UNIVERSITY OF TORONTO

Faculty of Arts and Science

August 2012 EXAMINATIONS

CSC207H1Y

Instructor: Abbas Attarwala

Duration - 3 hours

No Aids Allowed

Student Name:

Student ID:

Question	Mark
1) Short Answers	/15
2) Regular Expressions	/16
3) Publish/Subscribe Design Pattern	/20
4) Iterator Design Pattern	/15
5) Stack and Heap	/12
6) JUnit	/13
7) Single Precision (Floating Point)	/9

Total: /100

Question 1) Short Answers **[15]**

a) What do you mean by *immutable* and *mutable* objects in Java? Give one example of *immutable* objects in Java. [3]

b) What is *deep* and *shallow* copy? [3]

c) Define *generics* and *exceptions* and state one key difference between the two. [3]

d) Explain the difference between *Validation* and *Verification*? [3]

e) Explain the difference between '*extends*' and '*implements*' in Java. [3]

Question 2) Regular Expressions: **[16]**

Write a regular expression for each of the following sets of binary strings (a string that is made up of 1's and 0's) **[10]**

a) All binary strings including empty string [2]

b) All binary strings except empty string [2]

c) Beings with 1, ends with 1 [2]

d) Ends with 00 [2]

e) Contains at least three 1's [2]

Multiple Choice: (There may be more than one right answer) **[6]**

1) The regular expression **a(bc)+de** will match: [1]

A) abcde
B) ade
C) abcbcde
D) abc

2) The regular expression **a(bc)?de** will match: [1]

A) ade
B) abc
C) abcde
D) abcbcde

3) The regular expression **[a-m]*** will match: [1]

A) blackmail
B) above
C) imbecile
D) below

4) The regular expression **[^aeiou]** will match: [1]

A) b
B) a
C) c
D) e

5) The regular expression **[a-z]{4,6}** will match: [1]

A) spider
B) tiger
C) jellyfish
D) cow

6) The regular expression **[a-z\s]*hello** will match: [1]

A) hello
B) Othello
C) say hello
D) 2hello

Question 3) **[20]**

Publish/Subscribe Design Pattern:

a) What are the benefits of the publish/subscribe design pattern? [3]

b) [17]

class **Observable** has the following methods defined:
--void addObserver(Observer o);
--int countObservers();
--void deleteObserver();
--void deleteObserver(Observer o);
--boolean hasChanged(Observer o);
--void notifyObservers();
--void notifyObservers(Object arg);

AND the interface **Observer** is defined as following:

```
interface Observer
{
    public void update(Observable obs, Object obj)
}
```

Using the **class observable** and the **interface Observer** you are to write the following software system. The software system is made up of the *weather station* that gets the actual weather data from sensors (humidity, temperature and pressure). The Weather Data object (that tracks the data coming from the Weather Station), and updates the UI displays that shows users the current weather conditions. You have following incomplete code at your disposal. **You must complete it, i.e. add code in incomplete methods, add missing methods and specify if the classes inherit or implement any of the above super class or interface.** YOU CAN ALSO ASSUME THAT THE *StatisticsDisplay* AND *ForecastDisplay* CLASS ARE ALREADY IMPLEMENTED.

//YOU DO NOT NEED TO CHANGE THE WEATHERSTATION CLASS.

```
public class WeatherStation
{
   public static void main(String[] args)
   {
      WeatherData weatherData = new WeatherData();
      CurrentConditionsDisplay currentDisplay =
         new CurrentConditionsDisplay(weatherData);
      StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
      ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);
      weatherData.setMeasurements(80, 65, 30.4f);
      weatherData.setMeasurements(82, 70, 29.2f);
      weatherData.setMeasurements(78, 90, 29.2f);
   }
}
```

//THIS IS INCOMPLETE CLASS. YOU MUST COMPLETE IT

```
public class WeatherData
{
   private float temperature;
   private float humidity;
   private float pressure;

   public WeatherData() { }

   public void measurementsChanged()
   { //complete this method




   }

   public void setMeasurements(float temperature, float humidity, float pressure)
   {//complete this method




   }



}
```

```
public class CurrentConditionsDisplay
{
   Observable observable;
   private float temperature;
   private float humidity;

   public CurrentConditionsDisplay(Observable observable)
  {
      //THIS IS INCOMPLETE METHOD
       this.observable = observable;




   }
//1) THERE IS ONE KEY FUNCTION MISSING HERE. THIS FUNCTION THAT YOU
WRITE HERE, MUST CALL THE display() FUNCTION BELOW.
//2) WRITE ANOTHER FUNCTION THAT ALLOWS IT TO UNSUBSCRIBE




   public void display()
  {
      System.out.println("Current conditions: " + temperature
        + "F degrees and " + humidity + "% humidity");
   }
}
```

Question 4) **[15]**

Iterator Design Pattern:

You have a list (or List in Java) where you want to iterate over the contents for different purposes, but sometimes you want to iterate over the elements in reverse order. You are required to write an iterator called ***ReverseListIterator<Integer>*** **that implements Iterator<E> interface** and that will iterate over the list in the reverse order. Your ReverseListIterator<Integer> must work on the given code. Note: You do not have to make any change to the class ReverseListTest.

```
public class ReverseListTest {
	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());
		}

	}
}
```

The iterator <E>interface is defined as follow:

```
interface Iterator<E>
{
    //returns true if the iteration has more elements
    boolean hasNext();
    //returns the next element in the iteration
    E next();
    //removes from the underlying collection the last
    //element returned by the iterator (optional operation)
    void remove();
}
```

The ListIterator<E> interface is defined as follow:

```
interface ListIterator<E> extends Iterator<E>
{
//Returns true if this list iterator has more elements
//when traversing the list in the forward direction
    boolean hasNext();
//Returns true if this list iterator has more elements
//when traversing the list in the reverse direction
    boolean hasPrevious();
//returns the next element in the list
    E next();
//returns the previous element in the list
    E previous();
}
```

Here are some instance methods that you can call on the List<E> (inside java.util) object.

Return Type	**Function**
Iterator<E>	*iterator()* Returns an iterator over the elements in this list in proper sequence
int	*lastIndexOf(Object o)* Returns the index in this list of the last occurance of the specified element, or -1 if this list does not contain this element
ListIterator<E>	*listIterator()* Returns a list iterator of the elements in this list (in proper sequence)

ListIterator<E>	*listIterator(int index)* Returns a list iterator of the elements in this list(in proper sequence), starting at the specified position in this list.
E	*remove(int index)* Removes the first occurrence in this list of the specified element (optional operation)
boolean	*remove(Object o)* Removes the first occurance in this list of the specified element (optional operation)
E	*set(int index, E element)* Replaces the element at the specified position in this list with the specified element
int	*size()* Returns the number of elements in this list.

Solution to Question 4):

Question 5) **[12]**

a) Compare and contrast the *'Waterfall'*, *'Spiral'* and *'Scrum* software development processes. [3]

b) Explain in your own words the difference between a stack memory and a heap memory? How is memory **deallocated** on the stack and the heap? [3]

c)
```
openWord[] wordFiles1=new openWord[10];
openWord[] wordFiles2=new openWord[100];
```

What is allocated on the stack and on the heap in the following code? Draw a picture (stack/heap) clearly and show. You can assume that you have a class called *openWord* and that the above code is being called from a `main(String[] args)` function. [3]

d) What is the size of the reference wordFiles1? What is the size of reference wordFiles2? Explain your answer. If you are making any assumptions state them clearly. [3]

Question 6) **[13]**

a) Explain the benefits of Unit Testing. [3]

b) You are given the following three classes: [10]

```
public class Counter
{
	int count=0;

	public int increment() throws OverLimitException
	{
		if(count>=3)
			throw new OverLimitException("Crossed the limit of 3");
		return ++count;
	}

	public int decrement() throws UnderLimitException
	{
		if(count>=-3)
			throw new UnderLimitException("Crossed the limit of -3");
		return --count;
	}

	public int getCount()
	{
		return count;
	}

}
```

```
public class OverLimitException extends
Exception
{
        public OverLimitException(String
message)
        {
                super(message);
        }
}
```

```
public class UnderLimitException extends
Exception
{

        public UnderLimitException(String
message)
        {
                super(message);
        }

}
```

Write a test class using JUnit that will test your Counter class. Fixtures that are common across test methods, must be appropriately refactored in the setUp() and if necessary deallocated in the teardown().

Question 7) **[9]**

a) How many bits are allocated towards sign, mantissa and exponent in the single precision floating-point numbers? [3]

b) How would you represent a decimal number $(1)_{10}$ which is in base 10 into a single precision 32 bit floating point number? Your answer should clearly show all the steps and the final 32 bits of single precision representation. [6]