Question 1. [8 MARKS]

For each question below, a correct answer earns 2 marks, "I don't know" earns 0 marks, and an incorrect answer earns -1 mark. **Do not guess.**

Part (a) [2 MARKS] Consider the following program.

```
public class MyProgram {
    public static void throwit() {
        throw new RuntimeException();
    }
    public static void main(String args[]) {
        try {
            System.out.println("Hello!");
            throwit();
            System.out.println("Done!");
        } finally {
            System.out.println("Finally!");
        }
    }
}
```

Which of the following most closely describes the behaviour of the program above? Circle one.

a. The program will not compile.

b. Print `Hello!`, then report a `RuntimeException`, then print `Done!`, then print `Finally!`.

c. Print `Hello!`, then report a `RuntimeException`, then print `Finally!`,

d. Print `Hello!`, then print `Finally!`, then report a `RuntimeException`.

e. I don't know.

Part (b) [2 MARKS] Consider the following program.

```
public class MyProgram {
    public static void throwit() {
        throw new Exception();
    }
    public static void main(String args[]) {
        try {
            System.out.println("Hello!");
            throwit();
            System.out.println("Done!");
        } finally {
            System.out.println("Finally!");
        }
    }
}
```

because no catch block here

Which of the following most closely describes the behaviour of the program above? Circle one.

a. The program will not compile.

b. Print `Hello!`, then report an `Exception`, then print `Done!`, then print `Finally!`.

c. Print `Hello!`, then report an `Exception`, then print `Finally!`,

d. Print `Hello!`, then print `Finally!`, then report a `Exception`.

e. I don't know.

Part (c) [2 MARKS] Consider this program.

```
public class TryIt {
    public static void main (String [] args) {
        Integer i = new Integer(42);
        Integer j = new Integer(42);
        System.out.println((i == j) + " " + i.equals(j));
    }
}
```

What is the output of the program above? Circle one.

a. `true true`

b. `true false`

c. `false true`

d. `false false`

e. I don't know.

Integer i and j are different Objects and store in different memory location

Part (d) [2 MARKS] Consider this program.

```
public class Document {
    public static int quantity = 0;
    public Document() {
        quantity = quantity + 1;
    }
    public static int getQuantity() {
        return quantity;
    }
}
public class Book extends Document {
    private int numChapters;
    public Book(int numChapters) {
        this.numChapters = numChapters;
    }
    public static void main(String[] args) {
      Book b1 = new Book(4);
      Book b2 = new Book(6);
      Document d1 = new Document();
      System.out.println(Document.getQuantity());
    }
}
```

What is the output of the program above? Circle one.

a. 0

b. 1

c. 2

d. 3

e. I don't know.

Question 2. [10 MARKS]

Consider the following Java code.

```
public interface Circle {
    public double getArea();
}

public class ThisCircle implements Circle {
    private double radius;
    public static double getArea(double r) {
        return Math.PI * r * r;
    }
    public double getArea() {
        return getArea(this.radius);
    }
}
   public abstract class Triangle {
       public static double getArea(double b,
                                    double h) {
           return b * h / 2;
       }
   }

   public class IsoscelesTriangle extends Triangle {
       private double base, height;
   }
```

For each of the code fragments below, *circle one answer*. A correct answer earns 1 mark, "I don't know" earns 0 marks, and an incorrect answer earns -0.5 marks. **Do not guess.**

1. `interface FunnyCircle implements Circle {}`

   interface can only extends another interface

   compiles [does not compile] I don't know

2. `interface FunnyCircle extends Circle {public double approxArea();}`

   [compiles] does not compile I don't know

   interface has public variable/field only

3. `interface ColouredCircle extends Circle {private String colour;}`

   compiles [does not compile] I don't know

4. ```
   class FunnyTriangle extends Triangle {
       public double wrongArea(double b, double h) {
           return getArea(b, h) * 42;
       }
   }
   ```

   [compiles] does not compile I don't know

5. ```
   class Strange extends Triangle implements Circle {
       double x;
       public double getArea() {
           return getArea(x, x);
       }
   }
   ```

   [compiles] does not compile I don't know

6.
```
public class Main {
    public static void main (String [] args) {
        Circle c = new ThisCircle();
    }
}
```

[compiles] does not compile I don't know

7.
```
public class Main {
    public static void main (String [] args) {
        IsoscelesTriangle t = new Triangle();
    }
}
```

Triangle is abstarct so cant instantiate

compiles [does not compile] I don't know

8.
```
public class Main {
    public static void main (String [] args) {
        Triangle t = new IsoscelesTriangle();
    }
}
```

[compiles] does not compile I don't know

9.
```
public class Main {
    public static void main (String [] args) {
        System.out.println((new IsoscelesTriangle()).base);
    }
}
```

base not initialized

compiles [does not compile] I don't know

10.
```
public class Main {
    public static void main (String [] args) {
        System.out.println((new IsoscelesTriangle()).getArea(4.2));
    }
}
```

[compiles] does not compile I don't know

Question 3. [10 MARKS]

In the space provided on the next page, give the output of the following program. You may use the bottom part of the next page for rough work. You may want to draw the Java memory model to help you trace the program. This model will not be graded; only the output of the program will be considered for grading.

```
public class A {
    public int i = 0;
    public static String s = "";

    public A(int i) {
        System.out.println(i);
        s += "x";
    }

    public A debug() {
        if (this instanceof B) {
            System.out.println("Spam");
            s += "s";
        }
        return this;
    }
}

public class B extends A {
    public int i = 100;
    public static String s = "s";

    public B(int i, String s) {
        super(i);
        this.i += 5;
        this.s = s;
    }

    public static void main (String [] argv) {
        String s = "";
        B b = new B(0, s);
        System.out.println(b.i + " " + b.s);
        s += "foo";
        A a = new B(42, s);
        System.out.println(a.i + " " + a.s);
        System.out.println(b.debug().s + " " + b.i + " " + b.s);
        System.out.println(a.debug().s + " " + a.i + " " + a.s);
    }
}
```

note creation of A rewrites the static s in B as well as in A

Question 3. (CONTINUED)

```
0
105
42
0 xx
Spam
xxs 105 foo
Spam
xxss 0 xxss
```

Question 4. [8 MARKS]

The following questions are related to Android application development. For each question below, a correct answer earns 2 marks, "I don't know" earns 0 marks, and an incorrect answer earns -1 mark. **Do not guess.**

Part (a) [2 MARKS]

Name the Android type that is used to pass data between two `Activity`s.

Answer: `Intent` or `I don't know`

Part (b) [2 MARKS]

Name an interface that a class must implement in order for instances of that class to be passed between `Activity`s:

Answer: `Serializable` (alternative: `Parceable`) or `I don't know`

Part (c) [2 MARKS]

Name the XML field that needs to be set in order to associate a particular Java method with a `Button` click event.

Answer: `onClick` or `I don't know`

Part (d) [2 MARKS]

A file is created in the default internal storage location for an application on a particular emulator. When that emulator is shut-down and relaunched, will the file persist? Circle one.

[yes] no I don't know

Question 5. [4 MARKS]

Complete the table below by naming the artifact from your course project that correponds to the artifact from Scrum. If relevant, specify which project phase. For each question below, a correct answer earns 1 mark, "I don't know" earns 0 marks, and an incorrect answer earns -0.5 marks. **Do not guess.**

Artifact from Scrum	**Corresponding artifact from your course project**	
Scrum Meeting	status meeting	or `I don't know`
Planning Meeting	planning meeting	or `I don't know`
Product Backlog	feature list from phase I	or `I don't know`
Sprint Backlog	feature list from phase II or III	or `I don't know`

Question 6. [8 MARKS]

Part (a) [2 MARKS] Write all strings that match this regular expression: `x?(0|1)y`

```
x0y
x1y
0y
1y
```

Part (b) [2 MARKS] Write a regular expression for a username of this form: `c` or `g`, followed by one digit, and followed by 1 to 6 lowercase letters.

```
(c|g)[0-9][a-z]{1,6}
```

Part (c) [2 MARKS] For each string below, circle the right answer to indicate whether or not it matches the regular expression: `[ab]c*d+e?`

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

Part (d) [2 MARKS] For each string below, circle the right answer to indicate whether or not it matches the regular expression: `([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

Question 7. [12 MARKS]

In this question you will implement a bounded stack – a stack with a fixed capacity. In addition, the stack is only allowed to hold objects of the same type. You will accomplish this using Java generics. Here is an example use of the class `BoundedStack` that you will write.

```
public class UseStack {
    public static void main (String [] argv)  {
        BoundedStack<String> s =
            new BoundedStack<String>(2, new ArrayList<String>());
        try {
            s.push("foo");
            s.push("bar");
            s.push("won't fit");
        } catch (StackFullException e) {
            System.out.println("Stack is full!");
        }
        try {
            System.out.println(s.pop());
            System.out.println(s.pop());
            s.pop();
        } catch (StackEmptyException e) {
            System.out.println("Stack is empty!");
        }
    }
}
```

This program should produce the following output:

```
Stack is full!
bar
foo
Stack is empty!
```

You may assume that `StackFullException` and `StackEmptyException` have been implemented as subclasses of `Exception`.

Complete the implementation of `BoundedStack`. Do not forget that this class should be **generic**. Do **not** use Java's `Stack` type. Use a `List` in your implementation.

```
// Complete the class declaration.
```

```
    public class BoundedStack<T> implements Iterable<T> {
```

```
    // Add instance variables, if needed.
    private List<T> content;
    private int capacity;
```

```
/**
 * Constructs a new BoundedStack with capacity cap with initial content con.
 * @param cap the capacity of the new BoundedStack.
 * @param con the initial content of the new BoundedStack.
 */
```

```
public BoundedStack(int cap, List<T> con) {
    capacity = cap;
    content = con;
}
```

```
/**
 * Removes and returns the item that was added last to this BoundedStack.
 * @return the item that was added last to this BoundedStack.
 * @throws StackEmptyException if this BoundedStack is empty.
 */
```

```
public T pop() throws StackEmptyException {
    if (this.content.isEmpty()) {
        throw new StackEmptyException();
    }
    T ret = this.content.get(this.content.size() - 1);
    this.content.remove(this.content.size() - 1);
    return ret;
}
```

```
/**
 * Pushes item onto this BoundedStack.
 * @param item the item to push onto this BoundedStack.
 * @throws StackFullException if the BoundedStack is at capacity.
 */
```

```
public void push(T item) throws StackFullException {
    if (this.content.size() == this.capacity) {
        throw new StackFullException();
    }
    this.content.add(item);
}
```

Question 8. [6 MARKS]

This question uses class `BoundedStack` from the previous question. Write JUnit methods to test that `BoundedStack`'s method `push` works correctly.

```
@Test
public void testStackEmpty() {
    BoundedStack<String> s = new BoundedStack<String>(2, new ArrayList());
    try {
        s.push("one");
        assertEquals("one", s.pop());
    } catch (StackFullException e) {
        fail("Unexpected " + e);
    } catch (StackEmptyException e) {
        fail("Unexpected " + e);
    }
}

@Test
public void testStackNonEmpty() {
    List<String> content = new ArrayList<>();
    content.add("one");

    BoundedStack<String> s = new BoundedStack<String>(3, content);
    try {
        s.push("two");
        assertEquals("two", s.pop());
    } catch (StackFullException e) {
        fail("Unexpected " + e);
    } catch (StackEmptyException e) {
        fail("Unexpected " + e);
    }
}

@Test(expected = StackFullException.class)
public void testStackFull() throws StackFullException {
    BoundedStack<String> s = new BoundedStack<String>(2, new ArrayList());
    s.add("hi");
    s.add("bye");
    s.push("full");
}
```

Question 9. [10 MARKS]

This question uses class `BoundedStack` from the previous two questions. You will now use the **iterator design pattern** to add iterator support for `BoundedStack`. Here is an example use of a foreach loop to iterate over the elements of a `BoundedStack`:

```
BoundedStack<Integer> st =
     new BoundedStack<Integer>(3, new ArrayList<Integer>());
for (int i = 0; i < 3; i++) {
    st.push(i);
}
for (Integer i : st) {
    for (Integer j : st) {
        System.out.print("(" + i + " " + j + ") ");
    }
}
```

The above snippet of code should produce the following output:

```
(2 2) (2 1) (2 0) (1 2) (1 1) (1 0) (0 2) (0 1) (0 0)
```

Notice that the items are visited in the order in which they would be popped, i.e. a last-in-first-visited order.

Part (a) [1 MARK]

Show how you need to modify the declaration of `BoundedStack`.

```
public class BoundedStack<T> implements Iterable<T> {
```

Part (b) [9 MARKS]

Now add whatever is necessary to the class `BoundedStack` to ensure that the foreach loop above works. You may continue on the next page.

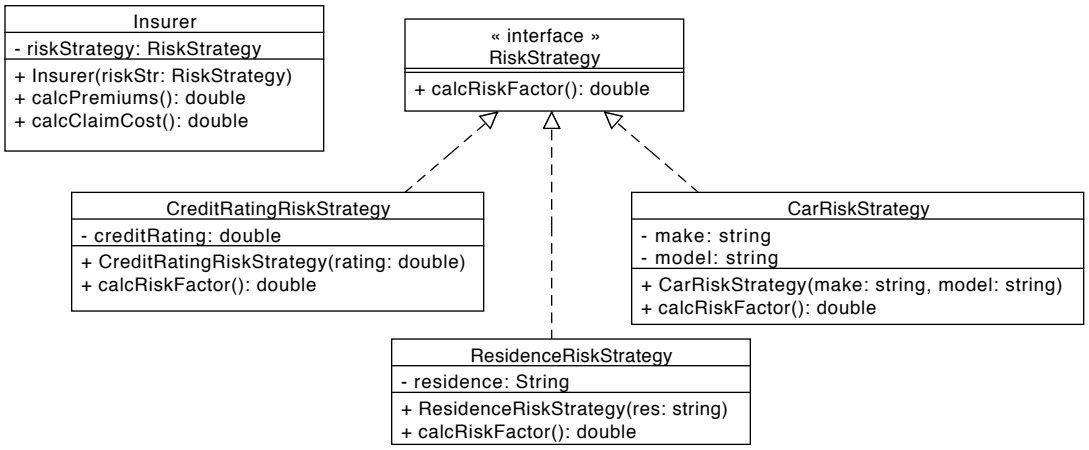
```
    @Override
    public Iterator<T> iterator() {
        return new StackIterator();
    }

    private class StackIterator implements Iterator<T> {
        private int cursor = content.size() - 1;

        @Override
        public boolean hasNext() {
            return this.cursor >= 0;
        }

        @Override
        public T next() {
            T ret = content.get(cursor);
            cursor--;
            return ret;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Not supported yet.");
        }
    }
```

Question 10. [8 MARKS]

Recall the Strategy Design Pattern.

Consider the following problem. You need to design a software system that deals with calculating car insurance premiums for auto insurance companies. Car insurance premiums are calculated based upon something called a *risk factor*, which is a number (a `double`) that predicts how likely it is that a customer with a particular set of circumstances will make a claim. There are several ways the risk factor could be calculated for a customer, including (a) based on customer's credit rating, (b) based on customer's place of residence, and (c) based on the car make and model. The risk factor is used to calculate, among other things, the car insurance premiums (a `double`) and how much a possible claim will likely cost (a `double`).

Draw a UML class diagram that represents your solution to the above problem using the Strategy design pattern.



Question 11. [6 MARKS]

Consider the following Java implementation of an `AirConditioner` class that makes use of a `BasicThermostat`.

```
public class AirConditioner {

    /** This AirConditioner's thermostat. */
    private BasicThermostat thermostat;

    /**
     * Creates a new AirConditioner.
     */
    public AirConditioner() {
        thermostat = new BasicThermostat();
    }

    /**
     * Turns on this AirConditioner if necessary.
     */
    public boolean turnOn() {
        if thermostat.aboveTarget() { ... }
        else { ... }
}

public class BasicThermostat {

    /** This BasicThermostat's target temperature. */
    private double targetTemperature;

    /**
     * Creates a new BasicThermostat.
     */
    public BasicThermostat() {}

    /**
     * Sets the target temperature of this BasicThermostat to temperature.
     * @param temperature the new target temperature of this BasicThermostat.
     */
    public void setTemperature(double temperature) { ... }

    /**
     * Returns whether the current temperature is above target.
     * @retrun true, if the current temperature is above target,
     * and false, otherwise.
     */
    public boolean aboveTarget() { ... }
}
```

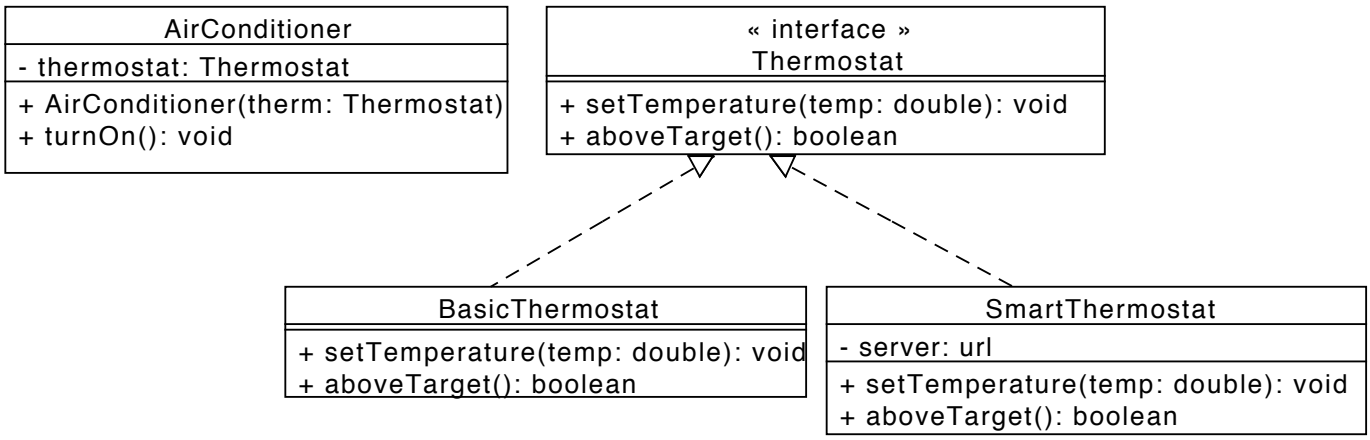
Part (a) [1 MARK]

Name the OO Design Principle that this design violates.

Dependency inversion

Part (b) [5 MARKS]

Provide a UML class diagram of a good solution that addresses the problem you identified in the previous part.



Question 12. [8 MARKS]

In Phases I, II, and III of the team project, you developed an object-oriented design of a hospital emergency room triage application.

Provide **two** examples of poor design decisions you made while designing the application. Explain why these were poor design choices. Avoid generic statements, such as "We used too few classes." or "It was not modular.". Instead, explain specifically what problems you encountered during your implementation/development as a result of the poor design choices.

Poor design decision #1:

Consequences of poor design decision #1:

Poor design decision #2:

Consequences of poor design decision #2:

Short Java APIs:

```
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
```

```
    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 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
```