Student Name:
Student #:

# UNIVERSITY OF TORONTO
# AUGUST 2014 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. Short Answers | /15 |
| 2. Java Floating Point Arithmetic and Regular Expressions | /12 |
| 3. Generics and Collections | /13 |
| 4. Garbage Collector and Java Memory Model | /10 |
| 5. Builder Design Pattern | /14 |
| 6. Iterator Design Pattern | /14 |
| 7. Publish Subscribe Design Pattern | /12 |
| Total: | /90 |

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

a) Define *shallow copy* and *deep copy.* [2]

b) Define *mutable* and *immutable* objects. Give one example of immutable object in Java JDK. [2]

c) What are *fixtures* in unit testing? Give one example of a fixture that you created in JUnit for *Assignment2* AND *Assignment3*? [2]

d) Assume that a certain integer data type is of size 4 bits.

d.1) If this data type were unsigned, what would its *minimum* and *maximum* value be? Explain your answer clearly. [2]

d.2) If this data type were signed, what would its *minimum* and *maximum* value be? Explain your answer clearly. [2]

e) Given the following `Car` class:

```
Class Car
{
      private String carModel;
      private int numberOfGears;
      private int mileageOfCar;
      private StreeringWheel swheel;
      private Tyre[4] tires;
      .
      .
      .
      public Car( String model, int gears)
      {
            .
            .
            .
      }
      .
      .
      .

}
```

You are now asked to provide two public methods i.e

```
1.) double convertMilesPerGallonToKilometersPerLitre (double mpg)
2.) void setMileage (double mpg)
```

e.1) How would you reason which of these methods must be *static* or *non static?* Just provide your reasoning. You do not need to implement any of the above methods. [3]

e.2) Based on your answer to d.1) above, how would you *call* the above two methods? [2]

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









      }
}
```

## Question 2) Java Floating Point Arithmetic [14]

a) What is *IEEE 754* standard? [2]

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 number? [2]

d) Write *regular expression* ONLY for the following:

d.1) The 24 hour clock is the convention of time keeping in which the day runs from midnight to midnight and is divided into 24 hours, indicated by the hours passed since midnight from 0 to 23. You like to validate time format in the 24 hour clock in format such as the following: [4]

**hh:mm:ss**

Where **hh** is the Hour, **mm** is the minutes and **ss** is the seconds.

**<u>Note</u>**: that the **<u>first</u>** h in hh, the **<u>first</u>** m in mm and the **<u>first</u>** s in ss may contain an optional 0. Following are some valid example of 24 hour clock.

```
00:00:00
01:43:59
 1:45:44
14:47:12
23:59:45
```

d.2) Write regular expression to validate an email address.
An email address does not contain any white space characters AND contains the symbol @ [3]

## Question 3) Generics and Collections. [13]

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. [10]

```
public class DateTest
{
      public static void main(String args[])
      {
            Date d1=new Date(1,21,2000);
            Date d2=new Date(4,30,2004);
            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: [11/21/1999,1/21/2000,4/30/2004,1/21/2012]
            System.out.println("Sorted List:"+dateList);
      }
}
```

The expected output of the above code in ascending order is:
Sorted List: [11/21/1999,1/21/2000,4/30/2004,1/21/2012]

```
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)`

—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.

Student Name:
Student #:

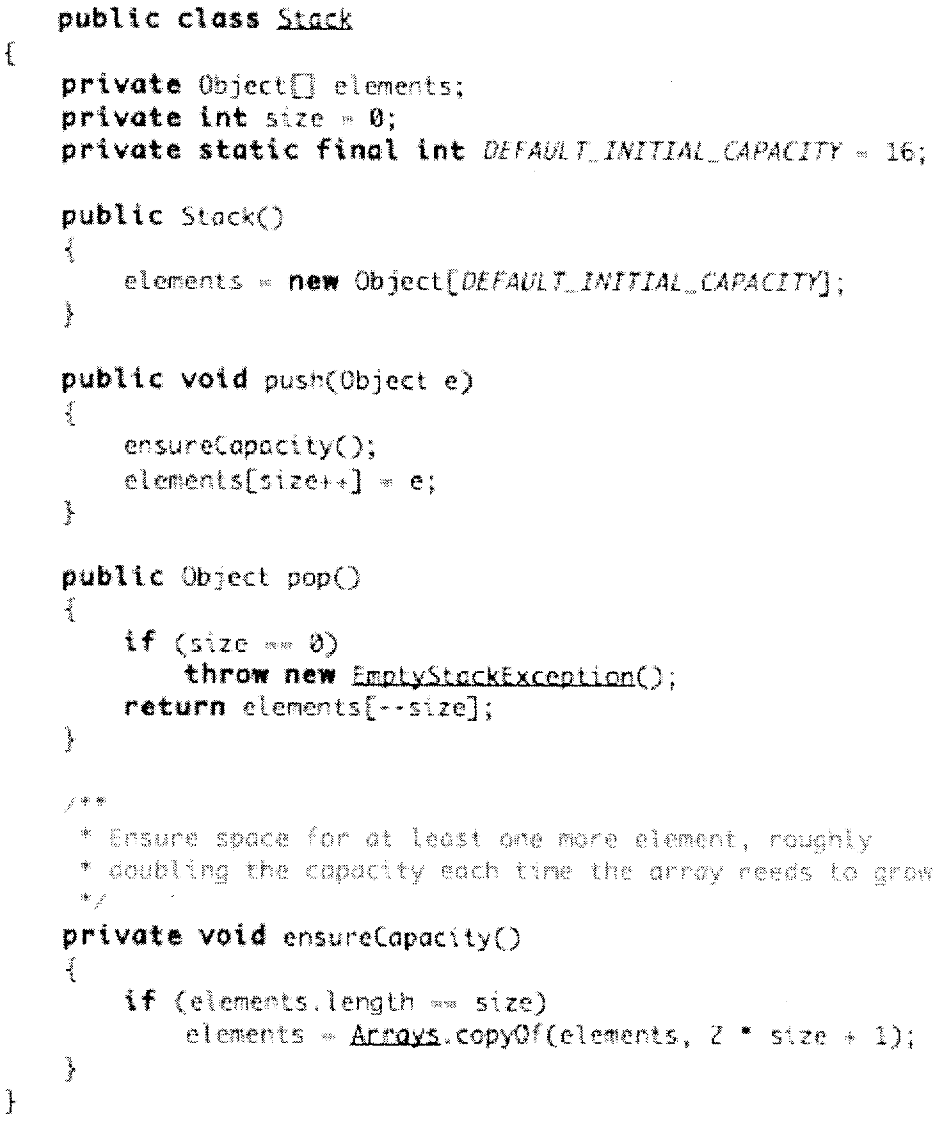
**Solution to question 3) comes here:**

Student Name:
Student #:

**Question 4) Garbage Collector and Java Memory Model** **[10]**

a) How is memory allocated on the *Stack*? How is memory deallocated on the *Stack*? [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);
    }
}
```

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. You can use the *main* function provided in appendix for this. [2]

b.3) Rewrite the `pop()` function such that the objects that are popped off the `Stack` class are now garbage collected. [2]

b.4) Draw the memory diagram again to show that objects popped off the `Stack` class (using your solution to b.3 above) are now garbage collected. You can use the *main* function provided in the appendix for this. [2]

**Question 5) Builder Design Pattern.** **[14]**

a) What is the *builder design pattern*? [1]

b) Give two advantages of the builder design pattern. [2]

c) Assume that the following `Person` class is used by the Government of Canada for census purposes:

```
class Person
{
      private final String firstName;       //required parameter
      private final String lastName;        //required parameter
      private final int sinNumber;          //required parameter

      private final String middleName;      //optional parameter
      private final String salutation;      //optional parameter
      private final boolean isEmployed;     //optional parameter
      private final boolean isHomeOwner;    //optional parameter

      public Person (int sin, String fName, String lName)
      {
            firstName=fName;
            sinNumber=sin;
            lastName=lName;
      }
```

```
    public Person (int sin, String fName, String lName, String mName)
    {
        firstName=fName;
        sinNumber=sin;
        lastName=lName;
        middleName=mName;
    }


    public Person (int sin, String fName, String lName, String mName,
    boolean employment)
    {
        firstName=fName;
        sinNumber=sin;
        lastName=lName;
        middleName=mName;
        isEmployed=employment
    }
    .
    .
    .
}
```

c.1) Write code in the given main function that will create a `Person` with the following parameters using the above `Person` class: [1]
**`FirstName`** = Gabriel, **`MiddleName`** = Garcia, **`LastName`** = Marquez, **`SINNumber`** = 1234

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




    }
}
```

c.2) What are two disadvantages related to creation of `Person` Objects of the above class? [2]

c.3) Rewrite the `Person` class using the *Builder Design Pattern.* [5]

c.4) Write code in the given main function that will create a `Person` with the following parameters for the `Person` class in c.3 above. [3]

**`FirstName`** = Gabriel, **`MiddleName`** = Garcia, **`LastName`** = Marquez, **`SINNumber`** = 1234

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







      }
}
```

**Question 6 ) Iterator Design Pattern.** **[14]**

a) What is one advantage of the *Iterator Design Pattern*? Give one example in relation to your Assignment2, where the Iterator design pattern is most applicable. [2]

b) Consider the following definition for a class that represents a **<u>Set</u>** of positive integers. [12]

```
import java.util.*;

public class Set implements Iterable<Integer> {
      // Represent a set of integers from 1 to myMaxPossible.
      // For each element k of the set, myValues[k-1] is true;
      // myValues entries for elements not in the set are all false.
      private boolean [ ] myValues;
      private int myMaxPossible;

      public Set (int maxElement) {
            myValues = new boolean [maxElement];
            myMaxPossible = maxElement;
      }

      // Add the element k to the set.
      public void addElement (int k) {
            myValues[k-1] = true;
      }

      public Iterator<Integer> iterator ( ) {
            //Complete this.




      }
}
```

You are now asked to write an Iterator (called **SetIterator**) for the above Set class. Supply the bodies of the **SetIterator** constructor and the **hasNext()** and **next ()** methods. You can add any new instance members to the SetIterator class as you see fit. You do not have to implement the **remove()** method. Execution of each of these methods should leave the **myIndex** variable indexing the next element to return in the set, or indexing past the end of the array if no more elements remain to be returned.

```
pubic class SetIterator implements ______________________
{
                // index of the next element of the set to return
                private int myIndex;


                public SetIterator(boolean [] array, int sizeOfArray)
                {







                }

                public boolean hasNext ( )
                {










                }

                public Integer next ( )
                {






                }
}
```

On a correct implementation of the `SetIterator`, the `main` function will print the following:
**Note:** You DO NOT have to modify or change the `SetTest` class.

```
class SetTest
{
          public static void main (String [ ] args) {
               Set s = new Set (9);
               s.addElement (3);
               s.addElement (6);
               s.addElement (9);
               Iterator<Integer> iter = s.iterator ( );
               System.out.println ("Set elements: ");
               while (iter.hasNext ( )) {
                    System.out.println (iter.next( ) + " ");
               }
          }

}
```

```
Set elements:
3
6
9
```

## Question 7) *Publisher/Subscriber* and *Singleton* Design Pattern [12]

a) Define *coupling* and *cohesion*. Give an example of each. [2]

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 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.** [8]

```
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 22 and page 23):**
**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

**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
```

The OperatingSystem 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()
      {













      }
}
```

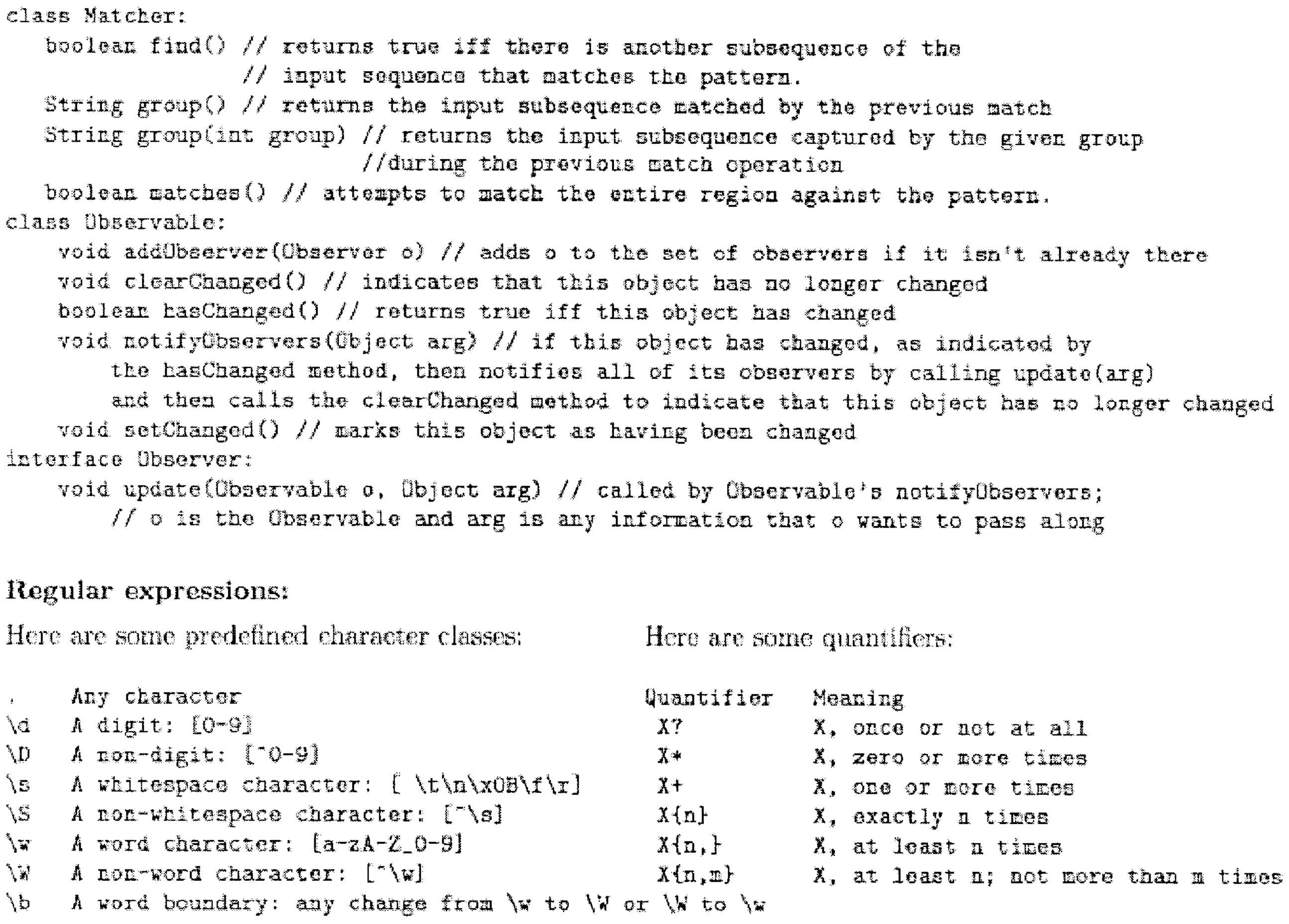
Student Name:
Student #:

```
public class WordFile ______________________________
{
      private String dataContents;
      private String fileName;
      private OperatingSystem referenceToOS=null;

      //Complete the body of the constructor and assign referenceToOS
      public WordFile (String fileName)
      {
            this.fileName=fileName;



      }
      //appends any new data to the dataContents and subscribes to the OS.
      //Complete this method
      public void setData(String content)
      {
            dataContents=dataContents+content;

      }
      //prompts the user whether to save the data or not
      //Complete this method
      private void promptUserToSaveData()
      {
             Scanner sc = new Scanner(System.in);
             System.out.println("Do you like to save the data for”+fileName+”?");
             if (sc.nextLine().equals("Yes")){
                    saveTheData();}
            else
            {
                    //The user does not wish to save the data.
                    //Complete this to unsubscribe from the OS.



            }
      }

      //this method will save/write the data on the file system and unsubscribes from the OS.
      //Complete this method
      public void saveTheData()
      {
            //You can assume that writeContentsOfFileToFileSystem() is already
            //implemented.
            writeContentsOfFileToFileSystem();
            System.out.println(fileName+"file has been successfully saved”);

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



}
```

## Appendix

```
    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
```

```
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
```



```
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 |

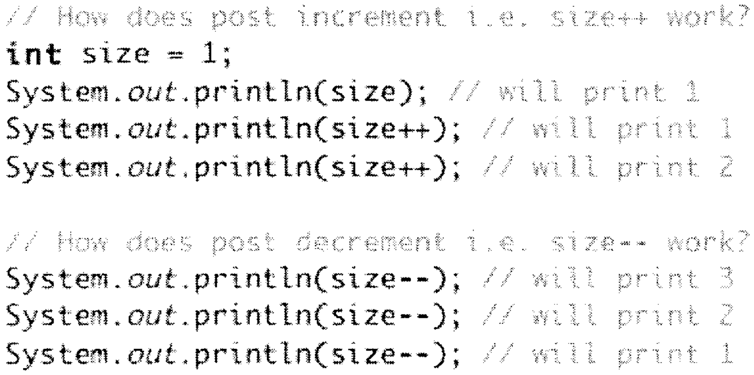
Here are some quantifiers:

| 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 |

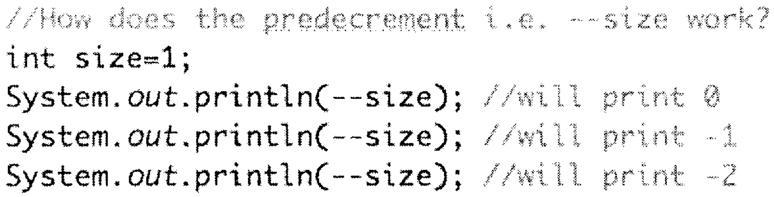
```
Class Observable:
-void addObserver(Observer o) //Adds an observer
-void clearChanged()
-int countObservers() //Counts the observer that are in the collection
-void deleteObserver(Observer o) //Deletes the observer
-void deleteObservers() //Deletes all the observers
-boolean hasChanged()
-void notifyObservers()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

**For Question 4):**

```
Class StackTest
{
     public static void main(String args[])
     {

          Stack s1=new Stack();
          .
          .
          .
          /*You can now assume at this point, that the Stack s1 is
populated with 5 arbitrary objects*/

          //Start and draw the memory diagram for the next two lines.
          s1.pop();
          s1.pop();
     }
}
```



```
// How does post increment i.e. size++ work?
int size = 1;
System.out.println(size); // will print 1
System.out.println(size++); // will print 1
System.out.println(size++); // will print 2

// How does post decrement i.e. size-- work?
System.out.println(size--); // will print 3
System.out.println(size--); // will print 2
System.out.println(size--); // will print 1
```



```
//How does the predecrement i.e. --size work?
int size=1;
System.out.println(--size); //will print 0
System.out.println(--size); //will print -1
System.out.println(--size); //will print -2
```

Student Name:
Student #:

## Extra Sheet: