University of Western Ontario
Department of Computer Science
Computer Science 1027b Midterm Exam
March 8th, 2014, NS-1, 10am-noon, 2 hours

PRINT YOUR NAME:

PRINT YOUR STUDENT NUMBER:

DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO!

Instructions

- Fill in your name and student number above immediately.
- You have **2 hours** to complete the exam.
- Part 1 of the exam consists of Multiple Choice questions. Circle your answers on this exam paper.
- Part 2 consists of questions for which you will provide written answers. Write your answers in the spaces provided in this exam paper.
- Multiple choices question are worth 1 mark, unless indicated otherwise; other than that, the marks for each individual question are given. Allow approximately 1 minute per mark on average.
- There are pages for rough work at the back of the exam. You may detach them if you wish, but hand them in with the rest of the exam paper.
- Calculators are not allowed!

Mark summary

1	2	3	4	5	6	total
/20	/20	/15	/15	/15	/15	/100

Problem 1: true/false (20 marks)

Choose **one** answer for each question.

1. (1 mark) If Class A extends Class B (A ISA B), then Class A implements Class B.
   (a) true
   (b) false
2. (1 mark) A Java Class always has a constructor.
   (a) true
   (b) false
3. (1 mark) All Java Class methods can overload methods from the Object Class.
   (a) true
   (b) false
4. (1 mark) All the methods in a class A can see A's private variables.
   (a) true
   (b) false
5. (1 mark) Private variables are the same as Local variables.
   (a) true
   (b) false
6. (1 mark) If class A extends class B, then all the methods in A can see B's protected variables.
   (a) true
   (b) false
7. (1 mark) If class A extends class B, then all the methods in B can see A's prototected variables.
   (a) true
   (b) false
8. (1 mark) If a class does not have a constructor method it uses the appropriate constructor method of a superclass with a constructor method in the class' ISA hierarrchy.
   (a) true
   (b) false
9. (1 mark) A child class cannot override a parent method that is declared as public
   (a) true
   (b) false
10. (1 mark) The push, pop, peek and isEmpty methods of the StackADT class are $O(1)$ operations.
    (a) true
    (b) false

11. (1 mark) An algorithm with time complexity $O(2^n)$ runs in the same time as one with time complexity $O(n^2)$.

    (a) true

    (b) false

12. (1 mark) Data encapsulation requires that local variables be declared as private variables.

    (a) true

    (b) false

13. (1 mark) An interface can be used only of it is implemented by another class.

    (a) true

    (b) false

14. (1 mark) An object of Class A can have its private variables set by methods in an unrelated Class B using setter methods defined in Class A.

    (a) true

    (b) false

15. (1 mark) It is possible to have global variables in Java.

    (a) true

    (b) false

16. (1 mark) Overloaded methods distinguish themselves by the number and type of their formal parameters.

    (a) true

    (b) false

17. (1 mark) Method overriding is where a subclass adds additional implementations of one or more of its parent's methods.

    (a) true

    (b) false

18. (1 mark) Polymorphism allows a reference variable to point to objects in a ISA hierarchy.

    (a) true

    (b) false

19. (1 mark) dequeue is an $O(n)$ operation for queues implemented using a linked list.

    (a) true

    (b) false

20. (1 mark) dequeue is an $O(n)$ operation for queues implemented using an array.

    (a) true

    (b) false

Problem 2 (20 marks)

Consider the following Java program:

```
 1 public class midterm2014 {
 2
 3   // private variables
 4   private StackADT<Integer> s = new LinkedStack<Integer>();
 5   private StackADT<Integer> s2 = new LinkedStack<Integer>();
 6
 7   // Constructor
 8   public midterm2014(Integer[] numbers){
 9     for(int i=0;i<numbers.length;i++)
10     {
11       s2.push(new Integer(numbers[i]));
12     }
13   }  // midterm2014 Constructor
14
15   // run method
16   public void run(){
17     s=whatDoesThisDo(s2);
18   } // run
19
20   public String toString(){
21     StackADT<Integer> c = whatDoesThisDo(s);
22     String stg;
23
24     if(c.isEmpty()) stg="Result stack is empty\n";
25     else
26     {
27       stg="Result stack: ";
28       while(!c.isEmpty())
29       stg=stg+c.pop() + " ";
30     }
31     return stg;
32   } // toString
33
34   public StackADT<Integer> whatDoesThisDo(StackADT<Integer> s) {
35     StackADT<Integer> c = new LinkedStack<Integer>();
36     StackADT<Integer> c2 = new LinkedStack<Integer>();
37     Integer temp;
38     // Empty s and put contents in reverse order onto c
39     while(!s.isEmpty())
40     {
41       temp=s.pop();
42       c.push(temp);
43     }
44     // Empty c1 and put results in right order into s and c2
45     // Thus s is in its original state and c2 is a copy of it
46     while(!c.isEmpty())
47     {
48       temp=c.pop();
```

```
49       s.push(temp);
50       c2.push(temp);
51     }
52     return c2;
53   } // whatDoesThis Do
54
55 } // midterm2014 class
56
57 class Test2014 {
58   public static void main(String[] args) {
59     // allocate an array of 10 elements with no objects
60     Integer numbers[]=new Integer[10];
61     numbers[0]=new Integer(-9);
62     numbers[1]=new Integer(9);
63     numbers[2]=new Integer(-4);
64     numbers[3]=new Integer(4);
65     numbers[4]=new Integer(-29);
66     numbers[5]=new Integer(29);
67     numbers[6]=new Integer(-1024);
68     numbers[7]=new Integer(1024);
69     numbers[8]=new Integer(-1123);
70     numbers[9]=new Integer(1123);
71     System.out.println("Input:");
72     for(int i=0;i<10;i++)
73       System.out.print("number[" + i + "]=" + numbers[i] + "\n");
74     midterm2014 p = new midterm2014(numbers);
75     System.out.println(p.toString());
76     p.run();
77     System.out.println(p.toString());
78   }
79 } // Test2014
```

Answer the following questions:
(2a) (2 marks) What methods for stacks are used in this program?

`push`, `pop`, `isEmpty` (the `toString` called belongs to class midterm2014 and not stack)

(2b) (2 marks) When "java Test2014" is run, what does line 76 do?

For line 76, `p.run()` is executed

If line 76 is changed to line 74 (the original intention):
`Creates p as an object of class midterm2014, constructed with 10 integers: -9, 9, -4 4, -29, 29, -1024, 1024, -1123, 1123.`

(2c) (2 marks) What is `s` in line 4?

`s is an empty stack of type Integer`

(2d) (2 marks) What does method `whatDoesThisDo` do?

```
   Copies stack a from b, while maintaining the values of b
```

(2e) (12%) Hand trace the program:

```
Input:
number[0]=-9
number[1]=9
number[2]=-4
number[3]=4
number[4]=-29
number[5]=29
number[6]=-1024
number[7]=1024
number[8]=-1123
number[9]=1123
Result stack is empty

Result stack: 1123 -1123 1024 -1024 29 -29 4 -4 9 -9
```

Problem 3 (15 marks)

Consider the following java code:

```
class fct2014 {
  static void computeLoop(Integer q) {
    int sum=0;
    int n=100;
    for(int i=q;i<n & Math.abs(i)<=10;i=i*2) sum=sum+i;
    System.out.println("sum=" + sum + " for q=" + q);
  }

  public static void main(String[] args) {
    Integer p;
    p=new Integer(-2);
    computeLoop(p);
    p=new Integer(2);
    computeLoop(p);
    p=new Integer(-10);
    computeLoop(p);
    p=new Integer(10);
    computeLoop(p);
    p=new Integer(-100);
    computeLoop(p);
    p=new Integer(100);
    computeLoop(p);
    p=new Integer(0);
    computeLoop(p);
  }
}
```

1. (2%) What is `sum` for `p` being -2?

2. (2%) What is `sum` for `p` being 2?

3. (2%) What is `sum` for `p` being -10?

4. (2%) What is `sum` for `p` being 10?

5. (2%) What is `sum` for `p` being -100?

6. (2%) What is `sum` for `p` being 100?

7. (3%) What is `sum` for `p` being 0?

The solutions:

```
sum=-14 for q=-2
sum=14 for q=2
sum=-10 for q=-10
sum=10 for q=10
sum=0 for q=-100
sum=0 for q=100
The code does not return for q=0 as it is an infinite loop
But one could argue that although this is true sum is always
0 for all the infinite iterations of the loop.
```

Problem 4 (15 marks)

In each of the following situations, use big-O notation to express the amount of work being done in terms of n.

1. (2%) An element is inserted in an `ArrayStack` of size n, which had not reached full capacity.

   Answer: $O(1)$

2. (2%) An element is inserted in a `LinkedQueue` of size n

   Answer: $O(1)$

3. (2%) We test whether a `LinkedQueue` of size n is empty using `isEmpty`

   Answer: $O(1)$

4. (2%) An element is removed from a `LinkedStack` of size n

   Answer: $O(1)$

5. (2%) We execute the following code segment

```
for (int i = 1; i < n; i++)
  for (int j = i; j <= i; j++)
    System.out.println(i+j);
```

   Answer: $O(n)$

6. (3%) We execute the following code segment

```
int j = 1;
for (int i = 1; i < n; i++)
  j = j+1;
for (int i = 1; i < j; i++)
  System.out.println(i);
```

   Answer: $O(n)$

7. (2%) We execute the following code segment

```
for (int i = 1; i < n*n+1000*n; i++)
  System.out.println(i);
```

   Answer: $O(n^2)$

Problem 5 (15 marks)

We use a stack to check whether an html file is well-formed. As input, we are given an array of `String`'s. Some of them are *tags*:

- A opening tag is a string of the form `<bla>`. The precise definition is that a string is an opening tag if and only if it has length at least 2, its first character is `<`, its last character is `>` and its second character is *not* `/`.
- A closing tag is a string of the form `</bla>`. Precisely, a string is a closing tag if and only if it has length at least 3, its first character is `<`, its second character is `/` and its last character is `>`.
- An opening tag such as `<bla>` and a closing tag such as `</bla>` are called a *match*.

If `s` is a string, `s.length()` gives its length and `s.charAt(i)` returns its ith character as a `char`. Calling `s.substring(i,j)` returns the substring starting at i and finishing at $j-1$ (inclusive). For instance, if `s` is `<bla>`, `s.substring(1,4)` is `bla`.

Write the following methods:

1. (3%) `public static boolean isOpeningTag(String s)` that returns `true` if and only if `s` represents an opening tag.

```
public static boolean isOpeningTag(String s){
 int ell = s.length();
 if (ell < 2)
   return false;
 if (s.charAt(0) != '<')
   return false;
 if (s.charAt(1) == '/')
   return false;
 if (s.charAt(ell-1) != '>')
   return false;
 return true;
}
```

2. (3%) `public static boolean isClosingTag(String s)` that returns `true` if and only if `s` represents a closing tag.

```
public static boolean isClosingTag(String s){
  int ell = s.length();
  if (ell < 3)
    return false;
  if (s.charAt(0) != '<')
    return false;
  if (s.charAt(1) != '/')
    return false;
  if (s.charAt(ell-1) != '>')
    return false;
  return true;
}
```

3. (4%) `public static boolean isMatch(String s, String t)` that returns `true` if and only if `s` is opening, `t` is closing and `s` and `t` are a match.

```
public static boolean isMatch(String s, String t){
  if (! isOpeningTag(s))
    return false;
  if (! isClosingTag(t))
    return false;
  return s.substring(1, s.length()-1).equals(t.substring(2, t.length()-1));
}
```

Now, we want to recognize whether a text is well-formed (all opening tags must be closed by a closing tag, forming a match). As input, we take an array of strings of length n, and we use the following algorithm. Create a stack of strings; for $i = 0, \ldots, n-1$, take the ith string from the array; call it `t`. If it is an opening tag, put it on the stack. If it is a closing tag, try to pop a string `s` from the stack (if the stack is empty, return `false`) and check whether `s,t` is a match; if true, continue, if not, return `false`. If `t` is neither an opening nor a closing tag, do nothing. If you finish the loop without exiting the method, return `true` if and only if the stack is empty.

(5%) Write a method `public static boolean check(String[] array)` that implements the algorithm above.

```
public static boolean check(String[] array){
  LinkedStack<String> stack = new LinkedStack<String>();
  for (int i = 0; i < array.length; i++){
    String t = array[i];

    if (isOpeningTag(t))
      stack.push(t);
    if (isClosingTag(t)){
      if (stack.isEmpty())
        return false;
      String s = stack.pop();
      if (! isMatch(s, t))
        return false;
    }
  }
  return stack.isEmpty();
}
```

Problem 6 (15 marks)

We consider two-dimensional arrays of `Integer`'s, such as for instance

1	3	7	4	0
9	9	3	1	2

These arrays will be represented by queues of `Integer`'s. The *row-major* representation stores one row after the other, so for our example it would be $(1, 3, 7, 4, 0, 9, 9, 3, 1, 2)$, with 1 at the front and 2 at the rear. The *column-major* representation stores one column after the other, so in our example it would be $(1, 9, 3, 9, 7, 3, 4, 1, 0, 2)$, with 1 at the front and 2 at the rear.

In this problem, you will write code to go from row-major to column-major (all the code should be written in the method `rowToColumn` that we give below). The input is the row-major queue, the number m of rows (2, in our example) and the number n of columns (5, in our example).

1. (2.5%) We give the code to create an array `queues` of queues of `Integer`'s of length n. Write a loop that initializes every queue in it.

2. (5%) write two nested loops (one of length m, one of length n) that dequeue all elements from `rowMajor`, and enqueue them in `queues[0]`, ..., `queues[n-1]`, `queues[0]`, ..., `queues[n-1]`, ..., `queues[0]`, ..., `queues[n-1]`. In the previous example, they should enqueue 1 in `queues[0]`, then 3 in `queues[1]`, then 7 in `queues[2]`, then 4 in `queues[3]`, then 0 in `queues[4]`, then 9 in `queues[0]`, then 9 in `queues[1]`, then 3 in `queues[2]`, then 1 in `queues[3]`, then 2 in `queues[4]`. At the end, `rowMajor` is empty.

3. (2.5%) give the contents of all queues in the array `queues` at this stage, for our example (front on the left)

   - `queues[0]`= $(1, 9)$
   - `queues[1]`= $(3, 9)$
   - `queues[2]`= $(7, 3)$
   - `queues[3]`= $(4, 1)$
   - `queues[4]`= $(0, 2)$

4. (5%) write code that creates a new queue `columnMajor`, and uses two nested loops to dequeue all entries from `queues` and enqueue them in `columnMajor`. In our example, we would dequeue 1 from `queues[0]`, then 9 from `queues[0]`, then 3 from `queues[1]`, then 9 from `queues[1]`, then 7 from `queues[2]`, then 3 from `queues[2]`, then 4 from `queues[3]`, then 1 from `queues[3]`, then 0 from `queues[4]`, then 2 from `queues[4]`. Finally, add a `return` at the end.

```
public static LinkedQueue<Integer> rowToColumn(LinkedQueue<Integer> rowMajor, int m, int n){

   LinkedQueue<Integer>[] queues = new LinkedQueue[n];

   for (int i = 0; i < n; i++)
     queues[i] = new LinkedQueue<Integer>();

   for (int i = 0; i < m; i++)
     for (int j = 0; j < n; j++)
       queues[j].enqueue(rowMajor.dequeue());
```

```
    LinkedQueue<Integer> columnMajor = new LinkedQueue<Integer>();

    for (int j = 0; j < n; j++)
      for (int i = 0; i < m; i++)
        columnMajor.enqueue(queues[j].dequeue());

     return columnMajor;

}
```

Stacks and Queues Interfaces

```
public interface StackADT<T>{
  /**  Adds one element to the top of this stack.
   *   @param element element to be pushed onto stack  */
  public void push (T element);

  /**  Removes and returns the top element from this stack.
   *   @return T element removed from the top of the stack */
  public T pop();

  /**  Returns without removing the top element of this stack.
   *   @return T element on top of the stack */
  public T peek();

  /**  Returns true if this stack contains no elements.
   *   @return boolean whether or not this stack is empty */
  public boolean isEmpty();

  /**  Returns the number of elements in this stack.
   *   @return int number of elements in this stack */
  public int size();

  /**  Returns a string representation of this stack.
   *   @return String representation of this stack */
  public String toString();
}
```

```
public interface QueueADT<T>{
   /**
    * Adds one element to the rear of this queue.
    * @param element  the element to be added to the rear of this queue  */
   public void enqueue (T element);

   /**
    * Removes and returns the element at the front of this queue.
    * @return  the element at the front of this queue */
   public T dequeue();

   /**
    * Returns without removing the element at the front of this queue.
    * @return  the first element in this queue */
   public T first();

   /**
    * Returns true if this queue contains no elements.
    * @return  true if this queue is empty */
   public boolean isEmpty();

   /**
    * Returns the number of elements in this queue.
    * @return  the integer representation of the size of this queue */
   public int size();

   /**
    * Returns a string representation of this queue
    * @return  the string representation of this queue */
Public String toString();
```

Rough work 1/4

Rough work 2/4

Rough work 3/4

Rough work 4/4