utility class or `Low-Level API`
  - `Fluent API` (exposes methods on various **dependency** objects)
  - `Bindings API` (`Bindings` class provides **static factory methods**)
  - `Low-Level API` (provides advanced binding classes)

# JavaFX Properties & Bindings

- The Fluent API relies on the creation of `Expression` objects, which are similar to properties (with observable values)
  - Properties can also be bound to expressions, hence the output of any manipulation can be used to update the bound property
  - Being observable and depending on an object for a value, allows chaining…

e.g. create chains of **strings** to concatenate them together:

```
StringProperty sourceProperty = new SimpleStringProperty(”One”);
StringExpression expression = sourceProperty.concat(", Two”);
StringProperty targetProperty = new SimpleStringProperty("");
//Bind targetProperty to expression, which concats the two strings
targetProperty.bind(expression);
System.out.println(targetProperty.get()); //Output: One, Two
```

e.g. create chains of **numbers**, e.g., convert degrees to radians:

```
DoubleProperty degrees = new SimpleDoubleProperty(180);
DoubleProperty radians = new SimpleDoubleProperty();
//Create two observable values to obtain radians from degrees
radians.bind(degrees.multiply(Math.PI).divide(180));
```



# JavaFX Properties & Bindings

- It is also possible to establish bindings between different types of base-type properties, e.g. `String` and `Double`-based properties

- Consider a calculator GUI where needed to automatically convert the screen `String` content (`TextField`) into a `Double` value:
  - Create a **Binding** between the `StringProperty screenTxtProp` and the `SimpleDoubleProperty displayValue` object

```
//Binding between txtProp -> displayValue, i.e. whenever the
//txtProp String content changes it will update Double displayValue
txtProperty.bind(Bindings.format("%.1f", displayValue));
```

- Also possible to change content of associated JavaFX components properties:
  - **Unidirectional** binding between a `Label` and a `TextField`

```
toLable.textProperty().bind(fromText.textProperty());
```

  - **Bidirectional** binding between two `TextFields`

```
oneText.textProperty().bindBidirectional(anotherText.textProperty());
```

# JavaFX Properties & Bindings

- The JavaFX `Bindings API` provides the `Bindings` utility class containing a panoply of methods for property binding
  - Create bindings between observable objects of various types
  - Combining properties with values, such a Strings and numbers
- There are 3 main binding areas:
  - Operations on **values**
    - Mathematics (+, – ÷, ×)
    - Selecting maximum or minimum
    - Value comparison (=, !=, <, >, <=, >=)
    - String formatting
  - Operations on **collections**
    - Binding two collections (lists, maps, sets)
    - Binding values to objects at a certain position in a collection
    - Binding to collection size
    - Whether a collection is empty
  - **Others**
    - Multiple-object bindings
    - Boolean operators (and, not, or)
    - Selecting values



# JavaFX Properties & Bindings

- The JavaFX `Bindings API` operation on **values**:
  - Support for mathematical operations: +, -, * and /
  - Provides methods for use of these with `float`, `double`, `integer` and `long` values, as well as between `ObservableNumberValue` objects (e.g. `DoubleProperty`)

```
//Define two properties, on which, changes will trigger calculation update
IntegerProperty cost = new SimpleIntegerProperty(15);
DoubleProperty multiplier = new SimpleDoubleProperty(25);
//and a non-property value (which will not trigger any changes)
double flatRate = 4;
//Create binding to calculate totalCost = cost * multiplier + flatRate
DoubleBinding totalCost =
      Bindings.createDoubleBinding(
        () -> cost.get() * multiplier.get() + flatRate,
        cost,
        multiplier
      );
```