

JavaFX Layout and Event Handling

- last time
 - which control to use
 - how to read its value
 - how to validate input
 - dialogs
 - a bit of layout
- today
 - registering event handlers for nodes
 - choosing a layout container

Sources of Information

- JavaFX reference – linked in lab 7
 - provides overview, a list of resources, pointers into the Java text
 - the assignment handout itself
 - the exercise itself – look at the whole thing before starting
 - preliminaries or similar sections
 - slides and examples from class (on the schedule page)
 - Introduction to Programming Using Java (Eck)
 - chapter 6 – basics of JavaFX
 - chapter 13 – advanced elements
 - JavaFX documentation
 - JavaFX API
- go into the resources with a specific goal in mind

Lab 7 Debrief

- sources of information
- don't modify the scene graph after construction
 - instead change what is displayed in the control
 - disable/hide/unmanage rather than removing from the scene graph
- local variables vs instance variables for controls
 - need instance variables if you need to refer to the control later

The Event Model

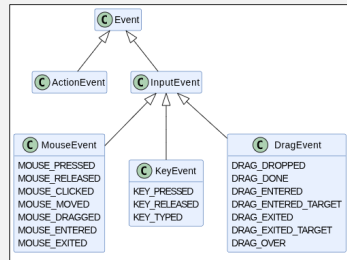
- the JavaFX Application thread runs the event loop

```
while (running) {
    Event e = eventQueue.take(); // wait until something happens
    dispatch(e);                // find the handler(s) and call them
}
```

 - OS delivers input, JavaFX turns it into Event objects and calls your handlers
- your code does not call handlers – it only registers them

The Event Model

- every event has a *type* (`MouseEvent.MOUSE_CLICKED`, ...) and details about what happened (where the mouse was, which key, ...)



Registering Handlers

- nodes provide convenience methods for specific event types
 - e.g. `button.setAction(...)`, `node.setMouseClicked(...)`, `node.setKeyPressed(...)`
- general form
 - `node.addEventHandler(MouseEvent.MOUSE_PRESSED, handler)`
- a handler is an object with a `handle(Event)` method
 - can be defined with a named class, with an anonymous class, or (most common) via a lambda

```

pileNode.setMouseClicked(e ->
    System.out.println("clicked at " + e.getX() + ", " + e.getY()));
confirmButton.setAction(e -> startNewGame());
  
```

Mouse Events

- mouse
 - `MOUSE_PRESSED`, `MOUSE_RELEASED`
 - `MOUSE_CLICKED`
 - `MOUSE_MOVED`
 - `MOUSE_DRAGGED`
 - goes to the node where the press started, even when the pointer leaves it
 - `MOUSE_ENTERED`, `MOUSE_EXITED`
- useful `MouseEvent` data
 - `getX()`, `getY()` – local coordinates of the node the handler is attached to, measured from the node's upper left corner
 - `getSceneX()`, `getSceneY()` – position measured from the scene's upper left corner
 - `getButton()` – which button caused the event (`MouseButton.PRIMARY` – left, `MouseButton.SECONDARY` – right, `MouseButton.MIDDLE`)
 - `getClickCount()` – how many quick clicks in a row (2 = double-click)
 - `isShiftDown()`, `isControlDown()`, `isAltDown()` – whether a modifier key was held

Keyboard Events

- keyboard
 - `KEY_PRESSED`
 - `KEY_RELEASED`
 - `KEY_TYPED` – use for text

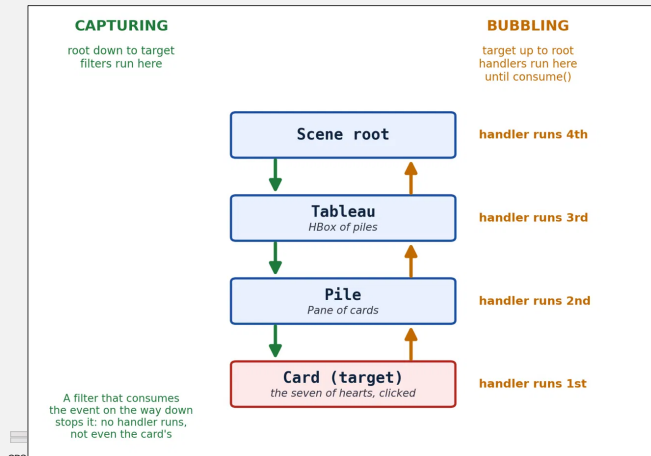
use for arrows, Enter, Escape, shortcuts – non-text keys
- useful `KeyEvent` data
 - `getCode()` – a `KeyCode` constant for the physical key; only meaningful for `KEY_PRESSED`, `KEY_RELEASED`
 - e.g. `KeyCode.LEFT`, `KeyCode.ENTER`, `KeyCode.A`
 - `getCharacter()` – the typed character as a string; only meaningful for `KEY_TYPED`
- keys go only to the *focused* node
 - mouse clicks do not give focus automatically on a plain `Pane` – need explicit handling via `requestFocus()` and `setFocusTraversable(true)` for custom nodes

Propagation and consume ()

- event handlers are not limited to the leaves of the scene graph
 - e.g. the card that was clicked
- target* is the node where the event occurred
 - e.g. the card that was clicked
- capturing* proceeds from the scene graph root down to the target
 - each node might intercept the event, preventing it from being handled at all
- bubbling* proceeds from the target back up to the scene graph root
 - handler for each node is invoked in turn, until the event is consumed
 - current node is the *source*
- consume an event when it has been fully handled and ancestors should not react
 - e.g. clicking on a card shouldn't also deselect the whole pile

61

Propagation and consume ()

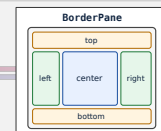


62

How Layout Works

- layout algorithms are bundled with containers
 - e.g. VBox, HBox
- a parent container sizes and positions its children
 - a child only states preferences – minWidth/Height, prefWidth/Height, maxWidth/Height
 - for a fixed size child, set minimum, preferred, and maximum to equal values
 - for an unbounded size (node stretches), set maximum to Double.MAX_VALUE
 - to let the content determine the size, set preferred size to Region.USE_COMPUTED_SIZE (default)
 - each container has its own rule for honoring preferences
- layout runs top-down when the window resizes or a child changes

BorderPane



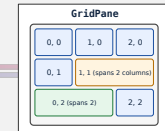
- five regions: top, left, center, right, bottom
- typical use: menu or toolbar on top, status on bottom, main content in center
 - good top-level container for a window
- top/bottom keep preferred height and stretch wide
- left/right keep preferred width and stretch tall
- center absorbs the rest
- empty regions take no space – use only the ones you need

BorderPane

- there aren't any spacing settings – use margins or padding to separate regions

call	what it does	default
setTop(node), setLeft(node), setCenter(node), setRight(node), setBottom(node)	put a node in a region one node per region – use a container to put several	empty
BorderPane.setMargin(node, new Insets(top, right, bottom, left))	empty space around a node (inside its region)	0
BorderPane.setAlignment(node, Pos.TOP_CENTER)	where a node sits inside its region when the node is smaller than the region	top/left: TOP_LEFT bottom: BOTTOM_LEFT right: TOP_RIGHT center: CENTER
setPadding(new Insets(amt))	empty space inside the BorderPane's edge, around all regions	0

GridPane



- rows and columns of cells
- typical use: forms (labels in column 0, fields in column 1), regular grids
- with no constraints, a column/row is as wide/tall as its widest/tallest child's preferred size
- extra space is not shared unless specified

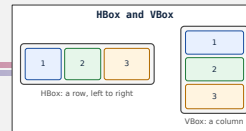
GridPane

call	what it does	default
setHgap(amt) setVgap(amt)	space between columns / between rows (not at the grid's edges)	0
setPadding(new Insets(amt))	empty space inside the grid's edge	0
GridPane.setMargin(node, new Insets(amt))	empty space around one node inside its cell	0
GridPane.setHalignment(node, HPos.RIGHT) GridPane.setValignment(node, VPos.TOP)	where a node sits inside its cell when smaller than the cell	LEFT / CENTER
GridPane.setHgrow(node, Priority.ALWAYS) GridPane.setVgrow(...)	this node's column / row takes spare space	NEVER
getColumnConstraints().add(...) getRowConstraints().add(...)	one object per column (widths) or row (heights) added in order	
setGridLinesVisible(true)	draws the grid lines (for debugging)	false

ColumnConstraints / RowConstraints

ColumnConstraints (widths) RowConstraints (heights)	what it does	default
setMinWidth(50) setMinHeight(30)	smallest the column or row may shrink to	computed from the children
setPrefWidth(120) setPrefHeight(40)	column/row's preferred size	computed from the children
setMaxWidth(200) setMaxHeight(80)	largest the column/row may grow to	computed from the children
setHgrow(Priority.ALWAYS) setVgrow(Priority.ALWAYS)	takes spare space when the grid is larger than its preferred size	none (does not grow)
setPercentWidth(25) setPercentHeight(50)	a fixed share of the grid's width or height overrides the other size settings and grow for that column or row	not used
setHalignment(HPos.RIGHT) setValignment(VPos.TOP)	where nodes sit when smaller than their cell; a node's own GridPane.setHalignment / setValignment wins	the grid's default: left, center
setFillWidth(false) setFillHeight(false)	if true, nodes stretch to fill the cell (if their max size allows) if false, they keep their preferred size	true

VBox and HBox

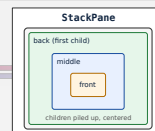


- VBox stacks children from top to bottom
- HBox lays out children from left to right
- typical uses
 - row of buttons (HBox) inside a form (VBox), toolbar, sidebar
- each child gets its preferred size along the box's direction and is widened (up to max size) to fill in the other direction
- extra space along the direction goes only to children marked to grow
 - multiple ALWAYS children share equally

VBox and HBox

call	what it does	default
<code>setSpacing(amt)</code>	gap between neighboring children (none before the first or after the last)	0
<code>setPadding(new Insets(amt))</code>	empty space inside the box's edge, around all the children; the box's background still fills it	0
<code>setAlignment(Pos.CENTER_RIGHT)</code>	where the children go, as a group, when the box has spare room	TOP_LEFT
<code>VBox.setFillWidth(true)</code> <code>HBox.setFillHeight(true)</code>	stretch children across the box (up to their max) or let each keep its preferred size	true
<code>HBox.setHgrow(node, Priority.ALWAYS)</code> <code>VBox.setVgrow(...)</code>	who gets spare space along the box: ALWAYS shares it; SOMETIMES gets it only if no ALWAYS child exists; NEVER gets none	NEVER
<code>HBox.setMargin(node, new Insets(top, right, bottom, left))</code>	empty space outside one child, set through the parent's class	0

StackPane



- children are drawn on top of each other – later children on top
 - centered by default
- typical uses
 - status message over a game area
 - selection highlight above content
 - background image under everything
- each child is sized between its min and max and placed by alignment
- stack's own preferred size comes from its largest child
- overlays that must not steal clicks need `setMouseTransparent(true)`

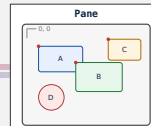
```
StackPane layers = new StackPane(feltBackground, cardGrid,
                                selectionHighlight);
selectionHighlight.setMouseTransparent(true);
```

StackPane

- no spacing setting – children (layers) are meant to overlap

call	what it does	default
<code>new StackPane(a, b, c)</code>	add children bottom to top	
<code>setAlignment(Pos.TOP_LEFT)</code>	where children are placed inside the stack, for all children	CENTER
<code>StackPane.setAlignment(node, Pos.BOTTOM_RIGHT)</code>	override the alignment for one child	the stack's setting
<code>StackPane.setMargin(node, new Insets(amt))</code>	empty space around one child inside the stack	0
<code>setPadding(new Insets(amt))</code>	empty space inside the stack's edge	0
<code>node.setVisible(false)</code>	hide a layer: it is not drawn and not clicked (it still takes part in sizing; <code>setManaged(false)</code> removes it)	true

Pane



- plain container
- typical use – as a base for custom component
- no layout rule
 - children are sized to their preferred sizes
 - position of children is set manually

Pane

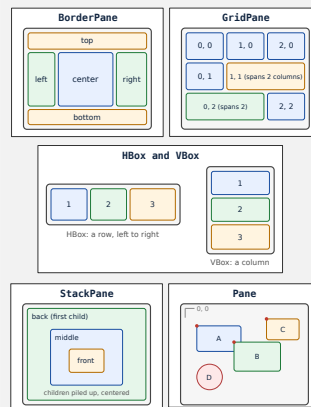
call	what it does	default
<code>getChildren().addAll(a,b)</code>	add children; a later child is drawn on top	no children
<code>node.setLayoutX(40)</code> <code>node.setLayoutY(25)</code>	place the node's top-left corner, in the pane's coordinates	0, 0
<code>node.relocate(40, 25)</code>	set layoutX and layoutY in one call	
<code>node.setPrefSize(80, 30)</code>	the size the pane gives a resizable child (Label, Button, Region)	computed from the content
<code>rect.setX(20)</code> <code>rect.setY(50)</code> <code>circle.setCenterX(...)</code>	a Shape places itself with its own coordinates; the pane does not resize it	0
<code>node.setTranslateX(15)</code> <code>node.setTranslateY(10)</code>	shift the node away from its layout position; the pane does not overwrite it, so it suits dragging	0
<code>node.toFront()</code> <code>node.toBack()</code>	move the node to the top / bottom of the drawing order	order added
<code>setPrefSize(400, 300)</code>	the pane's own preferred size	computed from the farthest right and bottom edges of the children, plus padding

Choosing a Layout

- pick the outer container by the overall shape of the window – how does the window divide into regions?

need...	...use
top-level window with menubar, toolbar, sidebar, status bar	BorderPane
form or regular grid	GridPane
row or column of items	HBox / VBox
layers or overlays	StackPane
free positioning	Pane

- nest inner containers for each region
 - prefer fewer containers over deep VBox/HBox chains



Resizing Strategy

- decide per region
 - should it grow with the window (set grow/max) or stay fixed (set min = pref = max)?
- for fixed size content in a resizable window, put it in a container that centers it
 - e.g. StackPane or BorderPane

when the window is resized –

- wider – the tableau piles spread out evenly, the side panel stays 150 pixels wide, the foundations stay at the right edge of the board and the stock and waste at the left
- taller – the tableau gets the extra height, the other sizes don't change
- the three labels stay at the top of the side panel, hint and undo stay at the bottom
- new game stays at the right end of the status bar

17