1. 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.

2. Briefly describe the difference (for objects) between `a.equals(b)`, `a==b`, `a.compareTo(b)`, and `Comparator.compare(a,b)`.
3. Convert the following code to use generics.

```java
interface StringCondition {
    boolean checkString(String s);
}

interface IntegerCondition {
    boolean checkInteger(Integer i);
}

class StringContainer {
    ArrayList<String> values;
    // Add and other methods are defined correctly here...

    String getFirstWhereHolds(StringCondition condition) {
        for (String s : values) {
            if (condition.checkString(s))
                return s;
        }
        return null;
    }
}

class IntegerContainer {
    ArrayList<Integer> values;
    // Add and other methods are defined correctly here...

    Integer getFirstWhereHolds(IntegerCondition condition) {
        for (Integer i : values) {
            if (condition.checkInteger(i))
                return i;
        }
        return null;
    }
}
```
4. Reimplement the following function using an **Iterator** instead of a **for-each** loop.

```java
public static Integer sum ( ArrayList<Integer> lst ) {
    Integer total = 0 ;
    for ( Integer elem : lst ) {
        total = total + elem ;
    }
    return total ;
}
```

5. Assume the following line of code is given:

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

What, then, is wrong with the following? Correct any errors:

(Hint: there are no syntax errors.)

```java
for( int i = 0; i < 20; ++i )
    t.add(i);
for( int i=0; i < t.size(); ++i )
    System.out.println(t.get(i));
```
6. Briefly explain the differences between the three kinds of exceptions: checked exceptions, runtime exceptions, and errors.

- **Checked Exceptions**: Exceptions that a method signature must specify it throws. If a method may throw a checked exception, all calls to that method must be within a `try`-`catch` block. Checked exceptions should be used exclusively for foreseeable run-time mistakes, and any reasonably robust system should be able to recover from one. A classic example is `IOException`.

- **Runtime Exceptions**: Not declared in a method signature and not anticipated to be thrown. Usually arise due to software bugs and often cause the program to crash. Classic examples are `NullPointerException` and `ArrayIndexOutOfBoundsException`.

- **Errors**: Represent a serious issue outside of the control of the programmer (hard drive failure, not enough memory, device issue). Examples are `IOException`, `VirtualMachineError`, and `ThreadDeath` (see Java's `Error` class).
7. Use the following code to answer the questions listed on the next page.

```java
import java.util.ArrayList;
import java.util.Collection;

public class UFO{
    private Collection<Probable> toProbe;
    private int aggression;

    public UFO(int agg){
        toProbe = new ArrayList<Probable>();
        aggression = agg;
    }

    public void probeEverything(){
        for( Probable p : toProbe ){
            p.probe();
        }
    }

    public void sendHome(){
        for( Probable p: toProbe ){
            p.returnHome();
        }
    }

    public void abduct(Collection<Probable> potentialAbductees){
        for( Probable p: potentialAbductees ){
            if( p.getMentalResolve() < aggression){
                toProbe.add(p);
            }
        }
    }
}

public static void main(String[] args){
    UFO ufo = new UFO(1199999999);
    ArrayList<Probable> field = new ArrayList<Probable>();

    field.add(new Cow("Bessie");
    // 10 = mentalResolve
    Human cleatus = new RedNeck("Cleatus", 10);
    // 50 == mentalResolve and 100 == academicRespect
    Human brown = new Professor("Brown", 50, 100);

    field.add(cleatus);
    field.add(brown);

    ufo.abduct(field);
    ufo.probeEverything();
    ufo.sendHome();
}
```
(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?

8. Name the design pattern used in the following snippet of code.

```java
class Car {
    private String make;
    private String model;
    private int mileage;
    private Car(String make, String model, int mileage) {
        this.make = make;
        this.model = model;
        this.mileage = mileage;
    }
    public static Car makeCar(String str) {
        String arr[] = str.split(" ");
        return new Car(arr[0], arr[1], Integer.parseInt(arr[2]));
    }
    public static void main(String[] args) {
        Car myCar = Car.makeCar("Toyota Camry 200000");
    }
}
```

The Factory design pattern is used.
9. **NullPointerExceptions**

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

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

   ```java
   import java.util.ArrayList;
   public class Example {
       ArrayList<Integer> myList;
       public Example() {
           for (int i = 0; i < 20; i++) {
               myList.add(i);
           }
       }
       public static void main(String[] args) {
           Example foo = new Example();
       }
   }
   
   Above, we declare `myList`, but we don’t actually give it a value. This is not a compile-time error; Java will recognize 'myList' as a valid variable for use in the rest of your code, but it will set its value to `null`, since we didn’t initialize the variable. Since `myList` is still `null` by the time we reach the inside of the `for` loop, the line that is executed would effectively read "null.add(0)", which is clearly invalid. Note that not initializing local variables declared inside methods will produce an error at compile time. However, instance variables and class (‘static’) variables are initialized to a default value, which is ‘null’ for object types. We may remedy this situation by initializing our list, by adding the following line at the beginning of our constructor:

   ```java
   myList = new ArrayList<>();
   ```

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

   The most common mistake that programmers make which causes `NullPointerException`s is the one we have exemplified above - forgetting to initialize your variables before you use them.

   ```java
   ```
10. Suppose you have the classes DecodeStream and EncodeStream which decodes and encodes text in a particular format. Assume that both classes buffer I/O and have readLine() which returns a string and writeLine( String s ) respectively to read and write a line at a time.

Write code to read the file "secret.txt", decode it using the DecodeStream, and print the contents.

```java
String line ;
DecodeStream in = new DecodeStream ( new FileReader ( "secret.txt" ) );
while ( ( line = in.readLine() ) != null ) {
    System.out.println ( line );
}
```

Write code to encode and write a string called "message" to a file.

11. 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.

```
A -- B -- D
|   |   |
| v v v
C   E
```

(c) What path from A to F does the DFS algorithm return?
12. 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?

13. When is a vertex’s sum weight finalized in Dijkstra’s algorithm?

14. In Dijkstra’s algorithm, what role does the priority queue play in finding the shortest path? When do we use it?

15. Why does Dijkstra’s algorithm not work correctly on graphs with negative edge weights?
16. Consider the following graph.

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

<table>
<thead>
<tr>
<th>Finalized</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
<td>(0, None)</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
</tr>
</tbody>
</table>

(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?

17. Given a node in an array-based binary heap at index $i$, where are the indices of both its children? What is the index of its parent?

18. For a binary heap containing $n$ elements, what is the maximum number of swaps occurring after an insert operation?
19. Chris made a mistake in his hash table implementation!

```java
import java.util.ArrayList;

public class WonkyHashTable {
    private int size;
    private ArrayList<String> table;

    public WonkyHashTable(int s) {
        size = s;
        table = new ArrayList<String>(size);
        for(int i = 0; i < s; i++) {
            table.add(""svs);
        }
    }

    public void add(String element) {
        int hash = bad_hash(element);
        table.set(hash, element);
        System.out.println(table.toString());
    }

    private boolean contains(String element) {
        for(String t : table) {
            if(element.equals(t)) {
                return true;
            }
        }
        return false;
    }

    private int bad_hash(String element) {
        return element.length() % table.size();
    }

    public static void main(String args[]) {
        WonkyHashTable htable = new WonkyHashTable(4);
        for(String s : "I wrestled a bear once".split(" ")) {
            htable.add(s);
        }
    }
}
```

(a) Show what the hash table looks like after the for loop in main completes.

(b) What is wrong with the code? What can we do to make the function behave as Chris expects it to behave?
(c) Draw the table of the properly behaving hash function.

(d) Assuming that this hash table will only be used on strings, is the hashing function being used a good one? Why or why not?

20. 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:

```java
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?

21. **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:

```java
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?
22. **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.

```java
public class NicksHeavyThreads {
    // jeans' size [1-5], or 0 when none on display
    private static int awesomeJeans = 0;

    // keep this updated as customers arrive and leave
    private static int customers = 0;

    private static MeanWorker hank = new MeanWorker();

    public static void main( String[] args )
    {
        for( int i = 0; i < 10; ++i )
        {
            (new LameCustomer()).start();
        }

        // wait one second before introducing Hank
        try { Thread.sleep( 1000 ); } catch( InterruptedException pleaseDont ) {}
        hank.start();
    }

    private static class LameCustomer extends Thread
    {
        // (implementation omitted)
    }

    private static class MeanWorker extends Thread
    {
        // (implementation omitted)
    }
}
```

(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.
23. 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.
24. 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?

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

Server: echoes one line of data sent to it

```java
ServerSocket pubServer = new ServerSocket(0);
System.out.println(pubServer.getLocalPort());
Socket client;
BufferedReader reader = null;
try {
    reader = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
} catch (IOException e) {
    System.out.println("IOException: " + e.getMessage());
}
String response = null;
try {
    response = reader.readLine();
} catch (IOException e) {
    System.out.println("IOException: " + e.getMessage());
}
System.out.println(response);
pubServer.close();
```

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

```java
InetAddress server = null;
try {
    server = InetAddress.getByName(args[0]);
} catch (UnknownHostException e) {
    System.out.println("Unknown host");
}
int port = Integer.parseInt(args[1]);
Socket conn = null;
try {
    conn = new Socket(server, port);
} catch (IOException e) {
    System.out.println("IOException: " + e.getMessage());
}
try {
    System.out.println(args[2]);
} catch (IOException e) {
    System.out.println("IOException: " + e.getMessage());
}
conn.close();
```

(a) What are the four core components of any backtracking `solve` function?

(b) Write a generic `solve()` function for a given configuration, which returns either the solution configuration or `null` (if no solution exists):

(Hint: make up a function name for each of the parts above, if necessary)

```
public Configuration solve(Configuration config) {
    if (config.isGoal()) {
        return config;
    } else {
        for (Configuration child : config.getSuccessors()) {
            if (child.isValid()) {
                Configuration ans = solve(child);
                if (ans != null) {
                    return ans;
                }
            }
        }
        // implicit backtracking happens here
    }
    return null;
}
```