COMP6700/2140 Widgets, Events and Listeners

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

1. Events and Callbacks
2. Programming callbacks:
   - Inner classes and their use in GUI
   - Lambda-expressions
3. Event-propagation effects
4. Event-filtering

User Interaction with a GUI Program

The interaction is carried out via a (new) flow of control mechanism — events:

- Events are *asynchronous*
- They alter the state of execution environment
- They are detected by widgets (active control elements, provided by API)
- Widgets are programmed to respond to an event occurrence, these responses are *callbacks*
- Events can also be caused by the program itself (not by a user):
  - Timeout (or other type of time event)
  - A callback execution completion (*ie*, closing a progress bar)
- Some callbacks are already defined by API
  - window events (resize, expose *etc*)

A "Hello World" of GUI events program is MouseEvents.java (an older, *Swing*-based example is MouseSpy.java). The structure is standard:

1. A `scene` and a few `shape` objects on it are created, all can detect events (mouse events in the example)
2. Each object ("widget") is programmed to respond to one of the mouse events in a particular way (by executing `setOnMouseXXX(..)` method on the object)
3. The scene (frame) and the event listener class (*MouseListener*) are combined into one in the simpler example `MouseSpy.java`.

Events and Event Properties

When a change occurs (user input *etc*), the application gets notified by an appropriate event. In JavaFX, *event* is an instance of `javafx.event.Event` or its subclass — *DragEvent, KeyEvent, MouseEvent, ScrollEvent* and others. To define custom event types, extend the *Event* class.

Event has properties:

- **Event type** — an instance of *EventType* class, it's determined by the "physical" source of the event, *eg* `KeyEvent.KEY_PRESSED`, `MouseEvent.MOUSE_RELEASED`, `ActionEvent.ACTION`. The event type form its own hierarchy rooted in `Event.ANY`. The "real" events are the leaves in this tree, while the nodes correspond to particular widgets.
- **Source** — an origin of the event, with respect to the location of the event in the event dispatch chain. The source changes as the event is passed along the chain.
- **Target** — a node on which the action occurred and the end node in the event dispatch chain. A target is an instance of any class which implements *EventTarget* interface. The target does not change, however if an event filter consumes the event during the event capturing phase, the target will not receive the event.

Subclasses of *Event* have additional properties: *MouseEvent* includes information about which button pushed, how many times, and the mouse position at that moment.

Once an event is detected by some node, it travels through an *event dispatch chain*, a sequence of nodes connected by their presence on the same scene graph. Standard JavaFX elements have their event dispatch chain already defined.

Event: Propagation, Filtering and Handlers

An event propagates through its event dispatch chain to the target where a programmed action will take place. The event delivery from source to target takes four stages:

1. **Target selection** — based on internal rules, *eg* for a key event, target is the node which has focus;
2. **Route construction** — initially, set by implementation of `EventTarget.buildEventDispatchChain()` method, but can be modified by *event filters* along the route process the event (sometimes event can be consumed before reaching its target); in the example SimpleShapesAndTransitions.java, the dispatch chain for the "mouse entered" event is `primaryStage --> scene --> root --> circle (c1)`.
3. **Event capturing** — when the event reaches its target (we ignore filters at the intermediary nodes of the chain which may consume the event) and is processed by a dedicated event handler (which executes its code):

```
c1.onMouseEnteredProperty().set(e -> {
        ft.stop();
        ft.playFromStart(); // fading the colour of circle
});
```

4. **Event bubbling** — after event is processed by its target, it travels back to the root, and all intermediary nodes, if they have registered event handlers, execute their code too (*Ponder: what effects can be realised in such scheme?*). To prevent event bubbling from a node up, the method `consume()` is called.

To learn (important!) details about the event delivery, study the JavaFX Tutorial chapter Events/Event Handlers.

Implementation of Event Handlers: Anonymous Inner Classes

To make an element (node, leaf) react to an event, one must set an event handler property to be an object which executes the handler code. Before Java 8, this required an anonymous inner class — a standard Java technique to pass around code to execute. Although this object may be an instance of any type that implements the `EventHandler` interface, in practice, it is usually an anonymous inner class:

```
final Scene scene = new Scene(root, 600, 450, Color.WHITE);
final Rectangle rect = new Rectangle(...);
// setting rect's properties
// setting rect to rotate once by the angle 45 deg, around the pivot (410,200)
rect.getTransforms().add(new Rotate(45, 410, 200));
root.getChildren().addAll(rect);
/* scene is set to react to pressing "R"-key event; rotation (22.5 deg around
 * rect's centre) is performed every time the key event is detected */
scene.onKeyPressedProperty().set(new EventHandler<KeyEvent>() {
    @Override
    public void handle(KeyEvent ke) {
        if (ke.getCode() == KeyCode.R)
            rect.getTransforms().add(new Rotate(22.5, 410, 200));
    }
});
```

An example SimpleShapesAndTransitions.java illustrates event handler use for fading and motion effects triggered by the user inputs.

Implementation of Event Handlers: λ-expressions

A modern and *better* approach to defining callbacks is to use λ-expressions or method references instead:

```
// 2D arrays of circles and FadeTransitions (fts) are declared beforehand
CircleBuilder cb = CircleBuilder.create().radius(20);// implicitly final
for (int i = 0; i < 5; i++)
   for (int j = 0; j < 5; j++) {
      Circle theCircle = cb.centerX(100 + 60*i).centerY(100 + 60*j).build();
      circles[i][j] = theCircle; // to avoid changing circles
      theCircle.setFill(Color.color(i*0.2,j*0.2,1.0));
      FadeTransition theTransition = ftb.build();
      theTransition.setNode(theCircle);
      fts[i][j] = theTransition; // to avoid changing fts (fade-transitions)
      theTransition.setCycleCount(1);
      circles[i][j].onMouseEnteredProperty().set(e -> theTransition.play());
      circles[i][j].onMouseExitedProperty().set( e -> theTransition.pause());
      circles[i][j].onMouseClickedProperty().set(e -> theCircle.setOpacity(1.0));
   }
```

An example BuilderAtWork.java.java would create 3 more bytecode classes if anonymous inner classes were used to implement callbacks. Even in a modest GUI applications there can be tens or even hundreds of callbacks.

Further Reading

- Hortsmann's Java SE 8 for the Really Impatient, Ch. 4.
- Oracle JavaFX: Handling Events