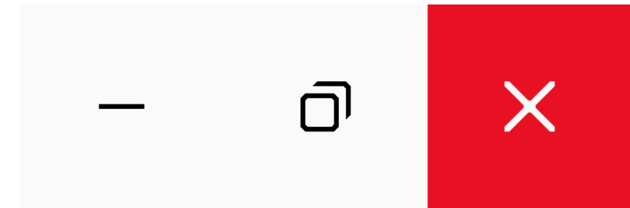
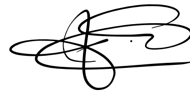
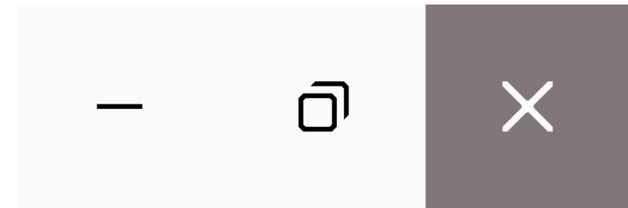


CS349 – User Interfaces

Events
Basic Event Handling
Event Dispatch



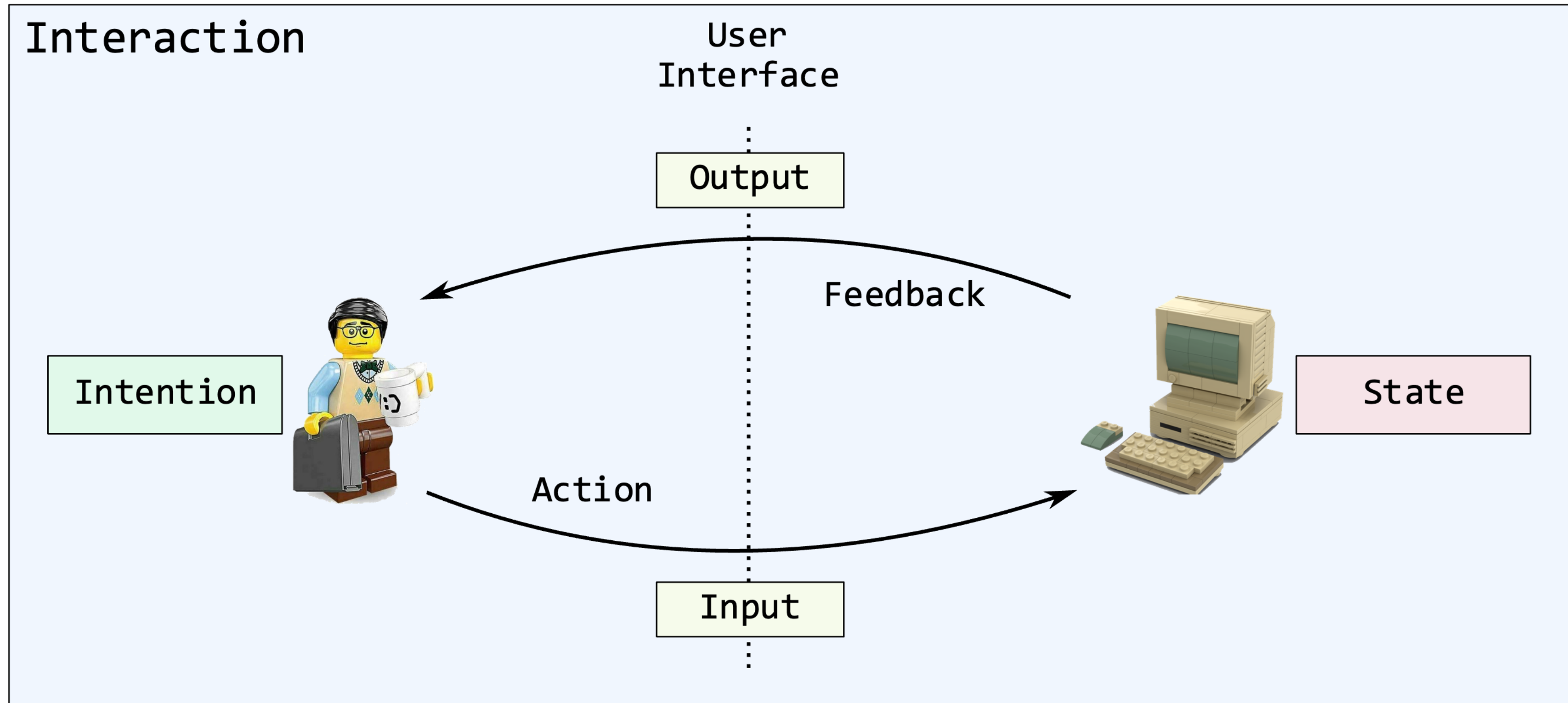
May 29



Events

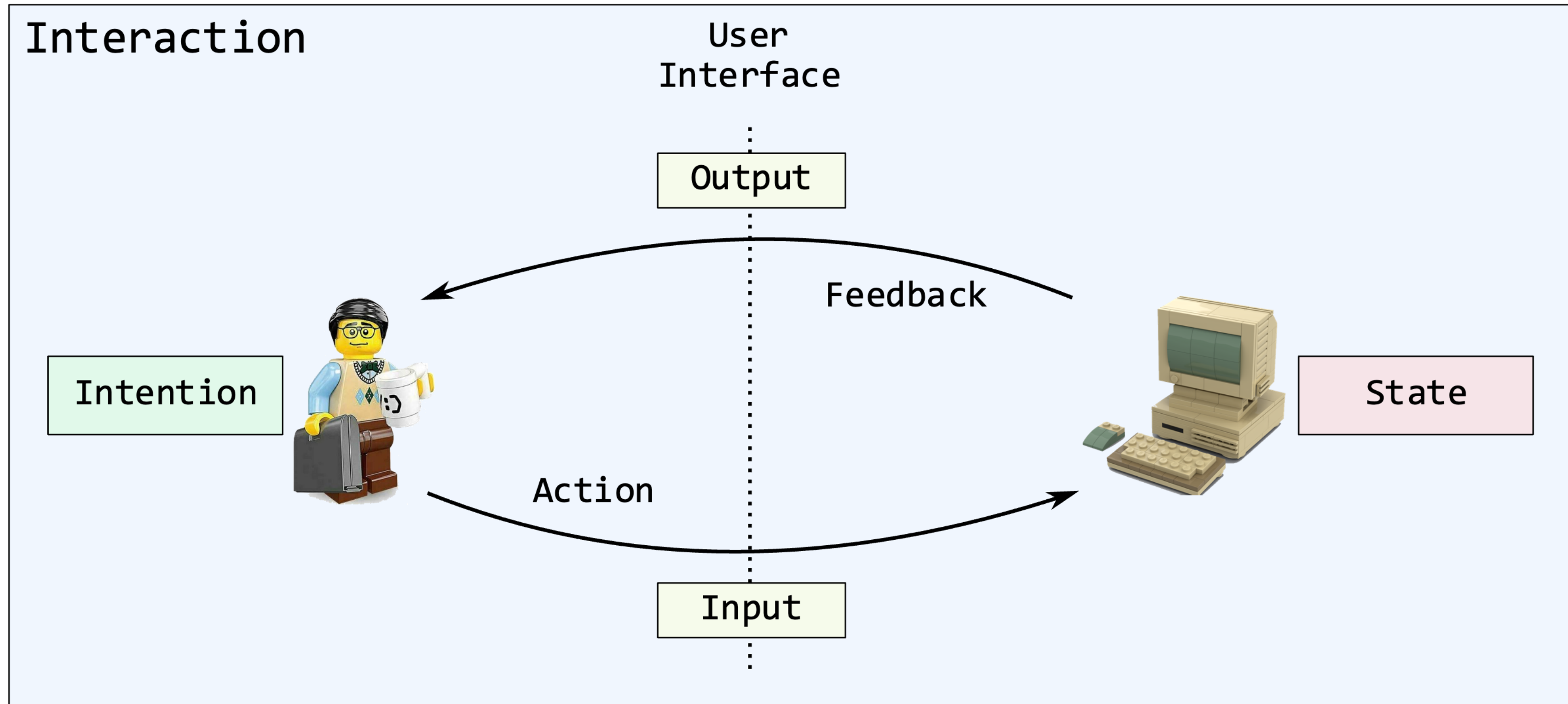
What are events? Why are they used?

Event-Driven Programming



In GUI, users expect immediate feedback by the system from their actions.

Event-Driven Programming



Event-driven programming is a programming paradigm that bases program execution flow on **events**. Events can represent user actions or other triggers in the system.

By using multiple threads and prioritizing certain events, we can create GUIs that retain a responsive feel to users.

What are events?

An event is a message to notify an application that something happened; a message of interest to an interface.

An efficient event-based system allows us to prioritize user-initiated actions, and carefully prioritize other work while still remaining responsive to a user.

Foreground Events

- Events initiated by the user.
- Created as a result of the user interacting with the user interface.
- Typically generated within the application.
- e.g. typing text, clicking a mouse button.

Background Events

- Events generated by the system.
- May be received and processed by the user interface.
- e.g. timer ticking, cloud data updating.

Types of Events

Events can be initiated by a user, e.g., as mouse input, or by the system, e.g., through a timer.

Types of events include:

- Property changes (text, checked, selected, value)
- Keyboard actions (key press, key release)
- Pointer actions (button press, button release, mouse move, mouse enter, mouse leave)
- Input focus changes (focus gained, focus lost)
- Window events (window resized, window minimized, window exited)
- Timer events (tick)

Show demos!

Events in JavaFX

Events are messages representing something of interest.

JavaFX has an event class (`javafx.event.Event`), with several predefined event subtypes.

All event classes include the following fields:

`EventType`	Type of the event that occurred, e.g., **`KEY_EVENT`**
`source`	node from where the event originated
`isConsumed`	Whether this event has been processed

All user interface classes also include:

`target`	node where the event should be delivered

Events may also include fields specific to their subtype; e.g., a mouse event will contain the mouse cursor coordinates, a key event will contain keypress information.

Event Subclasses

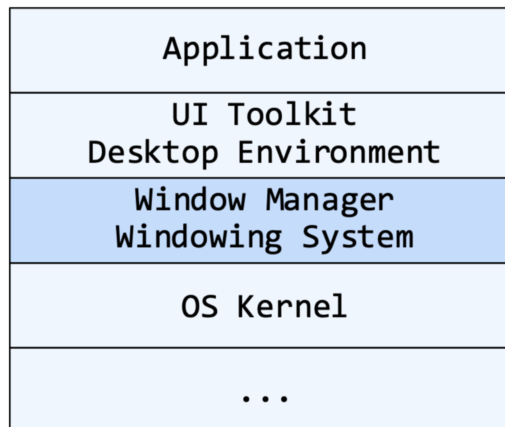
The following `Event` subclasses are commonly used:

- An **`ActionEvent`** occurs, for example, when a button is activated. This is typically used when there is only a single ”main” event generated by a class.
- A **`MouseEvent`** occurs when a mouse is used. This includes actions such as, clicking, pressing, releasing, moving, entering, and exiting.
- A **`DragEvent`** replaces mouse events during drag-and-drop gestures. This includes actions such as, drag entered, drag over, drag dropped, and drag exited.
- A **`KeyEvent`** indicates the key stroke occurred on a node. This includes actions such as, key pressed, key released, and key typed.
- A **`WindowEvent`** is related to window manipulation. This includes actions like window hiding and window showing. These can be background or foreground tasks (e.g. clicking on the X on a window will generate a WindowClosed event).

Event Propagation – System Event Loop

How do events get created? Typically, the Window Manager receives notification about a low-level system event, packages details into an event, and passes that to the appropriate UI toolkit. It performs these steps in order:

1. Collect event information from the underlying OS Kernel.
2. Store relevant information in an event structure, and store events in an event queue.
3. Dispatch events from the queue to the UI toolkit / Application.†



† This is typically the application window that triggered the event but could also be another window that is intercepting events.

Event Propagation – JVM Event Loop

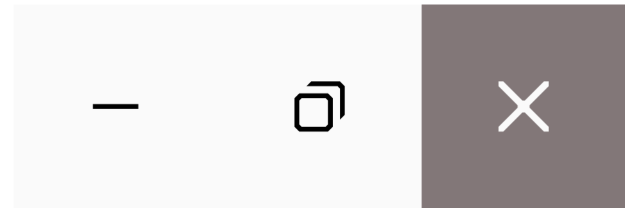
Kotlin/JVM applications modify this slightly -- they rely on the JVM to queue, manage, and dispatch events for each running application.

The JVM has an event-handling thread that performs these steps:

1. Manage an Application event queue where the Window Manager stores events intended for that application.
2. Pull events from this JVM event queue.
3. Format them as Kotlin events.
4. Dispatch them within the Application.

Event Propagation – Application Event Loop

An application might have its own event loop and secondary event queue that accumulate events until it is ready to handle them.



Basic Event Handling

What have we seen so far?

Basic Event Handlers

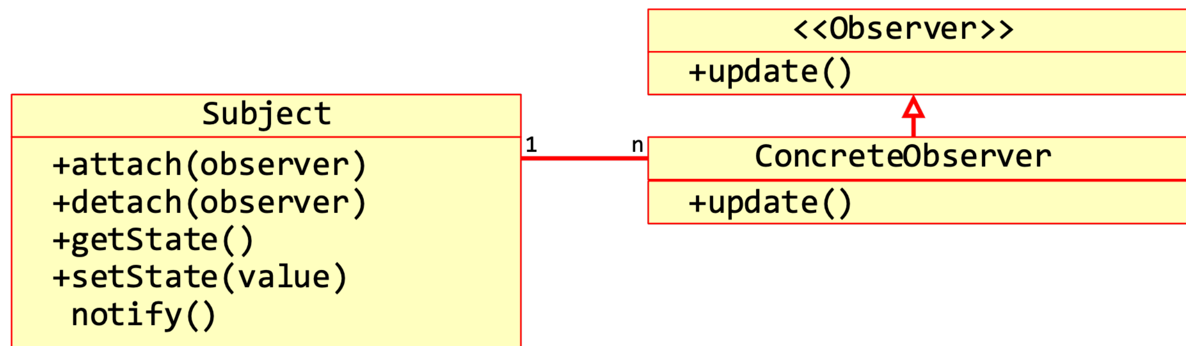
JavaFX defines interfaces for *specific event types* (or *device types*), e.g., `MouseEvent`, `KeyEvent`, `TouchEvent`, etc.

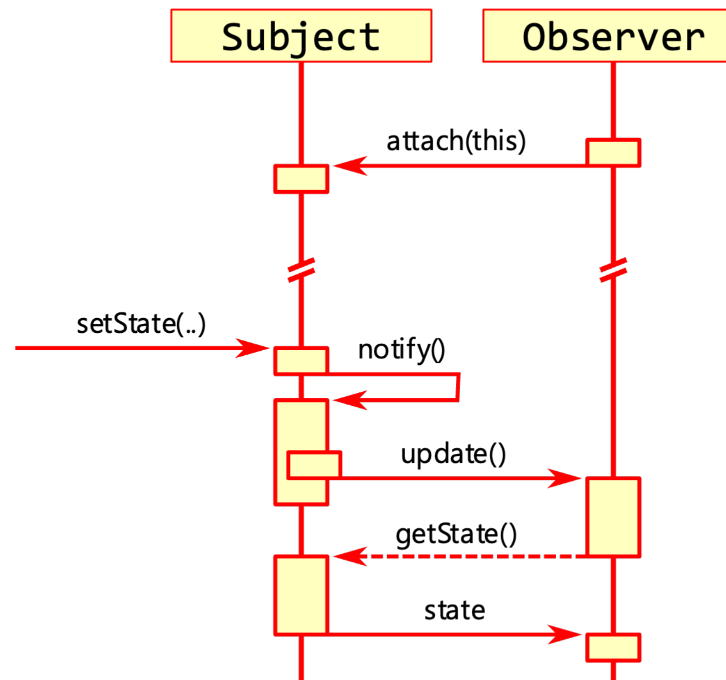
1. Register a listener function (e.g. *a lambda*) that can process each specific event type that we care about.
2. When an event is dispatched, the relevant listener function is called for that event.

This is approach follows the Observer-pattern:

Observer Pattern

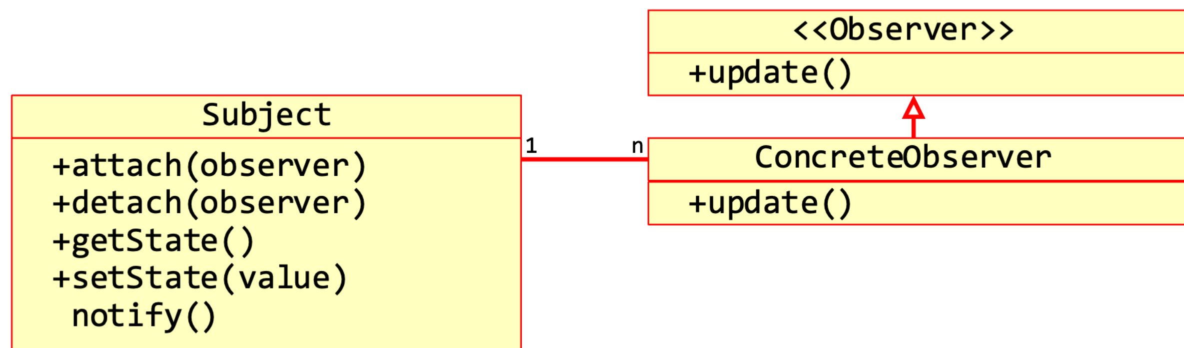
With the observer pattern, a subject maintains a list of observers, and notifies them of any state changes, usually by calling one of their methods.





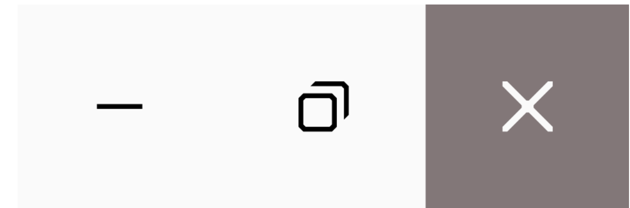
Basic Event Handlers

In JavaFX, observers are implemented as `EventHandlers`, and normally associated with the widget that generates the event.




```
// Pseudo-code: Associate this event handler code with this widget.
// When a mouse-click is generated, call the code (it is the event).

scene.onMouseClicked =                                  // subject / attach
    EventHandler { it ->                                // observer
        stage.title =
            "Click ${it.sceneX}/${it.sceneY}" } // get state from event
```



UI CS 349

Event Dispatch

How do events get delivered?

Event Dispatch in Java FX

The event dispatch process contains the following steps:

1. **Target selection** Which node should receive the event?
2. **Route construction** What is the path through the scene graph to a node?
3. **Event capturing** Traverse path downwards from Root to the node
4. **Event bubbling** Traverse path upwards from the node back to Root

Target Selection & Route Construction

Target Selection is determined by the type of Event:

- For **key events**, the target is the node that has focus
- For **mouse events**, the target is the node at the location of the cursor
- For **touch screen events**, target selection may be more complex, e.g., a **continuous gesture** (like pinch-to-zoom) might select the target node at the center point of all touches at gesture start, whereas a **swipe** (like swipe right) might select the target node at the center of the entire path of all fingers

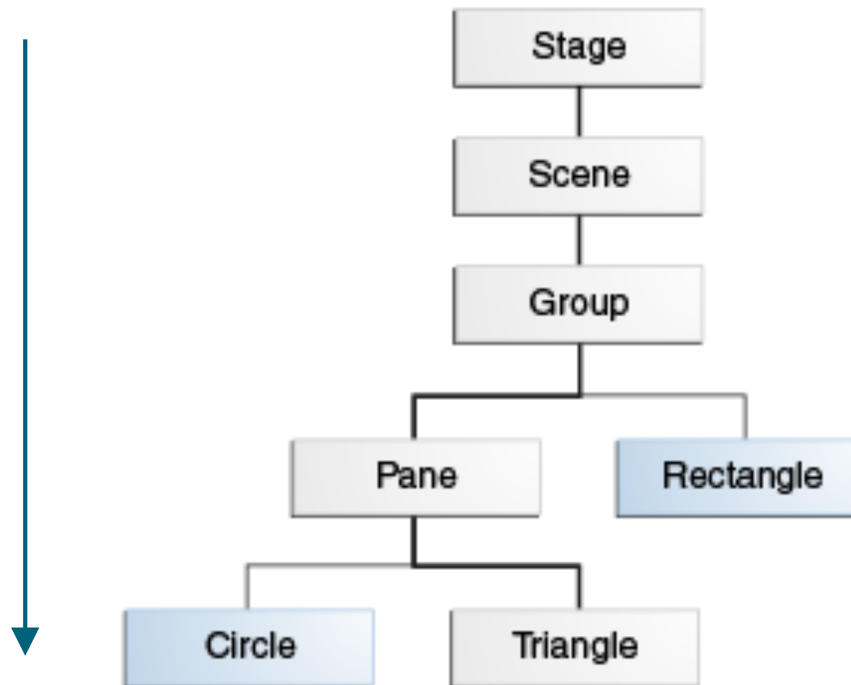
Route construction is the path to a particular Node through the tree:

- Stored as chain, using the Source and Target fields of the event.

Event Capturing and Bubbling

JavaFX supports both top-down and bottom-up processing. Events propagate from the Root to the Target (“Capture phase”), then back up to the Root (“Bubble phase”). Any Node in the path can intercept (“consume”) the Event, on either pass.

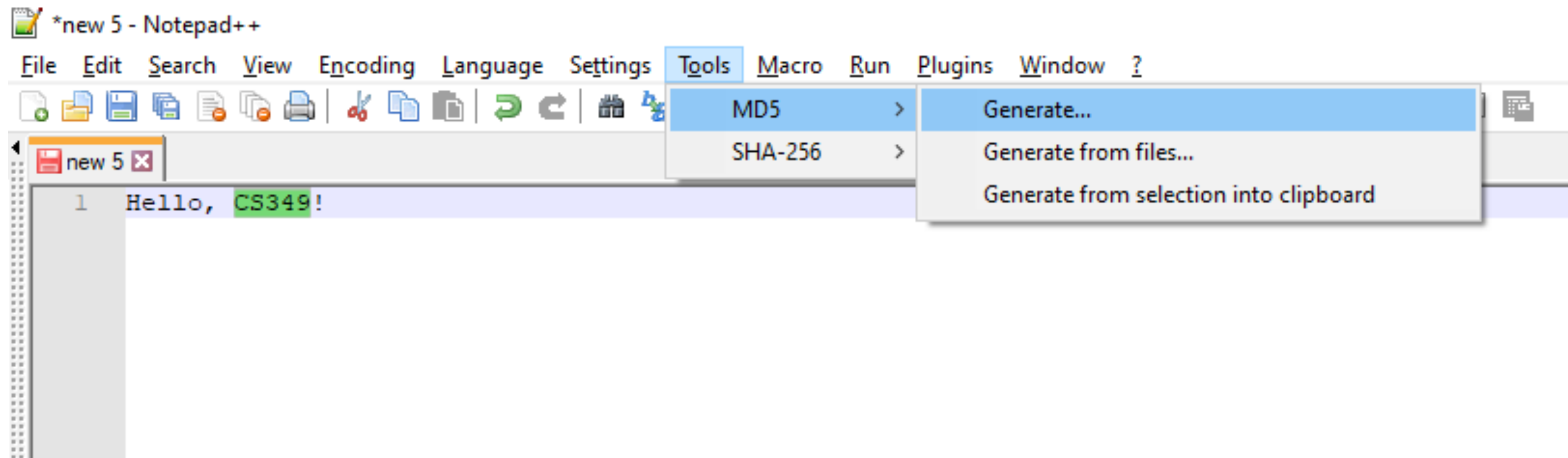
The **Capture phase** walks **down** the tree from the Stage (root) through each Node until it reaches the Triangle.



The **Bubble phase** walks **up** the tree from the Triangle, through each Node until it reaches the Stage (Root)

Positional Dispatch

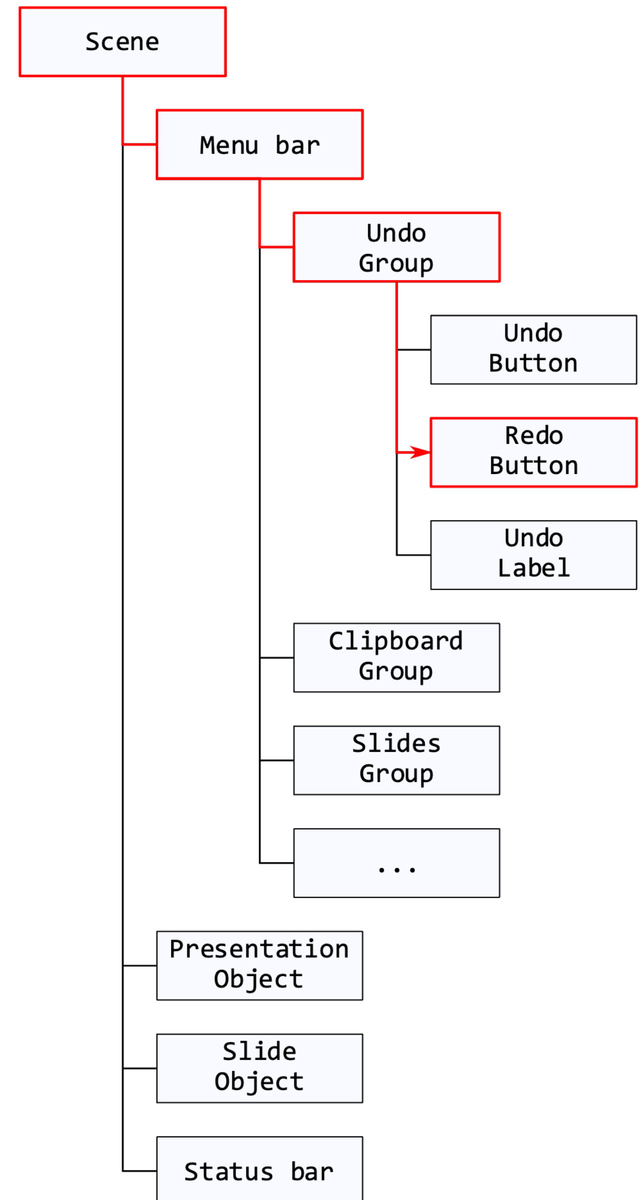
Sending events to the Node under the cursor is called **positional dispatch**.



Top-down Positional Dispatch (using Capture phase)

Event is dispatched to the root node of the scene graph first, and than travels through the scene graph to the target node. This means that the target node receives the event first.

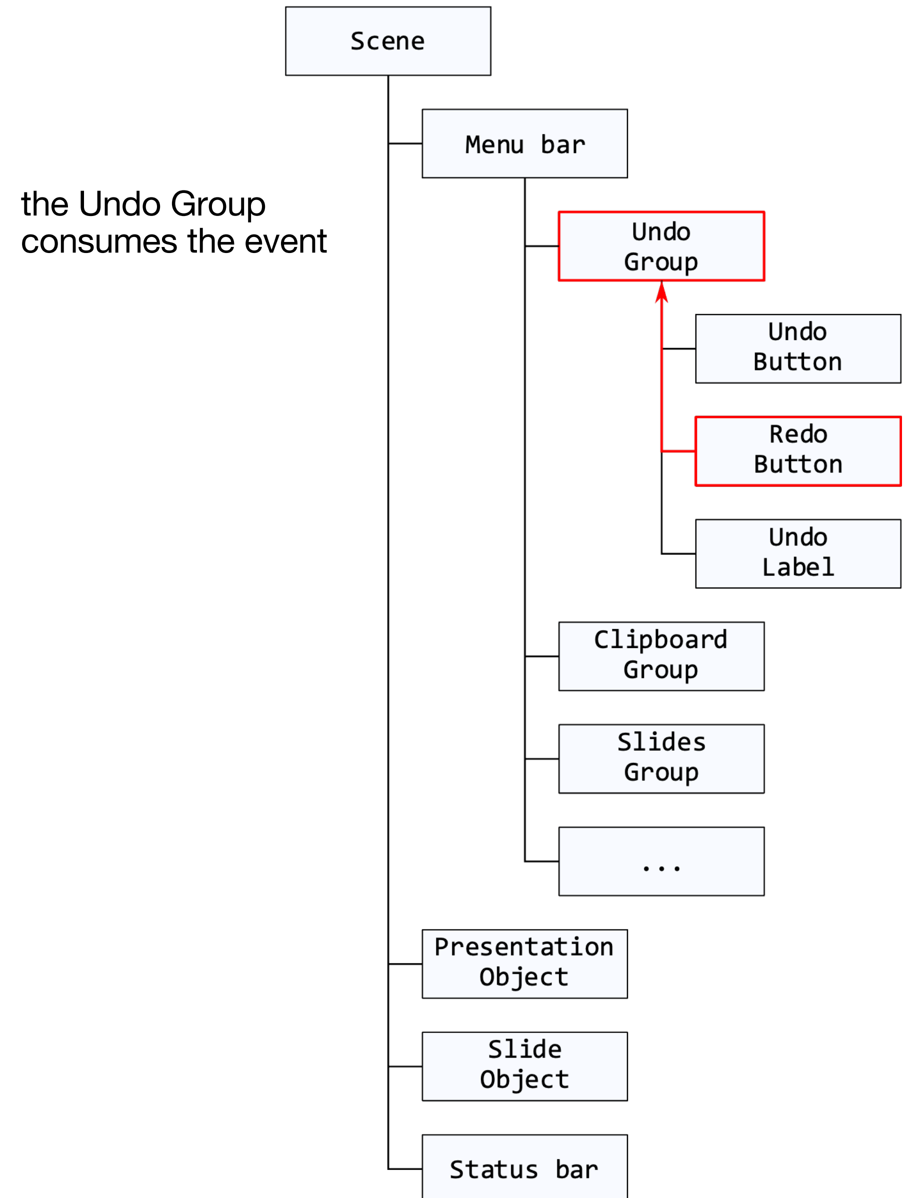
Any intermediate node can decide to consume the event, in which case it will not further propagate.



Bottom-up Positional Dispatch (using Bubble phase)

Event is dispatched to the target node first, and than travels through the scene graph towards the root node.

Event is dispatched to leaf node widget in the UI tree that contains the mouse cursor (using a Handler, registered at the Node for that event)

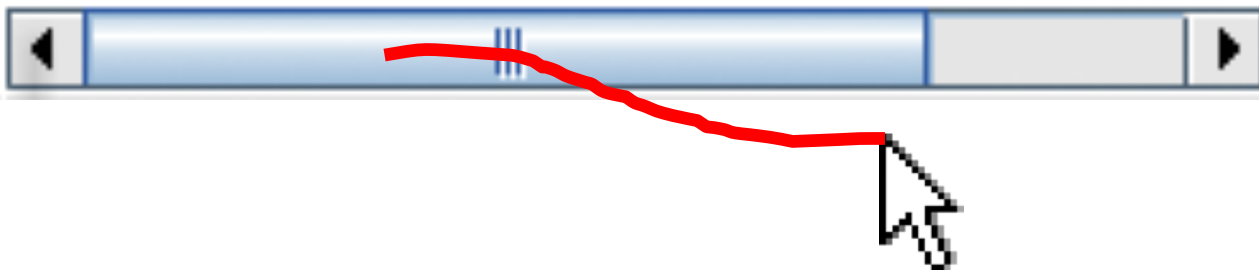


Positional Dispatch Limitations

Positional dispatch can lead to odd behaviour:

- Mouse drag starts in a scrollbar, but then moves outside the scrollbar: send the events to the adjacent widget?
- Mouse press event in one button widget but release is in another: each button gets one of the events?

Sometimes position is not enough, also need to consider which widget is “in focus”

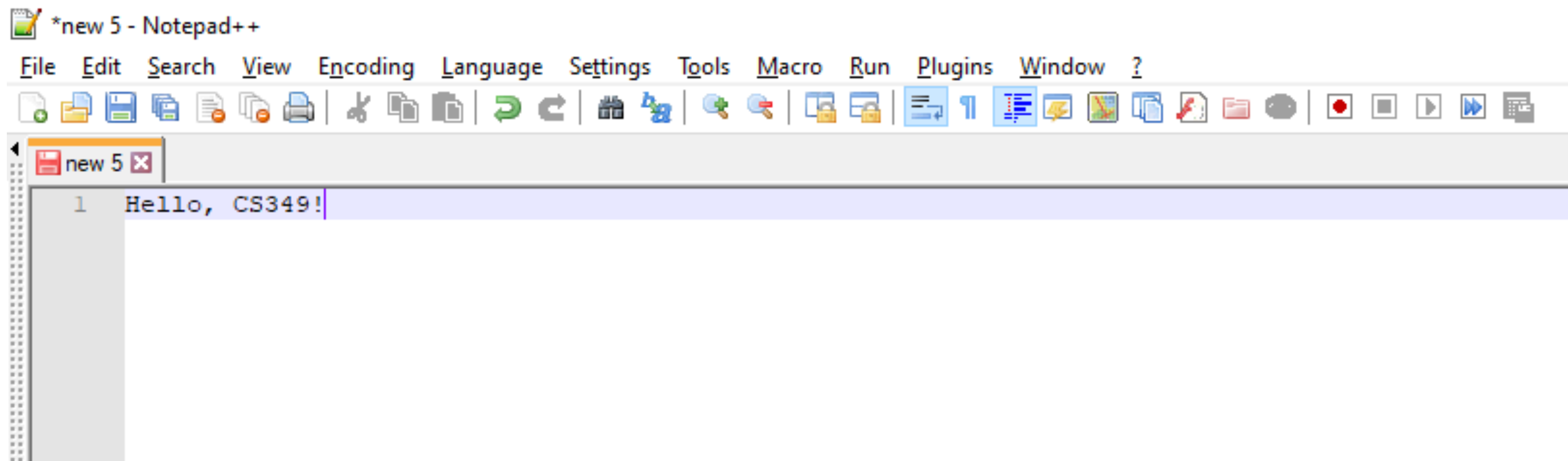


Focus Dispatch

Events dispatched to widget regardless of mouse cursor position

Needed for all keyboard and some mouse events:

- **Keyboard focus**: Click on text field, move cursor off, start typing
- **Mouse focus**: Mouse down on button, move off, mouse up … also called "mouse capture"

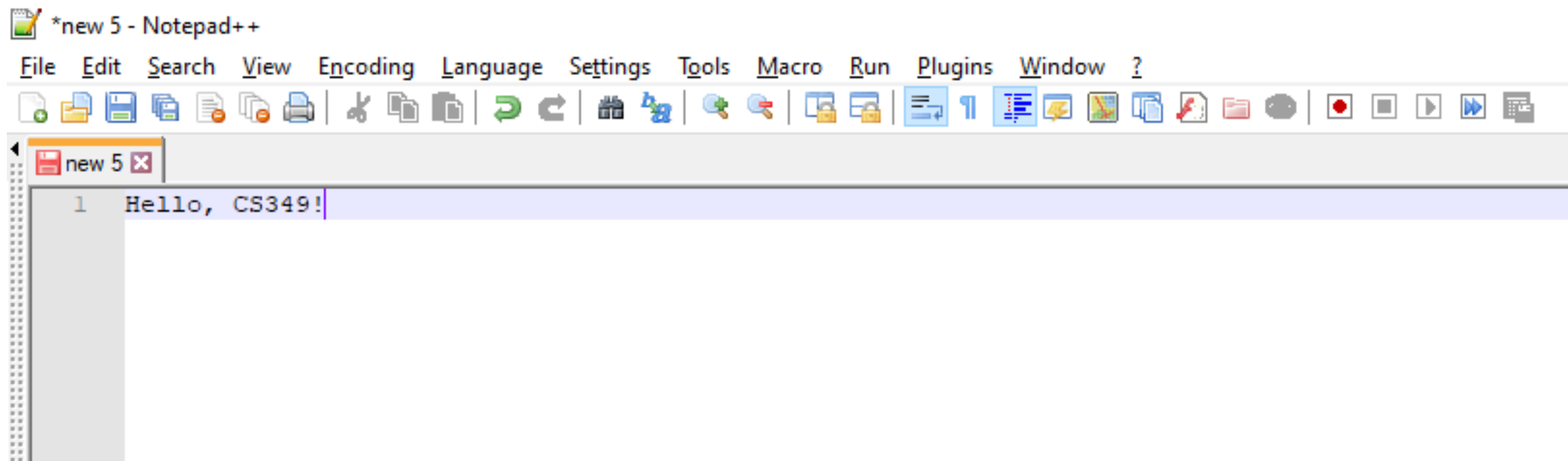


Focus Dispatch

Maximum one keyboard focus and one mouse focus (*why?*)

Need to gain and lose focus at appropriate times

- Transfer focus on mouse down (“capture”)
- Transfer focus when `TAB` key is pressed
- Ignore non-visible or non-enabled components
- Cycle through widgets based on order in scene graph



Setting up Dispatch Handlers

1. Define an event handler to capture a specific type of device, e.g., Mouse, Keyboard, or Touch.
2. Register the handler with a node
   - as a **filter** to process an event during the **capture** phase
   - as a **handler** to process during the **bubble** phase.

All “interested” nodes that could potentially process an event (i.e., that are in the route for dispatch) can register for an event and will have an opportunity to process it. They can also choose to “consume” an event, in which case it stops being propagated further along the route.

Setting up Dispatch Handlers

Add to a specific `Node` either using `EventFilter` (Capture phase) or `EventHandler` (Bubble phase):

```
scene.addEventFilter(MouseEvent.MOUSE_CLICKED) {
    println("Click:Filter ${it.sceneX}/${it.sceneY}") }

scene.addEventHandler(MouseEvent.MOUSE_CLICKED {
    println("Click:Handler ${it.sceneX}/${it.sceneY}") }



// Output:
// Click:Filter 349.0/32.0
// Click:Handler 349.0/32.0
```

Show demos!

Setting up Dispatch Handlers

```
override fun start(stage: Stage) {
    val filter = {it: MouseEvent ->
        println("Filter source: ${it.source.javaClass}")
        println("Filter target: ${it.target.javaClass}")
    }
    val handler = {it: MouseEvent ->
        println("Handler source: ${it.source.javaClass}")
        println("Handler target: ${it.target.javaClass}")
    }
    val rect = Rectangle(120.0, 120.0, Color.RED).apply {
        addEventFilter(MouseEvent.MOUSE_CLICKED, filter)
        addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
    }
    val root = Pane().apply {
        addEventFilter(MouseEvent.MOUSE_CLICKED, filter)
        addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
        translateX = 60.0
        background = Background(BackgroundFill(…))
        children.add(rect)
    }
    stage.apply {
        scene = Scene(root, 320.0, 240.0).apply {
            addEventFilter(MouseEvent.MOUSE_CLICKED, filter)
            addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
        }
        title = "Hello CS349!"
    }.show()
}
```

```
Filter source: Scene
Filter target: Rectangle
Filter source: Pane
Filter target: Rectangle
Filter source: Rectangle
Filter target: Rectangle
Handler source: Rectangle
Handler target: Rectangle
Handler source: Pane
Handler target: Rectangle
Handler source: Scene
Handler target: Rectangle
```

End of Chapter

Any further questions?