Programming Languages and Techniques (CIS1200)

Lecture 20

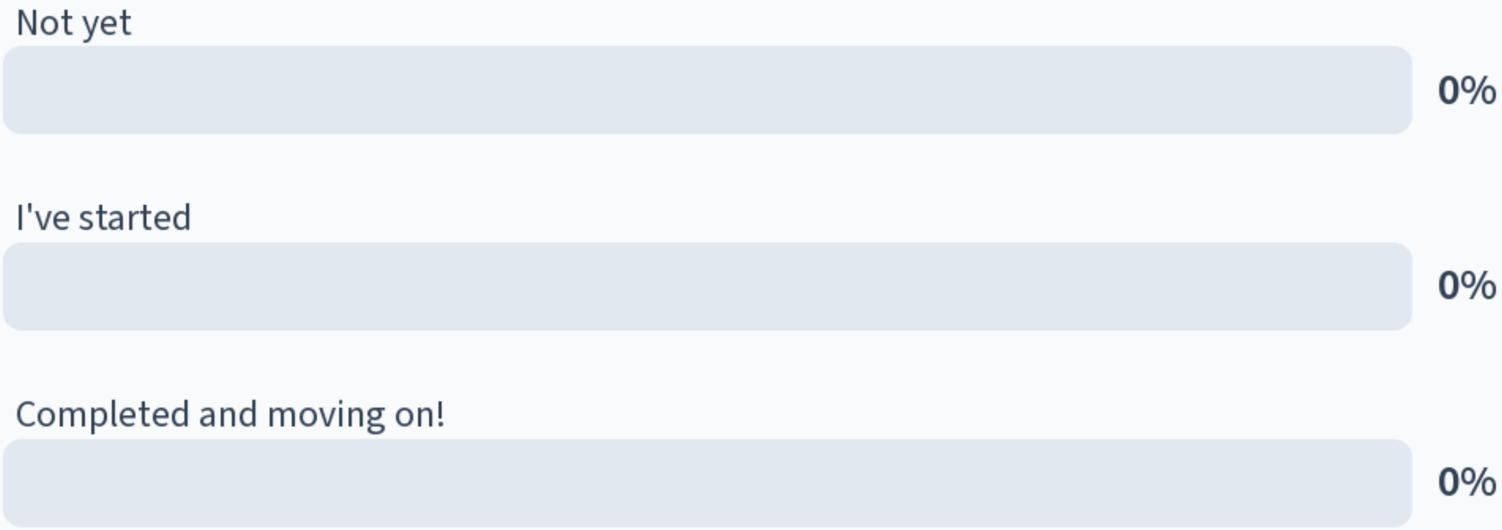
GUI library: Events and State

Chapter 18

Announcements

- HW05 available, due *Thursday*, October 24th (at 11.59pm)
  - The project is structured as *tasks*, not *files* (one task may touch multiple files)
    - Tasks 0-4 can be done already
    - Tasks 5-6 can be done after class today

Have you done Task 0 yet?



Not yet

0%

I've started

0%

Completed and moving on!

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

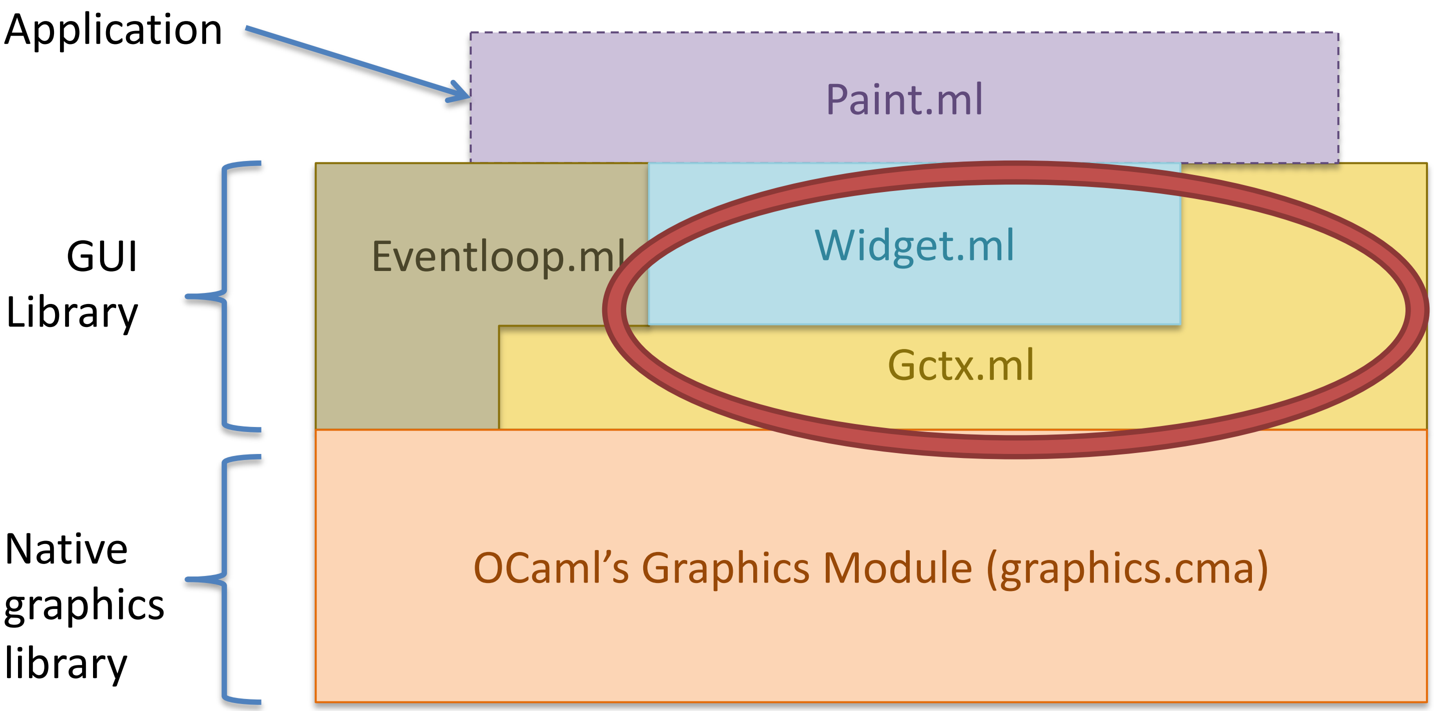
Review: Widget Layout

Building blocks of GUI applications

see simpleWidget.ml in GUI Demo Code project

Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position



The graphics context isolates the Widgets module from the Graphics module.

Simple Widgets

simpleWidget.mli

```
(* An interface for simple GUI widgets *)
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

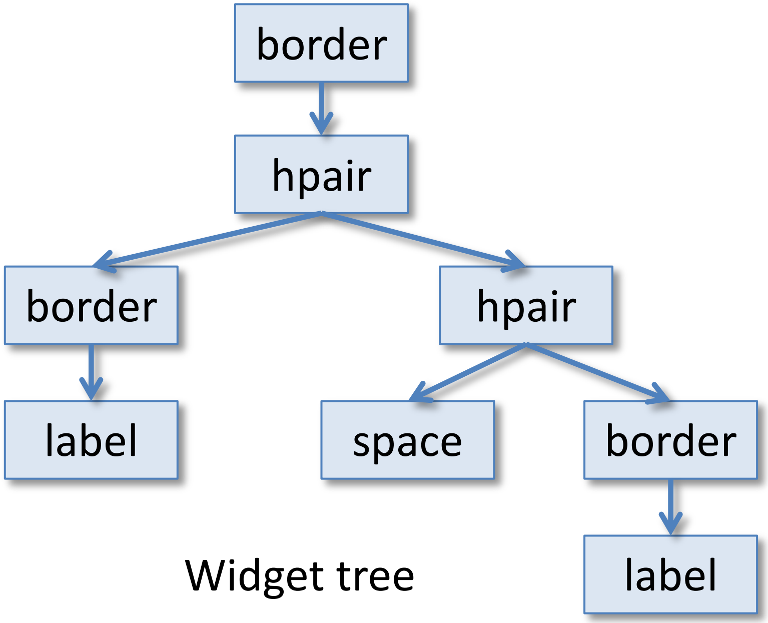
- You can ask a simple widget to repaint itself
- You can ask a simple widget to tell you its size
- (We'll talk about handling events later)

- Repainting is relative to a graphics context

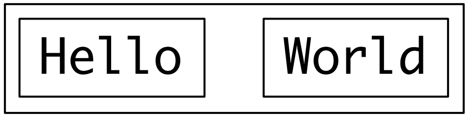
Widget Hierarchy Pictorially

swdemo.ml

```
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"
(* Compose them horizontally, adding some borders *)
let h = border (hpair (border l1)
                      (hpair (space (10,10)) (border l2)))
```



Widget tree

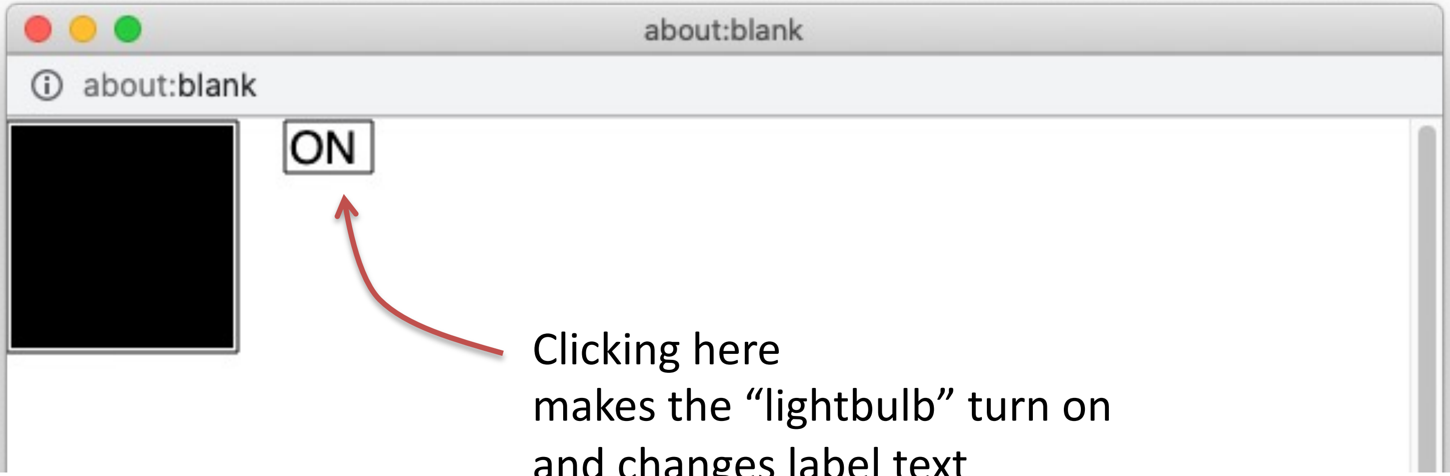


On the screen

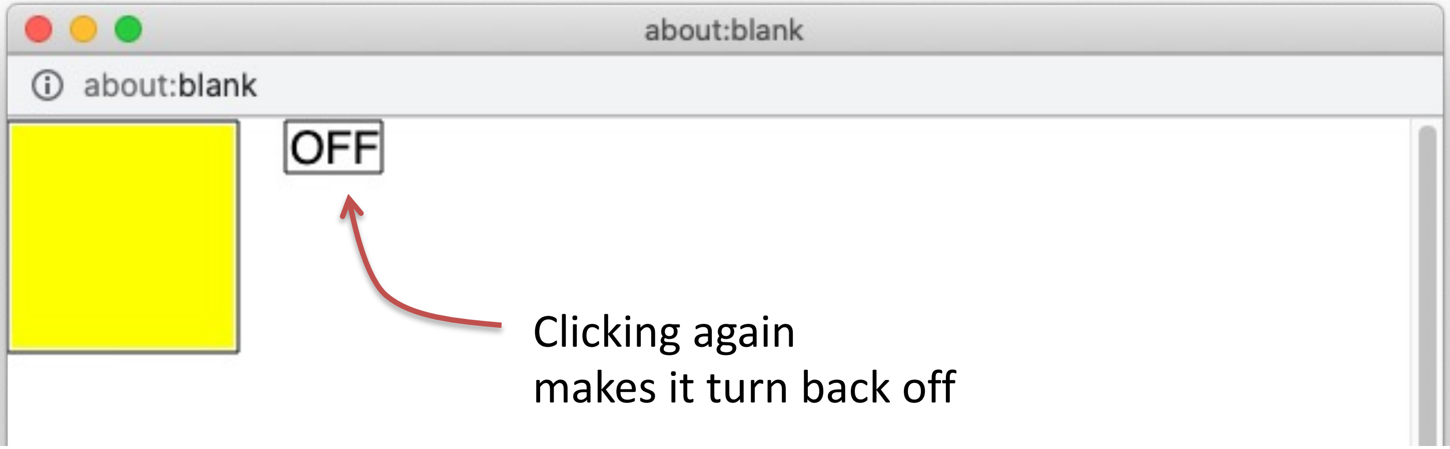
Coding with Simple Widgets

see swdemo.ml

"lightbulb" demo



Clicking here
makes the “lightbulb” turn on
and changes label text



Clicking again
makes it turn back off

19: Do you know how you would use the (simple) widget library to define the layout of this application?

I don't know how to start

0%

I may have it, but I'm not sure

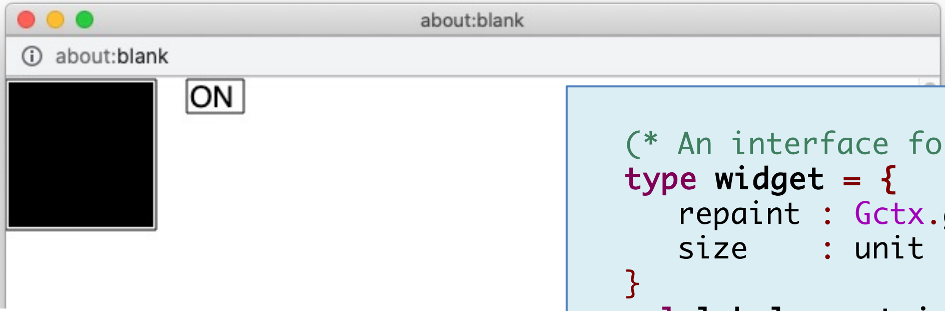
0%

I'm sure I've got it

0%

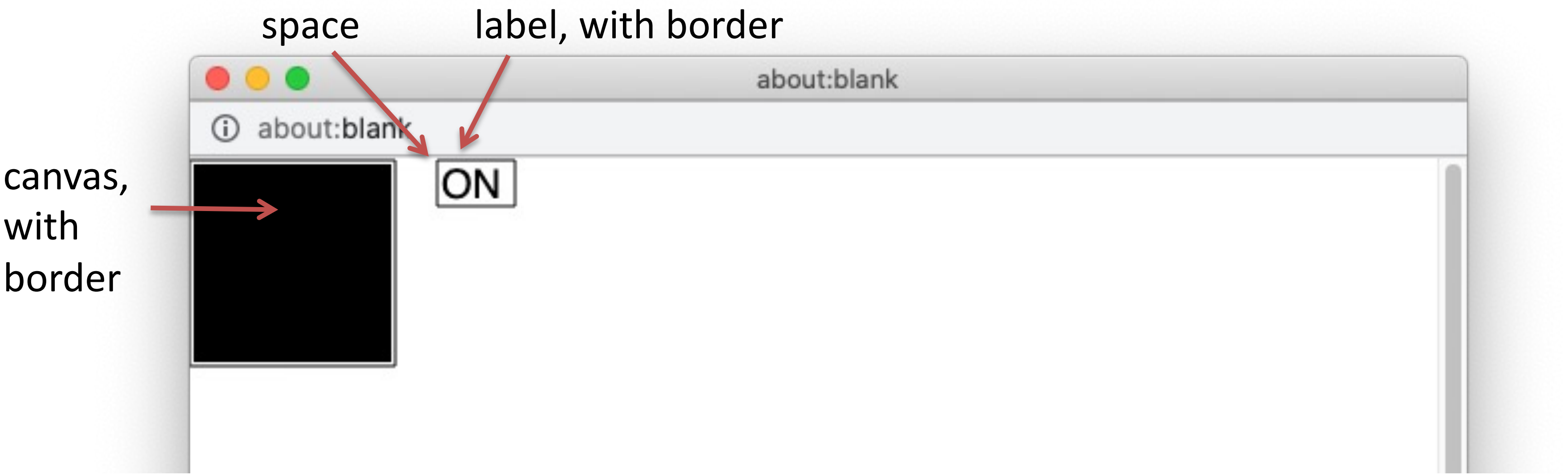
Do you know how you would use the (simple) widget library to define the layout of this lightbulb application?

1. I'm not sure how to start.
2. I may have it, but I'm not sure.
3. Sure! No problem.



```
(* An interface for simple GUI widgets *)
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

"lightbulb" demo layout



space

label, with border

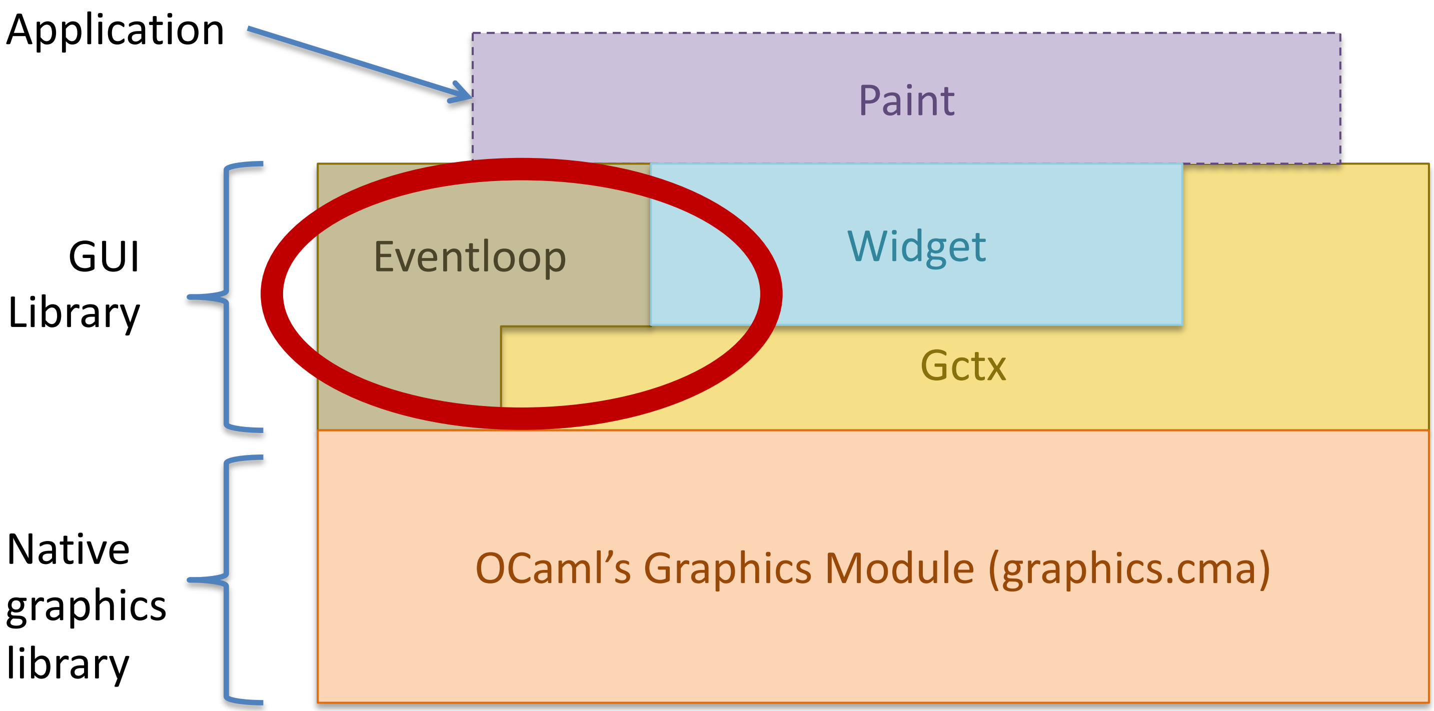
canvas, with border

swdemo.ml

```
let onoff = border (label "ON")

let paint_bulb (g: Gctx.gctx) : unit = …

let bulb = border (canvas (100, 100) paint_bulb)

let top : widget = hpair bulb (hpair (space (20, 20)) onoff)
```

Events and Event Handling

Project Architecture



Event loop with event handling

```
let run (w:widget) : unit =
  let g = Gctx.top_level  in      …create the initial gctx…
  w.repaint g;                    …display the widget
  Graphics.loop                   …wait for user input
    (fun e ->
       clear_graph ();
       w.handle g e;              …inform widget about the event…
       w.repaint g)               …update the widget's appeara
```

Eventloop

```
let rec loop (f: event -> unit) : unit =
  let e = wait_next_event () in
  f e;
  loop f
```

Graphics

Events

gctx.mli

```
type event

val wait_for_event : unit -> event

type event_type =
   | KeyPress of char  (* User pressed a key                 *)
   | MouseDown         (* Mouse Button pressed, no movement  *)
   | MouseUp           (* Mouse button released, no movement *)
   | MouseMove         (* Mouse moved with button up         *)
   | MouseDrag         (* Mouse moved with button down       *)

val event_type : event -> event_type
val event_pos  : event -> gctx -> position
```

Remember:
The graphics context translates the location of the event *to widget-local coordinates*

Reactive Widgets

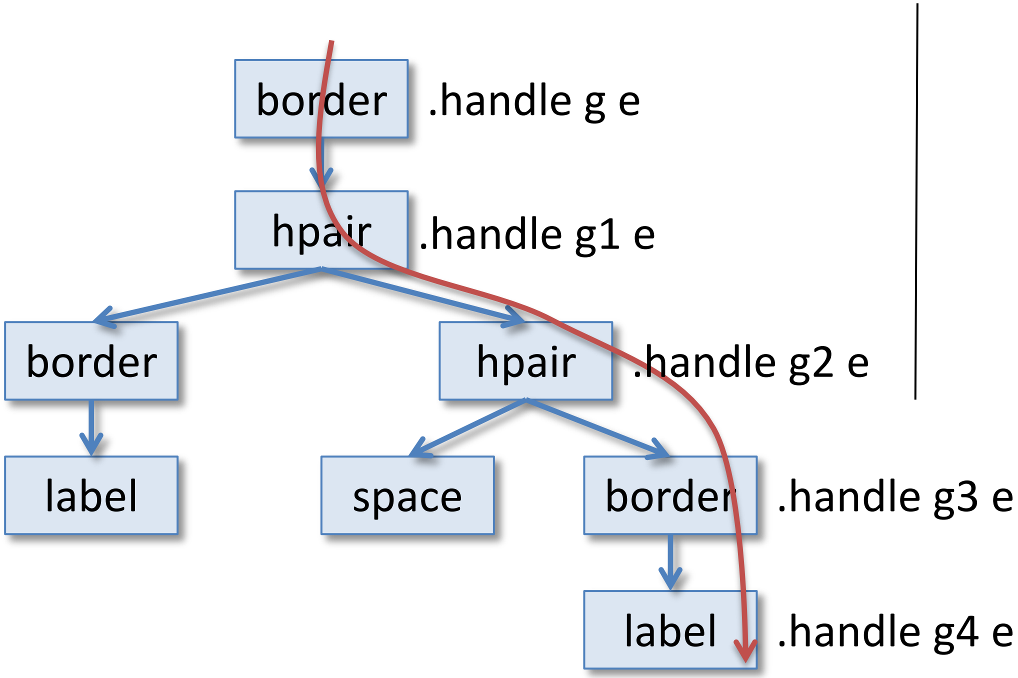
widget.mli

```
type widget = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit
}
```

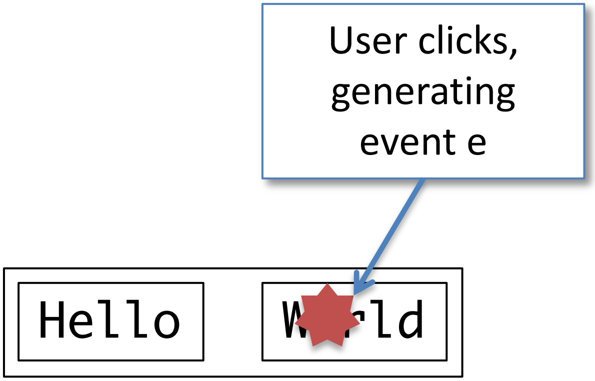
- Widgets now have a “method” for handling events
  - The eventloop waits for an event and then gives it to the root widget
  - The widgets forward the event down the tree, according to the position of the event

Event-handling: Containers

Container widgets propagate events to their children:



Widget tree



On the screen

```
g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (hello_width,0)
g3 = Gctx.translate g2 (space_width,0)
g4 = Gctx.translate g3 (2,2)
```

Routing events through container widgets

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.gctx must be translated so that the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:widget):widget =
  { repaint = …;
    size = …;
    handle = (fun (g:Gctx.gctx) (e:Gctx.event) ->
       w.handle (Gctx.translate g (2,2)) e);
  }
```

19: Consider routing an event through an hpair widget constructed as shown. The event will always be propagated either to w1 or w2.

True

0%

False

0%

Consider routing an event through an hpair widget constructed by:

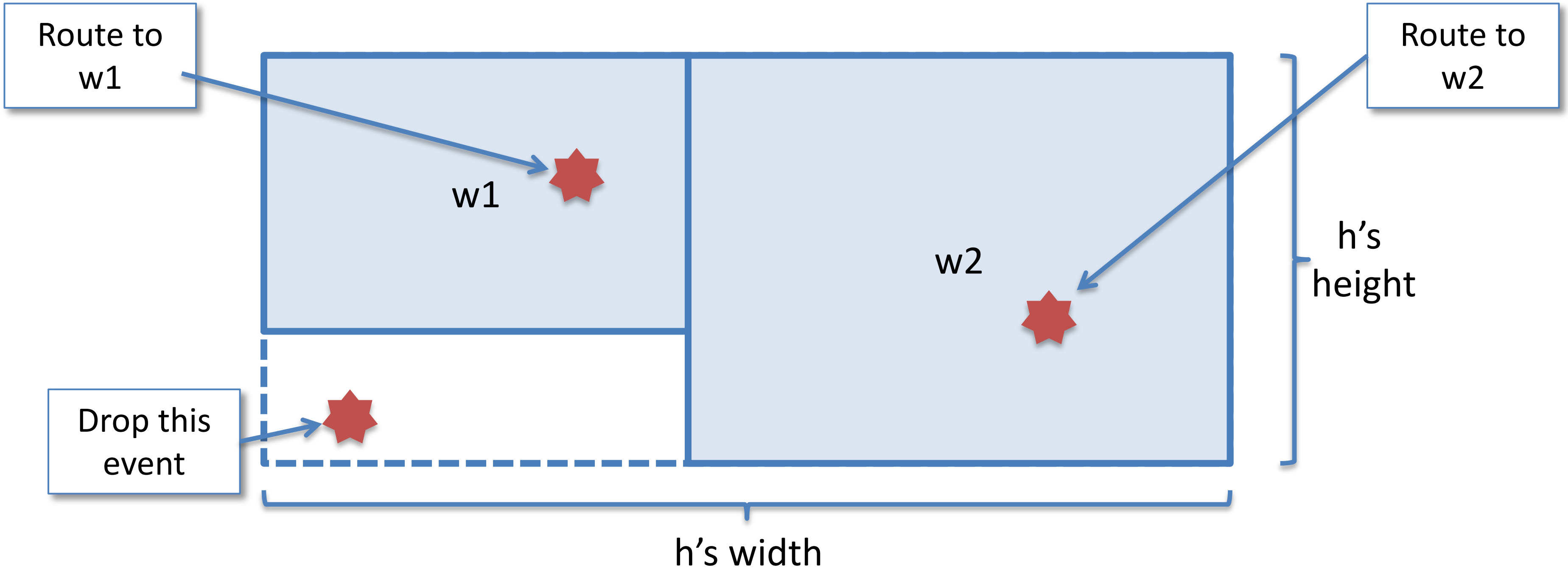
```
let hp = hpair w1 w2
```

The event will always be propagated either to w1 or w2.

1. True
2. False

Answer: False

Routing events through hpair widgets



- There are three cases for routing in an hpair.
- An event in the “empty area” should not be sent to either w1 or w2.

Routing events through hpair widgets

- The event handler of an hpair must check to see whether the event should be handled by the left or right widget.
  - Check the event’s coordinates against the *size* of the left widget
  - If the event is within the left widget, let it handle the event
  - Otherwise check the event’s coordinates against the right child’s
  - If the right child gets the event, don’t forget to translate its coordinates

```
handle =
 (fun (g:Gctx.gctx) (e:Gctx.event) ->
   if event_within g e (w1.size ())
   then w1.handle g e
   else
   let g = (Gctx.translate g (fst (w1.size ()), 0)) in
     if event_within g e (w2.size ())
     then w2.handle g e
     else ())
```

Stateful Widgets

How can widgets react to events?

A plain (stateless) label widget

```
let label (s:string) : widget =
{
  repaint = (fun (g:Gctx.gctx) -> Gctx.draw_string g (0,0) s);
  handle  = (fun _ _ -> ());
  size = (fun () -> Gctx.text_size s)
}
```

first stab at a

A stateful label Widget

```
let label (s: string) : widget =
  let r = { contents = s } in
  { repaint = (fun (g: Gctx.gctx) ->
                 Gctx.draw_string g (0,0) r.contents);
    handle  = (fun _ _ -> ());
    size    = (fun () -> Gctx.text_size r.contents)
  }
```

- The label object can make its string mutable. The “methods” can refer to this mutable string.
- But how can we change this string in response to an event?
- (r is "local" state accessible only by repaint/size funs --- see Ch. 17)

A stateful `label` Widget

widget.ml

```
type label_controller = { set_label: string -> unit;
                          get_label: unit -> string  }

let label (s: string) : widget * label_controller =
  let r = { contents = s } in
    ({ repaint = (fun (g: Gctx.gctx) ->
                  Gctx.draw_string g (0,0) r.contents);
       handle  = (fun _ _ -> ());
       size    = (fun () -> Gctx.text_size r.contents)
     }
    ,
    { set_label = (fun (s: string) -> r.contents <- s);
      get_label = (fun () -> r.contents);
    }
)
```

– Here, the `label_controller` object returned by `label` provides a way to set and get the label string notifierdemo.ml — increasingly sophisticated approaches to event handling

DEMO: NOTIFIER

Event Listeners

See notifierdemo.ml

(distributed with the lecture demos in Codio)

Handling multiple event types

- Problem: *Widgets may want to react to many different events*
- Example: Button
  - mouseclick: activates the button, primary reaction
  - mouse movement: tooltip?
  - key press: keyboard access to the button functionality?
- These reactions should be independent
  - Each event handled by a different *event listener* (i.e. first-class function)
  - Widgets may have *several* listeners to handle a triggered event
  - Listeners react in sequence; all are notified about the event
- Many different kinds of widgets react to events
  - Don't want to repeat the code for buttons in other widgets in the library
- Solution: notifier!

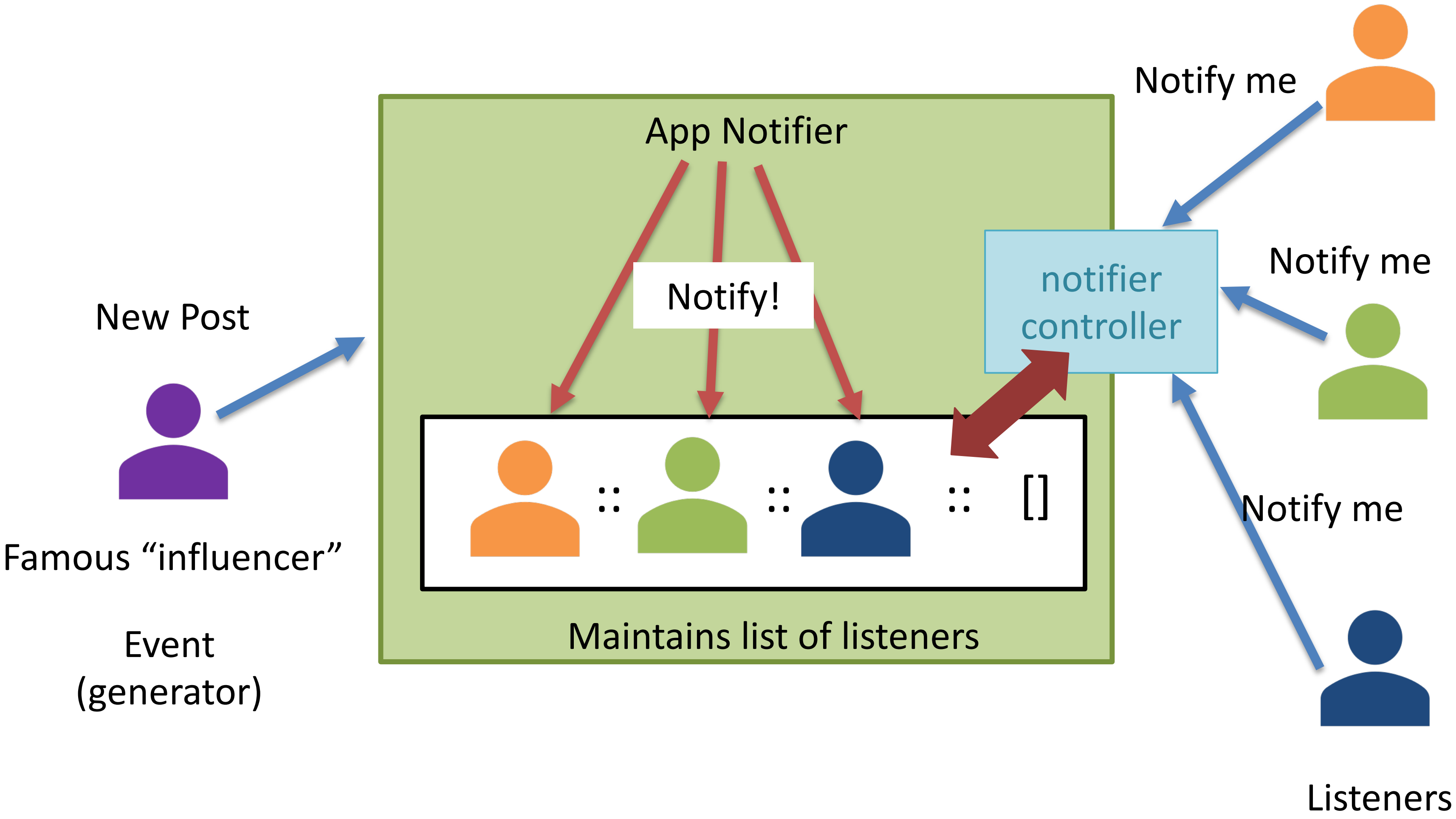
Analogy: Handling multiple event types

- Problem: Imagine a photo/video sharing app where you want to react to when your friend shares a new post

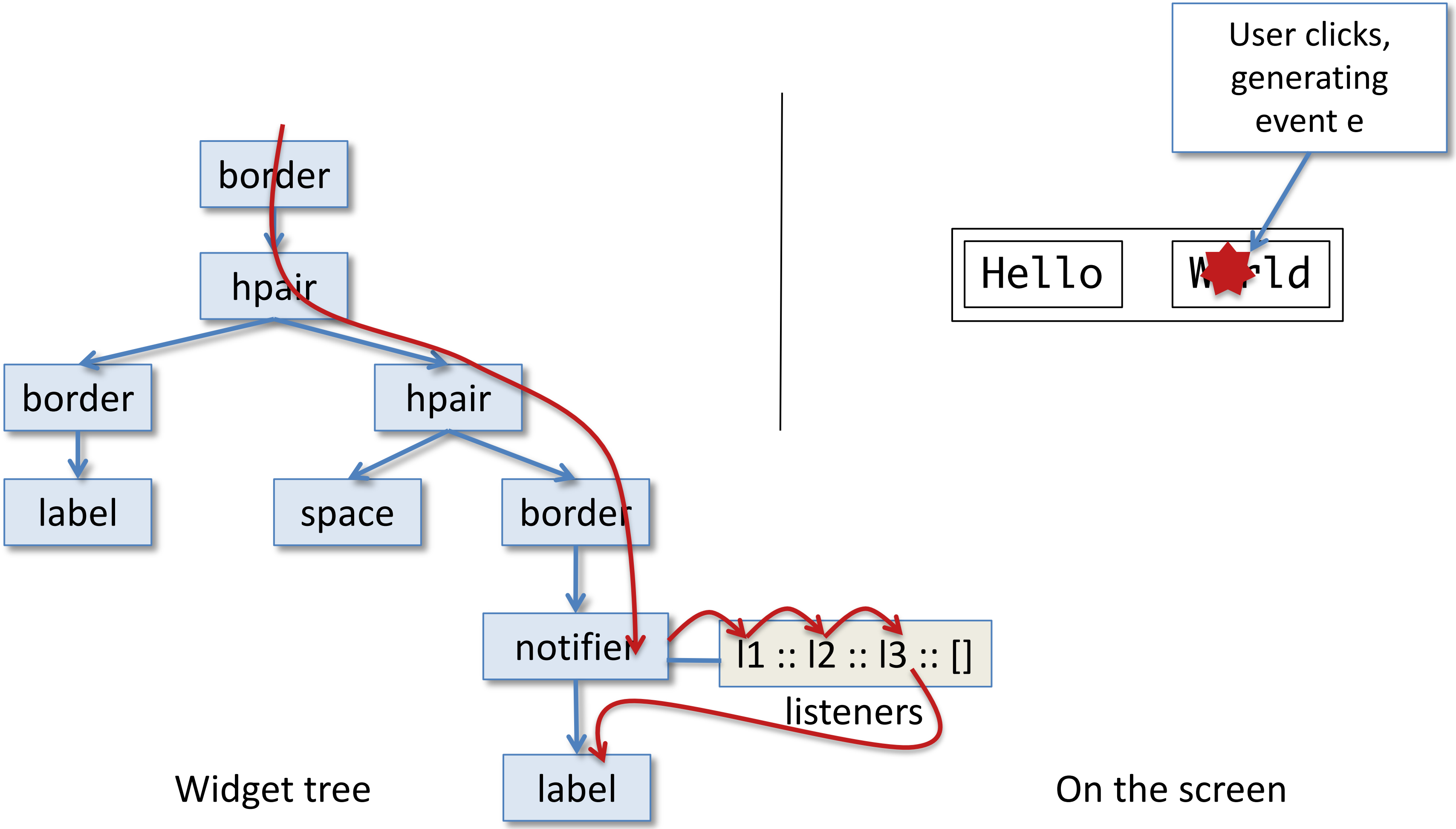
- Option 1 – Manual (Terrible idea!)
  - Keep refreshing the page every minute to see if there’s new content
  - Wasteful!

- Option 2 – Push Notifications
  - You can sign up to be *notified* when there is new content
  - Other people can sign up for the same notification too
  - If there is new content, you might “react” in a different way depending on the content – if it’s a picture, you want to reshare it; if it’s a video, you want to comment on it; ...
  - Your (and other people’s reactions) should be independent!

Analogy: Listeners and Notifiers Pictorially



Listeners and Notifiers Pictorially



User clicks, generating event e

Widget tree

On the screen

Notifiers

- A *notifier* is a container widget that adds event listeners to a node in the widget hierarchy
  - Note: this way of structuring event listeners is based on Java's Swing Library design (we use Swing terminology).
- *Event listeners* "eavesdrop" on the events flowing through the notifier
  - The event listeners are stored in a list
  - They react in order
  - Then the event is passed down to the child widget
- Event listeners can be added by using a `notifier_controller`

Listeners

widget.ml

```
type event_listener = Gctx.gctx -> Gctx.event -> unit

(* Performs an action upon receiving a mouse click. *)
let mouseclick_listener (action: unit -> unit)
                        : event_listener =
  fun (g:Gctx.gctx) (e: Gctx.event) ->
    if Gctx.event_type e = Gctx.MouseDown
    then action ()
```

Note: the type `event_listener` *is* the type of the `handle` method from the `widget` type.

widget.mli

```
type widget = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit
}
```

Notifiers and Notifier Controllers

widget.ml

```
type notifier_controller =
     { add_listener : event_listener -> unit }

let notifier (w: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
  { repaint = w.repaint;
    size    = w.size
    handle  =
      (fun (g: Gctx.gctx) (e: Gctx.event) ->
          List.iter (fun h -> h g e) listeners.contents;
          w.handle g e);
},
  { add_event_listener =
      fun (newl: event_listener) ->
          listeners.contents <-
                 newl :: listeners.contents
  }
```

Loop through the list of listeners, allowing each one to process the event. Then pass the event to the child.

The notifier_controller allows new listeners to be added to the list.

Buttons (at last!)

widget.ml

```
(* A text button *)
let button (s: string) : widget
                       * label_controller
                       * notifier_controller =
  let (w, lc)  = label s in
  let (w', nc) = notifier w in
    (w', lc, nc)
```

- A button widget is just a label wrapped in a notifier
- Add a mouseclick_listener to the button using the notifier_controller
- (For aesthetic purposes, we could also put a border around the label widget.)

Event Handling Summary

- An *event* is a signal
  - e.g., a mouse click or release, mouse motion, or keypress
  - Events carry data, such as e.g., state of the mouse button, the coordinates of the mouse, the key pressed
- An event can be *handled* by some widget
  - The top-level loop waits for an event and then gives it to the root widget
  - The widgets forward the event down the tree
  - e.g., a button handles a mouse click event
- Typically, the widget that handles an event *updates some state* of the GUI
  - e.g., to record whether the light is on and change the label of the button
  - state is usual updated via a *controller,* e.g., a label_controller
- A *listener* associates an action with a particular type of event
  - e.g., a mouseclick_listener does something on a mouse click
  - listeners are triggered when a *notifier* widget handles an event
- User sees the reaction to the event when the GUI repaint itself
  - e.g., button has new label, canvas is a new color

onoff.ml — changing state on a button click

DEMO: ONOFF

20: True or False: It is possible to create a single button that toggles the states of two separate lightbulbs.

True

0%

False

0%

True or False: One can use a notifier and label to create a button that toggles the states of *two* separate lightbulb canvases.

Answer: True