Name: _____

Pennkey: _____

CIS 120 Midterm II

March 30, 2012

1	/20
2	/20
3	/16
4	/16
5*ab*	/13
5*c*	/15
Total	/100

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 100 total points.
- There are 13 pages in this exam.
- Make sure your name and Pennkey (a.k.a. username) is on the top of this page.

1. OO terminology (20 points)

Fill in the blank with the correct term from the list at the bottom of the page. Each term will be used at most once and not all terms will be used.

a. A ___`static`___ method may not refer to the `this` reference.

b. Object-oriented programs use __`encapsulation`__ to preserve the invariants of data structures.

c. The static type of a variable is always a ___`supertype`___ of the dynamic class of the object value that it refers to.

d. In a method call `o.m()`, the ASM uses the ____`dynamic class`___ of the object `o` to determine which version of `m` to copy from the class table to the workspace.

e. Any Java type is a ____`subtype`____ of type `Object`.

f. The **`new`** expression creates an object value through class ____`instantiation`_____ .

g. When an object value is created, the class runs code defined in a ___`constructor`___ . This code often initializes the fields of the object.

h. Object values are stored in the ___`heap`____ by the Abstract Stack Machine.

i. A class may _____`implement`_____ multiple interfaces if it includes definitions for all of the methods declared in each of the interfaces.

j. The interface `Queue<E>` uses _____`generics`____ . The variable `E` is a type parameter, and may appear elsewhere in the definition of the interface.

Terms:

class	class table
constructor	dynamic class
encapsulation	extension
extend	generics
heap	implement
inherit	instantiation
object	stack
static	static type
supertype	subtype
workspace	

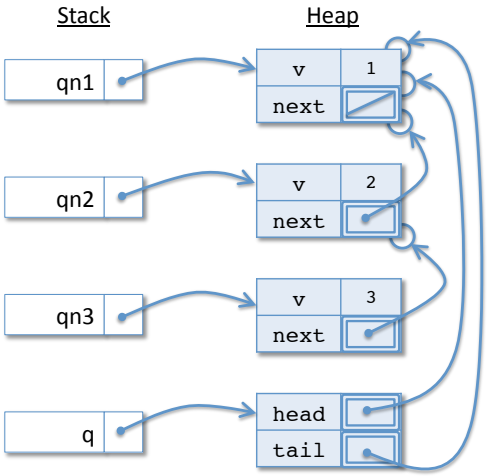
2. Queues in OCaml (20 points)

Recall the OCaml queue type from lecture and HW05.

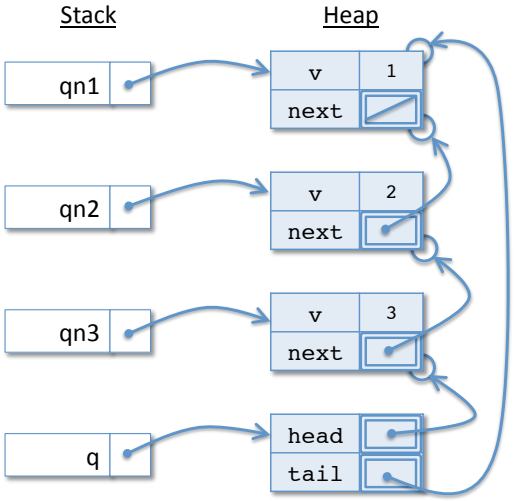
```
type 'a qnode = { v: 'a;
                  mutable next: 'a qnode option}

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

The next few questions concern the four stack/heap configurations shown below and labeled **(A)**, **(B)**, **(C)** and **(D)**. There are no responses required on page, so if you would like to *carefully* remove this page from the exam booklet you may.

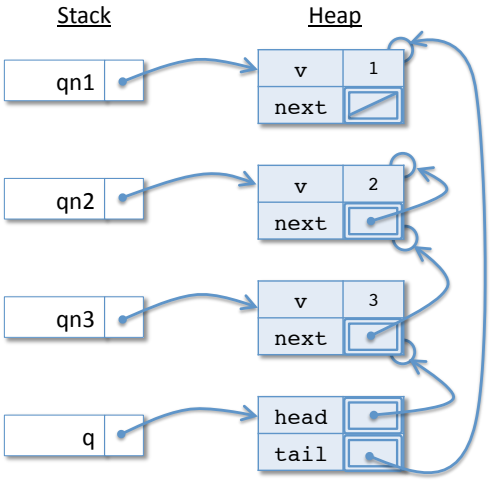
(A)



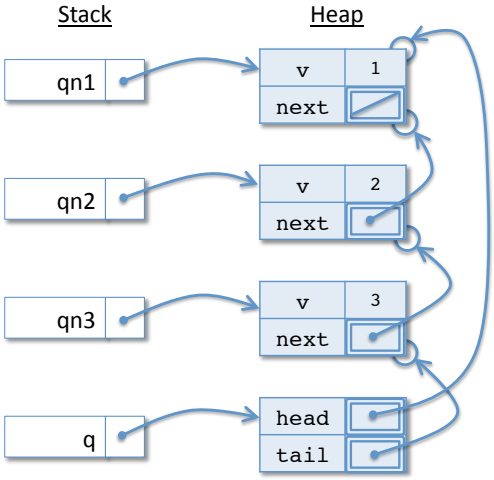
(B)



(C)



(D)



a. For each of the code segments below, which configuration displays the final state of the stack and heap after it has been executed on the workspace? Circle the correct letter.

i.
```
let qn1 = { v = 1; next = None }
let qn2 = { v = 2; next = Some qn1 }
let qn3 = { v = 3; next = Some qn2 }
let q = { head = Some qn3 ; tail = Some qn1 }
```

(A) **(B)** **(C)** **(D)**

ii.
```
let qn1 = { v = 1; next = None }
let qn2 = { v = 2; next = Some qn1 }
let qn3 = { v = 3; next = Some qn2 }
let q = { head = Some qn1 ; tail = Some qn3 }
```

(A) **(B)** **(C)** **(D)**

iii.
```
let qn1 = { v = 1; next = None }
let qn2 = { v = 2; next = Some qn1 }
let qn3 = { v = 3; next = Some qn2 }
let q = { head = Some qn1 ; tail = Some qn1 }
```

(A) **(B)** **(C)** **(D)**

b. Recall that the *queue invariant* for an `'a queue` value `q` is:

- `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`

Which of the queues `q` depicted in configurations **(A)**, **(B)**, **(C)** and **(D)** satisfy the queue invariant? Circle either SATISFIES or DOES NOT SATISFY below. For the ones that do not satisfy the invariant, describe the part of the invariant that they violate.

i. **(A)** SATISFIES, or DOES NOT SATISFY, because ...

ii. **(B)** SATISFIES, or DOES NOT SATISFY, because ...

iii. **(C)** SATISFIES, or DOES NOT SATISFY, because ... *it has a cycle*

iv. **(D)** SATISFIES, or DOES NOT SATISFY, because ... *the tail is not reachable from the head*

c. Consider the definition of the `contains` function, which determines whether a particular value is contained within a queue.

```
let contains (elt:'a) (q:'a queue) : bool =
   let rec loop (q: 'a qnode option) (vv: 'a) : bool =
    begin match q with
       | None -> false
       | Some a -> a.v = vv || loop a.next vv
    end
   in
   begin match q.head with
     | None -> false
     | Some n -> loop (Some n) elt
    end
```

Suppose the expression `contains 1 q` was placed on the workspace with the stack and heap as depicted in each of **(A)**, **(B)**, **(C)** and **(D)**. Circle whether the expression evaluates to **`true`**, **`false`**, or goes into an infinite loop.

i. With configuration **(A)**, `contains 1 q` evaluates to

`true` **`false`** infinite loop

ii. With configuration **(B)**, `contains 1 q` evaluates to

`true` **`false`** infinite loop

iii. With configuration **(C)**, `contains 1 q` evaluates to

`true` **`false`** infinite loop

iv. With configuration **(D)**, `contains 1 q` evaluates to

`true` **`false`** infinite loop

Grading scheme: 6 points for (a), 6 points for (b), 8 points for (c)

3. Subtyping, Interfaces, and Static types (16 points)

The following question involves the interface `Area` and classes `Circle` and `Rectangle` presented in lecture. For reference, these definitions are included in the Appendix—the last two pages of the exam.

Consider the static methods defined in the class below:

```
class C {
  public static Area asArea (Area s) {
    return s;
   }

  public static void main(String[] args) {
    Rectangle r = new Rectangle (1,2,1,1);
    Circle c = new Circle (1,1,3);
    Area s1  = r;
    Area s2  = c;

    _____ x = C.asArea (r);
    _____ y = C.asArea (s1);

  }
}
```

a. For each variable, fill in its *static type* and the *dynamic class* of the object that it refers to in the `main` method. Note that the type declarations for `x` and `y` have been intentionally omitted above.

variable	static type	dynamic class
s1	___Area____	_Rectangle_
s2	___Area____	__Circle___
x	__Area_____	_Rectangle_
y	__Area_____	_Rectangle_

b. Which of these assignments would type check if added to the end of the `main` method? Circle GOOD if the line would not cause a compile-time error, and TYPE ERROR otherwise.

`r = c;`	GOOD	**TYPE ERROR**
`r = s1;`	GOOD	**TYPE ERROR**
`s1 = c;`	**GOOD**	TYPE ERROR
`s1 = s2;`	**GOOD**	TYPE ERROR

c. Which of these variable declarations would type check if added to the end of the `main` method? Circle one.

`Area a = new Area();`	GOOD	**TYPE ERROR**
`double d = s1.getRadius();`	GOOD	**TYPE ERROR**
`double d = C.asArea(c).getRadius();`	GOOD	**TYPE ERROR**
`double d = C.asArea(c).getArea();`	**GOOD**	TYPE ERROR

Grading scheme: 1 point per blank, circle

4. Array programming (16 points)

Implement a static method, called `removeZeros`, that takes as input an array of `int`s and returns a new array that contains all of the nonzero values from the input array.

For example, if the input array is:

```
int [] arr = { 0, 1, 0, 0, 2, 2, 3, 0, 4 }
```

the output of `removeZeros(arr)` should be

```
{ 1, 2, 2, 3, 4 }
```

If the input to this method is **`null`**, then the result should also be **`null`**.

Answer:

```
public static int[] removeZeros(int[] arr) {
      if (arr == null) { return null; }
      int newsize = 0;
      for (int i : arr) {
        if (i != 0) newsize++;
      }
      int[] newdata = new int[newsize];
      int j = 0;
      for (int i : arr) {
        if (i != 0) {
          newdata[j] = i;
          j++;
        }
      }
      return newdata;
    }
```

Grading scheme:

16 points total:
1 - declaring correct input type
1 - declaring correct output type
2 - checking input for null
4 - correctly calculating size of new array
1 - allocating the new array
2 - loop over old array completely
3 - calculate index for new array update
2 - returning an array
- other errors at discretion

5. Program Design (28 points total)

Recall the definition of queues in Java. For reference, the interface `Queue` and the implementation of the classes `QNode` and `QueueImpl` appear in the Appendix.

Suppose you would like to add a public method, called `cutoff`, to the queue interface that cuts the queue off after a given value. In other words, given a value `x`, this method `x` the last element that appears in the queue by removing all elements that come after it. The method should return **`true`** if it modifies the queue and **`false`** otherwise.

a. (0 points) The first step of the program design process is to *understand the problem.* There is nothing to write here, but there are a few details to pay attention to.

- If there are multiple occurrences of the value, `cutoff` should truncate the queue after the *first* occurrence of the element.
- If the value does not appear in the queue, the queue should not be changed.
- You may assume that the queue does not contain nulls, and that the argument to this method is not `null`.
- Queues are generic and can store any type of objects. Therefore you should use `.equals` to compare values stored in the queue.

b. (3 points) The second step is to *define the interface* of the method. Write the method declaration that you should add to the `Queue<E>` interface.

answer:

```
public boolean cutoff(E x);
```

Grading scheme: One point for return type, one point for type of parameter, one point for public.

c. (10 points) The next step is to *write test cases*. For example, one possible test case creates a new queue containing three elements and then cuts off the queue after the middle element.

```
@Test
public void testCutOff() {
  QueueImpl<Integer> q = new QueueImpl<Integer>();
  q.enq(1);
  q.enq(2);
  q.enq(3);
  assertTrue(q.cutoff(2)); // the method DOES modify the queue
  assertTrue(q.deq() == 1);
  assertTrue(q.deq() == 2);
  assertTrue(q.is_empty()); // there is no value 3 in the queue
}
```

On the next page, write two more test cases for this method, accompanied by a short explanation of why they are interesting. As above, your test cases should test **both** the result of the `cutoff` method and the complete structure of the queue after the method call. **You will LOSE points on this problem if you don't include the explanation.**

Test cases for the `cutoff` method.

Valid interesting test cases include:

- empty queue
- cutoff for element that is not in the queue
- multiple values in the queue
- cutoff for an element at the end of the queue
- cutoff for first element

The test cases should test the result of cutoff and the structure of the queue.

Grading scheme: 5 points per test case: 1 for correct result, 1 for testing queue afterwards correctly, 2 for explanation (must match the behavior), 1 for test case actually being interesting.

d. (15 points) The final step is to *implement the method* as a new member of the `QueueImpl<E>` class.

Answer:

```
public boolean cutoff(E elt) {
  QNode<E> curr = this.head;
  while ( curr != null ) {
        if (curr.v.equals(elt) && this.tail != curr) {
          curr.next = null;
          this.tail = curr;
          return true;
        }
        curr = curr.next;
     }
     return false;
   }
```

Grading scheme:

1 - method declaration correct
3 - finds the correct queue node
2 - updates the tail
2 - updates the next pointer of a queue node to null
2 - returns true when elt in queue and queue changes
2 - returns false when elt is the last one
2 - returns false when elt not in queue
1 - uses .equals to compare elt
- other errors at discretion

Reference Appendix

```
interface Area {
   public double getArea ();
}

class Rectangle implements Area {
   private double x, y; // upper left coordinates
   private double w, h; // width and height
   public Rectangle (double x0, double y0, double w0, double h0) {
     x = x0; y = y0;
     w = w0; h = h0;
   }
   public double getArea () {
     return w * h;
   }
   public double getWidth () {
     return w;
   }
   public double getHeight () {
     return h;
   }
}

public class Circle implements Area {
   private double r;
   private double x,y;
   public Circle (double x0, double y0, double r0) {
      r = r0; x = x0; y = y0;
   }
   public double getArea () {
      return 3.14159 * r * r;
   }
   public double getRadius () { return r; }
}

   ************************** Queues ******************************
public interface Queue<E> {

   /** Determine if the queue is empty*/
   public boolean is_empty ();

   /** Add a value to the end of the queue */
   public void enq (E elt);

   /** Remove the front value and return it ( if any) */
   public E deq ();
}
```

```
public class QNode<E> {

  public final E v; // the value in the qnode is immutable
  public QNode<E> next; // the next pointer can be arbitrarily changed
  public QNode (E v0, QNode<E> next0) {
    v = v0;
    next = next0;
  }
}

public class QueueImpl<E> implements Queue<E> {

  private QNode<E> head;
  private QNode<E> tail;

  /** Constructor */
  public QueueImpl () {
    this.head = null;
    this.tail = null;
    return;
  }

  /** Determine if the queue is empty. */
  public boolean is_empty() {
    return (this.head == null);
  }

  /** Add a new value to the end of the queue */
  public void enq(E x) {
    QNode<E> newnode = new QNode<E>(x, null);
    if (tail == null) {
      head = newnode;
      tail = newnode;
    } else {
      tail.next = newnode;
      tail = newnode;
    }
  }
   /** Remove the front value and return it (if any).
    *
    * @throws a NullPointerException if the queue is empty. */
  public E deq() {
    E x = head.v;
    QNode<E> next = head.next;
    head = next;
    if (next == null) {
      tail = null;
    }
    return x;
  }
}
```