Name:
Student #:

UNIVERSITY OF TORONTO
APRIL 2014 FINAL EXAMINATION
CSC207H1S
Software Design
Duration – 3 hours
Aids: None

Name:
Student #:

Question	Mark
1) Short Answers	/13
2) Garbage Collector	/12
3) Java Exceptions	/10
4) Regular Expressions	/11
5) Single Precision (Floating Point)	/12
6) Software Development Processes	/12
7) Singleton/Publisher/Subscriber	/15
8) Iterator	/15

/100

Name:
Student #:

Question 1) Short Answers: **[13]**

a) What is *code refactoring*? When should you refactor code? [3]

b) Java does not support *multiple inheritances* due to *'diamond of death'* problem. Why doesn't this problem occur when implementing *multiple interfaces*? [2]

c) What are *CRC Cards?* And give one advantage of it. [2]

Name:
Student #:

d) What is the difference between *compile time* and *runtime binding*? What kinds of methods are binded at compile time? What kinds of methods are binded at run time? [2]

e) What is the difference between *local* and *instance* variables? Give an example of each. [2]

f) Explain what *Deep Copy* and *Shallow Copy* is? Give one example of each. [2]

Name:
Student #:

2) Garbage Collector [12]

a) How is memory *allocated* on the heap? How is memory *deallocated* on the heap? [2]

b) Following is an implementation of a *stack* data structure. A stack is a ***Last In, First Out*** data structure. The following implementation supports two methods on a stack i.e. `push` and `pop`. The `push` method adds the `Object` onto the stack and the `pop` method removes the *last or the top* most `Object` from the stack.

```
public class Stack
{
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack()
    {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e)
    {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop()
    {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity()
    {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

Name:
Student #:

b.1) In the current implementation, the object *removed* from the **Stack** via the pop method is not garbage collected. Why is this? Do you think this can be a problem? [2]

b.2) Draw the memory diagram to show that the objects that are *popped* off the *stack class* (in the code provided above) are not garbage collected. [3]

b.3) Modify and rewrite the pop function such that the objects that are popped off the *Stack class* will now be garbage collected. [3]

b.4) Use the memory diagram again to show that your solution to b.3) objects popped off the *Stack class* are now garbage collected. [2]

Name:
Student #:

Question 3) Exceptions **[10]**

a) Define *generics* and *exceptions* and state one key difference between the two in relation to *error checking*. [2]

b) Suppose we have a method B as follows: [8]

```
public int B(String s) throws XException, YException {
    // Body omitted
}
```

Write a public method called **A** to go in the same class as **B**. Method **A** should satisfy the following requirements:

- It takes a *String* parameter and returns an *int*
- It calls **B**, passing it the same String it was given
- If **B** throws, a *YException*, **A** prints "Eek!!"
- If **B** throws an *XException*, **A** doesn't deal with it; **A** just passes it along.
- If **B** executes without throwing any exception, **A** either
  - -throws a *ZException* if **B's** answer is less than zero, or
  - -Simply returns **B's** answer if it is at least zero.

Name:
Student #:

Solution to 3.b comes here:

Name:
Student #:

Question 4) Regular Expressions: **[11]**

a) Given the following text **(highlighted in bold)** and the regular expression on the next line *(highlighted in italics)*, what will the regular expression match in the text? Explain your answer. [3]

** this is line1   this is line2 **
* .*? *

b) Write a complete Java program such that:

You need to validate Canadian postal code. A Canadian postal code is a six-character string. The format for the postal code is A1A 1A1 where A is a letter (case insensitive) and 1 is a digit with a space OR a hyphen – separating the third and fourth characters. The string *worldAddress* (see below) may or may not contain Canadian postal code. If it does, print out ALL the matched postal codes only. [5]

Eg:
worldAddress="12 SomeStreet Apt 999, City GA 12345-6789 USA" (Your regex will find nothing)
worldAddress="12 SomeStreet Apt 999, City ON L8S 1A1 Canada" (Your regex will find L8S 1A1)
worldAddress="12 SomeStreet Apt 999, City ON L8S 1A1 or L8S 1A2 Canada"
(Your regex will find L8S 1A1 and L8S 1A2)

Name:
Student #:

```
import java.util.Scanner;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class ValidateUSPostalCode
{
      public static void main(String[] args)
      {

            String worldAddress;
            Scanner in= new Scanner(System.in);
            System.out.println("Enter a US or a non-US Address: ");
            worldAddress=in.nextLine();
            //Complete rest of the code in here:









      }
}
```

c) Create a regular expression that matches "magical" dates in *yyyy-mm-dd* format. A date is magical if the year minus the century, the month and the day of the month are all the same numbers. For example, 2008-08-08 is a magical date. You must only provide your *regular expression* and explain your answer. You are **not** asked to write a Java program for this. [3]

Name:
Student #:

Question 5) Java Floating Point Arithmetic **[12]**

a) What is *IEEE 754* standard? [1]

b) The real number ***.125*** in Java or Python gets correctly rounded off (2 decimal places) to ***.13***. Why is it that the real number ***2.675*** gets rounded off (2 decimal places) to *2.67* and not *2.68* in Java and Python? How would you reason this out? [3]

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

Name:
Student #:

d) Following is a single precision 32 bit floating point number in the IEEE 754 representation. What is the equivalent representation of this number in decimal or base 10? [5]

11000001 00000000 00000000 00000000

Name:
Student #:

Question 6) Software Development Processes. **[12]**

a) What is *software development process*? [2]

b) Define *unit testing*? What framework of unit testing did you use for your Assignment2? [2]

c) What is *Test-Driven Development*? What are three benefits of *Test Driven Development*? [3]

Name:
Student #:

d) What is the major difference between a *Waterfall software development process* and a *Scrum software development process*? [2]

e) Give two disadvantages, if you had followed the *Waterfall software development process* instead of the *Scrum software development process* for Assignment2. [3]

Name:
Student #:

Question 7) *Publisher/Subscriber* and *Singleton* Design Pattern **[15]**

a) Define coupling and cohesion. Give an example of each. [3]

b) Give one advantage and one disadvantage of the *publish subscribe* design pattern. [2]

c) You are designing a new operating system. One of the requirements that you are asked to implement for this operating system is that before it is shut down, applications with unsaved data must prompt the user whether the user like to save it or not.

For instance if you have four Word files and two of these files have unsaved data, then the operating system must notify these two files. The two files will then prompt the user whether to save the data before the OS completes its shut down process.

Note: The *TestOS* class contains the main function that is used to simulate the operating system and three word files in it. **You DO NOT have to modify/change this class.** [10]

Name:
Student #:

```
public class TestOS
{
        public static void main(String[] args)
        {

                operatingSystem os=operatingSystem.getReferenceToOS();
                wordFile file1=new wordFile("file1");
                wordFile file2=new wordFile ("file2");
                wordFile file3=new wordFile ("file3");

                //file4 will not get any notification at shutdown, as it has just been opened.
                wordFile file4=new wordFile("file4");

                //file1 has not been saved
                file1.setData("This is line1 in file1");

                //file2 has not been saved
                file2.setData("This is line1 in file2");

                //The user has saved file3 hence file3 will not get any notification at shutdown
                file3.setData("This is line1 in file3");
                file3.saveTheData();

                //The call to shut down the OS
                os.shutDownComputer();
        }

}
```

You need to complete the following two classes (on page 18 and page 19):
1) Implement all the incomplete methods, **2)** add any missing methods, **3)** satisfy the requirements for the singleton design pattern used on the operating system class, **4)** satisfy the requirements for publisher/subscribe design and **5)** Indicate if the classes implement the *observer* interface or extend the *observable* class.
Hint: Look at the *Observer interface* and *Observable class* in Appendix

Name:
Student #:

Note: The completed and correct code will work as follows, when the *main* function of ***TestOS*** is executed and assuming that the user types *Yes* for saving file2 and Yes for saving file1.

```
file3 file has been saved successfully

Do you like to save the data for file2?
Yes
file2 file has been saved successfully

Do you like to save the data for file1?
Yes
file1 file has been saved successfully
```

Name:
Student #:

The *operating system* class, follows the *singleton design pattern* and is as follows:

```
public class operatingSystem _____
{
        private static operatingSystem osReference=null;

        //You do not have to modify the constructor. You can assume this is completed for you.
        private operatingSystem()
        {
                //OS specific initialization here.
                .
                .
                .
        }

        //Complete this method, so that it follows the singleton design pattern
        public static operatingSystem getReferenceToOS()
        {









        }

        //This method is called, when the user shuts down the computer
        //Complete this method, so that all unsaved files are notified that the OS is about to shut down.
        public void shutDownComputer()
        {









        }
}
```

Name:
Student #:

```
public class wordFile _____
{
        private String dataContents;
        private String fileName;
        private boolean isDataSaved;
        public wordFile (String fileName)
        {
                this.fileName=fileName;
                isDataSaved=true;
        }

        //appends any new data to the dataContents and marks itself as having unsaved data
        //Complete this method
        public void setData(String content)
        {
                dataContents=dataContents+content;
                isDataSaved=false;



        }

        //prompts the user whether to save the data or not
        public void promptToSaveData()
        {
              Scanner sc = new Scanner(System.in);
              System.out.println("Do you like to save the data for"+filename+"?");
              if (sc.nextLine().equals("Yes"))
              {
                        saveTheData();
              }
        }

        //this method will save/write the data on the file system and unmarks itself as having unsaved data
        //Complete this method
        public void saveTheData()
        {


        //You can assume that writeContentsOfFileToFileSystem() is already implemented for you.
                writeContentsOfFileToFileSystem();
                System.out.println(filename+"file has been successfully saved");


        }
        //Add any missing methods here (if any?)

}
```

Name:
Student #:

Question 8) Iterator Design Pattern. **[15]**

a) What are two advantages of the Iterator design pattern? Give two examples in relation to your Assignment2, that the Iterator design pattern is most applicable. [5]

b) In this question, you must implement the iterator design pattern. Class java.lang.Number is an abstract class in Java 6. It is the super class of classes such as Integer and Double. The following class, Numberlist, maintains a list of Numbers. [10]

```
public class Numberlist{
      private int numItems;
      private Number[] numbers;

      public NumberList(int size) {
            this.numbers = new Number[size];
            this.numItems = 0;

      }
       public void add(Number n) {
             this.numbers[this.numItems++] = n;
       }

}
```

Name:
Student #:

Here is code that uses *Numberlist*

```
public void printPairs() {
      NumberList numbers = new NumberList(50);
      numbers.add(4);
      numbers.add(5);
      numbers.add(6);

      for (Number n1 : numbers) {
            for (Number n2 : numbers) {
                  System.out.println("" + n1 + n2);
            }
      }
}
```

printPairs does't work because *Numberlist* does properly support the foreach loop. If it did work, *printPairs* would print this output, one per line: 44 45 46 54 55 56 64 65 66.

Modify the *Numberlist* code so that *printPairs* works. You can cross off or add code to *Numberlist*, and you can continue the class on the next page. You are welcome to define as many additional classes and methods as you like.
Hint: Look at the *Iterator interface* and *Iterable interface* in Appendix

Name:
Student #:

Solution for question 8) comes in here.

Name:
Student #:

Extra Sheet that you can use.

Name:
Student #:

Name:
Student #:

Appendix

```
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(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>
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
```

Name:
Student #:

```
    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
   static Pattern compile(String regex) // compiles regex into a pattern
   Matcher matcher(CharSequence input) // creates a matcher that will match
                                       // input against this pattern
```

Name:
Student #:

```
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 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
```

Regular expressions:

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

```
Class Observable:
-void addObserver(Observer o) //Adds an observer
-void clearChanged()
-int countObservers() //Counts the observer that are in the collection
-void deleteObserver(Observer o) //Deletes the observer
-void deleteObservers() //Deletes all the observers
-boolean hasChanged()
-void notifyObservers() //Notifies all the observers
-void notifyObservers(Object arg)
-void setChanged()
```