PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL 2015 EXAMINATIONS

CSC 207 H1S
Instructor: Campbell

Duration — 3 hours

Examination Aids: None

Student Number: |_|_|_|_|_|_|_|_|_|_|

Family Name(s): _____

Given Name(s): _____

---

*Do **not** turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully.*

You must get 40% or above on this exam to pass the course (at least 34 out of 85); otherwise, your final course grade will be no higher than 47.
This final examination paper consists of 8 questions on 22 pages (including this one). *When you receive the signal to start, please make sure that your copy of the final examination is complete.*

- Legibly write your name and student number on this page.
- Legibly write your student number at the bottom of every odd page (except this one), in the space provided.
- If you use any space for rough work, indicate clearly what you want marked.
- In all programming questions you may assume all input is valid.
- You do not need to write Javadocs or internal comments.

# 1:	_____/11
# 2:	_____/10
# 3:	_____/ 8
# 4:	_____/11
# 5:	_____/15
# 6:	_____/22
# 7:	_____/ 4
# 8:	_____/ 4
TOTAL:	_____/85

Good Luck!

CONT'D...

Question 1. [11 MARKS]

Consider this Java code, which compiles and runs without error:

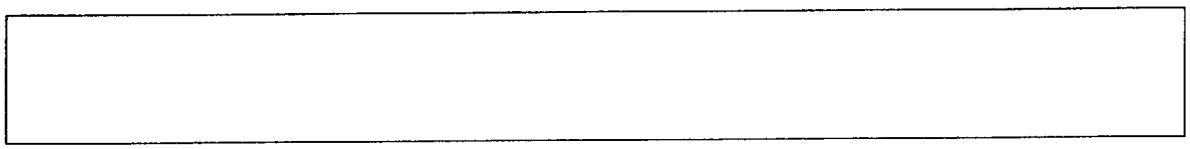
```
public abstract class Publication {

    public static int count = 0;
    private String title;
    private String date = "01012001";

    public Publication(String title) {
        this.title = title;
        count++;
    }
    public String getTitle() {
        return this.title;
    }
    public String getDate() {
        return this.date;
    }
    public String type() {
        return "Publication";
    }
    public String toString() {
        return "Title: " + getTitle();
    }
}
```

```
public class Newspaper extends Publication {

    private String date; // DDMMYYYY

    public Newspaper(String title, String date) {
        super(title);
        this.date = date;
    }
    public String getDate() {
        return this.date;
    }
    public String type() {
        return "Newspaper";
    }
}
public class Magazine extends Publication {

    private String date; // DDMMYYYY

    public Magazine(String title, String date) {
        super(title);
        this.date = date;
    }
    public String getDate() {
        return this.date;
    }
    public String type() {
        return "Magazine";
    }
}
```

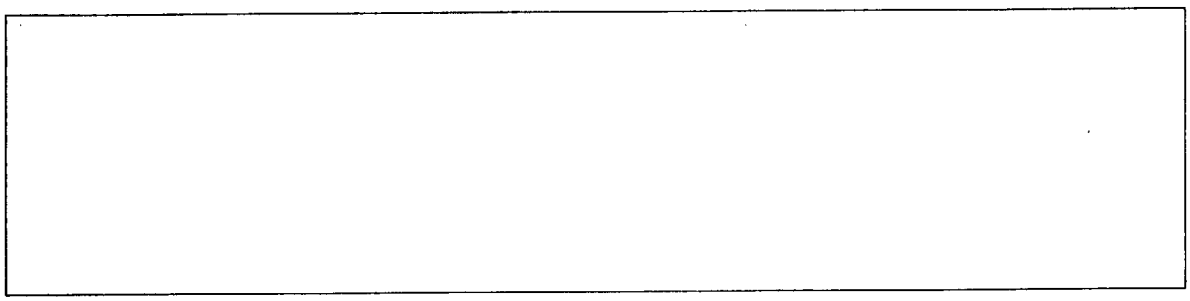
In the box below each code fragment, write what that code prints.

```
public class Main {

    public static void main(String[] args) {
        Publication globe = new Newspaper("globe", "05052005");

        System.out.println(Publication.count);
```

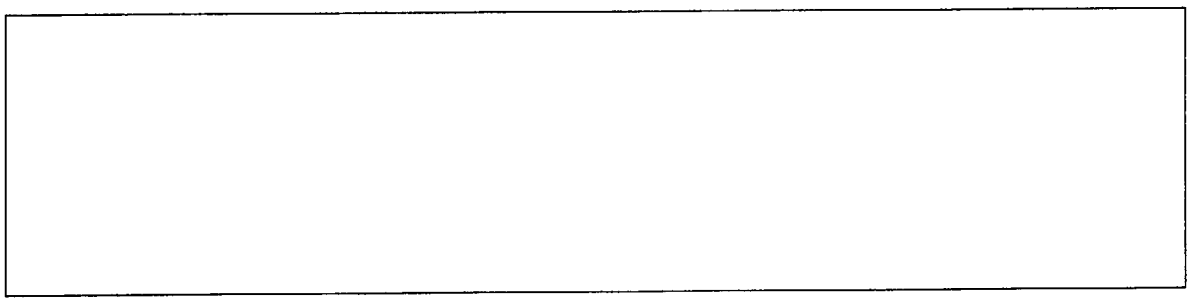
```
Newspaper post = new Newspaper("post", "08081980");
Magazine uoft = new Magazine("uoft", "04042004");

Publication[] pubs = new Publication[] {globe, post, uoft};

System.out.println(Publication.count);
```



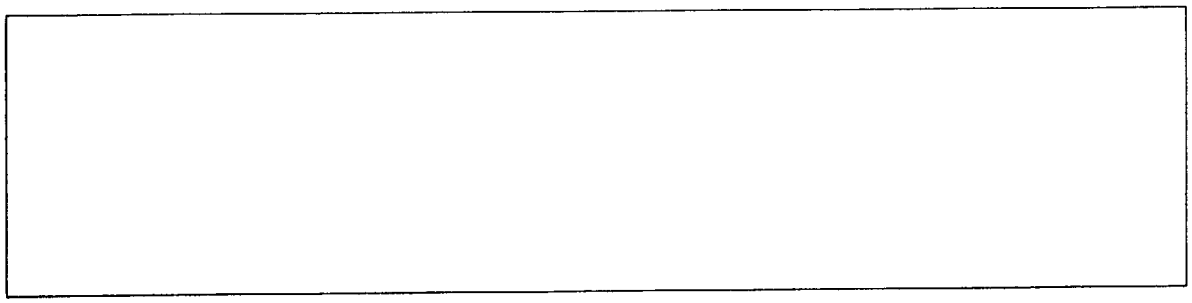
```
for (Publication p : pubs) {
    System.out.println(p);
}
```



```
for (Publication p : pubs) {
    System.out.println(p.type());
}
```



```
for (Publication p : pubs) {
    System.out.println(p.getDate());
}
```



```
}
```

Question 2. [10 MARKS]

Consider this Java code, which compiles and runs without error:

```
public abstract class Vehicle {
    public void move() {
        System.out.println("Moving");
    }
}

public abstract class MotorizedVehicle extends Vehicle {
    public void motor() {
        System.out.println("Motoring");
    }
}

public class Boat extends MotorizedVehicle {
    private int length;
    public Boat(int length) {
        this.length = length;
    }
}

public interface Wheeled {
    public int getNumWheels();
}

public class Car extends MotorizedVehicle implements Wheeled {
    public int getNumWheels() {
        return 4;
    }
}

public abstract class Cycle implements Wheeled {
    public abstract void roll();
}

public class Bicycle extends Cycle {
    public int getNumWheels() {
        return 2;
    }
    public void roll() {
        System.out.println("Rolling");
    }
}

public interface Winged {
    public void flies();
}

// This question continues on the next two pages.
```

For each of the code fragments below, *circle one answer.*

Part (a) [1 MARK]

```
public class SpeedBoat extends Boat {}
```

compiles does not compile

Part (b) [1 MARK]

```
public class Unicycle extends Cycle {
    public void roll() {
        System.out.println("Balancing");
    }
}
```

compiles does not compile

Part (c) [1 MARK]

```
public class Airplane implements Wheeled, Winged {
    public void flies() {
        System.out.println("Flying");
    }
    public int getNumWheels() {
        return 3;
    }
}
```

compiles does not compile

Part (d) [1 MARK]

```
public class Motorcycle extends MotorizedVehicle, Cycle {
    public int getNumWheels() {
        return 2;
    }
    public void roll() {
        System.out.println("Speeding");
    }
}
```

compiles does not compile

Part (e) [1 MARK]

```
public static void main(String[] args) {
    Car car = new Car();
    car.move();
}
```

compiles does not compile

Part (f) [1 MARK]

```
public static void main(String[] args) {
    Vehicle car = new Car();
    ((Car) car).getNumWheels();
}
```

compiles does not compile

Part (g) [1 MARK]

```
public static void main(String[] args) {
    Wheeled bike = new Bicycle();
    bike.getNumWheels();
}
```

compiles does not compile

Part (h) [1 MARK]

```
public static void main(String[] args) {
    Vehicle vehicle = new MotorizedVehicle();
}
```

compiles does not compile

Part (i) [1 MARK]

```
public static void main(String[] args) {
    MotorizedVehicle motorBoat = new Boat(27);
    System.out.println(motorBoat.length);
}
```

compiles does not compile

Part (j) [1 MARK]

```
public static void main(String[] args) {
    Cycle myRide = new Bicycle();
    ((Wheeled) myRide).getNumWheels();
}
```

compiles does not compile

Question 3. [8 MARKS]

Class `WordOrganizer` is used to organize words by their first letter. It reads lowercase words from a file containing one word per line and populates a `Map<String, Integer>` where each key is a lowercase letter and each value is the number of words from the file that begin with that letter. Implement class `WordOrganizer` so that the `main` method below works as shown. Your design should include helper methods.

```
public class WordOrganizer {
    private Map<String, Integer> letterToWords;

    public WordOrganizer(Scanner sc) {
```

```
    public static void main(String[] args)
      throws FileNotFoundException {
        WordOrganizer wo =
          new WordOrganizer(new Scanner(new File("words.txt")));
        System.out.println(wo.letterToWords);
}
```

Example words.txt:

```
apple
banana
celery
pear
peach
carrot
pepper
```

The program prints:
`{a=1, b=1, c=2, p=3}`
(the order of the key/value pairs is irrelevant)

Question 4. [11 MARKS]

Consider this Java code, which compiles and runs without error:

```
public class BankAccount {

    private double balance;

    public double deposit(double amount) {
        balance = balance + amount;
        return balance;
    }

    public double withdraw(double amount) {
        balance = balance - amount;
        return balance;
    }
}
```

Part (a) [1 MARK] Define a new checked exception class named `InsufficientFundsException`. It needs only one constructor with no arguments.

Part (b) [2 MARKS] In the space below, implement a revised version of method `withdraw`. In this version, if the amount to be withdrawn is greater than the available balance, an `InsufficientFundsException` is thrown and the balance is unchanged. You may assume the amount to be withdrawn is a positive, non-zero amount.

Part (c) [8 MARKS] Write JUnit methods to thoroughly test that `BankAccount`'s revised method `withdraw` (from Part (b)) works correctly. You may assume the amount to be withdrawn is a positive, non-zero amount.

```
public class BankAccountTest {


    @Before
    public void setUp() throws Exception {


    }

    @After
    public void tearDown() throws Exception {


    }

    // Write your test methods here:
```

Question 5. [15 MARKS]

Part (a) [11 MARKS]

In Exercise 3, you implemented several classes using Java's `java.util.Observer` interface and `java.util.Observable` class. You will now implement your own version of that interface and class: an interface `MyObserver` and a class `MyObservable`. The goal is to be able to use these implementations instead of Java's `Observer` and `Observable`. Here is the specification:

```
interface MyObserver:
    void update(MyObservable o, Object arg)
        - called by MyObservable's notifyObservers;
        - o is the MyObservable and arg is any information that o wants to pass along

class MyObservable:
    MyObservable()
        - initializes MyObservable's instance variables
    void addObserver(MyObserver o)
        - adds o to the set of observers if it isn't already there
    void setChanged()
        - marks this object as having been changed
    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
```

Here are a few guidelines:

- you must **not** use `java.util.Observer` or `java.util.Observable`
- implement only the methods listed above
- you must decide what, if any, instance variables to include

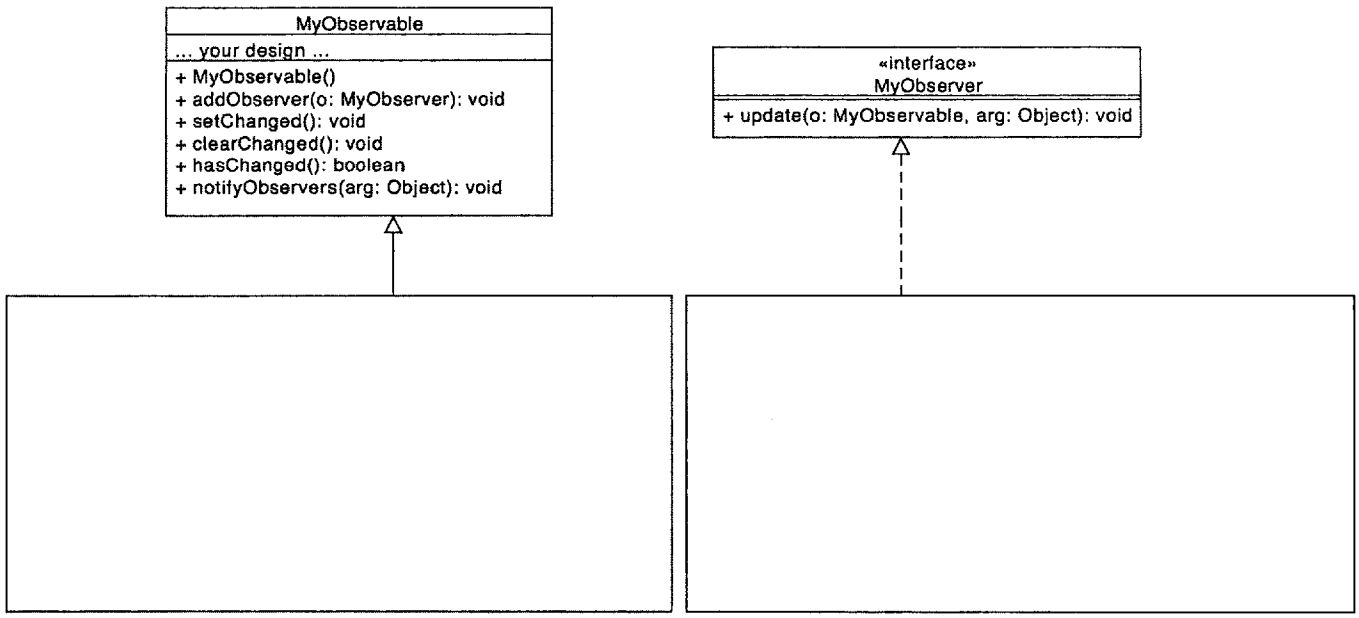
```
public interface MyObserver {




}
```

```
public class MyObservable {



    public MyObservable() {



    }
    public void addObserver(MyObserver o) {



    }
    public void setChanged() {


    }
    public void clearChanged() {


    }
    public boolean hasChanged() {


    }
    public void notifyObservers(Object arg) {









    }
}
```

Part (b) [4 MARKS]

An airline has a rewards program in which customers are awarded points for each flight that they take. Once a customer has earned a sufficient number of points, they are automatically considered a frequent flyer.

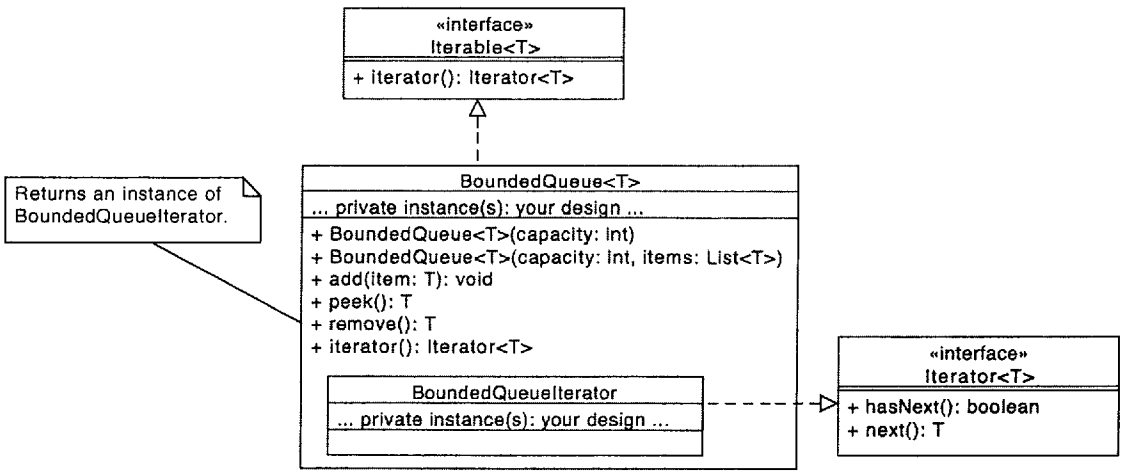
Use the Observer design pattern (using your `MyObserver` and `MyObservable`) to design your solution:



Use class names `Customer` and `FrequentFlyerProgram`. You do not need to design anything in these classes except for the variables methods that realize the Observer design pattern. You do not need to implement these classes.

Question 6. [22 MARKS]

In this question, you will implement a generic, iterable class named `BoundedQueue`:



You must use Java's `Iterator` and `Iterable` interfaces, which are described in the Java API provided at the back of the exam. Note: you may completely omit `BoundedQueueIterator`'s `remove` method (but you must implement `BoundedQueue`'s `remove` method).

A bounded queue is a first-in, first-out data structure with a fixed capacity. The queue is only allowed to hold objects of the same type and you will accomplish this using Java generics. In additional, the queue must be iterable. Here is an example use of the class `BoundedQueue` that you will write.

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

    BoundedQueue<String> queueOne = new BoundedQueue<>(3);
    Integer[] myArray = new Integer[] {1, 2, 3, 4};
    BoundedQueue<Integer> queueTwo = new BoundedQueue<>(5, Arrays.asList(myArray));

    try {
        queueOne.add("How");
        queueOne.add("are");
        queueOne.add("you");
        queueOne.add("today?");
    } catch (QueueFullException e) {
        System.out.println("The queue is full.");
    }

    for(String s : queueOne) {
        System.out.println(s);
    }

    queueTwo.remove();
    System.out.println(queueTwo.peek());

    for(Integer i: queueTwo) {
        System.out.println(i);
    }
}
```

This code prints:

```
The queue is full.
How
are
you
2
2
3
4
```

More space for your answer to the BoundedQueue question:

```
// Complete the header:
public class BoundedQueue

    // Add instance variables, if needed.




    /**
     * Constructs a BoundedQueue with the given capacity.
     * @param capacity The maximum number of elements that the new BoundedQueue
     *  can hold (a non-negative int).
     */
     public BoundedQueue(int capacity) {




    /**
     * Constructs a BoundedQueue with the given capacity and populates it with
     * the given items.  Precondition: items.size() <= capacity
     * @param capacity The maximum number of elements that the new BoundedQueue
     *  can hold (a non-negative int).
     * @param items The initial elements in this BoundedQueue.
     */
     public BoundedQueue(int capacity, List<T> items) {




    /**
     * Adds item to the end of this BoundedQueue.
     * @param item The item to add to the end of this BoundedQueue.
     * @throws QueueFullException If this BoundedQueue is full.
     */
```

```
/**
 * Returns, but does not remove, the head of this BoundedQueue,
 * or returns null if this BoundedQueue is empty.
 * @returns The head of this BoundedQueue or null if empty.
 */
```

```
/**
 * Removes and returns the head of this BoundedQueue, or
 * throws a NoSuchElementException if empty.
 * @return The head of this BoundedQueue.
 */
```

```
// Add whatever else is needed to implement Iterable<T>.
```

More space for your answer to the `BoundedQueue` question:

Question 7. [4 MARKS]

Part (a) [2 MARKS] Which of the following are lessons learned from our lecture on floating point notation? Circle all that apply.

a. When adding floating point numbers, avoid adding similar quantities.
b. When adding floating point numbers, avoid adding dissimilar quantities.
c. Use fewer arithmetic operations where possible.
d. Use more arithmetic operations where possible.

Part (b) [1 MARK] What is *proper rounding*?

Part (c) [1 MARK] Why is *proper rounding* done?

Question 8. [4 MARKS]

Part (a) [1 MARK] Write a regular expression that matches only non-empty binary strings of 1s and 0s (e.g., `1`, `0`, `101`, `1001101101101`, and so on).

Part (b) [2 MARKS] A *palindrome* is a word that is the same forward as backward. For example, *deed*, *noon*, *kayak*, and *level* are all palindromes. Write a regular expression that matches only lowercase palindromes with lengths of 4 or 5.

Part (c) [1 MARK] Administrators of a computing network require passwords to be:

- made up of only letters and digits, and
- to be 8-12 characters long.

Write a regular expression that matches only valid passwords. For full marks, be concise.

Total Marks = 85

[Use the space below for rough work. This page will **not** *be marked, unless you clearly indicate the part of your work that you want us to mark.]*

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()          // 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
```


CONT'D...

```
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