1. Rick owns a positively popular pizza place conveniently located just off of campus. Originally, he made all the pizzas himself, but rising campus food prices are making demand skyrocket. Luckily, the college students are as desperate for money as they are for food, a situation from which Rick, being a pragmatic individual, finds he can benefit. Drawing from the exploitable labor pool, Rick turns his already-hot kitchen into a sweat shop, ordering his workers like so:

```
for ( PizzaSlave student : laborPool ) {
    new Thread(student) . run();
}
```

   Sensing an early retirement, Rick promotes his first hire from slave to manager, rewarding him with slightly higher—but still illegal—pay. (Despite these perks and the envy of his peers, the manager is just like everyone else.) Alas, when Rick returns a few days later, he is so displeased with what he sees that he fires the manager on the spot. What made Rick so angry, and what should he have done differently to prevent its happening?

2. **The Life of Nick:** Nick, an aspiring entrepreneur, trained for 7 grueling years in the jungles of Zimbabwe. Nick's preparation was overseen by a group of trainers. His stages of learning were fueled by the following process:

```
public class Nick {
    private int experience = 0 ;
    public void train ( ) {
        this.experience += 1 ;
    }
}
```

   Imagine instructors are threads in Java. What are some problems we may encounter if Nick is having multiple people train him at the same time? How might we remedy these issues?

3. **Nick's Heavy Threads:** Nick now operates a store in Marketview Mall which has poor lighting, blasts black metal and sells jeans. Only one pair of jeans is available to purchase at a time, though there are more stored in the back. If a size is out that you don't want, you must wait for someone else to purchase the jeans. Nick's only employee, Hank, sits in a chair and stares at people angrily until someone makes a purchase, at which point he replaces the jeans with the same model of a random size. In order to prevent customers' waiting infinitely for an unavailable size, Hank will switch the jeans for a different size pair if no one has bought them after a period of three seconds.

```
1  public class NicksHeavyThreads
2  {
3      // jeans' size [1-5], or 0 when none on display
4      private static int awesomeJeans = 0;
5
6      // keep this updated as customers arrive and leave
7      private static int customers = 0;
8
9      private static MeanWorker hank = new MeanWorker();
10
11     public static void main( String[] args )
12     {
13         for( int i = 0; i < 10; ++i )
14         {
15             ( new LameCustomer() ).start();
16         }
17
18         // wait one second before introducing Hank
19         try { Thread.sleep( 1000 ); }
20         catch( InterruptedException pleaseDont ) {}
21         hank.start();
22     }
23
24     private static class LameCustomer extends Thread
25     {
26         // ( implementation omitted )
27     }
28
29     private static class MeanWorker extends Thread
30     {
31         // ( implementation omitted )
32     }
33 }
```

(Questions may be found on the next page. You may answer them in the space allotted here, or on the following page.)

(a) Complete the implementation of the `LameCustomer` class: Each instance must choose a jeans size and wait for it to be available, update the jeans to indicate that they have been taken, print the message "Customer: I got my size *size* jeans!" and inform all threads that the jeans selection has changed.
(Hint: Remember to keep an accurate count of how many customers are in the shop.)

(b) Now implement the `MeanWorker` class, which should choose a size and stock a pair of jeans of that size, print the message "Hank: I grumpily restocked with size *size*," and inform all threads that the selection has changed. It should then wait until someone has taken the jeans or until three seconds have elapsed, whichever comes first. These steps should be repeated until all customers have left the store.

4. Explain the differences between:
   (a) `class` vs `object`

   (b) constant vs non-constant field (variable). Declare a constant.

   (c) final vs non-final method.

   (d) class vs instance variable. Declare variables of both types.

5. Briefly describe the difference (for objects) between `a.equals(b)`, `a==b`, `a.compareTo(b)`, and `Comparator.compare(a,b)`.

6. Assume the following line of code is given:

```
Collection<Integer> t = new ArrayList<>();
```

What, then, is wrong with the following? Correct any errors:
(*Hint: there are no syntax errors.*)

```
for( int i = 0; i < 20; ++i )
    t.add(i);
for( int i=0; i < t.size(); ++i )
    System.out.println(t.get(i));
```

7. Briefly explain the differences between the three kinds of exceptions: checked exceptions, runtime exceptions, and errors.

8. What is the output when `LookAtDatMagic`'s main is executed?

```
1 public class HeySteve{
2     public int bananza(int in) throws NewException{
3         if ( in == 7 ){
4             throw new NewException("HeySteve, cut that out!");
5         }
6         return in;
7     }
8 }
9
10 public class NewException extends Exception{
11     public NewException(String message){
12         super(message);
13     }
14
15 }
16
17 public class WakaWaka{
18     public String BeachBash(Object a, Object b) throws NewException{
19         if ( a.equals(b) ){
20             throw new NewException("It's a Beach-bash! WakaWaka!");
21         }
22         return "Da-nananananan";
23     }
24 }
25
26 public class LookAtDatMagic{
27     public void magic() throws NewException{
28         int maraca = 5;
29         try{
30             HeySteve steve = new HeySteve();
31             maraca = steve.bananza(7);
32         }catch(NewException e){
33             System.out.println(e.getMessage());
34         }finally{
35             WakaWaka waka = new WakaWaka();
36             System.out.println(waka.BeachBash(maraca, 5));
37         }
38     }
39
40     public static void main(String[] args){
41         try{
42             LookAtDatMagic ladm = new LookAtDatMagic();
43             ladm.magic();
44         }catch(NewException e){
45             System.out.println(e.getMessage());
46         }
47     }
48 }
```

9. Use the following code to answer the questions listed on the next page.

```
1 import java.util.ArrayList;
2 import java.util.Collection;
3
4 public class UFO{
5     private Collection<Probable> toProbe;
6     private int aggression;
7
8     public UFO(int agg){
9         toProbe = new ArrayList<Probable>();
10        aggression = agg;
11    }
12
13    public void probeEverything(){
14        for( Probable p : toProbe ){
15            p.probe();
16        }
17    }
18
19    public void sendHome(){
20        for( Probable p: toProbe ){
21            p.returnHome();
22        }
23    }
24
25    public void abduct(Collection<Probable> potentialAbductees){
26        for( Probable p : potentialAbductees ){
27            if( p.getMentalResolve() < aggression){
28                toProbe.add(p);
29            }
30        }
31    }
32
33    public static void main(String[] args){
34        UFO ufo = new UFO(1199999999);
35        ArrayList<Probable> field = new ArrayList<Probable>();
36
37        field.add(new Cow("Bessie"));
38        //10 = mentalResolve
39        Human cleatus = new RedNeck("Cleatus", 10);
40
41        // 50 == mentalResolve and 100 == academicRespect
42        Human brown = new Professor("Brown", 50, 100);
43
44        field.add(cleatus);
45        field.add(brown);
46
47        ufo.abduct(field);
48        ufo.probeEverything();
49        ufo.sendHome();
50    }
51 }
```

(a) Why should `Probable` be an interface, rather than a class or an abstract class?

(b) Write the `Probable` interface.

(c) Should the `Redneck` and `Professor` classes implement `Probable` directly?

10. **Networking**

(a) What does TCP stand for? Where and why do we use TCP?

(b) What does UDP stand for? When and where do we use UDP?

(c) Which one does a stream socket use for data transmission? TCP (or) UDP?

(d) Which one does a datagram socket use for data transmission? TCP (or) UDP?

(e) What is a datagram?

(f) What is a socket?

11. Find at least 3 (total) errors in the following code:

Server: echoes one line of data sent to it

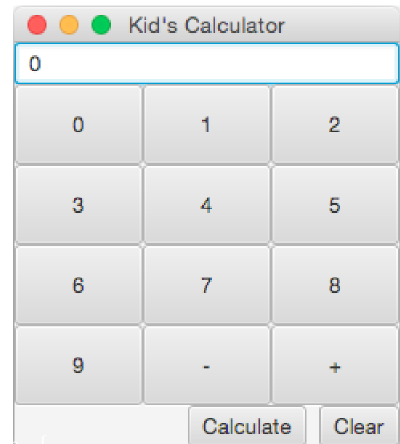
```
1     ServerSocket pubServer = new ServerSocket(0);
2     System.out.println(pubServer.getLocalPort());
3     Socket client;
4     BufferedReader reader = null;
5     try {
6         reader = new BufferedReader(
7             new InputStreamReader(client.getInputStream()));
8     } catch (IOException e) {
9         System.out.println("IOException: " + e.getMessage());
10    }
11    String response = null;
12    try {
13        response = reader.readLine();
14    } catch (IOException e) {
15        System.out.println("IOException: " + e.getMessage());
16    }
17    System.out.println(response);
18    pubServer.close();
```

Client: sends a line of text to a server: server address, port, text

```
1     InetAddress server = null;
2     try {
3         server = InetAddress.getByName(args[0]);
4     } catch (UnknownHostException e) {
5         System.out.println("Unknown host");
6     }
7     int port = Integer.parseInt(args[1]);
8     Socket conn = null;
9     try {
10        conn = new Socket(server, port);
11    } catch (IOException e) {
12        System.out.println("IOException: " + e.getMessage());
13    }
14    try {
15        System.out.println(args[2]);
16    } catch (IOException e) {
17        System.out.println("IOException: " + e.getMessage());
18    }
19    conn.close();
```

12. Create a class that constructs and displays the following GUI. If you can't remember exactly how to implement a certain part of the GUI in code, explain what the component is and how it would fit in with the rest of the calculator. (*Hint: draw out the GUI's* ***component hierarchy***.)

The buttons within the GUI do not need to be functional. You may or may not need the following: `Scene`, `BorderPane`, `FlowPane`, `HBox`, `TextField`, `Button`. The window should fit to all of the components and have a title.



13. Briely name and describe how the following Java features implement their respective design patterns. Iterator, Observer/Observable, and Streams.

14. **NullPointerExceptions**

    (a) Briefly describe what a `NullPointerException` is.

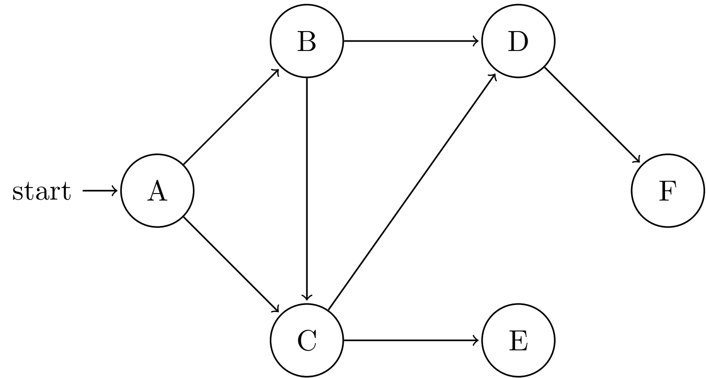
    (b) Provide a short code example that will throw a `NullPointerException` *without explicity writing "`null`" in your snippet*, explain why the exception will occur, and finally, explain how you would fix the problem.

    (c) What is the most common mistake programmers make that lead to `NullPointerException`s?

15. Suppose we are talking about the depth-first search (DFS) algorithm. Nodes are added to the data structures in alphabetical order.

    (a) What underlying data structure does this algorithm use?

    (b) Given the following graph, state the DFS traversal order and show the data structure at each step. Node `A` is the start node, and `F` is the destination node.



    (c) What path from `A` to `F` does the DFS algorithm return?

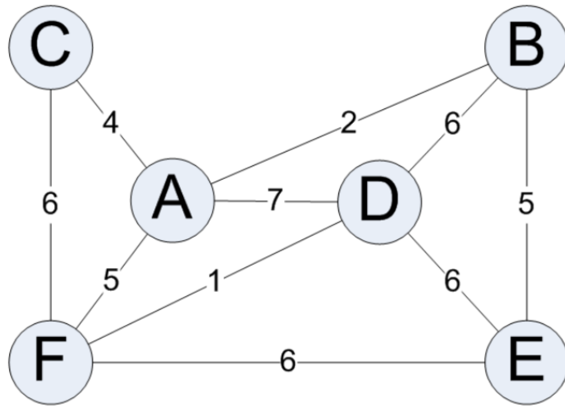
16. Now consider a BFS algorithm, again populating data structures in alphabetical order.

    (a) What changes would need to be made to a DFS implementation to turn it into a breadth-first search (BFS)?

    (b) Using the graph as described in Question 1, what is the BFS traversal order? Show the data structure at each step.

    (c) What path from `A` to `F` results from the BFS algorithm?

17. Consider the following graph.



(a) Perform Dijkstra's algorithm to find the shortest path between C and E.

Finalized	A	B	C	D	E	F
–	(∞, None)	(∞, None)	**(0, None)**	(∞, None)	(∞, None)	(∞, None)

(b) In general, when using Dijkstra's algorithm to find the shortest path between nodes, do you need to use every row of the table? Why or why not?

18. Define polymorphism and explain a situation in which you would use it.

19. If an instance variable is declared with `protected` access, who can access it?

20. What is the difference between an abstract class and an interface? Why might you want to use an interface over an abstract class?

21. What gets printed by the following code?

```
1 public class Class1 {
2     public Class1() {
3         System.out.println( "Class1()" );
4     }
5     public void print1() {
6         System.out.println( "Class1.print1()" );
7     }
8     public void print2() {
9         System.out.println("Class1.print2()" );
10     }
11 }
12
13 public class Class2 extends Class1 {
14     public Class2() {
15         System.out.println( "Class2()" );
16     }
17     public void print1() {
18         System.out.println("Class2.print1()");
19     }
20 }
21
22 public class Class3 extends Class1{
23     private Class2 class2;
24     public Class3() {
25         System.out.println( "Class3()" );
26         class2 = new Class2();
27     }
28     public void print1() {
29         class2.print1();
30     }
31     public void print2(){
32         System.out.println("Class3.print2()");
33         super.print2();
34     }
35 }
36
37 public class TestClass {
38     public static void main( String[] args ) {
39         Class1 c1 = new Class2();
40         c1.print1();
41         c1.print2();
42         System.out.println();
43         Class1 c2 = new Class3();
44         c2.print1();
45         c2.print2();
46     }
47 }
```