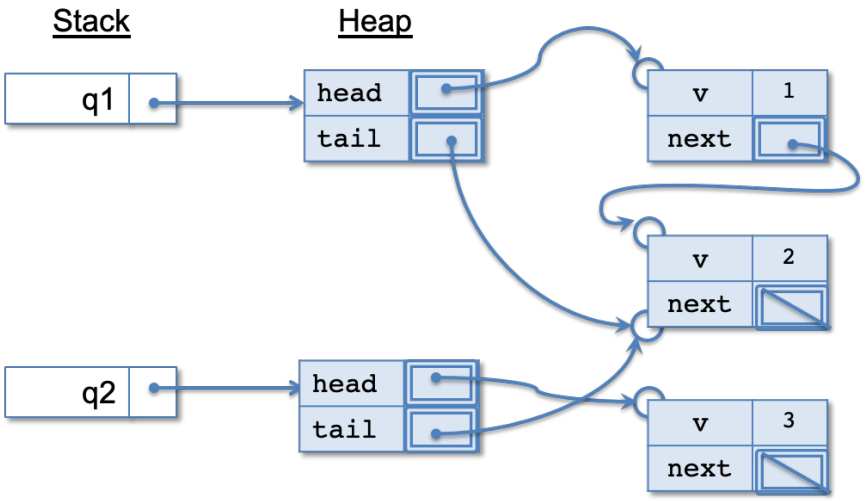
CIS 120 Midterm 2 November 13, 2020

SOLUTIONS

1. **Mutable Queues (invariants, mutable state, ASM)** (30 points total)

   Appendix A contains the implementation of the singly-linked queue data structure from Homework 4, including the (correct) implementations of the `create`, `enq`, and `deq` operations. It also summarizes the (singly-linked) queue invariants.

   Consider the Stack and Heap for the ASM shown below.



   (a) (2 points) Does the queue `q1` satisfy the queue invariants?

   ☒ Yes ☐ No

   (b) (2 points) Does the queue `q2` satisfy the queue invariants?

   ☐ Yes ☒ No

   (c) (7 points) Starting from an empty Stack and Heap, write code that will produce the ASM shown above. You can create additional stack bindings not displayed in the ASM diagram above, but you cannot create additional bindings in the heap.

   For this part, you ***must only*** create or modify records directly and ***cannot*** use any of the provided queue operations (`create`, `enq`, and `deq`).

   If it's not possible to create the ASM with these constraints, explain why.

```
let qn2 = {v = 2; next = None}

let q1 : 'a queue =
  {
  head = Some {v = 1; next = Some qn2};
  tail = Some qn2
  }

let q2 : 'a queue =
  {
```

```
head = Some {v = 3; next = None};
tail = q1.tail
}
```

(d) (7 points) Starting from an empty Stack and Heap, write code that will produce the ASM shown above. You can create additional stack bindings not displayed in the ASM diagram above, but you cannot create additional bindings in the heap.

For this part, you ***cannot*** create or modify records directly and ***must only*** use the provided queue operations (`create`, `enq`, and `deq`).

If it's not possible to create the ASM with these constraints, explain why.

Answer: Since `q2` *does not satisfy the queue invariants, it's not possible to create an invalid queue using code that always satisfies the queue invariants. Further, given the code implementation in the Appendix, it's not possible to create two different queues that share the same node.*

(e) (12 points) Given a valid queue, write a `delete_alternate` function that deletes every *alternate* element from that queue after the head. (That is, it would delete the 2nd, 4th, 6th, etc. elements from the queue.). E.g., if the queue contains the elements `1 -> 2 -> 3 -> 4` where the `head` is at 1 and the `tail` is at 4, the resulting queue will be `1 -> 3` where the `head` is still at 1 and the `tail` is now at 3.

The queue should be updated in-place and satisfy the queue invariants at the end. The function should be tail-recursive.

```
let delete_alternate (q: 'a queue) : unit =
  (* Hint: Do you need a recursive helper function? *)
  let rec loop (qn: 'a qnode option) : unit =
    begin match qn with
    | None -> ()
    | Some curr -> begin match curr.next with
                   | None -> ()
                   | Some n -> curr.next <- n.next;
                        if n.next = None then q.tail <- qn
                        else loop n.next
                    end
    end

  in loop q.head
```

2. **OCaml Programming: Encapsulation and Objects** (23 points total)

Recall the definition of OCaml's generic reference type:

```
type 'a ref = { mutable contents : 'a }
```

For this question, you'll implement two versions of the `counter` type defined below:

```
type counter = {
  get   : unit -> int;
  incr  : unit -> unit;
}
```

(a) (9 points) Version 1 with *NO* encapsulation

The first implementation will have no encapsulation for the internal state. Implement this version such that the code that follows will print out values as indicated.

```
(* Version 1 with NO encapsulation *)
let v : int ref = {contents = 0}

let make_counter_1 () : counter =
  {
    get = (fun () -> v.contents) ;
    incr = (fun () -> v.contents <- v.contents + 1) ;
  }

let ctr1 = make_counter_1 ()

(* this line will print 0 *)
;; print_endline (string_of_int (ctr1.get()))

;; ctr1.incr()
;; ctr1.incr()

(* this line will print 2 *)
;; print_endline (string_of_int (ctr1.get()))
```

Write a line of code that modifies the internal state, *without calling* `get` *or* `incr`, such that the following line *must* print a value other than 2.

```
(* some code unrelated to type counter (i.e., not get or incr) *)

;; v.contents <- 10

(* this line must print a value other than 2 *)
;; print_endline (string_of_int (ctr1.get()))
```

(b) (14 points) Version 2 with encapsulated coupled counters

The second implementation will have encapsulation for the internal state as follows:

- It will return *2 counters*, say `ctr1` and `ctr2`, and a function.
- The two counters will maintain their independent internal counts.
- The two counters are "coupled", which means that only one counter is unlocked at any given time. If a counter is unlocked, when `incr` is called on it, it will increment the value. If a counter is locked, when `incr` is called on it, it will do nothing.
- The function returned (i.e., the third item in the tuple) has type `unit -> unit`. When it is called, it will toggle which counter is locked and which is unlocked. E.g., if `ctr1` is locked (and therefore, `ctr2` is unlocked), calling toggle will result in `ctr1` being unlocked and `ctr2` being locked.
- The default initial values for the counts for both counters is **0.**
- The default locking state is that **`ctr1` is locked** and **`ctr2` is unlocked.**

```
(* Version 2 with encapsulated coupled counters *)

let make_counter_2 () : counter * counter * (unit -> unit) =
  let v1 : int ref = {contents = 0} in
  let v2 : int ref = {contents = 0} in
  let state : bool ref =
    {contents = true} in (* true "locks" first counter *)
  {
    get = (fun () -> v1.contents) ;
    incr = (fun () -> if (not state.contents)
                      then v1.contents <- v1.contents + 1 else ()) ;
  },
  {
    get = (fun () -> v2.contents) ;
    incr = (fun () -> if state.contents
                      then v2.contents <- v2.contents + 1 else ()) ;
  },
  (fun () -> state.contents <- (not state.contents))

let ctr1, ctr2, toggle = make_counter_2 ()

(* this line will print 0 since that's the initial value for ctr1 *)
;; print_endline (string_of_int (ctr1.get()))
(* this line will print 0 since that's the initial value for ctr2 *)
;; print_endline (string_of_int (ctr2.get()))

;; ctr1.incr()
;; ctr2.incr()

(* this line will still print 0 since ctr1 is locked by default *)
```

```
;; print_endline (string_of_int (ctr1.get()))
(* this line will print 1 since ctr2 is unlocked by default *)
;; print_endline (string_of_int (ctr2.get()))

(* ctr1 is now unlocked; ctr2 is now locked *)
;; toggle()

;; ctr1.incr()
;; ctr2.incr()

(* this line will now print 1 since ctr1 is unlocked *)
;; print_endline (string_of_int (ctr1.get()))
(* this line will still print 1 since ctr2 is locked *)
;; print_endline (string_of_int (ctr2.get()))
```

3. **Inheritance and Subtyping in Java** (22 points total)

The following questions refer to the Java interfaces and class definitions shown in Appendix B. Consider the class below, whose `main` method has some expressions omitted. (Assume all the code from the appendix is in the same source directory as this program, so there is no need to use **`import`** anywhere.)

```
1  public class ExampleCode {
2      public static void useFonJ(F obj, J tgt) {
3          obj.invokeMethod(tgt);
4      }
5
6      public static void main(String[] args) {
8          F f = /* omitted code X */
9          J j = /* omitted code Y */
10         useFonJ(f, j);
11     }
12 }
```

(a) (10 points) Suppose that when you run this `ExampleCode` program it raises a `NullPointerException`. Where in the program might the exception originate dynamically? (That is, on what line in which class will the exception be thrown when the program is run?) There are *at most five* possible sources of such an error. For each possibility, fill in the class and line number that throws the `NullPointerException`. Describe what the omitted code `X` or `Y` of `ExampleCode` would have to do to lead to that exception being raised. (Leave excess templates blank.)

RUBRIC:

- 4 points each for the `ExampleCode` line 3 and `F` line 4 answers
- 1 point each for line 8 and line 9, with any description that indicates that they throw the NPE

Class [`ExampleCode`], line [8] throws `NullPointerException` when omitted code [`X`] has the behavior of [raising the NPE exception]

Class [`ExampleCode`], line [9] throws `NullPointerException` when omitted code [`Y`] has the behavior of [raising the NPE exception]

Class [`ExampleCode`], line [3] throws `NullPointerException` when omitted code [`X`] has the behavior of [evaluating to **`null`**]

Class [`F`], line [4] throws `NullPointerException` when omitted code [`Y`] has the behavior of [evaluating to **`null`**]

Class [ ], line [ ] throws `NullPointerException` when omitted code [ ] has the behavior of [ ]

(b) (3 points) Assuming the code as given in the appendix, which types could be put in each blank below so that the variable declaration is statically well-typed?

```
I y = new _____();
```
☐ I ☐ J ☒ C ☒ D ☒ E ☐ F

```
J x = new _____();
```
☐ I ☐ J ☐ C ☒ D ☒ E ☐ F

```
D x = new _____();
```
☐ I ☐ J ☐ C ☒ D ☐ E ☐ F

(c) (9 points) Suppose you wanted to add a new class `G` to the code from the appendix such that:

- the statement `(new G()).method1();` successfully compiles and prints the string `"Class C's method1!"` when run
- the statement `(new F()).invokeMethod(new G());` successfully compiles and prints the string `"Class G's method2!"` when run

There are many ways to achieve this goal! List three different ways that this can be accomplished using ***simple inheritance***. In each case, indicate which superclass `G` **extends** (use only `Object` plus the classes available in the appendix); indicate which interface(s) `G` *must* explicitly implement (leave it blank if none); and for each method `method1` and `method2` indicate whether, for your choices above, `G` *must* explicitly provide the method or if `G` *may not* have to explicitly provide the method.

class `G` **extends** [Object] **implements** [J]

`G` ☒ must ☐ may not provide `method1()`

`G` ☒ must ☐ may not provide `method2()`

class `G` **extends** [C] **implements** [J]

`G` ☐ must ☒ may not provide `method1()`

`G` ☒ must ☐ may not provide `method2()`

class `G` **extends** [F] **implements** [J]

`G` ☒ must ☐ may not provide `method1()`

`G` ☒ must ☐ may not provide `method2()`

RUBRIC NOTE: These following answers use *overriding* and so are not examples of *simple inheritance*.

class G **extends** D **implements**

G ☐ must ☒ may not provide `method1()`

G ☒ must ☐ may not provide `method2()`

class G **extends** E **implements**

G ☒ must ☐ may not provide `method1()`

G ☒ must ☐ may not provide `method2()`

4. **Java Array Programming** (25 points)

Write a function `hPair`, that takes in two *rectangular* arrays of type **`int`**`[][]` (i.e. every inner array has the same length) and returns a new array that places the arrays horizontally adjacent to each other. Any "open" space left by the difference in array sizes should be filled with `0`'s. Pictorially, if `a` and `b` are as shown below, then the results of using `hPair` in the two orders are as shown.

```
  a       b       hPair(a,b)    hPair(b,a)
1 1 1     3 3     1 1 1 3 3     3 3 1 1 1
2 2 2     4 4     2 2 2 4 4     4 4 2 2 2
          5 5     0 0 0 5 5     5 5 0 0 0
```

You may assume that the input arrays `a` and `b` are not null, that they are rectangular, and that they contains no null sub-arrays. Note that `a[i]` refers to the row `i` in `a`. If `a` has no rows, then `hPair` should return a clone of `b` (and vice versa). You may find it useful to use the static methods `Math.max` and `Math.min`, or the array `clone` method.

```
public static int[][] hPair(int[][] a, int[][] b) {
    if (a.length == 0) {
        return b.clone();
    }
    if (b.length == 0) {
        return a.clone();
    }
    int h = Integer.max(a.length, b.length);
    int w = a[0].length + b[0].length;
    int[][] tgt = new int[h][w];
    for(int i=0; i<a.length; i++) {
        for(int j=0; j<a[0].length; j++) {
            tgt[i][j] = a[i][j];
        }
    }
    for(int i=0; i<b.length; i++) {
        for(int j=0; j<b[0].length; j++) {
            tgt[i][j+a[0].length] = b[i][j];
        }
    }

    return tgt;
}
```

A Appendix: Queue Implementation

```
type 'a qnode = { v: 'a;
                  mutable next: 'a qnode option }

type 'a queue = { mutable head: 'a qnode option;
                  mutable tail: 'a qnode option }

(* INVARIANT:
 - q.head and q.tail are either both None, or
 - q.head and q.tail both point to Some nodes, and
   - q.tail is reachable by following 'next' pointers
     from q.head
   - q.tail's next pointer is None
*)

let create () : 'a queue =
  { head = None; tail = None }

(* Add an element to the tail of a queue *)
let enq (elt: 'a) (q: 'a queue) : unit =
  let newnode = { v = elt; next = None } in
  begin match q.tail with
  | None ->
     (* Note that the invariant tells us that q.head is also None *)
     q.head <- Some newnode;
     q.tail <- Some newnode
  | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
  end

(* Remove an element from the head of the queue *)
let deq (q: 'a queue) : 'a =
  begin match q.head with
  | None ->
     failwith "deq called on empty queue"
  | Some n ->
     q.head <- n.next;
     if n.next = None then q.tail <- None;
     n.v
  end
```

B Appendix: Java Interfaces and Classes

```
1  public interface I {
2      void method1();
3  }
```

```
1  public interface J {
2      void method2();
3  }
```

```
1  public class C implements I {
2
3      @Override
4      public void method1() {
5          System.out.println("Class C's method1!");
6      }
7
8  }
```

```
1  public class D extends C implements J {
2
3      @Override
4      public void method2() {
5          System.out.println("Class D's method2!");
6      }
7
8  }
```

```
1  public class E implements I, J {
2
3      @Override
4      public void method1() {
5          System.out.println("Class E's method1!");
6      }
7
8      @Override
9      public void method2() {
10         System.out.println("Class E's method2!");
11     }
12
13 }
```

```
1  public class F {
2
3      void invokeMethod(J obj) {
4          obj.method2();
5      }
6
7  }
```