Programming Languages and Techniques (CIS120)

Lecture 20

October 19th, 2015

Wrapping up GUIs; Transition to Java

Announcements

HW05: GUI programming is available

- Due: THURSDAY OCT. 22 at 11:59:59pm
- *Graded manually*
  - *Submission only checks for compilation, no auto tests*
  - *Won't get scores immediately*
  - *Only LAST submission will be graded*
- This project is challenging:
  - Requires working with *multiple* levels of abstraction.
  - Managing state in the paint program is a bit tricky.

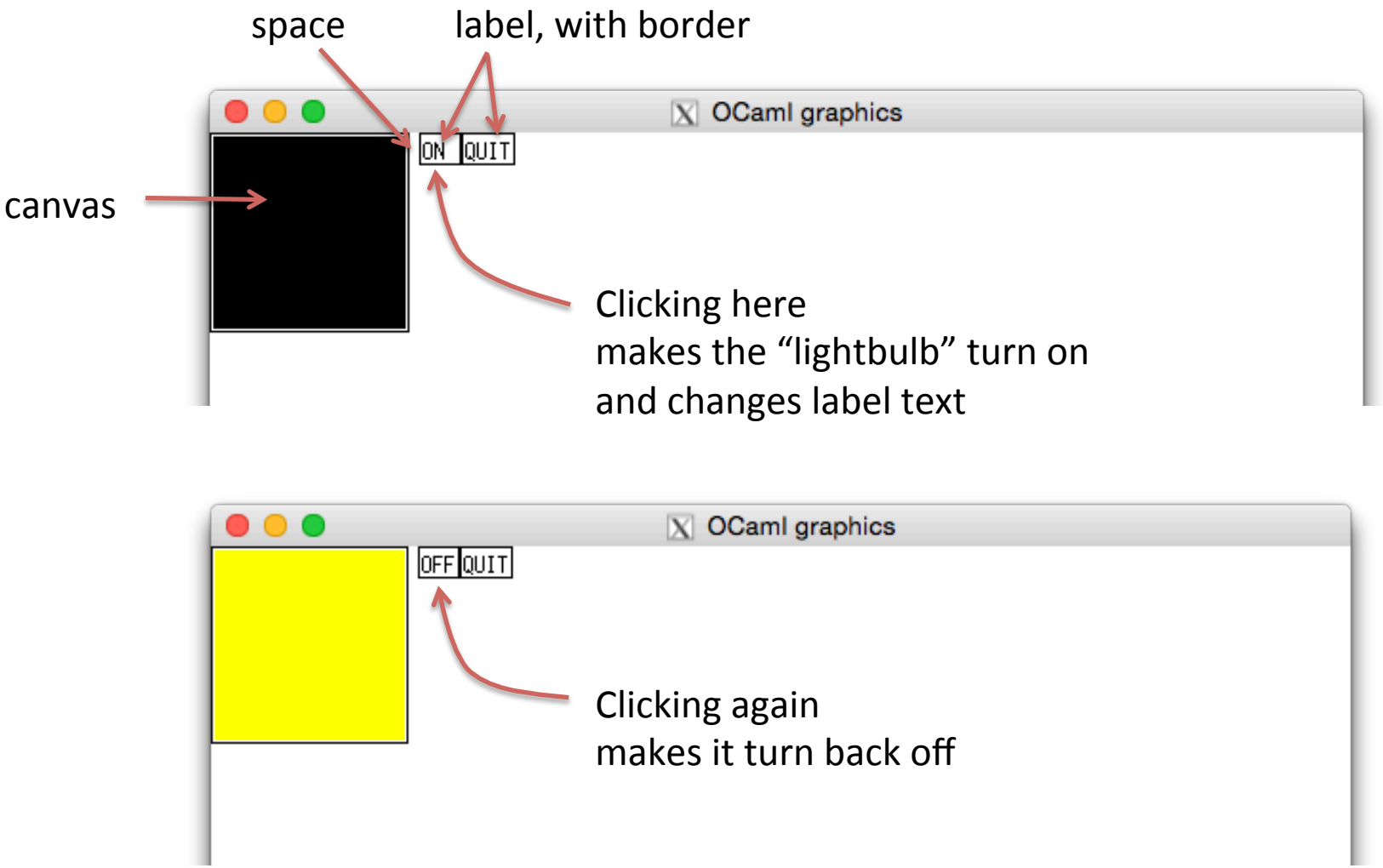
How far are you on HW 5?

1. Haven't started yet
2. Working on Tasks 1-4 (layout, drawing)
3. Working on Checkboxes
4. Working on Something Cool
5. I'm done!

Demo: lightswitchdemo.ml

Putting it all together.

lightbulb demo



Looking Back…

Course 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”

Recap: The Functional Style

- Core ideas:
  - value-oriented programming
  - 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)
- Good for:
  - simple, elegant descriptions of complex algorithms and/or data
  - parallelism, concurrency, and distribution
  - “symbol processing” programs (compilers, theorem provers, etc.)

Language Support for FP

- "Functional languages" (OCaml, Standard ML, F#, Haskell, Scheme, Clojure) promote this style as a default and work hard to implement it efficiently
- "Hybrid languages" (Scala, Python) offer it as one possibility among others
- "Object Oriented" languages (Java, C#, C++, Objective C) favor a different style by default
  - But many common OO idioms and *design patterns* have a functional flavor (e.g. the "Visitor" pattern is analogous to transform)
  - And most of them are gradually adding features (like anonymous functions) that make functional-style programming more convenient
  - Best practices discourage use of imperative state

Functional programming

OCaml

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

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

- No primitive data structures, no tail recursion
- Trees must be encoded by objects
- No first-class functions.* Must encode first-class computation with objects
- Generic types
- Abstract types through public/private modifiers

*until recently, in Java 8

OCaml vs. Java

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

let is_empty (t:'a tree) =
  begin match t with
    | Empty -> true
    | Node(_,_,_) -> false
  end

let t : int tree = Node(Empty,3,Empty)
let ans : bool = is_empty t
```

```
interface Tree<A> {
  public boolean isEmpty();
}
class Empty<A> implements Tree<A> {
  public boolean isEmpty() {
     return true;
  }
}
class Node<A> implements Tree<A> {
  private final A v;
  private final Tree<A> lt;
  private final Tree<A> rt;

  Node(Tree<A> lt, A v, Tree<A> rt) {
    this.lt = lt; this.rt = rt; this.v = v;
  }

  public boolean isEmpty() {
    return false;
  }
}

class Program {
  public static void main(String[] args) {
    Tree<Integer> t =
    new Node<Integer>(new Empty<Integer>(),
     3, new Empty<Integer>());
    boolean ans = t.isEmpty();
  }
}
```

Recap: Imperative programming

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

- Good for:
  - numerical simulations
  - implicit coordination between components

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 (and C, C++, C#)

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

Recap (and coming): The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - classes ("templates" for object creation)
  - dynamic dispatch ("receiver" of method call determines behavior)
  - subtyping (grouping object types by common functionality)
  - inheritance (creating new classes from existing ones)
- Good for:
  - GUIs!
    - and other complex software systems that include many different implementations of the same "interface" (set of operations) with different behaviors (cf. widgets)
  - Simulations
    - designs with an explicit correspondence between "objects" in the computer and things in the real world

OO programming

OCaml

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

- Flexibility through composition: objects can only implement one interface
  (i.e. 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
  (i.e. button <: widget)

Java and OCaml together

Dr. Stephanie Weirich
(teaches CIS 120 in the Spring)

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…

Looking Forward

Today: Objects, Classes and Interfaces in Java

How comfortable do you feel working with objects and classes?

1. I'm not sure what they are
2. I get the basics
3. Fairly comfortable
4. I've done a lot of OO programming

Do you understand the concept of Dynamic Dispatch?

1. Never heard of it
2. I've heard of it, but I'm not sure what it means
3. I think I know what I'm doing
4. I could implement an OO-language

Smoothing the transition

- DON'T PANIC

- Ask questions, but don't worry about the details until you need them.

- Java resources:
  - Lecture notes and lecture slides
  - Online Java textbook (http://math.hws.edu/javanotes/) linked from "CIS 120 Resources" on course website
  - Penn Library: Electronic access to "Java in a Nutshell" (and all other O'Reilly books)
  - Piazza!

Objects

from OCaml to Java

"Objects" in OCaml

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

(* Create a counter "object" with
hidden state: *)
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

Java Objects and Classes

- *Object*: a structured collection of *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*: code that is executed when the object is created (optional)
- Every Java object is an *instance* of some class

Objects in 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 name

class declaration

instance variable

constructor

methods

object creation and use

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

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

   }
}
```

constructor invocation

method call