Programming Languages and Techniques (CIS1200)

Lecture 21

GUI library: Events and State

Transition to Java

Chapters 19 & 20

Announcements

- HW05: GUI programming
  - Due: *Thursday* at 11.59pm

- Java Bootcamp / Refresher: Sunday, October 27
  - 1-3pm, Towne 100
  - Will be recorded
  - Look for more details on Ed

- HW06: Pennstagram
  - Java array programming
  - Available later this week
  - Due *Thursday*, October 31st at 11.59pm

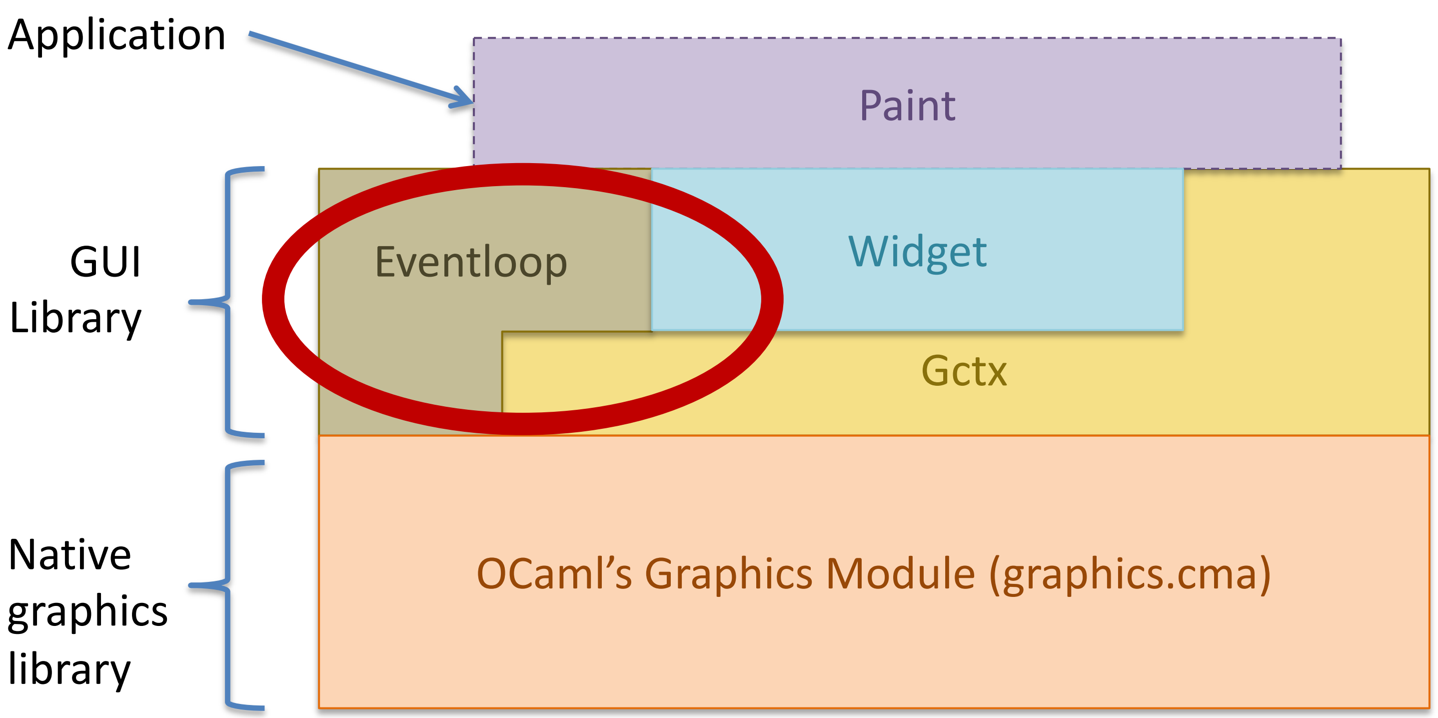
20: How far along are you in HW05: GUI Programming?

0

Not started yet	0%
Task 0 finished	0%
Working on tasks 1-4	0%
Working on Task 5	0%
Working on Task 6	0%
All done!	0%

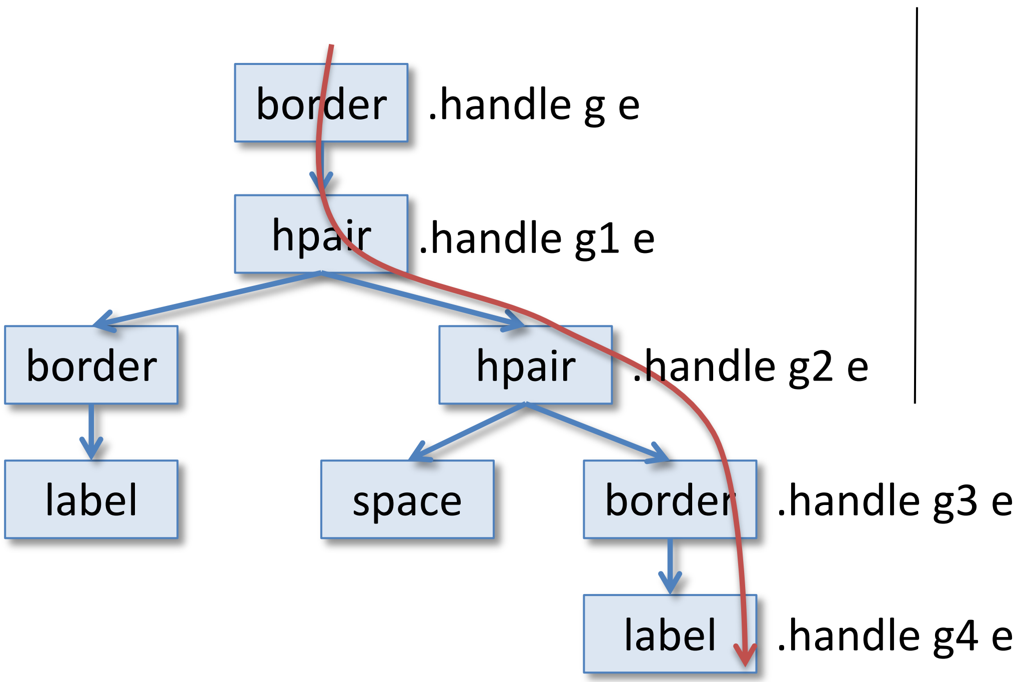
Review: Events and Event Handling

Project Architecture

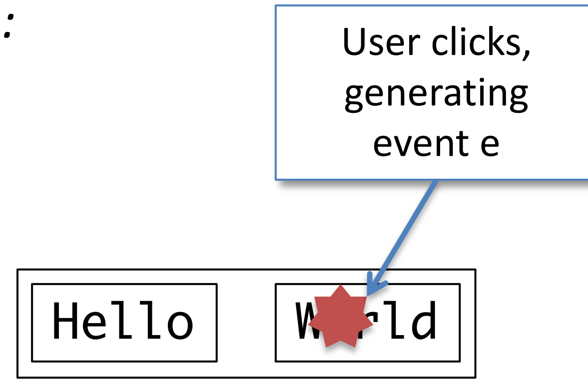


Event-handling: Containers

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)

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!

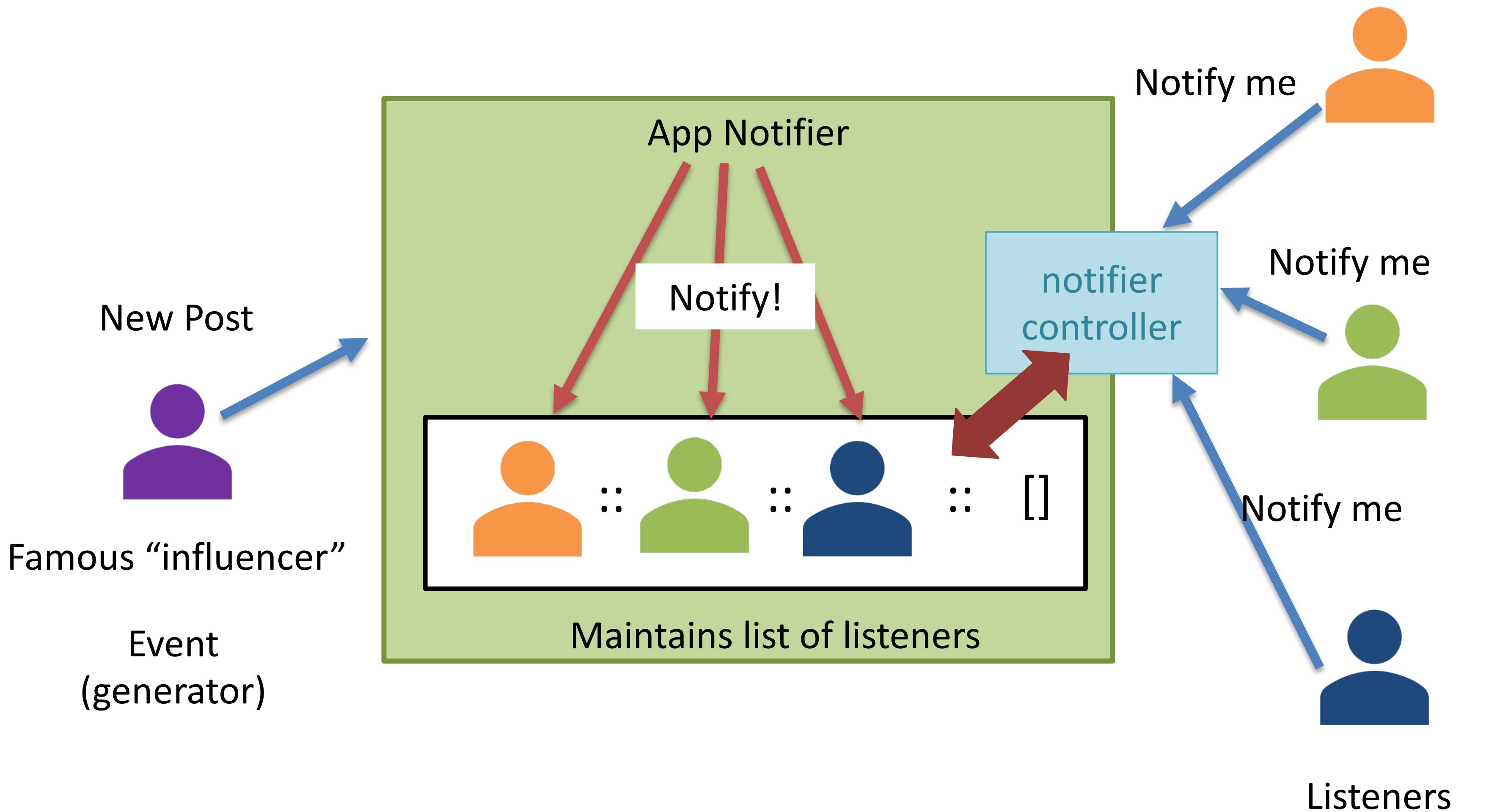
Analogy: Handling multiple event types

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

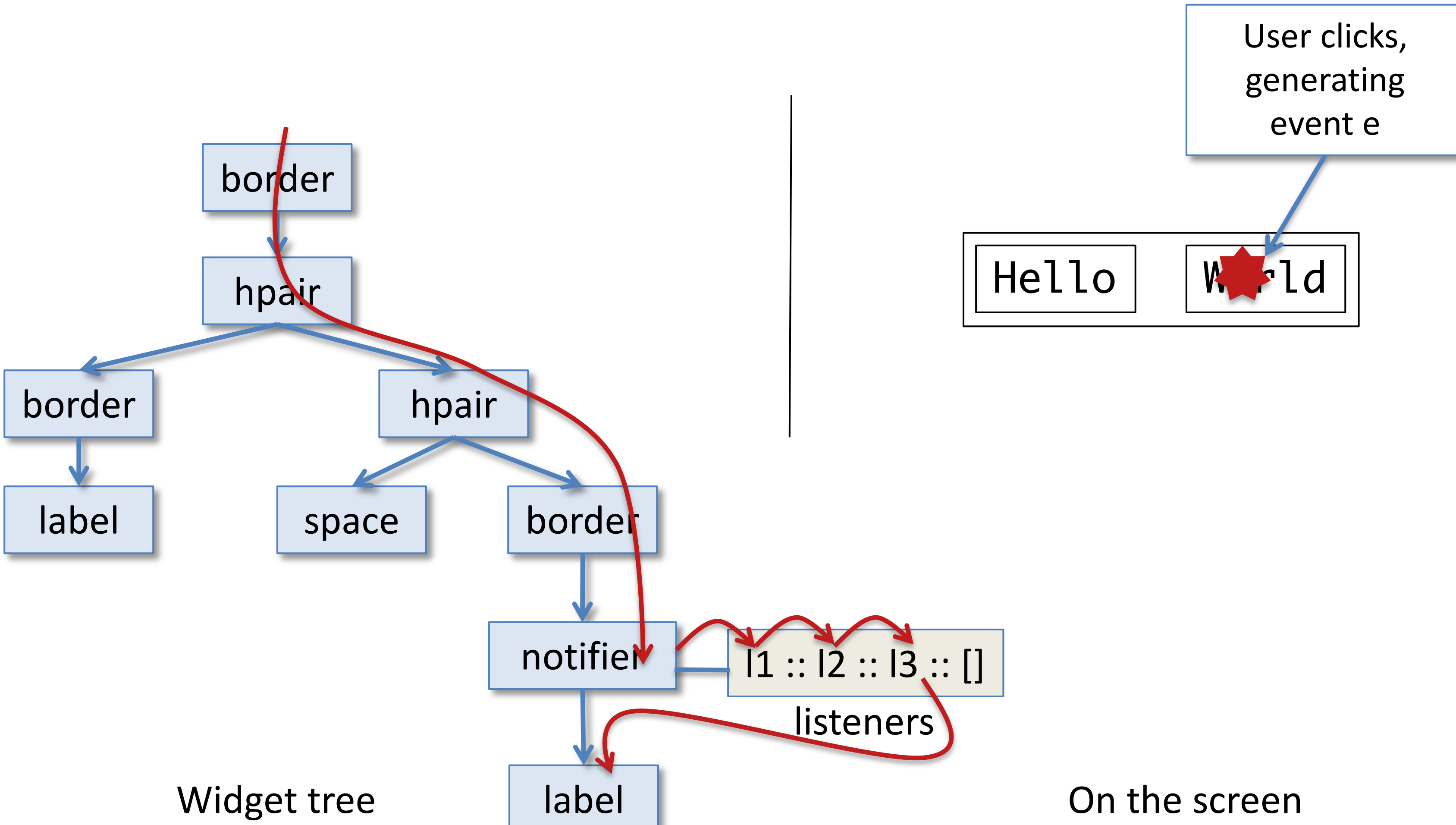
- Option 1 – Manual (Terrible idea!)
  - Keep refreshing the page every minute to see if there's new content
  - Wasteful!

- 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



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

Goodbye OCaml…

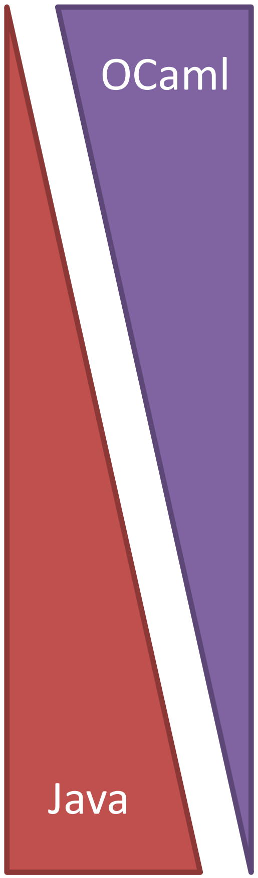
…Hello Java!

Smoothing the transition to Java

- General advice for the next few lectures:
  - Ask questions, but don't stress about the details
  - Wait till you need them
- Java resources:
  - Our lecture notes
  - Ed
  - CIS 1100 website and textbook
  - Online Java textbook (http://math.hws.edu/javanotes/) linked from "Resources" on course website

CIS 1200 Semester Overview

- Declarative (Functional) programming
  - *persistent* data structures
  - *recursion* is main control structure
  - frequent use of functions as data
- Imperative programming
  - *mutable* data structures (that can be modified “in place”)
  - *iteration* is main control structure
- Object-oriented (and reactive) programming
  - mutable data structures / iteration
  - heavy use of functions (objects) as data
  - pervasive “abstraction by default”



Java and OCaml together

Dr. Weirich (teaches CIS 1200 at Penn)

Guy Steele, one of the principal designers of Java

Xavier Leroy, one of the principal designers of OCaml

Moral: Java and OCaml are not so far apart…

Recap: The Functional Style

- Core ideas:
  - immutable (persistent / declarative) data structures
  - recursion (and iteration) over tree structured data
  - functions as data
  - generic types for flexibility (i.e. ‘a list)
  - abstract types to preserve invariants (i.e. BSTs)
  - *simple model of computation (substitution)*

- Good for:
  - elegant descriptions of complex algorithms & data
  - compositional design
  - “symbol processing” programs
    (compilers, theorem provers, etc.)
  - reliable software / verification
  - parallelism, concurrency, and distribution

Functional programming

OCaml

- Immutable lists primitive, tail recursion
- Datatypes and pattern matching for immutable tree structured data
- First-class functions, transform and fold
- Generic types
- Abstract types through module signatures

Java

- No primitive data structures, no tail recursion
- Trees must be encoded by objects, mutable by default, limited pattern matching*
- First-class functions less common**
- Generic types***
- Abstract types through interfaces and public/private modifiers

*feature of Java 17 (released 2021)

**late addition, encoded from objects

***not completely "first class" (see, e.g., Arrays)

OCaml vs. Java for FP

OCaml

```ocaml
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let rec lookup (t:'a tree) (n:’a : bool =
  begin match t with
   | Empty -> false
   | Node(lt, x, rt) ->
        x = n ||
        if n < x then lookup lt n
        else lookup rt n
  end
```

OCaml provides a succinct, clean notation for working with generic, immutable, tree-structured data. Java requires more "boilerplate".

```java
public abstract sealed class
  Tree<A extends Comparable<A>>
          permits Tree.Empty, Tree.Node {

  final static class
    Empty<A extends Comparable<A>> extends Tree<A> {}

  final static class
    Node<A extends Comparable<A>> extends Tree<A> {
      final A v;
      final Tree<A> lt;
      final Tree<A> rt;
      public Node(Tree<A> lt, A value, Tree<A> rt) {
        this.lt = lt; this.rt = rt; this.v = v;
      }

  public static <A extends Comparable<A>>
    boolean lookup(A x, Tree<A> t) {
      if (t instanceof Node<A> n) {
        return switch (x.compareTo(n.value)) {
            case -1 -> lookup(x, n.left);
            case 1 -> lookup(x, n.right);
            default -> n.value.equals(x);
        };
      } else {
        return false;
      }
    }
}
```

Java

Other Popular Functional Languages

F#: Most similar to OCaml,
Shares libraries with C#

Haskell (CIS 5520)
Purity + laziness

Swift
iOS programming

Verse: Functional/Logic language for unreal engine

Racket: LISP descendant; widely used in education

Scala
Java / OCaml hybrid

Recap: The imperative style

- Core ideas:
  - computation as *change of state over time*
  - distinction between primitive and reference values
  - aliasing!
  - linked data-structures and iteration control structures
  - generic types for flexibility (i.e., ‘*a* queue)
  - abstract types to preserve invariants (i.e., queue invariant)
  - *Abstract Stack Machine model of computation*

interior of a pocket watch

- Good for:
  - high performance, low-level code
  - numerical simulations
  - implicit coordination between components (queues, GUI)
  - explicit interaction with hardware

Imperative programming

OCaml

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

Java

- Most types have a **null** element. Partial functions can return **null**.
- Code is a sequence of **statements** that have effects, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

Explicit vs. Implicit Partiality

OCaml identifiers

- Cannot be changed once created; only mutable fields can change

```
type 'a ref = { mutable contents: 'a }
let x = { contents = counter () }
;; x.contents <- counter ()
```

- Cannot be null, must use options

```
let y = { contents = Some (counter ())}
;; y.contents <- None
```

- Accessing option values requires pattern matching

```
;; begin match y.contents with
     | None -> failwith "NPE"
     | Some c ->  c.inc ()
   end
```

Java variables

- Can be assigned to after initialization

```
Counter x = new Counter ();
x = new Counter ();
```

- Can always be null

```
Counter y = new Counter ();
y = null;
```

- Check for null is implicit whenever a variable is used

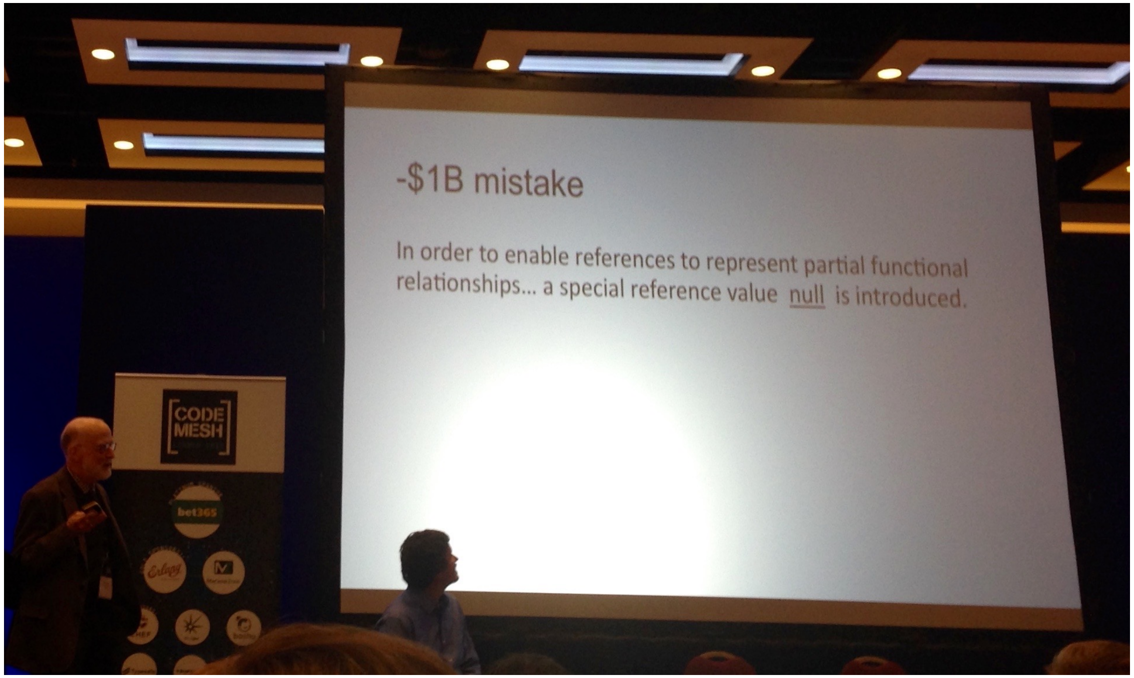
```
y.inc();
```

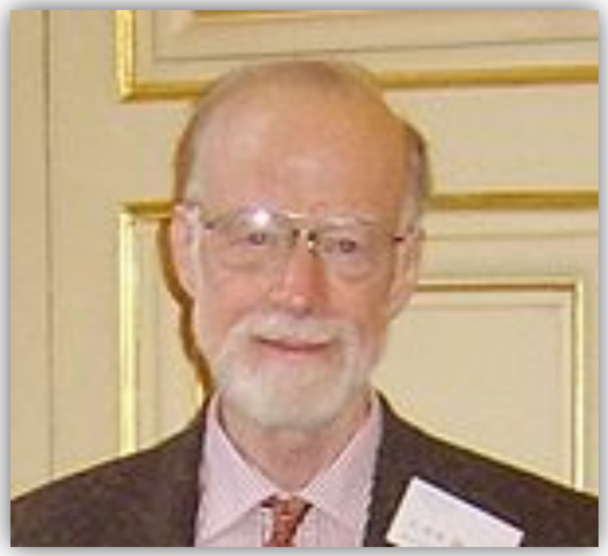
- If null is used as if it were an object (i.e. for a method call) then a **NullPointerException** occurs

The Billion Dollar Mistake

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. … This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. "

Sir Tony Hoare, QCon, London 2009





Java Core Language

differences between OCaml and Java

Structure of a Program

OCaml

- All code lives in (perhaps implicitly named) **modules**.
- Modules may contain multiple **type definitions**, **let**-bound value **declarations**, and top-level **expressions** that are executed in the order they are encountered.

Java

- All code lives in explicitly named **classes**.
- Classes are types (of objects).
- Classes contain **field declarations** and **method definitions**.
- There is a single "entry point" of the program where it starts running, which must be a method called `main`.

Expressions vs. Statements

- OCaml is an *expression language*
  - Every program phrase is an expression (and returns a value)
  - The special value () of type `unit` is used as the result of expressions that are evaluated only for their side effects
  - Semicolon is an *operator* that combines two expressions (where the left-hand one returns type unit)
- Java is a *statement language*
  - Two-sorts of program phrases: expressions (which compute values) and statements (which don't)
  - Statements are *terminated* by semicolons
  - Any expression can be used as a statement (but not vice-versa)
  - Some statements have expression variants (if, case)

Types

- As in OCaml, every Java *expression* has a type
- The type describes the value that an expression computes

Expression form	Example	Type
Variable reference	x	Declared type of variable
Object creation	new Counter ()	Class of the object
Method call	c.inc()	Return type of method
Equality test	x == y	boolean
Assignment	x = 5	don’t use as an expression!!

Type System Organization

	OCaml	**Java**
primitive types (values stored "directly" in the stack)	int, float, char, bool, …	int, float, double, char, boolean, …
structured types (a.k.a. *reference types* — values stored in the heap)	tuples, datatypes, records, functions, arrays (*objects encoded as records of functions*)	objects, arrays (*records, tuples, datatypes, strings, first-class functions are special cases of objects*)
generics	'a list	List<A>
abstract types	module types (signatures)	interfaces, abstract classes, public/private modifiers

Arithmetic & Logical Operators

<table>
<tr><th>OCaml</th><th>Java</th><th></th></tr>
<tr><td>=, ==</td><td>==</td><td>equality test</td></tr>
<tr><td><>, !=</td><td>!=</td><td>inequality</td></tr>
<tr><td>>, >=, <, <=</td><td>>, >=, <, <=</td><td>comparisons</td></tr>
<tr><td>+</td><td>+</td><td>addition</td></tr>
<tr><td>^</td><td>+</td><td>string concatenation</td></tr>
<tr><td></td><td></td><td>subtraction (and unary minus)</td></tr>
<tr><td>*</td><td>*</td><td>multiplication</td></tr>
<tr><td>/</td><td>/</td><td>division</td></tr>
<tr><td>mod</td><td>%</td><td>remainder (modulus)</td></tr>
<tr><td>not</td><td>!</td><td>logical “not”</td></tr>
<tr><td>&&</td><td>&&</td><td>logical “and” (short-circuiting)</td></tr>
<tr><td>||</td><td>||</td><td>logical “or” (short-circuiting)</td></tr>
</table>

Java: Operator Overloading

- The *meaning* of an operator in Java is determined by the *types* of the values it operates on:
  - Integer division

    `4/3` ⇒ `1`
  - Floating point division

    `4.0/3.0` ⇒ `1.3333333333333333`
  - Automatic conversion from int to float

    `4/3.0` ⇒ `1.3333333333333333`

- Method *overloading* is a general mechanism in Java
  - we'll see more of it later

Equality

- like OCaml, Java has two ways of testing reference types for equality:
  - "reference equality"

    o1 == o2
  - "deep equality"

    o1.equals(o2)

every object provides an "equals" method that should "do the right thing" depending on the class of the object

- Normally, you should use == to compare primitive types and ".equals" to compare objects

- Careful: Single-equals (=) means assignment, not equality comparison

Strings

- `String` is a *built in* Java class
- Strings are sequences of (unicode) characters

  `""`   `"Java"`   `"3 Stooges"`   `"富士山"`

- + means String concatenation (overloaded)

  `"3" + " " + "Stooges"` ⇒ `"3 Stooges"`

- Text in a String is immutable (like OCaml)
  - but variables that store strings are not
  - `String x = "OCaml";`
  - `String y = x;`
  - Immutability: can't do anything to `x` so that `y` changes
- The `.equals` method returns true when two strings contain the *same* sequence of characters

Aside: StringBuffers

- `StringBuffer` is a *mutable* Java String
- Alternative to "+" when constructing large strings

```
String s = "Hello";
for (int i=0; i<200; i++) {
   s = s + "!";
}
```

```
StringBuffer sb = new StringBuffer("Hello");
for (int i=0; i<200; i++) {
  sb.append("!");  // modify end of sb
}
String s = sb.toString();  // convert back to String
```

21: What is the value of *ans* at the end of this program?

0

true

0%

false

0%

NullPointerException

0%

What is the value of ans at the end of this program?

```
String x = "CIS 1200";
String z = "CIS 1200" ;
boolean ans = x.equals(z);
```

1. true
2. false
3. NullPointerException

Answer: true

This is the preferred method of comparing strings!

21: What is the value of *ans* at the end of this program?

0

true

0%

false

0%

NullPointerException

0%

What is the value of ans at the end of this program?

```
String x1 = "CIS ";
String x2 = "1200";
String x = x1 + x2;
String z = "CIS 1200";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: false
Even though x and z both contain the characters “CIS 1200”, they are stored in two different locations in the heap.

21: What is the value of *ans* at the end of this program?

✓✓ 0

true

0%

false

0%

NullPointerException

0%

What is the value of ans at the end of this program?

```
String x = "CIS 1200";
String z = "CIS 1200";
boolean ans = (x == z);
```

1. `true`
2. `false`
3. `NullPointerException`

Answer: true(!)
Why? Since strings are immutable, two identical strings that are known when the program is compiled can be aliased by the compiler (to save space).

Moral

Always use s1.equals(s2) to compare Strings!

Compare strings with respect to their content, not where they happen to be allocated in memory…

Object Oriented Programming

Preview: The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - dynamic dispatch ("receiver" of method call determines behavior)
  - classes ("templates" for object creation)
  - subtyping (grouping object types by common functionality)
  - inheritance (creating new classes from existing ones)

encapsulated state

- Good for:
  - GUIs
    - complex software systems that include many different implementations of the same "interface" (set of operations) with different behaviors
  - Simulations
    - designs with an explicit correspondence between "objects" in the computer and things in the real world
  - Games

"Objects" in OCaml

```
(* The type of counter objects *)
type counter = {
    inc  : unit -> int;
    dec  : unit -> int;
}

(* Create a counter "object" *)
let new_counter () : counter =
  let r = {contents = 0} in
  {
    inc = (fun () ->
      r.contents <- r.contents + 1;
      r.contents);
    dec = (fun () ->
      r.contents <- r.contents - 1;
      r.contents)
 }
```

Why is this an object?

- *Encapsulated local state* only visible to the methods of the object
- Object is *defined by what it can do*—local state does not appear in the interface
- There is a way to *construct* new object values that behave similarly

OO terminology

- *Object*: a structured collection of encapsulated *fields* (aka *instance variables*) and *methods*
- *Class*: a template for creating objects
- The class of an object specifies…
  - the types and initial values of its local state (fields)
  - the set of operations that can be performed on the object (methods)
  - one or more *constructors*: create new objects by (1) allocating heap space, and (2) running code to initialize the object (optional, but default provided)
- Every (Java) object is an *instance* of some class
  - Instances are created by invoking a constructor with the `new` keyword

OO programming

OCaml (part we've seen)

- Explicitly create objects using a record of higher order functions and hidden state

- Flexibility through ***composition***: objects can only implement one interface

```
type button =
    widget *
    label_controller *
     notifier_controller
```

Java (and C, C++, C#)

- Primitive notion of object creation (classes, with fields, methods and constructors)

- Flexibility through ***extension***: ***Subtyping*** allows related objects to share a common interface

```
class Button extends Widget {
  /* Button is a subtype
     of Widget */

}
```

Objects in Java

```java
public class Counter {

    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }

    public int dec () {
        r = r - 1;
        return r;
    }
}
```

class declaration

class name

instance variable

constructor

methods

object creation and use

```java
public class Main {

    public static void
        main (String[] args) {

            Counter c = new Counter();

            System.out.println( c.inc() );

    }
}
```

constructor invocation

method call

Encapsulating local state

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

r is *private*

constructor and methods can refer to r

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

      System.out.println( c.inc() );

    }
}
```

other parts of the program can only access public members

method call

Encapsulating local state

- *Visibility modifiers* make the state local by controlling access
- Basically*:
  - public : accessible from anywhere in the program
  - private : only accessible inside the class
- Design pattern — first cut:
  - Make *all* fields private
  - Make constructors and non-helper methods public

*Java offers a couple of other protection levels — “protected” and “package protected” for structure larger code developments and libraries. The details are not important at this point.

Constructors with Parameters

```
public class Counter {

  private int r;

  public Counter (int r0) {
    r = r0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```
public class Main {

  public static void
     main (String[] args) {

       Counter c = new Counter(3);

       System.out.println( c.inc() );

    }
}
```

constructor invocation

Creating Objects

- *Declare* a variable to hold a `Counter` object
  - Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for `Counter` to create a `Counter` instance with keyword "new" and store it in the variable

```
Counter c = new Counter();
```

Creating Objects

- Every Java variable is mutable

```
Counter c = new Counter(2);
c = new Counter(4);
```

- A Java variable of *reference* type can also contain the special value "null"

```
Counter c = null;
```

Remember!
Single = for assignment
Double == for reference equality testing

What is the value of ans at the end of this program?

```
Counter x;
x.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Raises `NullPointerException`

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

Answer: NullPointerException

What is the value of ans at the end of this program?

```java
Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. NullPointerException

```java
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

Answer: 3