UNIVERSITY OF TORONTO
Faculty of Arts and Science

April 2016 EXAMINATIONS

CSC207H1S
Software Design
Duration – 3 hours
Aids: None

You must obtain a mark of at least 40% on this exam, otherwise, a course grade of no higher than 47% will be assigned.

Enter your First Name: _____

Enter your Last Name: _____

Enter your Student#: _____

Question:	1	2	3	4	5	6	7	Total
Points:	8	10	12	9	9	10	10	68
Score:								

1. Read this page carefully, before attempting the follow up questions on the next page. You have the following correct and error free implementation of `Vector` class.

   - Vector: The `Vector` class implements a growable array of integers. Like an array, it contains methods (see code) that can push integers or pop integers at specified index in the vector.
   - Stack: The `Stack` class represents a last-in-first-out (LIFO) stack of integers. It must support the following two functional requirements:
     1. push the integer into the Stack as per LIFO
     2. pop the integer from the Stack as per LIFO

```
class Vector
{
.
.
   public Vector(int capacity)
   {
      /*Creates a empty Vector
      *of size capacity*/
      .
      .
   }

   public void push(int e)
   {
      /*appends 'e' to the end
      *of the vector*/
      .
      .
   }

   public void push(int index,int e)
   {
      /*inserts 'e' at the specified index
      *in the vector*/
      .
      .
   }

   public int pop()
   {
      /*removes the last(end) element
      *from the vector*/
      .
      .
   }

   public int pop(int index)
   {
      /*removes the integer at the
      *index position*/
   .
   .
   }
}
```

```
class Stack extends Vector
{
   public Stack(int capacity)
   {
      /*Creates a empty Stack of
      *size capacity*/
      super(capacity);
   }
}
```

i. Your friend correctly claims that the above design of `Stack` class does not duplicate code and supports the two required functional requirements. How does the `Stack` class, support the two functional requirements? Explain your answer clearly. (2)

ii. You are not happy with the design of the above `Stack` class. Using your knowledge of CSC 207, how would you explain to your friend that the above Stack class is a violation of the Liskov Substitution Principle? (3)

iii. Rewrite the `Stack` class so that instead of inheritance (Stack IS A Vector), you now use composition i.e. Stack HAS A Vector. You can safely assume that the above `Vector` class is correctly implemented. Your `Stack` class MUST support the two functional requirements AND is consistent as per the LIFO behaviour outlined on page 2. Your `Stack` class MUST not duplicate any code. (3)

Total for Question 1: 8

2. **Regular Expression**

(a) Write a regular expression for each of the following subquestion. A binary string is a string made up of only 1's and 0's. For example **11001** is an example of a binary string and so is **00000011**. YOU DO NOT HAVE TO WRITE ANY JAVA CODE.

i. Binary string that doesn't contain the substring 110 (2)

ii. Binary string that has an odd length (2)

iii. Binary string that starts with 0 and has odd length, or starts with 1 and has even length (2)

iv. Binary string that has length of atleast 1 and atmost 3 (2)

(b) Write regular expression that find words whose letters are in alphabetical order, e.g., *almost* and *beefily*. (2)

Total for Question 2: 10

3. What output does this program produce when it is executed? (**Note** that the main method is at the bottom. This program does compile and run without errors.) (12)

```
class A {
    public void f() { System.out.println("Af"); }
    public void f(String s) { f(1,s); }
    public void f(String s, int n) { System.out.println("Afsn: "+s+n); }
    public void f(int n, String s) { System.out.println("Afns: "+n+s); }
    public void f(int n) { System.out.println("Afn: "+n); }
}

class B extends A {
    public void f(int n) { System.out.println("Bfn: " + n); }
    public void f(String s, int n) { System.out.println("Bfsn: "+s+n); }
}

class C extends B {
    public void f(int n, String s) { System.out.println("Cfns: "+n+s); }
    public void f(int n) { f("hello", n); }
}


public class Methods {
  public static void main(String[] args) {
    B b = new B();
    b.f();
    b.f(17);
    A c = new C();
    c.f(" hi ");
    c.f(331);
    c.f(17," question ");
    c.f(" answer ", 42);
} }
```

Use this space to provide the output.

Total for Question 3: 12

4. **Mock Objects** Read this page carefully, before attempting the follow up questions on the next page. Your friend is implementing `JFileSystem` class. A `JFileSystem` by definition is a collection of `Files` and `Directories`. You are responsible for implementing the `Cat` command.

   The `Cat` command depends and collaborates with the `FileSystem`. You can also assume for this question that `Cat` command on `JFileSystem` takes in a single path argument. Here is the current design.

```
interface FileSystem{
     .
     .
     /*path can be absolute or relative.
     *If the path does not exist, InvalidPathException is thrown.
     *This method reads the file contents at *path and returns back the contents
     *as String.*/
     public String getFileContents(String path) throws InvalidPathException;
     .
     .
}

//Code that your friend is responsible for.
class JFileSystem implements FileSystem{
.
.
.
     public String getFileContents(String path) throws InvalidPathException{
          .
          .
          .
     }
.
}
```

   You have completed implementing the `Cat` command. Your code is as follows for the `Cat` command:

```
class Cat{
   private FileSystem fs;
   public Cat(FileSystem fs){
       this.fs=fs;
   }
   public String execute(String path)
   {
         String contents;
         try{
            contents=fs.getFileContents(path);
         }
         catch (InvalidPathException e){
             contents="Invalid Path Entered!";
         }
         return contents;
   }
}
```

   You like to get started immediately on the testing of the `Cat` command, however your friend hasn't completed the `JFileSystem` yet.

(a) Why do you want the `JFileSystem` completed before testing your `Cat` command? (1)

(b) You decide, to mock the `FileSystem` in order to proceed with the testing of the `Cat` class. Write a class called `MockJFileSystem` that mocks `FileSystem`. (4)

(c) Complete the following two test cases in `CatTest`. You can add any extra method(s) that you see fit in the class `CatTest` : (4)

```
class CatTest
{
    private Cat catCommand;


    //This test depending on your implementation of the MockJFileSystem,
    //passes in a valid file path to the execute method and must assert
    //equals on the contents of the file.
    @Test
    public void testGetValidStringFromExecute()
    {
        //COMPLETE THIS









    }

    //This test intentionally passes in an invalid path to the execute
    //method and must assert equals on the  error string
    //"Invalid Path Entered!"
    @Test
    public void testGetErrorStringFromExecute()
    {
        //COMPLETE THIS









    }
}
```

Total for Question 4: 9

5. Suppose we are defining a class to represent items stocked by an online grocery store. Here is the start of the class definition, including the class name and instance variables.

```
public class StockItem {
    String name; // item name (e.g., "Fancy Feast")
    String size; // size ("small", "12oz", etc.)
    String description; // item description (e.g., \yummy food")
    int    quantity; // number of copies of this item in stock


  /** Construct a new StockItem with the given data */
  public StockItem(String name, String size,
                   String decription, int quantity) { ... }
```

A summer intern was asked to implement an equals function for this class that treats two `StockItem` objects as equal if their name and size fields match. Here's the result:

```
/** return true if the name and size fields match */
public boolean equals(StockItem other)
  {
      return name.equals(other.name) && size.equals(other.size);
  }
```

i. This `equals` method seems to work some of the time but not always. Give an example (by writing Java code) showing a situation where it fails. Do not forget to provide an explanation. (4)

ii. Show how to fix the equals method given above so it works properly and has the intended meaning i.e. (`StockItems` are equal if their `name` and `size` are equal) (5)

Total for Question 5: 9

6. **Publish/Subscribe Design Pattern: Read this and the next page carefully. Followup questions appear on page 13.** Twitter is a social networking website where users post short messages called tweets. Posting a message is called tweeting. If user 'a' chooses to follow user 'b', it means that 'a' finds out about 'b's' tweets. For this question, you will define an *Account* class for keeping track of information about an individual Twitter account. An *AccountList* class, for keeping track of all Twitter accounts, has already been written. The following code demonstrates what these classes must be able to do: (10)

```
public class Twitter {
    public static void main(String[] args) {
        try {
            // Make an account list and create some accounts to go in it.
            AccountList accounts = new AccountList();
            Account a1 = accounts.createAccount("dianelynn");
            Account a2 = accounts.createAccount("barack");
            Account a3 = accounts.createAccount("Oprah");
            // Record the fact that a2 tweeted "Hello world".
            a2.tweet("Hello world.");
            // Record the fact that "dianelynn" now follows "barack".
            // From now on, she will find out about his tweets.
            accounts.recordFollows("dianelynn", "barack");
            // More twitter actions.
            a2.tweet("I love rutabagas!");
            a1.tweet("Me too!!");
            accounts.recordFollows("barack", "Oprah");
            a2.tweet("Totally.");
            a3.tweet("Are you kidding @mitt?");
        } catch (UsernameUnavailableException ex) {
            System.out.println("Username already taken");
        } catch (NoSuchUsernameException ex) {
            System.out.println("Unrecognized username");
 }
    }
}
```

With the two classes properly defined, the above code should produce the following output:

```
dianelynn (0 tweets) found out that barack (2 tweets) tweeted 'I love rutabagas!'
dianelynn (1 tweets) found out that barack (3 tweets) tweeted 'Totally.'
barack (3 tweets) found out that Oprah (1 tweets) tweeted 'Are you kidding @mitt?'
```

Note: the existing code brings up a number of interesting design issues, but for this question, you don't need to be concerned about that. Assume that classes *UsernameUnavailableException* and *NoSuchUsernameException* have been appropriately defined. On the next page, you will find class *AccountList*. On page 13, write the one missing class: *Account*. You must use the Publish/Subscribe also called the Observable/Observer design pattern. Hints:

- Design your code so that an Account object can be observed AND also can observe other Account objects.
- In class Account, store only the username and number of tweets made by that user. For the purposes of this question, you don't need to store the actual tweets.

Implement only the methods called in the existing code and anything needed to make them work as described by the output above. For example, you do not need to write any getters of setters.

To earn credit for your answer, you must use the observer pattern as described above. Some of the marks will be for good coding style. You do not need to write any Javadoc.

YOU DO NOT HAVE TO MODIFY THE ACCOUNTLIST CLASS.

```
public class AccountList {
    private HashMap<String, Account> list;
    public AccountList()
    {
        this.list = new HashMap<String, Account>();
    }
    /**
    * Creates a new account with the specified username and remembers it in
    * this AccountList.
    *
    * @param username the username for the new account.
    * @return the new account.
    * @throws UsernameUnavailableException if the specified username has
    * already been used for another account.
    */
    public Account createAccount(String username)throws UsernameUnavailableException {
        if (list.containsKey(username)) {
            throw new UsernameUnavailableException();
        }
        else {
            Account a = new Account(username);
            list.put(username, a);
            return a;
        }
    }
    /**
    * Records the fact that s1 follows s2.
    *
    * @param s1 the username of the person who follows s2.
    * @param s2 the username of the person followed by s1.
    * @throws NoSuchUsernameException if either s2 or s2 is not a username for
    * an existing account.
    */
    public void recordFollows(String s1, String s2)throws NoSuchUsernameException {
        Account a1 = this.list.get(s1);
        Account a2 = this.list.get(s2);
        if (a1 == null || a2 == null) {
            throw new NoSuchUsernameException();
        }
        else {
            a1.follows(a2);
        }
    }
}
```

(a)
```
//TODO: Complete the Account class as per the specs outlined in the question.
public class Account
{

}
```

Total for Question 6: 10

7. You want to iterate over the contents of a Java `List` in forward and reverse direction. In this question, you are required to write an iterator called `ReverseListIterator<Integer>` that implements `Iterator<E>` interface and will iterate over the `List` in the reverse order. Your `ReverseListIterator<Integer>` must work on the given `ForwardAndReverseListTest` class. Note: You do not have to make any change to the class `ForwardAndReverseListTest`. (**Hint:** `ReverseListIterator` MUST collaborate with `ListIterator`). Some of the marks will be for good coding style, and using proper collaboration with the `ListIterator`. (10)

```
public class ForwardAndReverseListTest {
    public static void main(String[] args) {
        /*Choose an arbitrary size for the lists*/
        int maxSize = 20;
        List<Integer> numberArray = new ArrayList<Integer>(maxSize);
        List<Integer> numberStack = new Stack<Integer>();
        /*Fill the lists with Integers*/
        for (int i = 0; i < maxSize; i++) {
            numberArray.add(new Integer(i));
            numberStack.add(new Integer(i));
        }
        /*Now iterate through the lists, forward and then backwards */
        /*First the arrayList */
        Iterator<Integer> fwdArrayIterator = numberArray.iterator();
        System.out.println("Iterating forward through the (array) list:");
        while (fwdArrayIterator.hasNext())
        {
            System.out.println(fwdArrayIterator.next());
        }

        Iterator<Integer> reverseListIterator =
                              new ReverseListIterator<Integer>(numberArray);
        System.out.println("Iterating backward through the (array) list:");
        while (reverseListIterator.hasNext())
        {
            System.out.println(reverseListIterator.next());
        }

        /*Then the stack */
        Iterator<Integer> fwdStackIterator = numberStack.iterator();
        System.out.println("Iterating forward through the (stack) list:");
        while (fwdStackIterator.hasNext())
        {
            System.out.println(fwdStackIterator.next());
        }
        reverseListIterator = new ReverseListIterator<Integer>(numberStack);
        System.out.println("Iterating backward through the (stack) list:");
        while (reverseListIterator.hasNext())
        {
            System.out.println(reverseListIterator.next());
        }
    }
}
```

```
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class ReverseListIterator<E> implements Iterator<E>
{




}
```

Total for Question 7: 10

1 Rough Page

This page will not be marked unless you tell us explicitly to mark it. If you like this page to be marked, you must write your **first name, last name, student#** on the very top. You must also clearly tell us what question does this page contain solution to.

2 Rough Page

3 Appendix

```
To convert from Float to String:
String s=Float.toString(25.0f);  //s now is the string \25.0"

class Throwable:
    // the superclass of all Errors and Exceptions
    Throwable getCause() // returns the Throwable that caused this Throwable to get thrown
    String getMessage() // returns the detail message of this Throwable
    StackTraceElement[] getStackTrace() // returns the stack trace info
class Exception extends Throwable:
    Exception()          // constructs a new Exception with detail message null
    Exception(String m) // constructs a new Exception with detail message m
    Exception(String m, Throwable c) // constructs a new Exception with detail message m caused by c
class RuntimeException extends Exception:
    // The superclass of exceptions that don't have to be declared to be thrown
class Error extends Throwable
    // something really bad
class Object:
    String toString() // returns a String representation
    boolean equals(Object o) // returns true iff "this is o"
interface Comparable<T>:
    int compareTo(T o) // returns < 0 if this < o, = 0 if this is o, > 0 if this > o
interface Iterable<T>:
    // Allows an object to be the target of the "foreach" statement.
    Iterator<T> iterator()
interface Iterator<T>:
    // An iterator over a collection.
    boolean hasNext() // returns true iff the iteration has more elements
    T next() // returns the next element in the iteration
    void remove() // removes from the underlying collection the last element returned or
                  // throws UnsupportedOperationException
interface Collection<E> extends Iterable<E>:
    boolean add(E e) // adds e to the Collection
    void clear() // removes all the items in this Collection
    boolean contains(Object o) // returns true iff this Collection contains o
    boolean isEmpty() // returns true iff this Collection is empty
    Iterator<E> iterator() // returns an Iterator of the items in this Collection
    boolean remove(E e) // removes e from this Collection
    int size() // returns the number of items in this Collection
    Object[] toArray() // returns an array containing all of the elements in this collection
interface List<E> extends Collection<E>, Iteratable<E>:
    // An ordered Collection. Allows duplicate items.
    boolean add(E elem) // appends elem to the end
    void add(int i, E elem) // inserts elem at index i
    boolean contains(Object o) // returns true iff this List contains o
    E get(int i) // returns the item at index i
    int indexOf(Object o) // returns the index of the first occurrence of o, or -1 if not in List
    boolean isEmpty() // returns true iff this List contains no elements
    E remove(int i) // removes the item at index i
    int size() // returns the number of elements in this List
    ListIterator<E> listIterator() //Returns a list iterator over the elements in this list
    (in proper sequence).
    ListIterator<E> listIterator(int index) //Returns a list iterator over the elements in this list
    (in proper sequence), starting at the specified position in the list.

class ArrayList<E> implements List<E>
class Arrays
```

```
    static List<T> asList(T a, ...) // returns a List containing the given arguments


interface Map<K,V>:
    // An object that maps keys to values.
    boolean containsKey(Object k) // returns true iff this Map has k as a key
    boolean containsValue(Object v) // returns true iff this Map has v as a value
    V get(Object k) // returns the value associated with k, or null if k is not a key
    boolean isEmpty() // returns true iff this Map is empty
    Set<K> keySet() // returns the Set of keys of this Map
    V put(K k, V v) // adds the mapping k -> v to this Map
    V remove(Object k) // removes the key/value pair for key k from this Map
    int size() // returns the number of key/value pairs in this Map
    Collection<V> values() // returns a Collection of the values in this Map
class HashMap<K,V> implements Map<K,V>
class File:
    File(String pathname) // constructs a new File for the given pathname
class Scanner:
    Scanner(File file) // constructs a new Scanner that scans from file
    void close() // closes this Scanner
    boolean hasNext() // returns true iff this Scanner has another token in its input
    boolean hasNextInt() // returns true iff the next token in the input is can be
                         // interpreted as an int
    boolean hasNextLine() // returns true iff this Scanner has another line in its input
    String next() // returns the next complete token and advances the Scanner
    String nextLine() // returns the next line and advances the Scanner
    int nextInt() // returns the next int and advances the Scanner
class Integer implements Comparable<Integer>:
    static int parseInt(String s) // returns the int contained in s
        throw a NumberFormatException if that isn't possible
    Integer(int v) // constructs an Integer that wraps v
    Integer(String s) // constructs on Integer that wraps s.
    int compareTo(Object o) // returns < 0 if this < o, = 0 if this == o, > 0 otherwise
    int intValue() // returns the int value
class String implements Comparable<String>:
    char charAt(int i) // returns the char at index i.
    int compareTo(Object o) // returns < 0 if this < o, = 0 if this == o, > 0 otherwise
    int compareToIgnoreCase(String s) // returns the same as compareTo, but ignores case
    boolean endsWith(String s) // returns true iff this String ends with s
    boolean startsWith(String s) // returns true iff this String begins with s
    boolean equals(String s) // returns true iff this String contains the same chars as s
    int indexOf(String s) // returns the index of s in this String, or -1 if s is not a substring
    int indexOf(char c) // returns the index of c in this String, or -1 if c does not occur
    String substring(int b) // returns a substring of this String: s[b .. ]
    String substring(int b, int e) // returns a substring of this String: s[b .. e)
    String toLowerCase() // returns a lowercase version of this String
    String toUpperCase() // returns an uppercase version of this String
    String trim() // returns a version of this String with whitespace removed from the ends
    boolean equalsIgnoreCase(String anotherString)
    //Compares this String to another String, ignoring
    //case considerations. Two strings are considered
    //equal ignoring case if they are of the same length
    //and corresponding characters in the two strings are
    //equal ignoring case.
class System:
    static PrintStream out // standard output stream
```

```
    static PrintStream err // error output stream
    static InputStream in // standard input stream
class PrintStream:
    print(Object o) // prints o without a newline
    println(Object o) // prints o followed by a newline


class Pattern:
   static boolean matches(String regex, CharSequence input) // compiles regex and returns
                                                            // true iff input matches it
   static Pattern compile(String regex) // compiles regex into a pattern
   Matcher matcher(CharSequence input) // creates a matcher that will match
                                       // input against this pattern
class Matcher:
   boolean find() // returns true iff there is another subsequence of the
                  // input sequence that matches the pattern.
   String group() // returns the input subsequence matched by the previous match
   String group(int group) // returns the input subsequence captured by the given group
                           //during the previous match operation
   boolean matches() // attempts to match the entire region/string against the pattern.
class Observable:
    void addObserver(Observer o) // adds o to the set of observers if it isn't already there
    void clearChanged() // indicates that this object has no longer changed
    boolean hasChanged() // returns true iff this object has changed
    void notifyObservers(Object arg) // if this object has changed, as indicated by
        the hasChanged method, then notifies all of its observers by calling update(arg)
        and then calls the clearChanged method to indicate that this object has no longer changed
    void setChanged() // marks this object as having been changed
interface Observer:
    void update(Observable o, Object arg) // called by Observable's notifyObservers;
        // o is the Observable and arg is any information that o wants to pass along

interface ListIterator<E>. SuperInterface is Iterator<E>
    void        add(E e)
    //Inserts the specified element into the list (optional operation).

    boolean        hasNext()
    //Returns true if this list iterator has more elements when traversing the list in the forward
    direction.

    boolean        hasPrevious()
    //Returns true if this list iterator has more elements when traversing the list in the reverse
    direction.

    E        next()
    //Returns the next element in the list and advances the cursor position.

    int        nextIndex()
    //Returns the index of the element that would be returned by a subsequent call to next().

    E        previous()
    //Returns the previous element in the list and moves the cursor position backwards.

    int        previousIndex()
    //Returns the index of the element that would be returned by a subsequent call to previous().

    void        remove()
```

```
    //Removes from the list the last element that was returned by next() or previous()
    //(optional operation).

    void        set(E e)
    //Replaces the last element returned by next() or previous() with the specified element
    //(optional operation).
```

```
Assertion of JUnit:
Class Assert:
        static void assertEquals(String message,String expected,String actual)
        static void assertEquals(String message,int expected,int actual)
        static void assertEquals(String message,float expected,float actual)
        static void assertEquals(String message,double expected,double actual)
```

```
Here are some predefined character classes:
.    Any character
\d   A digit: [0-9]
\D   A non-digit: [^0-9]
\s   A whitespace character: [ \t\n\x0B\f\r]
\S   A non-whitespace character: [^\s]
\w   A word character: [a-zA-Z_0-9]
\W   A non-word character: [^\w]
\b   A word boundary: any change from \w to \W or \W to \w
```

```
Here are some quantifiers:
Quantifier   Meaning
 X?          X, once or not at all
 X*          X, zero or more times
 X+          X, one or more times
 X{n}        X, exactly n times
 X{n,}       X, at least n times
 X{n,m}      X, at least n; not more than m times
```