CS-142 Final Exam Review

December 3, 2018

Presented by the RIT Computer Science Community

http://csc.cs.rit.edu

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?

   In his haste to make a profit, Rick mistakenly called the `Thread` class's `run()` method, which caused his manager to run synchronously and make pizzas while everyone else stood around waiting. Rick should instead have called `start()`, which would have whipped all of his workers into shape at roughly the same time.

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?

   The train function, and specifically its incrementation, is non-atomic, meaning the value of Nick's `experience` variable will be undefined after multiple threads attempt to call the function concurrently. One solution would be to synchronize on `Nick` to ensure only one thread at a time executes within the critical section.

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.)

```
static class LameCustomer extends Thread {
    public void run() {
        synchronized( hank ) {
            ++customers;
            // pick size
            int desiredSize = (int) ( Math.random()*(5) ) + 1;
            // wait for pair
            while( awesomeJeans != desiredSize ) {
                try { hank.wait(); }
                catch( InterruptedException pleaseDont ) {}
            }
            System.out.println( "Customer: I got my size "
                + awesomeJeans + " jeans!" );
            awesomeJeans = 0; // take the jeans
            hank.notifyAll(); // inform everyone they are gone
            --customers;
        }
    }
}
```

(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.

```
static class MeanWorker extends Thread {
    public synchronized void run() {
        do {
            // new size
            awesomeJeans = (int) ( Math.random()*(5) ) + 1;
            System.out.println( "Hank: I grumpily restocked "
                + "with jeans of size " + awesomeJeans );
            notifyAll(); // inform customers of the restocking
            try { wait( 3000 ); } // let people shop
            catch( InterruptedException pleaseDont ) {}
        }
        while( customers > 0 );
    }
}
```

4. Explain the differences between:

   (a) `class` vs `object`
   A class defines methods and fields—it can be viewed as a template. An object is an instance of a class. Think of classes as molds and objects as individual things created by those molds.

   (b) constant vs non-constant field (variable). Declare a constant.
   A constant field cannot be changed during run time (it may be set *once*).

   ```
   public final int NUM_PEOPLE_WHO_LIKE_JAVA = 1;
   ```

   Note that marking mutable types `final` only prevents them from being reassigned; they can still call methods that mutate themselves. For example, you can declare a `final ArrayList`, but this does not prevent you from modifying the contents of it, such as through the use of the `add(T toInsert)` method.

   (c) final vs non-final method.
   A final method can't be overridden by a subclass.

   (d) class vs instance variable. Declare variables of both types.
   Class (static) variables are associated with the class, not to instances of that class. All instances of a class access the same static variable. However, instance (non-static) variables are unique to each instance.

   ```
   class Types {
       public static int im_a_class_var = 1;
       public int im_an_instance_var = 2;
   }
   ```

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

   - `a.equals(b)` Compares objects for equality. Class `Object` provides a default implementation (to be precise, it is `==` by default) that can be overridden for behavior necessary for a certain class. Returns a `boolean`.
   - `a == b` Checks *memory locations* (if the two objects are the SAME object, as defined by whether or not `a` and `b` point to the same spot in memory). Can also be used to check whether `a` is `null`. Also returns a `boolean`.
   - `a.compareTo(b)` Returns an `int` indicating whether `a` is less than (-1), equal to (0), or greater than (+1) `b`, according to their natural ordering. Specified by the `Comparable` interface.
   - `compare(a,b)` Returns a negative a $<$ b, 0 if a $=$ b, a positive if a $>$ b.
     With two `Comparator` objects, `comp1` and `comp2`, `comp1.equals(comp2)` implies that
     `sgn(comp1.compare(o1,o2))==sgn(comp2.compare(o1, o2))` for every object reference `o1` and `o2`.

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));
```

`Collection` does not support `get(i)`. The better solution is:

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

7. 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 runtime mistakes, and any reasonably robust system should be able to recover from one. Classic example is `IOException`.

runtime exception - 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 `IOError`, `VirtualMachineError` and `ThreadDeath` (see Java's `Error` class).

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-nanananan";
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 }
```

HeySteve, cut that out!

It's a Beach-bash! WakaWaka!

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?

`Probable` needs to define that certain actions can be performed on `Probable` objects, but does not need to define what those actions should do.

(b) Write the `Probable` interface.

```
1 public interface Probable{
2     public void probe();
3     public int getMentalResolve();
4     public void returnHome();
5 }
```

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

No! Since `Redneck` and `Professor` objects are stored in variables of type `Human`, they must extend the `Human` class. `Human` must be a class rather than an interface because it has state information that must be inherited. In addition, since the `Human` variables are able to be added into a collection of `Probable` objects, the `Human` class must implement `Probable`, which will carry down into the `Redneck` and `Professor` classes.

10. **Networking**

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

TCP stands for Transmission Control Protocol. We use TCP in Telephone Connection because TCP guarantees packet delivery and thus can be considered "lossless and reliable".

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

User Datagram Protocol. We use UDP when we are managing a tremendous amount of state (ex: weather data, video transmission).

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

TCP.

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

UDP.

(e) What is a datagram?

A datagram is an independent, self-contained message sent over the network with no guarantees.

(f) What is a socket?

A socket refers to the endpoints of logical connections between two hosts, which can be used to send and receive data.

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();
```

(a) Server: reader is not always instantiated!

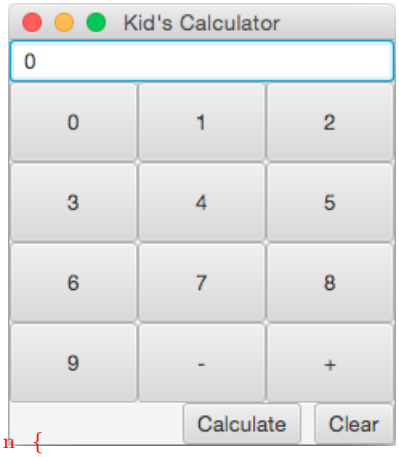
(b) Server: client never initialized, use pubServer.accept().

(c) Server: Missing client.close().

(d) Server and Client: *.close() should be in a try-catch block.

(e) Client: Need a PrintWriter writer; writer = new PrintWriter(conn.getOutputStream(), true); writer.println(args[2]); instead of System.out.println(args[2]);

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.

```
1  public class KidsCalc extends Application {
2      private final static int BUTTON_WIDTH = 80;
3      private final static int BUTTON_HEIGHT = 50;
4      private final static int MIN_WIDTH = BUTTON_WIDTH*3;
5
6      public void start(Stage primaryStage) throws Exception {
7          Scene scene = new Scene(this.makeMainPane());
8          primaryStage.setTitle("Kid's Calculator");
9          primaryStage.setScene(scene);
10         primaryStage.show();
11     }
12
13     private Parent makeMainPane() {
14         BorderPane bp = new BorderPane();    //BORDER PANE LAYOUT
15         bp.setPrefWidth(MIN_WIDTH);
16         bp.setTop(makeTopArea());          //TOP
17         bp.setCenter(makeCenterArea());  //CENTER
18         bp.setBottom(makeBottomArea());  //BOTTOM
19         return bp;
20     }
21
22     private Node makeTopArea(){
23         TextField t = new TextField("0");
24         t.setEditable(false);
25         return t;
26     }
27
28     private Node makeCenterArea(){
29         FlowPane flow = new FlowPane();
30         flow.setMinWidth(MIN_WIDTH);
31
32         for(int i = 0; i<10; i++){  // buttons 0 - 9
33             Button b = new Button(String.valueOf(i));
34             b.setPrefSize(BUTTON_WIDTH,BUTTON_HEIGHT);
35             flow.getChildren().add(b);
36         }
37
38         Button minus = new Button("-");
39         minus.setPrefSize(BUTTON_WIDTH,BUTTON_HEIGHT);
40         flow.getChildren().add(minus);
41         Button plus = new Button("+");
42         plus.setPrefSize(BUTTON_WIDTH, BUTTON_HEIGHT);
43         flow.getChildren().add(plus);
44         return flow;
45     }
46
47     private Node makeBottomArea(){
```

```
48          HBox bottom = new HBox(8);                  //HBox for bottom row
49          bottom.setAlignment(Pos.CENTER_RIGHT);
50          Button calculate = new Button("Calculate"); //calculate button
51          bottom.getChildren().add(calculate);
52          Button clear = new Button("Clear");         //clear button
53          bottom.getChildren().add(clear);
54          return bottom;
55      }
56  }
```

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

    The Iterator interface implements the iterator pattern. It allows access to a list data structure, put protects it by keeping the implementation hidden and only providing the functions needed. The Observer/Observable interfaces implement the Observer pattern. They allow for an observable class to notify its observers of changes, which they can then retreive by accessor methods. Streams implement the decorator pattern. The stream classes allow for a programmer to define read and write methods of their own, which can then be used on any other stream, allowing for "decoration" of any of the basic streams without knowing their implementation.

14. **NullPointerExceptions**

    (a) Briefly describe what a `NullPointerException` is.
    A `NullPointerException` occurs (is thrown) when an application attempts to make use of `null` where an object instance was required. In other words, the exception will occur if you make use of a reference as if it were a object, but it is actually `null`; the error occurs because objects have functionality that `null` does not.

    (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.

```
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:

```
myList = new ArrayList<>(); // java7 diamond operator ftw!
```

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

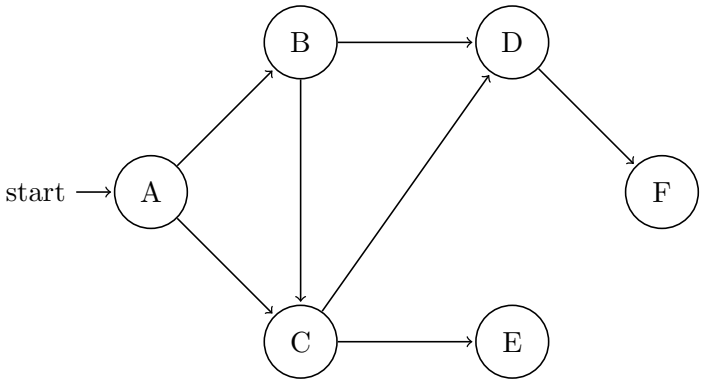
15. Suppose we are talking about the depth-first search (DFS) algorithm. Nodes are added to the data structures in alphabetical order.

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

(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.

← bottom of stack
|A
|B C
|B D E
|B D
|B F
|B

The traversal order is ACEDF.



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

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

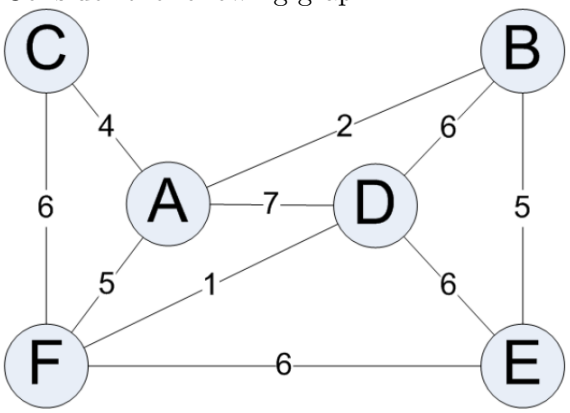
(a) What changes would need to be made to a DFS implementation to turn it into a breadth-first search (BFS)?
Use a queue data structure (instead of a stack).

(b) Using the graph as described in Question 1, what is the BFS traversal order? Show the data structure at each step.
← front of queue
A
B C
C D
D E
E F
F

The traversal order is ABCDEF.

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

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)
C	**(4, C)**	(∞, None)	(0, None)	(∞, None)	(∞, None)	**(6, C)**
A	(4, C)	**(6, A)**	(0, None)	**(11, A)**	(∞, None)	(6, C)
F	(4, C)	(6, A)	(0, None)	**(7, F)**	**(12, F)**	(6, C)
B	(4, C)	(6, A)	(0, None)	(7, F)	**(11, B)**	(6, C)
D	(4, C)	(6, A)	(0, None)	(7, F)	(11, B)	(6, C)
E	(4, C)	(6, A)	(0, None)	(7, F)	(11, B)	(6, C)

The shortest path is CABE with a total cost of 11. To reconstruct this solution, we start with the destination node, then move on to its recorded optimal predecessor. We repeat the process until we reach the starting node. (Note that this isn't the only possible 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?

No. The algorithm is finished as soon as the destination node has been finalized; Dijkstra's is a greedy algorithm so it will never change decisions once they are made.

18. Define polymorphism and explain a situation in which you would use it.
    Polymorphism: when objects of various types define a common interface of operations (in other words, objects of different subclasses appear and act as a shared superclass). References to and collections of a super class may hold instances of subclasses. Methods invoked on these objects determine the correct (type-specific) behavior at runtime. That is, an object instantiated from a subclass will use the subclass's implementation of a method, even if it is stored in a reference to its superclass. This allows different subclasses to be plugged in and used by superclass variable.

    Example:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();

// Square with side length of 1
shapes.add(new Square(1.0));

// Circle with radius of 9
shapes.add(new Circle(9.0));

// Triangle with base of 4 and height of 5
shapes.add(new Triangle(4.0, 5.0));

// Outputs the correct area of each shape
for( int i = 0; i < shapes.size(); i++ ){
    double area = shapes.get(i).getArea();
    System.out.println(area);
}
```

19. If an instance variable is declared with `protected` access, who can access it?
    The class that the variable is defined in, any subclasses of that class, and any classes in the same package as that class.

20. What is the difference between an abstract class and an interface? Why might you want to use an interface over an abstract class?
    Abstract classes can have fields, whereas interfaces cannot. Classes can implement multiple interfaces, but can only extend a single class. Because of this, if you want a class to be able to be treated as two or more unrelated types, make those types interfaces. In Java 7 and earlier, interfaces can only have method signatures, without any implementation. Abstract classes can have a mix of abstract methods and methods with an implementation. This allows abstract classes to provide default behavior for their subclasses, as well as common base behavior that subclasses can extend from.

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 }
```

```
Class1()                    Class1()
Class2()                    Class3()
Class2.print1()             Class1()
Class1.print2()             Class2()
```

```
Class2.print1()                    Class1.print2()
Class3.print2()
```