1. The following code snippets are legal.

   (a) `ArrayList<String> a = new LinkedList<String>();`
   False - There is no superclass-subclass relationship between ArrayList and LinkedList.

   (b) `List<Integer> l = new List<Integer>();`
   False - List is an interface

   (c) `Set<Double> s = new TreeSet<Double>();`
   True

2. What is the difference between a class and an object?
   A class is a definition for a type (members and methods). An object is an instance of a class.

3. What does the `super` keyword do? When must you use it?
   The `super()` keyword is used to call the constructor of a super class. Use it at the beginning of the subclass constructor. It can also be used to call super class methods in cases where you have overridden a method and need the super class version. (`super.someMethod();`)

4. When overriding equals() for a custom object, what other method should you also override and why? Do you need to use accessors and mutators in either of these methods to access the class' private variables and methods?
   hashcode() because we need to ensure we can rehash the same code for identical objects. And no, the private variables and methods are private to this class and are inaccessible from other classes.

5. Identify the errors in the following code:

```
1 public interface TestInterface {
2     public TestInterface() {}
3     public void splat( String message ) {}
4 }
5 public class AbstractClass {
6     abstract protected void myMethod();
7     public void callMyMethod() {
8         myMethod();
9     }
10 }
11 public class ConcreteClass extend AbstractClass implement TestInterface {
12     protected int myMethod() {
13         System.out.println( "I got called!" );
14         return 7;
15     }
16     public void splat( String message ) {
17         System.out.println( "SPLAT!" );
18         System.out.println( message );
19     }
20 }
21 public class SimpleProgram {
22     public static void main(String[] args) {
23         AbstractClass myclass = new AbstractClass();
24         myclass.callMyMethod();
25         //myclass.splat( "woohoo" );// Not defined in AbstractClass
26     }
27 }
```

6. What are the two type of exceptions? What is an Error?

   (a) Checked - Code calling a method that may throw a checked exception must either provided a try-catch-(finally) block around the call OR declare that it throws the same exception in its method signature.

   (b) Runtime - Unanticipated software error that occurs at runtime and from which a program cannot usually recover.

   (c) Error - Serious issue outside of the program. Possibly a memory or hardware issue.

7. What will be printed by the following code?

```
1  public class ExceptionTest {
2      public static void main( String args[] ) {
3          try {
4              for ( int i = 0; i < 5; i++ ) {
5                  process( i );
6              }
7          } catch ( Exception e ) {
8              System.err.println( e.getMessage() );
9          }finally{
10             System.out.println( "Done processing" );
11         }
12     }
13     public static void process( int num ) throws Exception {
14         if ( num != 2 ) {
15             System.out.println( "Good number: " + num );
16         } else {
17             throw new Exception( "Bad number: " + num );
18         }
19     }
20 }
```

Good number: 0
Good number: 1
Bad number: 2
Done processing

8. Instantiate a new `BufferedReader` which decorates a `ForumReader`. ForumReaders only use the default constructor but can throw a `TooLongDidntReadException` if the forum post is too long to be worth reading. Handle any exceptions thrown and gracefully close the file.

```
1      try {
2          BufferedReader troll = new BufferedReader(new ForumReader());
3          // Do some things here.
4      }catch(TooLongDidntReadException tex) {
5          System.err.println(tex.getMessage());
6          // Cool story, bro!
7      }catch(Exception ex) {
8          System.err.println(ex.getMessage());
9          // Handle other exceptions.
10     }finally{
11         // Close the reader
12         if (troll != null) {
13             try {
14                 troll.close();
15             }catch(IOException ex){
16                 ex.printStackTrace();
17             }
18         }
19     }
```

9. Why is buffering a good idea when doing file I/O?
   Buffering uses an internal data structure to store data before reading/writing. Without buffering, every read/write call would translate to a call at the OS level IO mechanism. Reading/writing a byte or so at a time is incredibly inefficient, so buffering stores data and then reads/writes the entire buffer. Buffering is drastically more efficient than not buffering.

10. Name and describe 3 JavaFX layouts.

    (a) GridPane - matrix representation (rows x columns)

    (b) FlowPane - Default. Uses one row until it runs out of space.

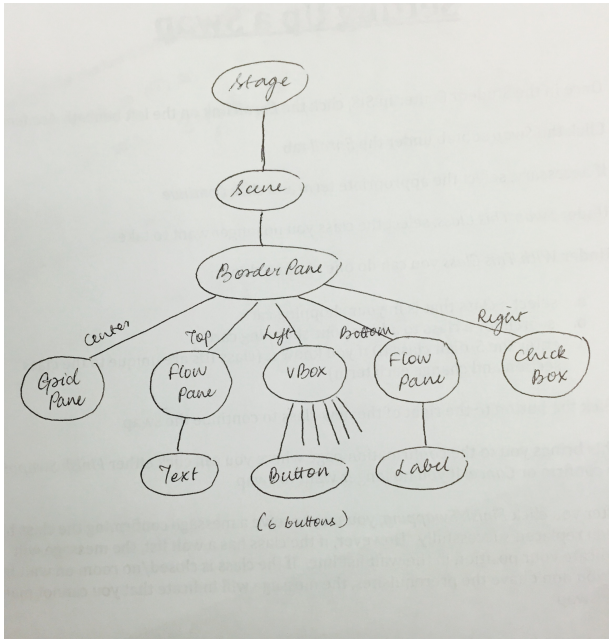
    (c) BorderPane - 5 sections: Top, Bottom, Left, Right and Center.

11. What is the default behavior of a button in JavaFX? What other events can be triggered?
    ActionEvent is generated and the event listener should handle it. MouseEvent, KeyEvent, ActionEvent, WindowEvent..

12. Write an `EventHandler` implementation called `SimpleAction` that changes a label('messageLabel') to say "Cheese!" when an `MouseEvent` is received.

```
1 public class SimpleAction implements ActionListener {
2     public void actionPerformed(MouseEvent e) {
3         messageLabel.setText("Cheese");
4     }
5 }
```

13. Draw the tree of the hierarchy of the layouts used in the following code.



14. Describe `List, Set` and `Map`. What interface(s) do `List` and `Set` extend?
`List` and `Set` implement `Collection`. `Map` does not implement any interface.

    (a) `List` - interface for an ordered collection that provides random access to elements

    (b) `Set` - interface for a collection containing no duplicate elements

    (c) `Map` - interface for a collection of key/value pairs.

15. Explain the difference between `Comparator` and `Comparable`.

    (a) `Comparable` - an interface implemented by a class to make it comparable to other objects, i.e. it specifies the natural ordering of instances of a particular class.

    (b) `Comparator` - an interface implemented by a class which can compare two objects of a particular type. This is useful for sorting algorithms in which the desired ordering is different than the natural ordering of the elements in the collection.

16. Use an iterator to count the number of elements in the Collection

```
1  public class VictimsIterator {
2      public static void main(String args[]) {
3          VictimsIterator vi = new VictimsIterator();
4          List<String> victims = new ArrayList<String>();
5          for(int i = 0; i < 20; i++)
6              victims.add("Java");
7          System.out.println("Count = " + vi.numVictims(victims));
8      }
9
10     public int numVictims(Collection<String> victims) {
11         Iterator<String> iter = victims.iterator();
12         int count = 0;
13         while(iter.hasNext()) {
14             count++;
15             iter.next();  //returns a string
16         }
17         return count;
18     }
19 } // VictimsInterator
```

17. Write a `Comparator` that compares strings by their size modulo (%) 13 and use this comparator for sorting a TreeSet.

```
1  public class ModuloComparator<T> implements Comparator<T> {
2
3      public static void main(String args[]) {
4          ModuloComparator<String> mc = new ModuloComparator<String>();
5          Set<String> mySet = new TreeSet<String>(mc);
6          mySet.add("Zack");
7          mySet.add("Sean");
8          mySet.add("Matthew");
9          mySet.add("Trudy");
10
11         Iterator<String> iter = mySet.iterator();
12         while(iter.hasNext()) {
13             System.out.println(iter.next());
14         }
15     }
16     public int compare(T o1, T o2) {
17         String str1 = (String) o1;
18         String str2 = (String) o2;
19         return ((str1.length() % 13) - (str2.length() % 13));
20     }
21 } // ModuloComparator
```

18. Write a lambda using stream operations to print the names of the dogs whose age is at most 2 and is of the breed Corgi.

```
1 List<Dog> dogs = Arrays.asList(
2     new Dog("Brady", "Corgi", 1),
3     new Dog("Bruno", "Pug", 2),
4     new Dog("Porter", "Corgi", 2),
5     new Dog("Darcy", "Dalmation", 3),
6     new Dog("Rodney", "Bulldog", 3)
7 );
```

```
1         dogs.stream().filter(d -> d.age <= 2 && d.breed.equals("Corgi"))
2             .forEach(n -> System.out.println(n));
```

19. Describe the following thread functions/terms

    (a) start
        i. causes a thread to begin execution of its run method.

    (b) sleep
        i. Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not lose ownership of any monitors.

    (c) join
        i. Causes the current thread to wait for the thread that join was called on to finish executing before continuing.

    (d) synchronized
        i. forces a only one at a time behavior on objects and methods which threads may share.

    (e) wait
        i. causes calling thread to idle until notified

    (f) notify
        i. Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

    (g) notifyAll
        i. Wakes up all threads that are waiting on this object's monitor.

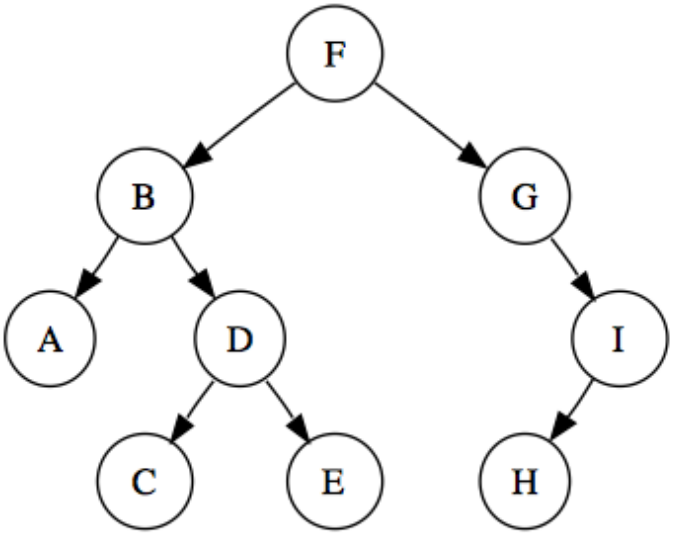
20. Write a program which creates 3 threads to print the numbers 0-999, 1000-1999, 2000-2999 respectively.

```
1
2 public class PrintThread extends Thread {
3
4     private int startNum;
5
6     public PrintThread( int startNum ) {
7         this.startNum = startNum;
8     }
9
10    public void run() {
11        for( int i = 0; i < 1000; i++ ) {
12            System.out.println( "Thread #" + startNum + ": " + ( startNum * 1000 + i ) );
13        }
14    }
15
16    public static void main( String[] args ) {
17        for( int i = 0; i < 3; i++ ) {
18            PrintThread p = new PrintThread( i );
19            p.start();
20        }
21    }
22
23 }
```

21. For the following two problems, always visit neighbors in alphabetical order.

(a) Given the following *tree*, perform a DFS from node F to E and give the order of visited nodes.

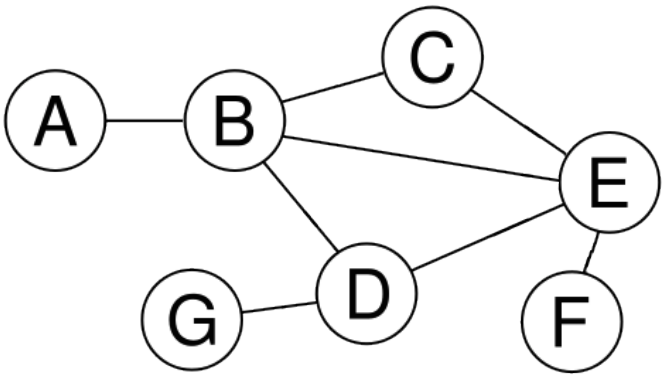
Repeat with a BFS.



DFS recursive: F, B, A, D, C, E
DFS iterative: F, G, I, H, B, D, E
BFS: F, B, G, A, D, I, C, E

(b) Given the following *graph*, perform a DFS from node A to F and give the order of visited nodes.

Repeat with a BFS.



DFS recursive: A, B, C, E, D, G, F
DFS iterative: A, B, D, G, E, F
BFS: A, B, C, D, E, G, F

22. Given the `searchBFS` function from lecture, trace the output when run between nodes A and F from the previous question (b).

(a) Show the resulting predecessor map.
'A' = None, 'B' = A, 'C' = B, 'D' = B, 'E' = B, 'G' = D, 'F' = E

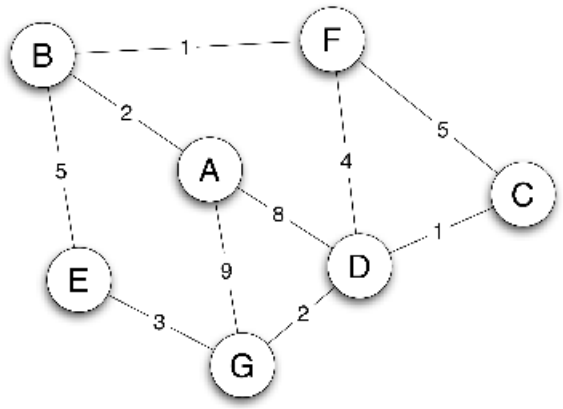
(b) Show how the queue and current change over time.
A; B; C,D,E; D,E; E,G; G,F; F; Empty

(c) What does constructPath() return?
A; B; C; D; E; E; G; F

```
1   public List<Node> searchBFS(String start, String finish) {
2           Node startNode, finishNode;
3           startNode = graph.get(start);
4           finishNode = graph.get(finish);
5
6           List<Node> dispenser = new LinkedList<Node>();
7           dispenser.add(startNode);
8
9           Map<Node, Node> predecessors = new HashMap<Node,Node>();
10          predecessors.put(startNode, startNode);
11
12          while (!dispenser.isEmpty()) {
13              Node current = dispenser.remove(0);
14              if (current == finishNode) {
15                  break;
16              }
17              for (Node nbr : current.getNeighbors()) {
18                  if(!predecessors.containsKey(nbr)) {
```

```
19                          predecessors.put(nbr, current);
20                          dispenser.add(nbr);
21                      }
22                  }
23              }
24
25             return constructPath(predecessors, startNode, finishNode);
26          }
27
28          private List<Node> constructPath(Map<Node,Node> predecessors,
29                  Node startNode, Node finishNode) {
30              List<Node> stack = new LinkedList<Node>();
31              List<Node> path = new ArrayList<Node>();
32
33              if(predecessors.containsKey(finishNode)) {
34                  Node currNode = finishNode;
35                  while (currNode != startNode) {
36                      stack.add(0, currNode);
37                      currNode = predecessors.get(currNode);
38                  }
39                  stack.add(0, startNode);
40              }
41
42              while (!stack.isEmpty()) {
43                  path.add(stack.remove(0));
44              }
45
46              return path;
47          }
```

23. Perform Dijkstra's shortest path algorithm on this graph to find the shortest distance from A to C. In the case of a tie, select a path based on the alphebetical ordering of the vertices.



Iteration	Finalized Vertex	A	B	C	D	E	F	G
		(0,None)	(∞,None)	(∞,None)	(∞,None)	(∞,None)	(∞,None)	(∞,None)
1	A	✓						
2								
3								
4								
5								
6								
7								

Perform Dijstra's
A : 0, None, Finalized iteration 1
B : 2, A, Finalized iteration 2 (updated from Infintiy, None when visited from A)
C : 8, F Finalized iteration 6 (updated from Infinitiy, None when visited from F)
D : 7, F Finalized iteration 4 (updated from Infinity, None when visited from A, and then updated from 8, A when visted from F)
E : 7, B Finalized interation 5 (updated from Infinity, None when visted from B)
F : 3, B, Finalized iteration 3 (updated from Infinity, None when visted from B)
G : 9, A Finalized iteration 7 (updated from Infinity, None when visited from A)

24. Give the least cost path from A to E and the total cost.
A, B, E 7

25. Give the least cost path from A to C, and the total cost.
A, B, F, C 8

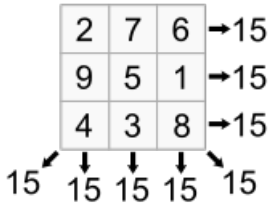
26. Give the definitions for the following:

- Serializable
an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
- Domain Name Server
Server which does translation from host name to IP addres
- TCP
Connection Oriented (stream socket), Reliable,Hand Shaking(telephone connection)
- UDP
Connection less (datagram), Unreliable , Postal system
- Datagram
is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

27. Complete the following code for the backtracking solve algorithm. Be sure to include pruning in your solution.

```
1   public static ArrayList< Configuration > solve(Configuration config){
2       //Backtracking algorithm
3       if (config.isGoal()){
4           return config;
5       } else {
6           ArrayList< Configuration > neighbors =  config.getSuccessors();
7       for (int i = 0; i < neighbors.size(); i++) {
8           if(neighbors[i].isValid()){
9           Configuration c = solve(neighbors[i]);
10          if(c != null){
11              return c;
12          }
13      }
14      return null;
15  }
```

28. `Magic Square:` An n x n square with numbers 1 to n^2 arranged, so that the sum of the rows, the columns, and the diagonals is the same. The figure shows the solution for a 3x3 magic square. What algorithm would you choose to solve the Magic Square Problem? Explain the details.



This is a Backtracking problem.
`Configuration:` Represented by a 2d integer array initialized with 0s.
int [][] config = new int[n][n];

`isGoal:` Check if the square is filled. No zeros.

`isValid:` Check if the the row, column and diagonal of the last inserted number does not contain zero's, check if the sum of the row, column and diagonal matches.
(Effective Pruning: Use the magic constant $M = n(n^2 + 1)/2$ which will be the sum of each row, column and diagonal)

`Child Config/Neighbors`: The next row/column with a zero. The next empty cell in the square.