CIS 1100 — Spring 2023 — Exam 2

Full Name: ____

PennID (e.g. 12345678): ___________________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

_______________________________   ___________________________

Signature   Date

Instructions are below. Not complying will lead to a 0% score on the exam.

- Do not open this exam until told by the proctor.
- You will have exactly 120 minutes to take this exam.
- Make sure your phone is turned OFF (not on vibrate!) before the exam starts.
- Food and gum are strictly forbidden. Masks are optional.
- You may not use your phone or open your bag for any reason
- This exam is closed-book, closed-notes, and closed computational devices.
- If you get stuck on a problem, it may be to your benefit to move on to another question and come back later.
- All code must be written in proper Java format, including all curly braces and semicolons.
- Do not separate the exam pages. Do not take any exam pages with you. The entire exam packet must be turned in as is.
- There are two blank pages near the end of the exam packet if you need extra space for any graded answers.
- Use a pencil, or blue or black pen to complete the exam.
- If you have any questions, raise your hand and a proctor will come to you.
- When you turn in your exam, you may be required to show your PennCard. If you forgot to bring your ID, talk to an exam proctor immediately.
- We wish you the best of luck!

There are 10 questions on the exam (plus an 11th bonus question at the end). We have also included an appendix at the very end of the exam that includes some functions that may be useful for you as you complete the exam.

All your answers must be placed in the answer box below each question. Only answers placed in the designated area will be graded.

Q1. Object Critiques

Question 1.1:

Consider the code below:

```
public class A {
    private int foo;
    public String bar;
    private double baz;

    public A(int a, String b) {
        foo = a;
        bar = b;
        baz = Math.random();
    }

    public int getFoo() {
        return foo;
    }

    public void setFoo(int newFoo) {
        if (newFoo > foo) {
            foo = newFoo;
        }
    }
}
```

Which **two** of the following statements are valid **critiques** about the *design* of the `A` class? (Points awarded for each line correctly marked/left unmarked.)

Answer:

- ☐ The class features a mixture of public and private instance variables
- ☐ The constructor for `A` initializes baz without taking in a corresponding parameter for baz as input.
- ☐ The constructor's assignment operations are missing "`this`", which would cause the program to behave incorrectly.
- ☐ `baz` is never used inside the class and is inaccessible outside of the class, making it useless.

For Q1.2, Q1.3 & Q1.4 consider the following code:

```
public class Counter {
	private int count;

	public Counter() {
		count = 0;
	}

	public void incr() {
		count++;
	}

	public int show() {
		return count;
	}

	public void reset() {
		count = 0;
	}

	public static void main(String[] args) {
		Counter c = new Counter();
		c.incr();
		c.incr();

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

Question 1.2:

Would the **main method** compile as written?

Answer:
☐ Yes ☐ No

Question 1.3:

Which of the methods written is functionally a "getter" (accessor) method?

```
A. public Counter()
B. public void incr()
C. public int show()
D. public void reset()
```

Answer: _____

Question 1.4:

Which one of the following is an invariant of the `Counter` class? (Remember that an invariant is a statement that is always true about the state of the object.)

A. For a given `Counter` object, we know that `count` is always exactly equal to the number of times that `incr()` has been called.
B. The value of `count` can only increase after initialization and it will never decrease.
C. The value of `count` can only decrease when the user calls the `incr()` method with a negative input.
D. For a given `Counter` object, we know that `count` is always less than or equal to the number of times that `incr()` has been called.

Answer: _____

Q2. References

Question 2.1:

```
public class Greetable {
     private String name;

     public Greetable(String name) {
          this.name = name;
     }

     public void setName(String newName) {
          this.name = newName;
     }

     public void greet() {
          System.out.println("Hi, I'm " + name);
     }

     public static void main(String[] args) {
          Greetable pilar = new Greetable("Pilar");
          Greetable anselmo = new Greetable("Anselmo");

          anselmo = pilar;
          anselmo.greet();
          pilar.setName("Maria");
          anselmo.greet();
          pilar.setName("Pablo");
          pilar.greet();
          pilar = new Greetable("Jordan");
          anselmo.greet();
     }
}
```

Fill in the blanks to represent what gets printed when we run `java Greetable`.

1	Hi, I'm ________
2	Hi, I'm ________
3	Hi, I'm ________
4	Hi, I'm ________

Question 2.2:

Inspect the following class and write a reasonable and meaningful implementation of the `equals()` method. There are several valid answers, but at a minimum your answer should behave differently from using == ("double equals") to compare two objects.

```
public class Student {
    // guaranteed to uniquely identify a "Student"
    private int uniqueID;

    // multiple "Student" objects could have the same data
    private String name;

    public Student(String name) {
        // generateUniqueID() implementation omitted,
        // but assume it exists!
        uniqueID = generateUniqueID();
        this.name = name;
    }

    // returns true when two objects are structurally equal to
    // each other and false otherwise.
    public boolean equals(Student other) {
        // your implementation here!

    }

}
```

Q3. Abstract Data Types

Question 3.1:

Given an interface `Drink` and a class `Soda` that implements the `Drink` interface, **select all of the lines below that would compile** when written in `main` of `Soda.java`. (Points awarded for each line correctly marked/left unmarked.)

Answer:

- ☐ `Drink d = new Soda();`
- ☐ `Drink d = new Drink();`
- ☐ `Soda s = new Soda();`
- ☐ `Soda s = new Drink();`

Question 3.2:

For the following question, it is given that `Soda` and `Drink` both compile, `Soda` implements `Drink`, and `Soda.java` contains a method with signature `public void open()`.

We can assume that `Drink.java` must contain an abstract method with the same signature (`public void open()`).

Answer:

☐ True ☐ False

Q4. List

For the following questions, you may find it useful to refer to the “Lists” section of the Appendix.

Question 4.1:

Inspect the following function:

```
public static int mystery(List<Double> ds) {
    int x = 0;
    for (int i = ds.size() - 1; i >= 0; i--) {
        if (ds.get(i) < 0) {
            ds.remove(i);
            x++;
        }
    }
    return x;
}
```

How does a list passed as input to `mystery` change as the function is executed? Specifically, **how are the contents of list *ds* different after the function finishes executing?**

What does the above function's ***return value*** represent?

Question 4.2:

Read the following function that takes an input array named `arr` and returns a new array that contains the contents of `arr` repeated `k` times:

```
public static String[] loopKTimes(String[] arr, int k) {
    String[] output = new String[A.length * k];
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < k; j++) {
            output[i + j * arr.length] = arr[i];
        }
    }
    return output;
}
```

For example, `loopKTimes(new String[] {"hi", "bye"}, 3)` returns `{"hi", "bye", "hi", "bye", "hi", "bye"}`.

Fill in the following function stub so that the function mimics the behavior of `loopKTimes` but takes in and returns an ArrayList instead. Remember that ArrayLists are *dynamically resizing.*

```
public static ArrayList<String> loopKTimes(ArrayList<String> lst, int k) {
    ArrayList<String> output = new ArrayList<String>();
    // Place your code here










    return output;
}
```

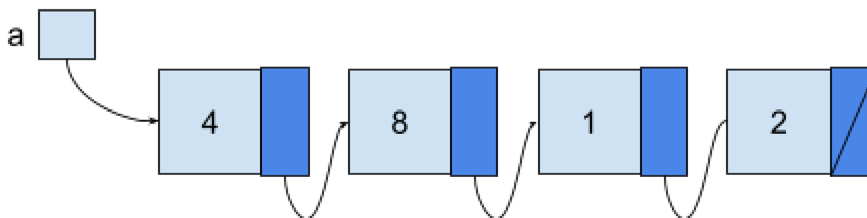
Q5. Nodes

For Question 5, assume we have the following `Node` class:

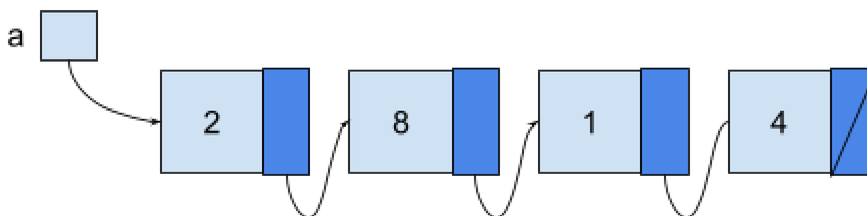
```
/**
 * This node class will store int values.
 */
public class Node {

    // public instance variables
    public int data;
    public Node next;

    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

Question 5.1:

Given the following state of nodes:



rearrange the nodes so that it looks like:



To be clear, you do not have to generalize this to work on any sequence of Nodes. You are given the chain drawn in the first image and you have to transform it into the chain drawn in the second image.

<u>NOTE</u>: you **are not allowed to** access or modify the value in any nodes `data` instance variable, and you are not allowed to allocate a `new node` (e.g. the code you write should not contain `new Node(..., ...)` in it.)

You **are allowed to** create Node reference variables with something like: `Node temp = a`

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

    // Sets up the initial state of nodes
    Node a = new Node(4, null);
    a.next = new Node(8, null);
    a.next.next = new Node(1, null);
    a.next.next.next = new Node(2, null);


    // TODO: put your code here:

}
```

Question 5.2:

Complete the following function:

```
/* Given a reference to the start of a chain of linked nodes
 * and an integer we are searching for.
 *
 * return true if the target is found in the chain,
 * and false if it is not found.
 */
public static boolean search(Node head, int target) {
     // TODO

}
```

Q6. Sorting

For both questions below, consider the following (broken) implementation of selection sort

Line Number	Code
0	`public static void selectionSort(double[] array) {`
1	`    for (int i = 0; i < array.length - 1; i++) {`
2	`        int indexOfSmallest = i;`
3	`        for (int j = 0; j < array.length; j++) {`
4	`            if (array[indexOfSmallest] < array[j]) {`
5	`                indexOfSmallest = j;`
6	`            }`
7	`        }`
8	
9	`        double temp = array[indexOfSmallest];`
10	`        array[indexOfSmallest] = array[i];`
11	`        array[i] = temp;`
12	`    }`
13	`}`

As mentioned, the code above does not work (though, it still compiles).

Question 6.1:

Consider the following array:
`double[] nums = {4.5, 1.2, -10, 30};`
If we were to pass this array into the function (e.g. calling `selectionSort(nums)`), what would the `nums` array look like after `selectionSort(nums)` returns?
Hint: if it was properly sorting, `nums` would look like: `{-10, 1.2, 4.5, 30}`
The code shown above will NOT leave `nums` in sorted order

Question 6.2

Let's try to fix the `selectionSort` code above so that it correctly sorts any array that is passed as a parameter. For each line that needs to be changed, list its line number and then what the code should be changed to be. Your solution should only need to change 2 lines of code.

Line Number	Revised Code

Q7. 2D Arrays

Question 7.1:

Fill in the contents of the 2D array as initialized by this code here:

```
int[][] output = new int[2][3];
for (int row = 0; row < output.length; row++) {
    for (int col = 0; col < output.length; col++) {
        if (row <= col) {
            output[row][col] = row + col;
        } else {
            output[row][col] = row - col;
        }
    }
}
```

Put your answers in the table below

	Column 0	Column 1	Column 2
Row 0			
Row 1			

Q8. Comparing Objects

Question 8.1:

For the following questions, consider the following `Treasure` class:

```
public class Treasure {
    public int value;
    public Treasure(int value) {
        this.value = value;
    }
    /* Other methods omitted */
}
```

While there are many valid implementations of a `compareTo()` method for `Treasure`, any valid implementation should have the following properties:

1. The method should compile!
2. `x.compareTo(y)` should return a negative number if `x` has a `value` less than y's `value`.
3. `x.compareTo(y)` should return a positive number if `x` has a `value` greater than y's `value`.
4. `x.compareTo(y)` should return `0` if `x` and `y` have the same `value`.

For each of the following `compareTo()` methods, decide if it is valid or invalid. If it is invalid, explain why with reference to the above properties. **Put your answers in the table on the next page**

Option 1:

```
public int compareTo(Treasure o) {
    if (this == o) {
        return 0;
    } else if (this < o) {
        return -1;
    } else {
        return 1;
    }
}
```

Option 2:

```
public int compareTo(Treasure o) {
    return o.value - this.value;
}
```

Option 3:

```
public int compareTo(Treasure o) {

    if (this.value == o.value) {
        return 0;
    } else if (this.value < o.value) {
        return -1 - (int) (Math.random() * 10);
    } else {
        return 1 + (int) (Math.random() * 10);
    }
}
```

Option	Valid/invalid and explanation
1	
2	
3	

Q9. Directories

For the following question, you may find it useful to refer to the "Directories" section of the Appendix.

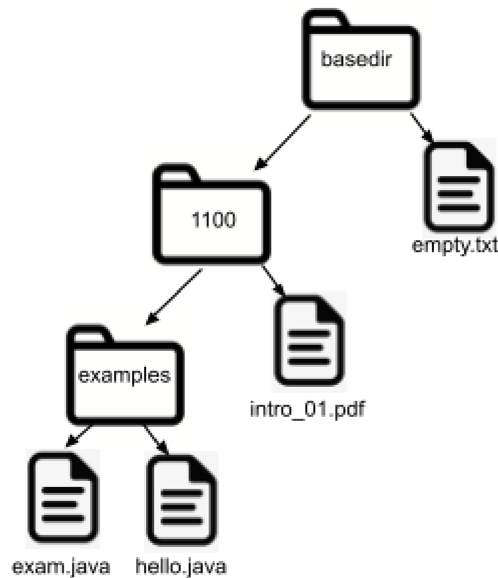
Question 9.1:

Consider the following function:

```
/* Given a path, determine if that path specifies a parent directory
 * A parent directory satisfies two conditions:
 * - it is a valid directory
 * - it contains at least one directory inside of it
 */
public static boolean isParentDirectory(String path) {
    File f = new File(path);
    if (!f.isDirectory()) {
        return false;
    }
    File[] entries = f.listFiles();
    for (int i = 0; i < entries.length; i++) {
       if (entries[i].isDirectory()) {
           return true;
       }
    }
    return false;
}
```

Problem continued on the next page

and the following file tree diagram:



Select all of the following paths that correspond to a parent directory and thus would make `isParentDirectory` return `true` when that path is passed in.
(Points awarded for each line correctly marked/left unmarked.)

- ☐ `basedir`
- ☐ `basedir/1100`
- ☐ `basedir/1100/intro_01.pdf`
- ☐ `basedir/examples`
- ☐ `basedir/examples/hello.java`

Q10. Long Coding

Map

Note: each of the following three questions can be completed without relying on a correct answer for the others. Don't panic if you get stuck on one of these questions!

Throughout this course, we have been storing data in arrays and ArrayLists. In both cases, we index into the data structure using an integer. In other words, we can access the elements in those containers by specifying an integer index that uniquely identifies a value in the array or ArrayList.

The goal of the following question will be for you to write a class that allows you to index data using a String as a key instead of an integer. Computer scientists refer to this data structure as a *Map*, so that's what we'll call the class.

The `Map` is implemented using an `ArrayList<Match>`; the `Match` interface, along with an implementing class `SimpleMatch`, is provided for you. The `Match` interface has three methods: `getKey()`, `getValue()`, and `setValue()`. **To allow the indexing to be well-defined, we will enforce the constraint that no two `Match` objects contained in the `ArrayList<Match>` can have the same key.**

Question 10.1:

Implement the `Map` constructor. This function takes a filename as input and reads in a list of key to value matches from the file using the `In` object for reading files. Note that the key is of type String, and the values are of type double. The file format is as follows, where `N` refers to the number of `Match`es represented in the file:

```
N
key1 value1
key2 value2
...
keyN valueN
```

You can assume that the file will be formatted correctly and that it will contain no duplicate keys. The constructor should read in the file and store the key to value matches in the `ArrayList<Match>` instance variable called `matches`.

```
public Map(String inputFile) {
     this.matches = new ArrayList<Match>();
     In inStream = new In(inputFile);
     // your code here








}
```

Question 10.2:

Next, write two tests for the provided `get()` method. (See appendix for implementation if you want)

The first test should create a `Map` object from the file `test1.txt` and then call the `get()` method on the `Map` object with the key `"Aditya"`. The test should then check that the value returned by the `get()` method is equal to `63.4` (representing the number of hours he spent in Office Hours this semester!). You can assume `test1.txt` already exists, and has the contents that are displayed on the right.

test1.txt:
3 Aditya 63.4 Bhrajit 94.5 Sukya 192.3

```
@Test
public void testGetElementPresent() {
    double DELTA = 0.001;
    // your code here








}
```

The second test should create a `Map` object from the file `test1.txt` and then call the `get()` method on the `Map` object with a key that is not present in the file. The test should then check that the value returned by the `get()` method is `Double.NEGATIVE_INFINITY`.

```
@Test
public void testGetElementNotPresent() {
    double DELTA = 0.001;
    // your code here








}
```

Question 10.3:

Next, implement the `insert()` method. This method takes a key and a value as input and updates our `Map` to match the key with that value.

If a `Match` object with the given key already exists in the `ArrayList<Match>`, then the method should **update the existing Match object with that key** and return `false` to indicate that the size of the `Map` has not changed.

Otherwise, the method should create a new `Match` object, insert it into `matches`, and return `true`.

We have provided two test cases that further demonstrate the intended behavior.

As a reminder: you can look at parts of the implementation for Map, Match and SimpleMatch in the appendix at the end of the exam.

```
@Test
public void insertKeyNotPresent() {
    Map m = new Map("test1.txt");
    boolean result = m.insert("key4", 3423);
    assertTrue(result);
    assertEquals(3423, m.get("key4"), 0.01);
    assertEquals(m.size(), 4);
}


@Test
public void insertKeyAlreadyPresent() {
    Map m = new Map("test1.txt"); // has a match for "Aditya"
    boolean result = m.insert("Aditya", 100.3);
    assertFalse(result);
    assertEquals(100.3, m.get("key1"), 0.01);
    assertEquals(m.size(), 3);
}
```

Please put your answer in the box on the next page

```
public boolean insert(String key, double value) {
        // your code here

}
```

Q11. (Bonus)

What is one piece of advice you'd give to a new student just starting out CIS 1100?

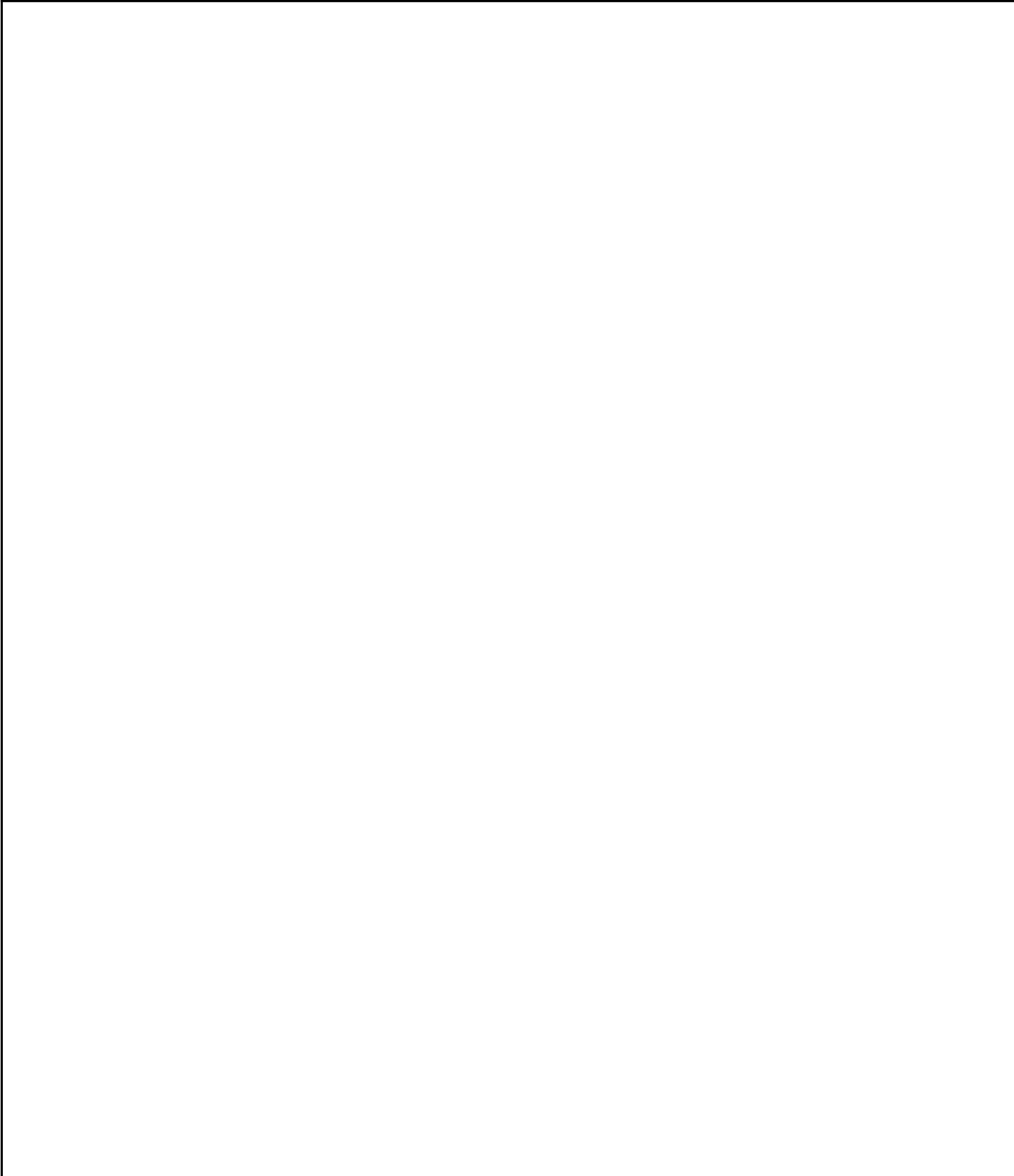
All exams will get full credit for this question, answer as seriously as you would like!

Extra Answers Page (This page is intentionally blank)

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.

Extra Answers Page (This page is intentionally blank)

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.



Appendix

Lists

method signature	purpose
`get(int i)`	return the value at position `i` in the List.
`set(int i, E e)`	set the value at position `i` in the List to be `e`.
`size()`	return the number of values stored in the List.
`remove(int i)`	remove and return the value stored at position `i` in the List.
`add(E e)`	insert the value `e` at the end of the List.
`add(int i, E e)`	insert the value `e` at position `i` in the List. `i` must be between `0` and `size()`.

Directories

method signature	purpose
`File(String path)`	Constructor, takes in a `path` to a file to the file to represent.
`isFile()`	Method that tests if the `File` called on is a regular file, `true` if it is, `false` otherwise
`isDirectory()`	Method that tests if the `File` called on is a directory, `true` if it is, `false` otherwise
`listFiles()`	Method that returns a `File[]`. If the `File` the method is called on is a directory, the array contains all files in the directory. If the `File` the method is called on is not a directory, it returns `null`.

In.java

method signature	purpose
`readInt()`	read the next characters from the file that represent an `int`
`readDouble()`	read the next characters from the file that represent a `double`
`readString()`	read the next characters from the file as a `String`, stopping at the first space character.
`readChar()`	read the next character
`isEmpty()`	returns true if the file is empty; false otherwise.

Map.java

```
public class Map() {
    private ArrayList<Match> matches;
    public Map(String inputFile) {...}

    public double get(String key) {
        for (int i = 0; i < matches.size(); i++) {
            Match m = matches.get(i);
            if (m.getKey().equals(key)) {
                return m.getValue();
            }
        }
        return Double.NEGATIVE_INFINITY;
    }

    public boolean insert(String key, double value) {...}

    public int size() {
        return matches.size();
    }
}
```

Match.java

```
public interface Match {
    public String getKey();
    public double getValue();
    public void setValue(double value);
}
```

SimpleMatch.java

```
public class SimpleMatch implements Match {
    private String key;
    private double value;

    public SimpleMatch(String key, double value) {
        this.key = key;
        this.value = value;
    }
    /* Other methods omitted for space; they behave as the
       method names suggest for getters and setters.    */
}
```