First Name:
Last Name:
Student #

UNIVERSITY OF TORONTO
AUGUST 2015 FINAL EXAMINATION
CSC207H1Y
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.

Name:
Student #:

Question	Mark
1. Factory Methods	/15
2. Builder Design Pattern	/14
3. Generics and Collections	/10
4. Regular Expressions	/15
5. Iterator Design Pattern	/10
6. Publish Subscribe Design Pattern	/15
Total:	/79

First Name:
Last Name:
Student #

Question 1) Factory Methods **[15]**

Given the following `ComplexNumber` class:

```
class ComplexNumber
{
     private float realPart;
     private float imaginaryPart;

     //Create ComplexNumber using CartesianCoordinates
     public ComplexNumber(float r, float i)
     {
          realPart=r;
          imaginaryPart=i;
     }

     //Create ComplexNumber using PolarCoordinates
     //The angle is measured in radians.
     public ComplexNumber(double modulus, double angle)
     {
          realPart=(float)(modulus*Math.cos(angle));
          imaginaryPart=(float)(modulus*Math.sin(angle));
     }

     public void setRealPart(float rP)
     {
          realPart=rP;
     }

     public void setImaginaryPart(float iP)
     {
          imaginaryPart=iP;
     }

     public float getRealPart()
     {
          return realPart;
     }

     public float getImaginaryPart()
     {
          return imaginaryPart;
     }

}
```

First Name:
Last Name:
Student #

a) A strict requirement is that the `ComplexNumber` class must remain *immutable* at all times. What change would you make to the above `ComplexNumber` class such that it is turned into *immutable*? [1]

b) A fellow developer now has written the following *instance* method inside the class `ComplexNumber` to add two complex numbers:

```
public ComplexNumber addTwoComplexNumbers(ComplexNumber a)
{
      a.realPart=a.realPart+this.realPart;
      a.imaginaryPart=a.imaginaryPart+this.imaginaryPart;
      return a;

}
```

b.1) Explain, how the immutability constraint is violated in the above `addTwoComplexNumbers(...)` method? [1]

b.2) Rewrite the above `addTwoComplexNumbers(...)` method such that the immutability constraint is no longer violated. [2]

c) Override the `toString()` method from the `Object` class in `ComplexNumber` such that the textual string representation of a `ComplexNumber` object is:

realPart + imaginaryPart i

i.e. the `realPart` followed by a *space* followed by the *plus* character followed by a *space* followed by the `imaginaryPart` followed by the character *i*.
For example, if a complex number has a real part of 5.0 and imaginary part of 10.0, then the textual string representation of this number would be:

5.0 + 10.0i [3]

(See appendix on how to convert from `Float` to `String`).

d) Rewrite the above `ComplexNumber` class such that it now uses Static Factory Methods. [3]

```
class ComplexNumber
{

}
```

First Name:
Last Name:
Student #

f) Provide *three* unit tests: i.e. one each to test the factory methods and one for testing the addition of two complex numbers.

TestCase1 for Cartesian Factory:

- The complex number has a real part of *5.0* and an imaginary part of *10.0*.

TestCase2 for Polar Factory:

- The complex number has a modulus of *10.0* and an angle of *3.14*

TestCase3 for addition of two complex numbers:

- You can create any two complex numbers to test the addition.

Write appropriate comments in your `ComplexTest` class and provide the `@Before` and `@After` annotations on `setup()` and `tearDown()` if you require them. [5]
You can safely assume that all the import statements for JUnit are in place.

(Write your tests on the next page).

First Name:
Last Name:
Student #

```
class ComplexNumberTest
      {



            @Test
            public void testCartesianFactoryMethod()
            {









            }

            @Test
            public void testPolarFactoryMethod()
            {









            }

            @Test
            public void testAddTwoComplexNumbers()
            {









            }

      }
```

First Name:
Last Name:
Student #

Question 2) Builder Design Pattern. **[14]**

a)Assume that the following Song class is used by a popular Radio Station in Canada.

```
class Song
{
      private final String artist;                  //required parameter
      private final String songTitle;               //required parameter

      private final int songLength=-1;              //optional parameter
      private final int yearOfSong=-1;              //optional parameter
      private final String genre="unknown";         //optional parameter
      private final boolean isIndie="false";        //optional parameter
      private final String lyrics="unknown";        //optional parameter

      public Song (String artist, String songTitle)
      {
            this.artist=artist;
            this.songTitle=songTitle;
      }


      public Song (String artist, String songTitle, int songLength)
      {
            this.artist=artist;
            this.songTitle=songTitle;
            this.songLength=songLength;
      }


      public Song (String artist, String songTitle, int songLength,int
      yearOfSong)
      {
            this.artist=artist;
            this.songTitle=songTitle;
            this.songLength=songLength;
            this.yearOfSong=yearOfSong;
      }


     public String getArtist() { return this.artist;}
     public String getsongTitle() { return this.songTitle;}
     public int getYearOfSong() { return this.getYearOfSong;}
     public int getSongLength() { return this.songLength;}
      .
      .
      .
}
```

First Name:
Last Name:
Student #

a.1) Write code in the given `main` function that will create a `Song` with the following parameters using the above `Song` class: [1]

`artist` = Norman Blake, **`songTitle`** = You are my sunshine, **`songLength`** = 264, **`yearOfSong`** = 2000

```
class DriverSongTest
{
      public static void main(String [] args)
      {
            //Write your code in here.









      }
}
```

a.2) The current design of the `Song` class, does't scale up well when new *optional* instance parameters are *added* or *removed*. Explain your answer in relation to constructor overloading of the `Song` class. [2]

a.3) Using your CSC 207 skillset, you propose a new design using the Builder design pattern on the `Song` class.

a.3.1) What is the Builder design pattern? (Just state the definition) [1]

First Name:
Last Name:
Student #

a.3.2) Besides addressing the issues raised in a.2), what are two other advantages of using the Builder design pattern? [2]

a.3.3) Rewrite the `Song` class using the *Builder Design Pattern.* [5]

First Name:
Last Name:
Student #

a.4) Using your code in a.3.3), now write a single JUnit test, that tests the Builder design pattern on the following Song:
artist = Norman Blake, **songTitle** = You are my sunshine,
songLength = 264, **yearOfSong** = 2000 [3]

```
class SongTest
{

      @Test
      public void testBuilderPatternInSong()
      {
            String artist = "Norman Blake";
            String songTitle = "You are my sunshine"
            int songLength = 264
            int yearOfSong = 2000

            //Complete this. (Hint: Use the getter methods in the Song class)









      }

}
```

First Name:
Last Name:
Student #

Question 3) Generics and Collections. [10]

a) Define *generics* and *exceptions* and state one key difference between the two in relation to error checking. [3]

b) Given a list of `Date`, you are asked to sort them in ascending order using the `Collections.sort(…)` method AND the interface `Comparator`. Details below. [7]

```
public class DateTest
{
      public static void main(String args[])
      {
            Date d1=new Date(1,21,2000);
            Date d2=new Date(4,30,2000);
            Date d3=new Date(1,21,2012);
            Date d4=new Date(11,21,1999);

            List<Date> dateList=new ArrayList<>();
            dateList.add(d1);
            dateList.add(d2);
            dateList.add(d3);
            dateList.add(d4);

            //Call Collections.sort(…) here




            //Sorted List: [11/21/1999,1/21/2000,4/30/2000,1/21/2012]
            System.out.println("Sorted List:"+dateList);
      }
}
```

The expected output of the above code in ascending order is:
Sorted List: [11/21/1999,1/21/2000,4/30/2000,1/21/2012]

First Name:
Last Name:
Student #

```
public interface Comparator<Date>
{
   public int compare(Date obj1, Date
obj2);
}
```

For ascending order:
—obj1 and obj2 are the Dates to be compared.

—The compare method MUST return *zero* if the Dates are equal.

—The compare method MUST return a *positive value* (any value > 0) if obj1 is greater than obj2.

—Otherwise a *negative value* (any value < 0) is returned.

```
public class Date
{
    public int month;  //range 1 to 12
    public int day;    // range 1 to 31
    public int year;  // Year number
    public Date(int m, int d, int y)
    {
      month = m;
      day = d;
      year = y;
    }
    public String toString()
    {
      return month+"/"+day+"/"+year;
    }

}
```

—You do not have to write any sorting algorithm for this question. Instead the sorting algorithm is already implemented for you inside the `Collections.sort(…)` method. However, you MUST call `Collections.sort(…)`, inside the `main` function of your `DateTest` class with the **correct arguments**.

—The signature of the `sort` method inside the `Collections` class is as follows:

```
void Collections.sort(List<Date> list, Comparator<Date> c)

/*The above sort method in the class Collections, sorts the specified list
*according to the order induced by the specified comparator. */
```

—Write code (you are welcome to write any number of new classes and modify the above code in any way you like) so that the output of the `main` function in `DateTest` matches the expected output.

(Write your solution on the next page).

First Name:
Last Name:
Student #

Solution to question 3) comes here:

First Name:
Last Name:
Student #

Question 4) Regular Expressions.

a) Write all strings that match the regular expression (in bold): [2]

x?(0|1)y

b) Write a regular expression for a username of this form: [2]

c or g, followed by one digit, and followed by 1 to 6 lowercase letters.

c) For each of the four string below, circle the right answer to indicate whether or not it matches the regular expression (in bold): [4]

[ab]c*d+e?

abcde	matches	does not match
ad	matches	does not match
acccdddde	matches	does not match
accddee	matches	does not match

First Name:
Last Name:
Student #

d) For each string below, circle the right answer to indicate whether or not it matches the regular expression (in bold): [4]

`([a-z0-9_\.-]+)@([a-z\.]+)\.([a-z]){2,6}`

`aa@bb.cc`	matches	does not match
`12+@34+.abc`	matches	does not match
`baker@yummy.apple.pie.com`	matches	does not match
`123.abc...@..com`	matches	does not match

e) Complete the method below. You may assume that all appropriate imports have been done. For efficiency, you must compile the pattern once and reuse it. [3]

```
/**
*Return the number of strings in data such that the entire string is matched
*against the regex.
*      @param regex a regular expression
*      @param data the strings to be matched against regex
*      @return the number of strings in data that match regex
*/
public static int numMatches(String regex, ArrayList<String> data) {
      int count=0;









      return count;
}
```

First Name:
Last Name:
Student #

Question 5) Iterator. **[10]**

Below is the beginning of a very simple `Table` class. Add to it whatever is necessary (inner class and overriding methods) to ensure that it does implement `Iterable` as it claims. See appendix for the `Iterable` interface.

Assume that we want to iterate over the individual cells of the table (as opposed iterating over the rows or over the columns), and that we want to go in *row-major order*. In other words, the second entry we should visit is `contents[0][1]`, not `contents[1][0]`.

Note: Throw the `UnsupportedOperationException` in the `remove` method of your *inner* class that implements the `Iterator` interface. See appendix for the `Iterator` interface.

```
public class Table implements Iterable {

    /**
    * The contents of this table
    */
    private Object [][] contents;

    // Constructors, setters, getters and other table methods omitted and not required

}
```

First Name:
Last Name:
Student #

Question 6) Publish/Subscribe Design Pattern. **[15]**

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:

```
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 20, 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.

First Name:
Last Name:
Student #

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);
      }
    }

}
```

First Name:
Last Name:
Student #

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

{









}
```

First Name:
Last Name:
Student #

Extra Page

If you like your work on this sheet to be marked, clearly specify what question you are attempting here **and** write your first name, last name and student number on this sheet.

First Name:
Last Name:
Student #

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
class ArrayList<E> implements List<E>
class Arrays
    static List<T> asList(T a, ...) // returns a List containing the given arguments
```

First Name:
Last Name:
Student #

```
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
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
```

First Name:
Last Name:
Student #

```
   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

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