2020-2021: Final Question

(1) (b) (2) Find errors and fix them.

(i) Corrected code:

Error: • Methods g() and h() are missing the static keyword
• they should have proper return types like void.

Corrected code:

```
static void g() {
    System.out.println("Inside method g");
}
Static void h() {
    System.out.println("Inside method h");
}
```

(ii) error: product() has no return type specified:

Corrected code:

```
static int product() {
    int a=6, b=5, c=4, result;
    result = a*b*c;
    System.out.printf("Result is %d\n", result);
    return result;
}
```

(iii) Error: sum(int x, int y) is missing a return type

Corrected code:

```
static int sum(int x, int y) {
    int result = x+y;
    return result;
}
```

(iv) Error: the variable a is not declared or initialized:

Corrected code:

```
static void f(float a){
    System.out.println(a);
}
```

(2) [a] What is static variable and method? why is the main method static?

→

Static variable:

- A Variable belongs to the class rather than any object.
- shared across all instance of the class.

Static method:

- A method that belongs to the class and can be called without creating an object.

Why main is static:

- The main method is static because it allows the JVM to call it without creating an instance of the class.
- This ensures that the program can start execution from the main method.

```
class Example {
    static int counter = 0;
    static void increment(){
        counter++;
    }
    public static void main (String[] args){
        increment();
        System.out.println("counter: " + count);
    }
}
```

[b] Constructor in OOP:

- A constructor is a special method used to initialize objects of a class.

Multiple Constructor:

- Yes, multiple constructors are possible in java using constructor overloading.
- Overloaded constructors have the same name as the class but different parameter lists.

```
class person {
        String name;
        int age;

person (String name) {
        this.name = name;
    }
person (String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

(3) [a] Piggy Bank:

```
class AddAmount {
    private int amount = 50;

// Constructor no parameter
    AddAmount();
// Constructor with parameter
    AddAmount (int addAmount) {
```

```
    this amount t = addAmount;
}
void displayAmount () {
    System.out.println ("final amount: " + amount);
}
}
public class Main {
    public static void main(String[] args){
        AddAmount obj1 = new AddAmount();
        AddAmount obj2 = new AddAmount(100);
        obj1. displayAmount(); // 50
        obj2. displayAmount(); // 150;
    }
}
```

[b] Define Encapulation and difference between Interface and abstract class.

Encapsulation: Wrapping data (variable) and code (method) together in a single unit.

Ex:

```
class BankAccount {
    private int balance;
public int getBalance() {
    return balance;
}
public void setBalance (int balance) {
    this.balance = balance;
}
}
```

• Interface Vs Abstract class

Interface	Abstract class
– Only method declarations, no implementations.	– Can have both abstract and concrete method.
– Only static and final variables (field).	– Can have instance variable (fields)
– Supports multiple inheritance	– Single inheritance

(4) [a] Difference Between OOP and procedural programing

OOP	procedural programming
– organized using objects and classes.	– organized using functions and procedures.
– promotes code reuse through inheritance	– Minimal support for code reuse.
– encapsulation ensures data security	– Data is exposed to all functions

[b] Abstract class and dog implementation:

```
Abstract class Animal {
        String name;
  abstract void eat();
Void SetName (String name) {
        this.name = name;
  }
  String getName (){
        return name;
  }
}
```

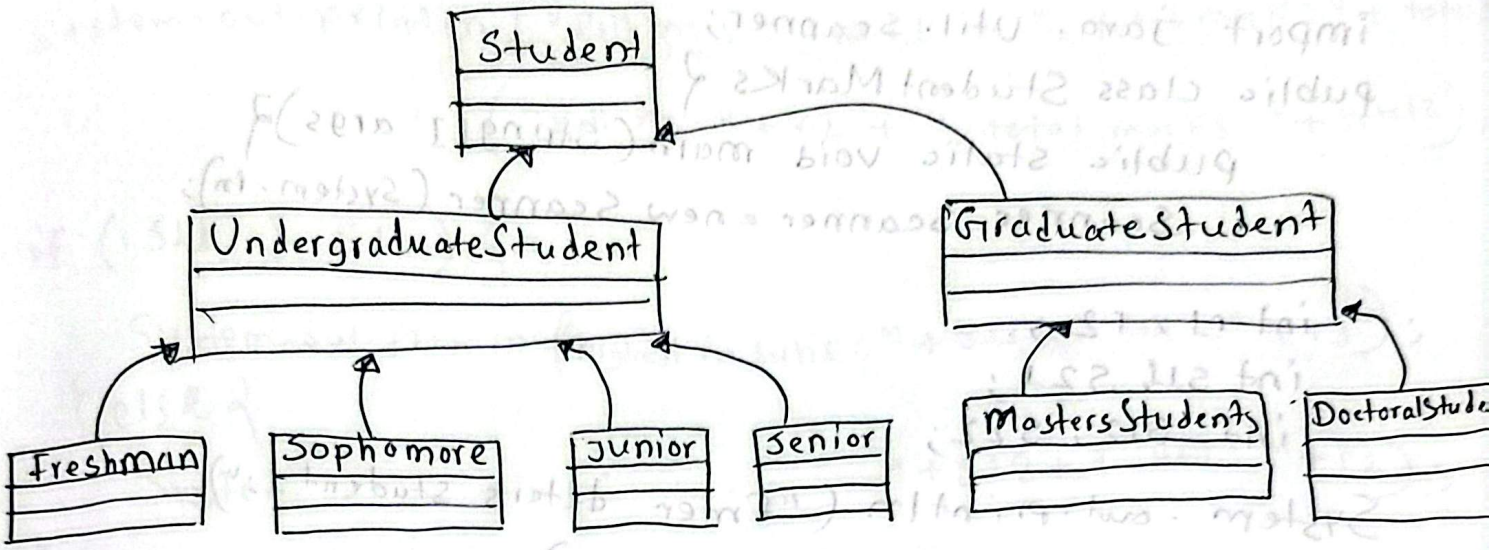
```
class Dog extends Animal {
    void eat () {
        System.out.println (name + " is eating");
    }
}
public class main {
    Public Static void main (String [] args) {
        Dog dog = new Dog ();
        dog.setName ("Buddy");
        System.out.println ("Dog's Nae: " + dog.getName());
        dog.eat ();
    }
}
```

Inheritance Hierarchy:

```
class Student {
    void info () {
        System.out.println ("I am a student");
    }
}
class UndergraduateStudent extends Student {
    void level () {
        System.out.println ("Undergraduate level student");
    }
}
class GraduateStudent extends Student {
    void level () {
        System.out.println ("Graduate level student");
    }
}
Class Freshman extends Undergraduate Student {
    Void year () {
        System.out.println ("Freshman year.");
    }
}
```

```
class sophomore extends UndergraduateStudent {
    void year(){
        System.out.println("sophomore year.");
    }
}
class junior extends UndergraduateStudent{
    void year(){
    System.out.println("junior year");
    }
}
class senior extends UndergraduateStudent{
    void year(){
    System.out.println("Senior year.");
    }
}
class Mastersstudents extends GraduateStudent{
    void degree(){
    System.out.println("Master's degree student");
    }
}
class DoctoralStudent extends GraduateStudent{
    void degree(){
    System.out.println("Doctoral student");
    }
}
public class University {
    public static void main(String[] args){
        Senior S = new Senior();
        S.info();   // from student.
        S.level();  // from UndergraduateStudent
        S.year();   // from senior;
    DoctoralStudent d = new DoctoralStudent();
        d.info();   // from student;
        d.level();  // from GraduateStudent
        d.degree(); // from Doctoral student
    }
}
```

• Hierarchy Diagram :



• Relationship :

— Student is the base class for all types of student.

– UnderGraduate and Graduate extend Student and represent two major Categories.

– Freshman, Sophomore, Junior, Senior are specific types of Undergraduate Students.

– Masters Student and Doctoral Student are types of graduatestudents.

– This is a multilevel inheritance and represent a real world academic hierarchy.

5 [a] (19-20) mid [5] No

[b] Wrapper classes and Autoboxing :

• Wrapper classes : Integer, Float, Double, Boolean etc they convert primitive data types to objects.

• Autoboxing : Converting primitive to object automatically.

int a = 5; Integer b = a; ← autoboxing.

• Unboxing : Object to primitive

Integer b = 5; int a = b; ← Unboxing.

[e] Program to read table and output:

```
import java.util.Scanner;
public class StudentMarks {
    public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);

    int r1, r2;
    int s11, s21;
    int s12, s22;
System.out.println("Enter details student 1:");
    System.out.print("Roll: ");
    r1 = scanner.nextInt();
    System.out.print("Marks sub 1 ");
    s11 = scanner.nextInt();
    System.out.print("Marks sub2");
    s21 = scanner.nextInt();
System.out.println("Enter details for Ste 2");
    System.out.print("Roll:");
    r2 = Scanner.nextInt();
System.out.print("Marks sub1");
    s12 = Scanner.nextInt();
System.out.print("Marks sub2");
    s22 = Scanner.nextInt();
int total1 = s11 + s21;
int total2 = s12 + s22;
```

```
System.out.println("total marks obtained by students");
System.out.println("Roll no: " + r1 + ", total mark: " + total1);
//          //("Roll no: " + r2 + ", total marks: " + total2);

if (s11 > s12){
    System.out.println("highest in sub1: " + s11 + " Roll: " + r1);
}else{
    Syste... ---.("..... " + s12 + " Roll: " + r2);
}
if (sub21 > sub22){
    ---------.(... + r1);
}else{
    ---  ---  ---(... + r2);
}
Scanner.close();
}
```

(6) [a] Output of given code:

i) `int i = 7, j = 9;`
`modresult = j % i;` // 2
`divresult = i / modresult;` 7/2 = 3
`cout << divresult;` // 3

ii) `i[1] != i[0]` so, `j = 1`.
`(j >= 1)`
`cout << "OK";`

Output: OK;

b

```
class ExceptionA extends Exception{
    public ExceptionA (String message){
        super (message);
    }
}
class ExceptionB extends ExceptionA {
    public ExceptionB ( String message){
        super (message);
    }
}
class ExceptionC extends ExceptionB{
    public ExceptionC (String message ){
        super (message);
    }
}
public class ExceptionDemo {
    public static void main (String[] args){
        try {
            throw new ExceptionB (" This is ExceptionB");
        } catch (ExceptionA e){
            System.out.println ("caught: ExceptionA:"
                    + e.getMessage());
        }
        try {
            throw new ExceptionC ("This is Exception c");
        } catch ( ExceptionA e){
            System.out.println ("caught ExceptionA: "+
                    e.getMessage());
        }
    }
}
```

[c] Max Min finder:

```
public class MaxMinFinder {
    public static void main (String[] args){
        ~~int arr~~
        int[] arr = {10, 0, 20, 30, 5, 24, 30};
        int max = arr[0];
        int min = arr[0];
    for (int num: arr) {
        if (num > max) {
            max = num;
        }
        if (num < min) {
            min = num;
        }
    }
    System.out.println ("Max = " + max);
    System.out.println ("Min = " + min);
}
}
```

(7) [a]

```
class TaskA extends Thread {
    public void run() {
        for (int i=0; i<100; i++) {
            System.out.println ("Hello World");
        }
    }
class TaskB extends Thread {
    Public void run() {
        int sum = 0;
        for (int i=0; i<=100; i++) sum += i;
        System.out.println ("Sum: " + sum);
    }
}
```

```
class TaskC extends Thread {
    public void run() {
        for (int i = 0; i < 200; i++) {
            System.out.println("I can do it");
            try { Thread.sleep(1000); } catch(Exception e) { }
        }
    }

public class Main {
    public static void main(String[] args) {
        new TaskA().start();
        new TaskB().start();
        new TaskC().start();
    }
}
```

[b] Forms of implementing Interfaces:

1. **using class:**

```
class myclass implements Runnable {
    pubic void run() {
        System.out.println("Thread running");
    }
}
```

2. **Using Anonymous class:**

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Thread running");
    };
};
```

3. **Lambda Expression:**

```
Runnable r = () ->
    System.out.println("Thread running");
```

⑧ [a] Java collection Framework Sort conditions:
- the elements must implement Comparable
- Use a custom Comparator during Sort.

Ex: Collections.sort(list);
Collections.sort(list, Comparator);

Final : 2019-20

① [b]

i) Explanation: This will cause a compilation error. In java the condition in the if statement must be a boolean. -1 is an integer, so this is not allowed.
Output: Compilation Error.

ii) Explanation: String concatenation happens left to right.
Output: CSE.

iii) Explanation: The loop runs once, prints "Hello", then break exits the loop.
Output: Hello.

iv) Explanation: This is valid method overloading. Java allows multiple main methods with different parameters.
Output: No output (runs successfully but does nothing).

② [a] 2018-19 mid (5) no.
[a]

③ [a] mid: 19-20 (4) no
[b] mid (19-20) 5 no.

④ [a] final (20-21) 2(b).
[b] final (20-21) 4(b).
final (21-22) 1(c).

(5) [a] mid (19-20) (3)

[b] How does String class differ from the StringBuffer class?

→

String class	String Buffer
- In string class the length of String class is fixed.	- The string buffer class the length of String Buffer not fixed.
- With the string class we cannot create modifiable String.	- String buffer class is used to create modifiable string
- Not Thread safe	- ~~not~~ thread safe.
- consumers more energy	- consumers less energy
- String constant memory	- Heap memory

[c] Voting program (Java):

```
import. java.util.Scanner;
public class ~~void main~~ VotingProgram {
    public static void main (String[] args) {
        int[] Votes = new int [5];
        int spoilBallots = 0;
    Scanner sc = new Scanner (System.in);
    System.out.println ("Enter total number of ballots: ");
    int n = sc.nextInt();
  System.out.println ("Enter votes (1 to 5): ");
   for (int i = 0; i<n; i++) {
          int vote = sc.nextInt();
          if (Vote >= 1 && vote <= 5) {
            votes [vote-1]++;
        } else {
              spoilBallots++;
        }
    }
```

```
for (int i = 0; i < votes.length; i++) {
    System.out.println("candidate " + (i+1) + " received: "
                       + votes[i] + " votes");
}
System.out.println("spoilt Ballots: " + spoilBallots);
}
}
```

(6) [a] mid-2 (21-22) (1).

[b] mid-2 (19-20) (2)

[c] mid-2 (21-22) (5)

(7) [a] Final (21-22) 7 (a).

[b] Elevator Control Application:

```
public class ElevatorTest {
    public static void main(String[] args) {
        Elevator elevator = new Elevator();
        elevator.setState(new DoorIsClosed());
        elevator.doAction();

        elevator.setState(new DoorIsOpen());
        elevator.doAction();
    }
}
```

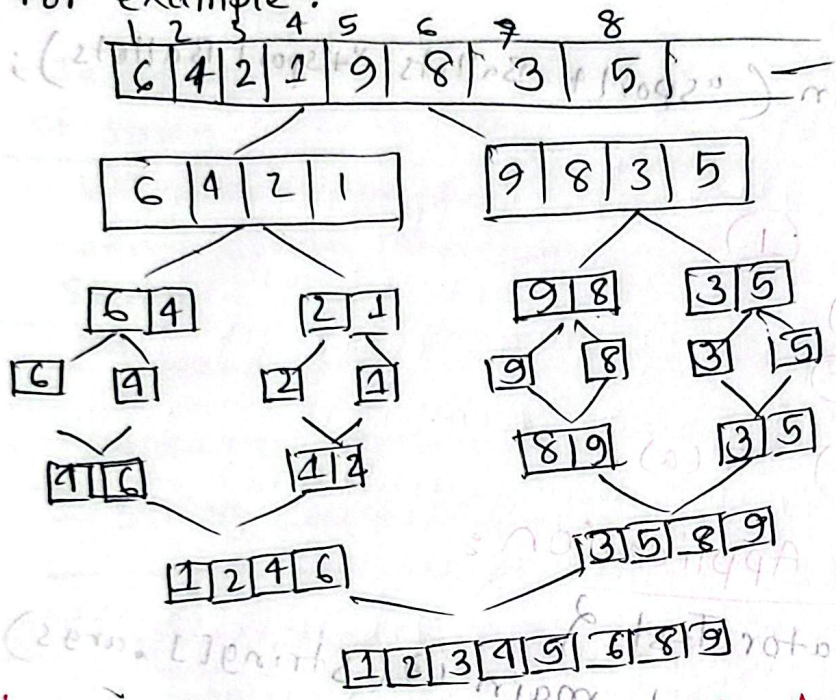
(8) [a] Collections: sort(), min(), max():

In java the collections framework provides a static method sort() that can be used to sort elements in a collection.

The sort() method of the collection framework uses the merge sort algorithm to sort elements of a collection.

The merge sort () algorithm is based on divide and conquers rule. The advantage of merge sort is it's time complexity which is nlogn.

for example:



[b] Explain why you would choose a LinKedHashSet over a HashSet.

→

Hashset → doesn't maintain order.
LinkedHashSet maintains insertion order.

- Use LinKHashSet When:
  - You need uniaue elements
  - You want to preserve the order in which items were added.

[c] Final (21-22) 6(c).