CIS 120 Final Exam May 7, 2014

SOLUTIONS

1. **Design Process** (8 points)

   List the four steps of the "design process" (or "recipe") that we used throughout the semester.

   a. *Understand the problem (and how the concepts relate to each other)*

   b. *Define the interface*

   c. *Write tests*

   d. *Implement (i.e., write the code)*

   Grading Scheme: 2 points per step.

2. True or False (10 points)

a. T [✓ F] In OCaml, if `x` is a variable of any type, `Some x == Some x` will always return true.

b. [✓ T] F In OCaml, data structures such as records and datatypes are immutable by default.

c. T [✓ F] Binary search trees can *only* be implemented in OCaml.

d. T [✓ F] Every mutable reference in OCaml could be `null`.

e. T [✓ F] In Java, a static method dispatch `C.m()` implicitly pushes the `this` reference onto the stack

f. [✓ T] F In Java, if an exception is thrown but not caught, it immediately terminates the program.

g. T [✓ F] In Java, any method that could throw a `NullPointerException` must include the clause **`throws`** `NullPointerException` in the method header.

h. [✓ T] F In the Java ASM, the dynamic class of an object is the name of the class that was used to create it.

i. T [✓ F] In the Java ASM, references can point to objects stored in the heap or on the stack.

j. [✓ T] F Two references are called *aliases* if they point to the same location in the heap.

Grading Scheme: 1 pt each.

3. Binary Search Trees (15 points)

Recall the type definitions for binary trees from Homework 3:

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

Also recall that binary search trees (BST) are trees with an additional invariant (which hopefully you remember), and recall the `insert` function for BSTs:

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

Circle the tree that corresponds to `ans` after the execution of each code snippet. *Also* circle YES if `ans` satisfies the BST invariant and NO otherwise.

a.
```
let t : tree = Node(Empty, 2, Empty)
let ans : tree = insert 3 t
```

```
  2            3          2       3
   \          /          /         \
    3        2          3           2

           YES         NO
```

The first one, YES

b.
```
let t : tree = Node(Empty, 4, Node(Empty, 4, Empty))
let ans : tree = insert 5 t
```

```
  4            5       4                4
   \          /         \                \
    5        4           4                5
                          \              /
                           5            4

           YES         NO
```

The third one, NO

c.
```
let t : tree = Node(Empty, 2, Node (Empty, 3, Empty))
let ans : tree = insert 3 t
```

```
      2         3       2            2
       \       / \       \            \
        3     2   3       3            3
                           \          /
                            3        3

              YES          NO
```

The first one, YES

d.
```
let t : tree = Node(Node(Empty, 2, Empty), 5, Node(Empty, 7, Empty))
let ans : tree = insert 4 (insert 3 t)
```

```
      5             5               2                 5
     / \           / \               \               / \
    3   5         2   7               3             2   7
   / \             \                   \             \
  2   4             3                   4             4
                     \                   \           /
                      4                   5         3
                                           \
                                            7

              YES          NO
```

The second one, YES

e.
```
let t : tree = Node(Node(Empty, 7, Empty), 5, Node(Empty, 8, Empty))
let ans : tree = insert 7 (insert 6 t)
```

```
      5             5               5                 5
     / \           / \               \               / \
    7   8         7   8               6             6   8
   /   /               /               \                 /
  6   7            6                    7             7
                    \                    \
                     7                    8

              YES          NO
```

The second one, NO

Grading Scheme: 3pts each, 2 pts for shape, 1 pt for invariant

4. Immutable Sets and Maps (14 points)

Consider the following OCaml modules, `Set` and `Map`, that are described by the module types `SET` and `MAP` shown in Appendix A. Although the implementations of these modules is not shown, you may assume that they implement *immutable* sets and finite maps as in Homework 3.

```
module Set : SET = struct ... end
module Map : MAP = struct ... end
```

Circle the value of the following OCaml expressions, or *Ill-typed* if the expression would not type check.

a.
```
let s1 = Set.add "Happy" [] in
Set.size s1
```

0 1 2 **true** **false** [✓ Ill-typed]

b.
```
let s1 = Set.add "Happy" Set.empty in
let s2 = Set.add "Birthday" s1 in
Set.member "Birthday" s1
```

0 1 2 **true** [✓ **false**] Ill-typed

c.
```
let s1 = Set.add "Happy" Set.empty in
let s2 = Set.add "Birthday" s1 in
Set.member "Happy" s2
```

0 1 2 [✓ **true**] **false** Ill-typed

d.
```
let s1 = Set.add "Happy" Set.empty in
let s2 = Set.add true s1 in
Set.size s2
```

0 1 2 **true** **false** [✓ Ill-typed]

e.
```
let s1 = Set.add "Happy" Set.empty in
let s2 = Set.add "Happy" s1 in
Set.size s2
```

0 [✓ 1] 2 **true** **false** Ill-typed

f.
```
let s1 = Set.add "Birthday" Set.empty in
let m1 = Map.add "Happy" s1 Map.empty in
let m2 = Map.add "Happy" Set.empty m1 in
Set.size (Map.find "Happy" m2)
```

[✓ 0] 1 2 **true** **false** Ill-typed

g.
```
let s1 = Set.add "Birthday" Set.empty in
let m1 = Map.add "Happy" s1 Map.empty in
let s2 = Set.add "Party" s1 in
let m2 = Map.add "It's my" s2 m1 in
Set.size (Map.find "Happy" m2)
```

0 [✓ 1] 2 **true** **false** Ill-typed

Grading Scheme: 2pts each

5. **Program Design (Arrays)** (16 points)

Implement a static method called `trimArray` that, when given an **`int`** n and an **`int`** array x, returns a new array with the same contents as x except for all occurrences of n at the *beginning* of the array.

For example, `trimArray( 1, new int[] {1,1,2} )` should yield a new array just containing a single element, `2`, and `trimArray (1, new int[] {2,1,2,1})` should return an identical array to the input, because there are no `1`s at the beginning of the array.

If the input array is **`null`**, this method should return **`null`**. In particular, it should never throw a `NullPointerException`.

```
public static int[] trimArray (int n, int[] x) {
   if (x == null) {
      return null;
   }
   int i;
   for (i = 0; i < x.length && x[i] == n; i++ );

   int[] y = new int[x.length - i];

   for (int j=0; j < y.length; j++) {
      y[j] = x[j+i];
   }
   return y;
}
```

Grading Scheme: 16 points total, rough guidelines

- *2 pt – method declaration correct takes (int, int[]), returns int[]*
- *2 pt – null check*
- *3 pt – identifies the index of the first value not equal to n*
- *2 pt – allocates the new array*
- *2 pt – new array has correct size*
- *3 pt – copies the values over, with shifting*
- *2 pt – returns new array*
- *other deductions at discretion*

6. Types (10 points)

Consider the classes and interfaces `Iterable`, `Collection`, `List`, `Iterator`, `ArrayList` and `LinkedList` from the Java Collections Framework. (More information about these interfaces and classes is shown in Appendix B.)

Write down a type for each of the following Java variable definitions. Due to subtyping, there may be more than one type that would work—you should put the **most specific type** permissible in that case. (For example, if `C implements I` then you should put `C` instead of `I`). Write **ill-typed** if the compiler would flag an error anywhere in the code snippet.

Hint: read through the Appendix carefully. Even though you may have used these methods before, you may not have given much thought to their types.

The first two have been done for you as a sample and are used in the remaining definitions.

```
________ArrayList<String>___________ x = new ArrayList<String>();

____ArrayList<List<String>>________ y = new ArrayList<List<String>>();
```

a. `____Iterator<String>___ a = x.iterator();`

b. `_______String____________ b = x.iterator().next();`

c. `____ill-typed___________ c = x.next();`

d. `_____List<String>_______ d = x.subList(1,2);`

e. `_____String_____________ e = x.get(2);`

f. `_____boolean____________ f = x.contains(x);`

g. `_____List<String>_______ g = y.get(0);`

h. `_____boolean____________ h = y.add(x);`

i. `____ill-typed___________ i = x.add(x);`

j. `_____boolean____________ j = x.contains("Whew!");`

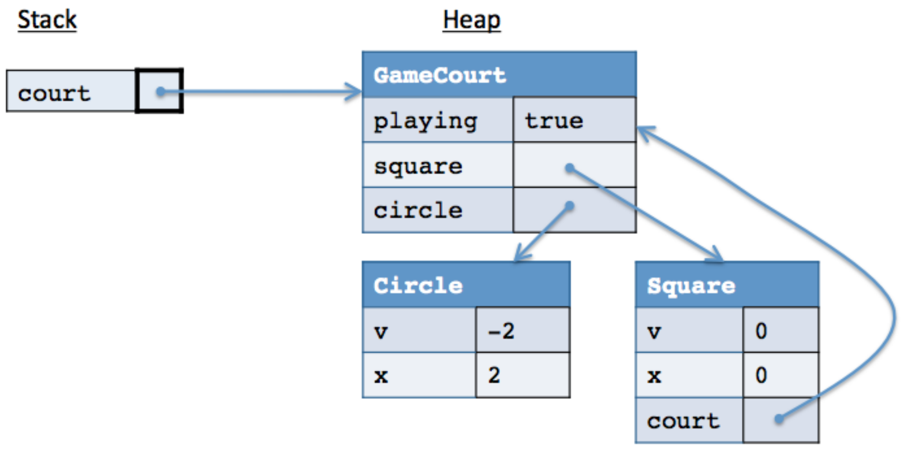
Grading Scheme: 2pts each

7. Java ASM (14 points)

Consider the classes `GameCourt` and `GameObj`, shown in Appendix C. These classes are loosely based on the *Mushroom of Doom*.

a. Draw the stack and the heap of the Java Abstract Stack Machine after the following code has executed on the workspace:

```
GameCourt court = new GameCourt();
```



Grading Scheme: 8 points total:

- *1 stack var "court"*
- *2 GameCourt object*
- *2 Circle object*
- *3 Square object*

Note, all boxes are mutable so should be marked. But ok to omit.

b. Now suppose that the following code is placed on the workspace, picking up where the above computation left off.

```
court.tick();
```

What are the values of the following Java expressions after this code executes?

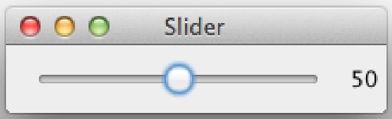
`court.playing`	false
`court.square.x`	0
`court.circle.x`	0

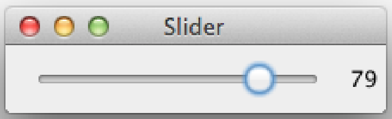
Grading Scheme: 2 points per blank

8. Reactive programming and Swing (13 points)

A `JSlider` is a Swing component that allows users to set a percentage by moving a bar. For example, the window on the left displays a slider that starts with an initial value of 50 percent. The slider can be used to select values in the range 0 to 100.

When the user drags the slider, it changes the value of the slider to a new percentage, as shown in the window on the right. In each window, to the right of each slider is a `Jlabel` that displays the current value of the slider. The label updates whenever the slider is changed.





This question asks you to complete the implementation of the simple application described above.

For reference, documentation for various components of the Swing library (including `JSlider` and `JLabel`) appears in Appendix D. Recall that you can convert an **`int`** to a string using the static method `Integer.toString(`**`int`**` x)`.

Hint: you will need to define an object that implements the `ChangeListener` *interface. You may do so with either a separate class or an anonymous inner class.*

Hint: all of the information you need to complete this problem is contained in the Appendix. In particular, you do not need to implement the `MouseListener` *interface.*

(See next page.)

```
public class Main {
  public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {

          @Override
          public void run() {
             JFrame frame = new JFrame("Slider");
             final JPanel panel = new JPanel();
             frame.setContentPane(panel);

             // Add any necessary code here. You may also define any
             // helper classes (on the next page) if you wish.

              final JLabel lab = new JLabel();
              final JSlider slider = new JSlider();
              final int INITIAL_VALUE = 50;
              slider.setValue(INITIAL_VALUE);
              label.setText(Integer.toString(INITIAL_VALUE));

              slider.addChangeListener(new ChangeListener() {
                public void stateChanged(ChangeEvent e) {
                  int v = slider.getValue();
                  String s = Integer.toString(v);
                  lab.setText(s);
                }
              });
              panel.add(slider);
              panel.add(lab);

             frame.pack();
             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
             frame.setVisible(true);
          }
      });
   }
}
```

(More space for question 8, if necessary).

An alternative answer defines the following external class, but must pass the slider and label references to the constructor.

```
class C implements ChangeListener {
  JSlider slider;
  JLabel lab;

  public C(JSlider s, JLabel l) {
    this.slider = s; this.lab = l;
  }

  public void stateChanged(ChangeEvent e) {
    int v = slider.getValue();
    String s = Integer.toString(v);
    lab.setText(s);
  }
}
```

Grading Scheme: 13 points total

a. *1 declare and initialize slider*

b. *1 declare and initialize label*

c. *1 set initial label text to "50"*

d. *2 stateChanged method can access to the slider and label references (by making them final, or making them instance vars of separate class).*

e. *2 in stateChanged method, get value of the slider*

f. *2 in stateChanged method, convert int to String*

g. *2 in stateChanged method, set label text*

h. *1 add change listener*

i. *1 add slider and label to panel*

9. Object encodings and Collections (20 points)

Recall that in HW 06, we defined a GUI library for representing widgets, and that in HW 08, we translated this library to Java. In particular, we represented OCaml values of type:

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

with Java classes that implement the following interface:

```
public interface Widget {
   public void repaint (Gctx gc);
   public void handle (Gctx gc, Event e);
   public Dimension minSize();
}
```

Because we had not covered Java collections at that point in the semester, HW 08 simplified the treatment of `Notifier` widgets. In that assignment, notifiers stored at most one event listener. However, in the OCaml version (shown in Appendix E), notifier widgets are more flexible; they store a *list* of event listeners. In that case, when an event occurs, the listeners processes the event in the order that they were added—the first listener to "finish" the event terminates the iteration. If none of the listeners finishes the event, then, and only then, is the inner widget notified of the event.

Your job is to implement a Java `Notifier` widget with this functionality, by translating the OCaml "object" shown in the appendix to Java classes and interfaces. Each method of your new `Notifier` class should do the "same thing" as the corresponding part of the OCaml version, with two difference:

- As in HW 08, the `minSize` method takes no arguments. However, like `size` in `notifier` it should just delegate to the inner widget.
- Instead of using OCaml lists, you should use classes from the Java Collections framework (see Appendix B).

To get started, consider the interface below, which is the analogue to the OCaml `event_listener` type. For brevity on the exam, we call it `EL`, short for `EventListener`.

```
/** A listener processes events; it returns true if the event has been
completely taken care of by the listener, and false if the event should
be propagated to other listeners along a widget's event handlers. */
interface EL {
    public boolean listen(Gctx gc, Event e);
}
```

Your first step is to complete the `NotifierController` interface corresponding to the `notifier_controller` type in the OCaml version.

```
/** A NotifierController is associated with a notifier widget.
   It allows the program to add event listeners to the notifier */
interface NotifierController {
        public void addEventListener(EL el);
}
```

Grading Scheme: 2pts for this blank

Next, below and on the next page, complete the `Notifier` class so that it implements the required interfaces.

Note: For simplicity on the exam, you may assume that all arguments to methods are non-null.

```
public class Notifier implements NotifierController, Widget {

  // instance variables
  private final Widget w;
  private final List<EventListener> listeners;

  // constructor
  public Notifier(Widget w) {
    this.w = w;
    this.listeners = new LinkedList<EL>();
  }

  // Required method for NotifierController interface
  public void addEventListener(EL el){
    listeners.add(el);
  }
```

(Implementation of `Notifier` **`class`** `continued`)

```
  // Required methods for Widget interface
  public void repaint(Gctx gc) {
    w.repaint(gc);
  }
  public Dimension minSize() {
    return w.minSize();
  }
  public void handle(Gctx gc, Event e) {
    Iterator<EventListener> it = listeners.iterator();
    boolean finished = false;
    while (it.hasNext() && !finished) {
      finished = it.next().listen(gc, e);
    }
    if (!finished) {
      w.handle(gc,e);
    }
  }



}
```

Grading Scheme:

- *must have instance variable w*
- *must have instance variable list for listeners*
- *instance variables must be private (may or may not be final)*
- *constructor must take widget as argument*
- *must initialize internal state*
- *interface of addEventlistener must match*
- *addEventListener must add to collection*
- *repaint, minsize must delegate*
- *handle calls listen on at least one event listener*
- *(2pts) stops when first listener is “finished”*
- *(2pts) only calls handle for w if no listener finishes*

A Sets and Maps in OCaml

```
module type SET = sig
  type 'a set
  (* The type of (generic) sets *)

  val empty : 'a set
  (* The empty set *)

  val add : 'a -> 'a set -> 'a set
  (* add x s returns a set containing all elements of s, plus x. *)
  (* If x was already in s, s is returned unchanged. *)

  val member : 'a -> 'a set -> bool
  (* member x s tests whether x belongs to the set s. *)

  val size : 'a set -> int
  (* size s returns the number of elements contained in the set s *)

end

module type MAP = sig
 type ('k,'v) map
 (* The type of maps from key 'k to values 'v *)

 val empty : ('k,'v) map
 (* The empty map *)

 val add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
 (* add x y m returns a map containing the same bindings as m,  *)
 (* plus a binding of x to y. If x was already bound in m, its  *)
 (*  previous binding disappears. *)

 val mem : 'k -> ('k,'v) map -> bool
 (* mem x m returns true if m contains a binding for x, and *)
 (* false otherwise. *)

 val find : 'k -> ('k,'v) map -> 'v
 (* find x m returns the current binding of x in m, or fails *)
 (* if no such binding exists. *)
end
```

B Excerpt from the Collections Framework

```
interface Iterator<E> {
  public boolean hasNext();
  // Returns true if the iteration has more elements.
  public E next();
  // Returns the next element in the iteration .
  public void remove();
  // Removes from the underlying collection the last element
  // returned by this iterator
}

interface Iterable<E> {
  public Iterator<E> iterator();
  // Returns an iterator over a set of elements of type E.
}

interface Collection<E> extends Iterable<E> {
  public boolean add(E o);
  // Ensures that this collection contains the specified element
  // Returns true if this collection changed as a result of the call .
  // (Returns false if this collection does not permit duplicates
  // and already contains the specified element.)
  public boolean contains(Object o);
  // Returns true if this collection contains the specified element.
}

interface List<E> extends Collection<E> {
  public E get(int i);
  // Returns the element at the specified position in this list
  public List<E> subList(int fromIndex, int toIndex);
  // Returns a view of the portion of this list between the
  // specified fromIndex, inclusive , and toIndex, exclusive .
}

class ArrayList<E> implements List<E> {
   public ArrayList();
   // Constructs an empty list with an initial capacity of ten

   // ... methods specified by interface
}

class LinkedList<E> implements List<E> {
   public LinkedList();
   // Constructs an empty list

   // ... methods specified by interface
}
```

C GameCourt and GameObj classes

```
class GameCourt {
  public Square square;
  public Circle circle;
  public boolean playing;

  public GameCourt() {
    square = new Square(this);
    circle = new Circle();
    playing = true;
  }

  public void tick() {
    circle.move();
    square.move();
  }
}
class GameObj {
  public int v; // velocity of the  object
  public int x; // position of the  object

  public GameObj(int v, int x) {
    this.v = v;
    this.x = x;
  }
  public void move(){
    x = x + v;
  }
}
class Square extends GameObj {
  GameCourt court;
  public Square(GameCourt c) {
    super(0,0);
    court = c;
  }
  public void move() {
    super.move();
    if (x == court.circle.x) {
      court.playing = false;
    }
  }
}
class Circle extends GameObj {
  public Circle() {
    super(-2,2);
  }

}
```

D Swing class documentation

```
class JSlider extends JComponent {
  public JSlider() { ... }
  // Creates a horizontal slider with the range 0 to 100 and
  // an initial value of 50
  public void addChangeListener(ChangeListener l) { ... }
  // Adds a ChangeListener to the slider
  public int getValue() { ... }
  // Returns the slider's current value
  public void setValue(int n)
  // Sets the slider's current value to n
}

public interface ChangeListener {
   public void stateChanged(ChangeEvent e);
   // Invoked when the target of the listener has changed its state.
}

public class ChangeEvent {
   public Object getSource() { ... }
   // The object on which the Event initially occurred.
}

public class JLabel extends JComponent {
   public JLabel() { ... }
   // Creates a JLabel instance with an empty string for the title.
   public JLabel(String text) { ... }
   // Creates a JLabel instance with the specified text.

   public String getText() { ... }
   // Returns the text string that the label displays
   public void setText(String text) { ... }
   // Defines the single line of text this component will display.
}

public class JPanel extends JComponent {
  public JPanel() { ... }
  // Creates a new JPanel with a flow layout.

  public void add(JComponent comp) { ... }
  // Appends the specified component to the end of this container.
}
```

E Notifiers

```
type event_listener_result =
  | EventFinished
  | EventNotDone

(** A listener processes events; it returns EventFinished if the event
    has been completely taken care of by the listener, and EventNotDone
    if the event should be propagated to other listeners along a widget's
    event handlers. *)

type event_listener = Gctx.gctx -> Gctx.event -> event_listener_result

(** A notifier_controller is associated with a notifier widget.
    It allows the program to add event listeners to the notifier.
*)
type notifier_controller = {
  add_event_listener: event_listener -> unit
}

(** A notifier widget is a widget "wrapper" that doesn't take up any
    extra screen space -- it extends an existing widget with the
    ability to react to events. It maintains a list of of "listeners"
    that eavesdrop on the events propagated through the notifier
    widget.

    When an event comes in to the notifier, it is passed to each
    event_listener in turn until one of them declares the event
    to be "finished".
*)
let notifier (w: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
   { repaint = w.repaint;
     handle =
       (fun (g:Gctx.gctx) (e: Gctx.event) ->
         let rec loop (l: event_listener list) : unit =
           begin match l with
            | [] -> w.handle g e
            | h::t -> begin match h g e with
                  | EventFinished -> ()
                  | EventNotDone -> loop t
                end
           end in
         loop listeners.contents;
     size = w.size
   },
  { add_event_listener =
     fun (newl:event_listener) ->
       listeners.contents <- listeners.contents @ [newl]
  }
```