1. **With relevant examples, explain the following concepts as used in Java programming.**
a. Mutable classes.

Explain what is meant by mutable class
A mutable class is one that can change its internal state after it is created.

Write a program that implements the concept of a mutable class

```
class MutableStudent{
    private String name;

        MutableStudent(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

}

public class Main {

    public static void main(String[] args) {
        MutableStudent mutableStudent = new MutableStudent("Jai");
        System.out.println("Student Name: " + mutableStudent.getName());
        mutableStudent.setName("Sandy");
        System.out.println("Student Name after modification: " +
mutableStudent.getName());
    }
}
```

b**. Immutable classes.**
Explain what is meant by immutable class
Immutable class in java means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Write a program that implements the concept of immutable class

```
// Java Program to Create An Immutable Class

// Importing required classes
import java.util.HashMap;
import java.util.Map;

// Class 1
// An immutable class
final class Student {

        // Member attributes of final class
        private final String name;
        private final int regNo;
        private final Map<String, String> metadata;

        // Constructor of immutable class
        // Parameterized constructor
        public Student(String name, int regNo,
                                Map<String, String> metadata)
        {

                // This keyword refers to current instance itself
                this.name = name;
                this.regNo = regNo;

                // Creating Map object with reference to HashMap
                // Declaring object of string type
                Map<String, String> tempMap = new HashMap<>();

                // Iterating using for-each loop
                for (Map.Entry<String, String> entry :
                        metadata.entrySet()) {
                        tempMap.put(entry.getKey(), entry.getValue());
                }

                this.metadata = tempMap;
        }

        // Method 1
        public String getName() { return name; }

        // Method 2
        public int getRegNo() { return regNo; }

        // Note that there should not be any setters

        // Method 3
        // User -defined type
        // To get meta data
        public Map<String, String> getMetadata()
        {

                // Creating Map with HashMap reference
                Map<String, String> tempMap = new HashMap<>();

                for (Map.Entry<String, String> entry :
                        this.metadata.entrySet()) {
```

```
                tempMap.put(entry.getKey(), entry.getValue());
            }
            return tempMap;
        }
}

// Class 2
// Main class
class GFG {

        // Main driver method
        public static void main(String[] args)
        {

            // Creating Map object with reference to HashMap
            Map<String, String> map = new HashMap<>();

            // Adding elements to Map object
            // using put() method
            map.put("1", "first");
            map.put("2", "second");

            Student s = new Student("ABC", 101, map);

            // Calling the above methods 1,2,3 of class1
            // inside main() method in class2 and
            // executing the print statement over them
            System.out.println(s.getName());
            System.out.println(s.getRegNo());
            System.out.println(s.getMetadata());

            // Uncommenting below line causes error
            // s.regNo = 102;

            map.put("3", "third");
            // Remains unchanged due to deep copy in constructor
            System.out.println(s.getMetadata());
            s.getMetadata().put("4", "fourth");
            // Remains unchanged due to deep copy in getter
            System.out.println(s.getMetadata());
        }
}
```

c. **Explain the situations where mutable classes are more preferable than immutable classes when writing a Java program.**

- The mutable objects can be changed to any value or state without adding a new object. Whereas, the immutable objects can not be changed to its value or state once it is created. In the case of immutable objects, whenever we change the state of the object, a new object will be created.

- Mutable objects provide a method to change the content of the object. Comparatively, the immutable objects do not provide any method to change the values.
- The mutable objects support the setters and getters both. Comparatively, the immutable objects support only setters, not getters.
- The Mutable objects are may or may not be thread-safe, but the immutable objects are thread-safe by default.
- The mutable class examples are StringBuffer, Java.util.Date, StringBuilder, etc. Whereas the immutable objects are legacy classes, wrapper classes, String class, etc.

2.

a. **Explain what a String buffer class is as used in Java, the syntax of creating an object of StringBuffer class and Explain the methods in the StringBuffer class.**

StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences

The syntax of creating a StringBuffer object is:

The methods in the StringBuffer class are;

append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method
capacity()	the total allocated capacity can be found by the capacity( ) method
charAt()	This method returns the char value in this sequence at the specified index.

b. **Write the output of the following program.**

```
class Myoutput
 {
     public static void main(String args[])
     {
         String ast = "hello i love java";
         System.out.println(ast.indexOf('e')+" "+ast.indexOf('ast')+" "+ast.lastIndexOf('l')+" "+ast .lastIndexOf('v'));
     }
 }
```

Output:

The program has no output

c. Explain your answer in (2b) above.

In the above code we have ast.indexOf('ast'). indexOf() does not take a String argument hence resulting to an error.

d. **With explanation, write the output of the following program**.

```
class Myoutput
1.  {
2.      public static void main(String args[])
3.      {
4.          StringBuffer bfobj = new StringBuffer("Jambo");
5.          StringBuffer bfobj1 = new StringBuffer(" Kenya");
6.          c.append(bfobj1);
7.          System.out.println(bfobj);
8.      }
9.  }
```

The program does not run because of an error in line 6. “c.append(bfobj1);”. The variable “c” was not created.

e. **With explanation, write the output of the following program.**

```
class Myoutput
{
    public static void main(String args[])
    {
       StringBuffer str1 = new StringBuffer("Jambo");
       StringBuffer str2 = str1.reverse();
       System.out.println(str2);
    }
}
```

Output: obmaJ

This is because the original str1 having “Jambo” has been reversed by the reverse() function and transferred to the str2 variable that is later printed.

f. **With explanation, write the output of the following program**.

```
class Myoutput
{
    class output
    {
      public static void main(String args[])
      {
         char c[]={'A', '1', 'b' ,' ' ,'a' , '0'};
         for (int i = 0; i < 5; ++i)
         {
              i++;
                if(Character.isDigit(c[i]))
                    System.out.println(c[i]+" is a digit");
                if(Character.isWhitespace(c[i]))
                    System.out.println(c[i]+" is a Whitespace character");
                if(Character.isUpperCase(c[i]))
                    System.out.println(c[i]+" is an Upper case Letter");
                if(Character.isLowerCase(c[i]))
                    System.out.println(c[i]+" is a lower case Letter");
                i++;
        }
    }
}
```

Output:

```
1 is a digit
a is a lower case Letter
```

At the first loop, we check if the second value is a digit, a whitespace, an uppercase or lowercase. Since it is “1”, then it is a digit, and we print to the console.
We then skip the third value, and check the forth value if it is a digit, a whitespace, an uppercase or lowercase. Since the forth value is “a”, then it is a lowercase, and we print to the console.
“I” is incremented two times in the loop.