PART- B

Q1) List of OOPs Concepts in Java with Examples:

1) Class: The class is one of the Basic concepts of OOPs which is a group of similar entities. It is only a logical component and not the physical entity. Example, if you had a class called “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc. the methods may be performed with these cars are driving, reverse, braking etc.

2) Object : An object can be defined as an instance of a class, and there can be multiple instances of a class in a program. For example - chair, bike, marker, pen, table, car, etc.

3) Inheritance : Inheritance is one of the Basic Concepts of OOPs in which one object acquires the properties and behaviors of the parent objectIt offers robust and natural mechanism for organizing and structure of any software.

4) Polymorphism: Polymorphism refers to one of the OOPs concepts in Java which is the ability of a variable, object or function to take on multiple forms. For example, in English, the verb *run* has a different meaning if you use it with *a laptop*, *a foot race*, and *business*

5) Abstraction: Abstraction is one of the OOP Concepts in Java which is an act of representing essential features without including background details. Example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

6) Encapsulation: Encapsulation is one of the best Java OOPs concepts of wrapping the data and code. In this OOPs concept, the variables of a class are always hidden from other classes. For example - in school, a student cannot exist without a class.

7) Association : Association is a relationship between two objects. It is one of the OOP Concepts in Java which defines the diversity between

objects. In this OOP concept, all object have their separate lifecycle, and there is no owner. For example, many students can associate with one teacher while one student can also associate with multiple teachers.

8) Aggregation

In this technique, all objects have their separate lifecycle. However, there is ownership such that child object can't belong to another parent object. For example consider class/objects department and teacher. Here, a single teacher can't belong to multiple departments, but even if we delete the department, the teacher object will never be destroyed.

9) Composition:. It is also called "death" relationship. Child objects do not have their lifecycle so when parent object deletes all child object will also delete automatically. For that, let's take an example of House and rooms. Any house can have several rooms. One room can't become part of two different houses. So, if you delete the house room will also be deleted.

Q2)

1.Simple: Java inherits all the best features from the programming languages like C, C++ and thus makes it really easy for any developer to learn with little programming experience. The concept of Object Oriented programming was not invented by Java but it was just adopted by the Java team

2.Secure: When Java programs are executed they don't instruct commands to the machine directly. Instead Java Virtual machine reads the program (ByteCode) and convert it into the machine instructions. This way any program tries to get illegal access to the system will not be allowed by the JVM.

3.Portable: Java programs are portable because of its ability to run the program on any platform and no dependency on the underlying hardware / operating system.

4.Object Oriented Programming Language: Java is object oriented programming language but everything in Java are not objects. Java manages to maintain balance and adopted what make sense in the current situation. The object oriented model in Java is simple and easy to extend and also the primitive types such as integers, are retained for high-performance.

5.Robust: Following features of Java make it Robust.

- Platform Independent
- Object Oriented Programming Language
- Memory management
- Exception Handling

6.Multithreaded: Java allows you to develop program that can do multiple task simultaneously.

7.Interpreter: The compiled code of Java is not machine instructions but rather its a intermediate code called ByteCode. This code can be executed on any machine that implements the Java virtual Machine. JVM interprets the ByteCode into Machine instructions during runtime.

8.High Performance: When java programs are executed, JVM does not interpret entire code into machine instructions. If JVM attempts to do this then there will huge performance impact for the high complexity programs.

9.Distributed: Java has a feature called Remote Method Invocation (RMI) using which a program can invoke method of another program across a network and get the output.

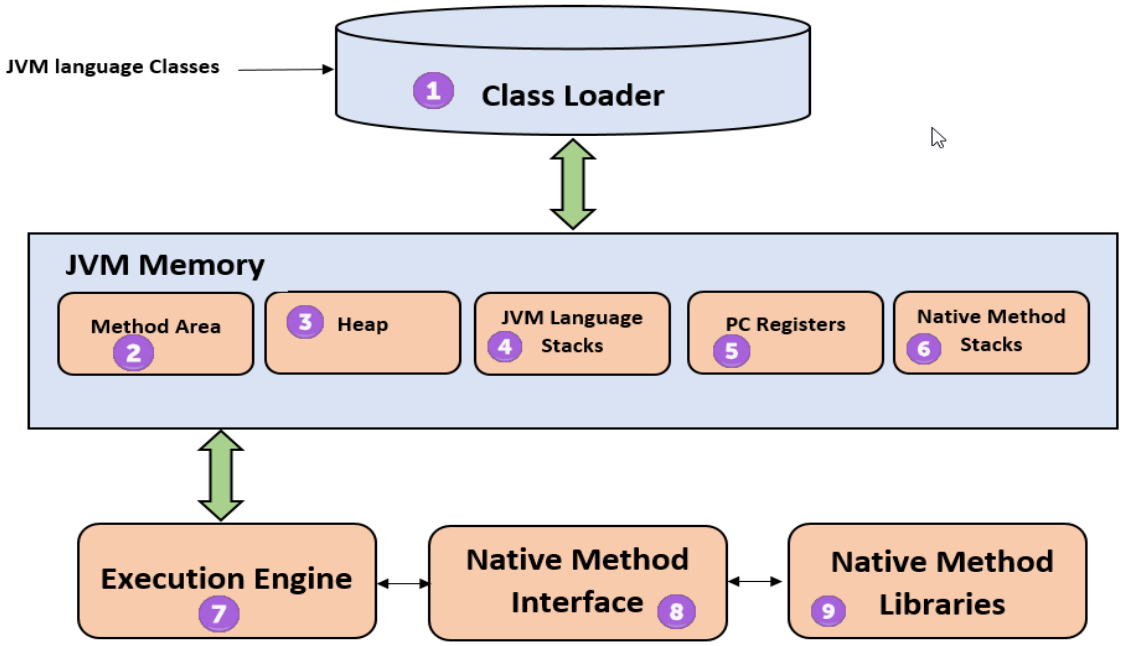
10.Dynamic: Java programs access various runtime libraries and information inside the compiled code (Bytecode). This dynamic feature allows to update the pieces of libraries without affecting the code using it.

4Q)

```
error: illegal character: '\u201c'
            System.out.pirnltn(“ y is :” + y);
                               ^
Main.java:16: error: ';' expected
            System.out.pirnltn(“ y is :” + y);
                               ^
```

5Q) **Java Virtual Machine (JVM)** is a engine that provides runtime environment to drive the Java Code or applications. It converts Java bytecode into machines language. JVM is a part of Java Run Environment (JRE). In other programming languages, the compiler produces machine code for a particular system.

JVM Architecture : JVM architecture in Java contains classloader, memory area, execution engine etc



1) **ClassLoader:** The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.
2) **Method Area:** JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.
3) **Heap:** All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple thread.
4) **JVM language Stacks**: Java language Stacks store local variables, and it's partial results. Each thread has its own JVM stack, created simultaneously as the thread is created.

5) **PC Registers:** PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.
6) **Native Method Stacks**: Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.
7) **Execution Engine**:It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.
8) **Native Method interface**: The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.
9) **Native Method Libraries**:Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine.

Q6) An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

Single dimensional array − A single dimensional array of Java is a normal array where, the array contains sequential elements (of same type) –

```
type var-name[ ];

int calendar_months[];

calendar_months = new int[12];
```

Example:

```
class AutoArray

 {

public static void main(String args[]) {
```

int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

System.out.println("April has " + month_days[3] + " days.");

}}

Multi-dimensional array – A multi-dimensional array in Java is an array of arrays. A two dimensional array is an array of one dimensional arrays and a three dimensional array is an array of two dimensional arrays.

int twoD[][] = new int[4][5];

Example:

```
public class Tester {

   public static void main(String[] args) {

      int[][] multidimensionalArray = {
{1,2},{2,3}, {3,4} };

      for(int i = 0 ; i < 3 ; i++){

         for(int j = 0 ; j < 2; j++){


System.out.print(multidimensionalArray[i][j] +
" ");

         }

         System.out.println();

      }

   }

}
```

7Q) Java provides the user with a block known as static block, which is mainly used for the static initializations of a class. The block consists of a set of statements which are executed before the execution of the main method. This is due to the fact that the class has to be loaded into the main memory before its usage, and static block is executed during

the loading of the class. On defining a number of static blocks in a program, the blocks execute from the top to the bottom. Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time the class is loaded into memory.

Example:

```
class Static {
static int p;
int q;
// creating the static block
static {
p = 18;
System.out.println("This is the static block!");
}
// end of static block
}
public class Main {
public static void main(String args[]) {
// Accessing p without creating an object
System.out.println(Static.p);
}
}
```

Output:

```
This is the static block!
18
```

It must be noted that static blocks are executed before constructors.

8Q) 1.if statement: The if statement tells our program to execute a certain section of code only if a particular test evaluates to true.

Syntax:

```
if(condition){
  Statement(s);
}
```

Sample Program:

```
package ClassThreeControlFlowStatements;

public class IfStatement {

public static void main(String[] args) {

int num = 100;

if (num<=100){

System.out.println("Value of num is "+num);
}}}
```

2. Nested if statement: An if statement inside another the statement. If the outer if condition is true then the section of code under outer if condition would execute and it goes to the inner if condition.

Syntax:

```
if(condition_1) {
  Statement1(s);

  if(condition_2) {
    Statement2(s);
  }}
```

Sample Program:

```
package ClassThreeControlFlowStatements;

public class NestedIfStatement {

public static void main(String[] args) {

int num = 100;
if (num<=100){
System.out.println("Value of num is "+num);
if (num>50){
System.out.println("Value of num is "+num);
}
```

}}}

3. if-else statement: If a condition is true then the section of code under if would execute else the section of code under else would execute.

Syntax:

```
if(condition) {
  Statement(s);
}
else {
  Statement(s);
}
```

Sample Program:

```
package ClassThreeControlFlowStatements;

public class IfElseStatement {

public static void main(String[] args) {

int num = 100;
if (num>100){
System.out.println("Value is greater than 100");
}else {
System.out.println("Value is less than 100");
}
}}
```

4. if-else-if statement:

Syntax:

```
if(condition_1) {
  /*if condition_1 is true execute this*/
  statement(s);
}
else if(condition_2) {
  /* execute this if condition_1 is not met and
   * condition_2 is met
   */
  statement(s);
```

```
}
else if(condition_3) {
  /* execute this if condition_1 & condition_2 are
   * not met and condition_3 is met
   */
  statement(s);
}
.
.
.
else {
  /* if none of the condition is true
   * then these statements gets executed
   */
      statement(s);
```

Sample Program:

```
package ClassThreeControlFlowStatements;

public class IfElseIfStatement {

public static void main(String[] args) {

int marks = 76;
      char grade;

      if (marks >= 80) {
         grade = 'A';
      } else if (marks >= 70) {
         grade = 'B';
      } else if (marks >= 60) {
         grade = 'C';
      } else if (marks >= 50) {
         grade = 'D';
      } else {
         grade = 'F';
```

```
        }
        System.out.println("Grade = " + grade);
}}
```

5.Switch Case: The switch statement in Java is a multi branch statement.

Syntax: switch(expression) {

```
    case valueOne:
        //statement(s
        break;
    case valueTwo:
        //statement(s
        break;
:
    default: //optional
    //statement(s) //This code will be executed if all cases are not
matched
```

Sample Program:

```
package ClassThreeControlFlowStatements;

public class SwitchCaseProgram {

public static void main(String[] args) {
/*The java switch statement is fall-through.
It means it executes all statement after first match if break statement
is not used with switch cases.*/
int num=200;
switch(num){
case 100:
System.out.println("Value of Case 1 is "+num);
case 200:
System.out.println("Value of Case 2 is "+num);
default:
System.out.println("Value of default is "+num);
}}}
```

Q9) When Java executes a program, the values are stored in containers called variables. It is the name of a memory location.It is also a basic unit of storage.
In Java ,variables are declared as
<datatype><variable_name>=<Variable_value>

For Example- int i=76, String s="DataFlair";

Types of Variables in Java :

1.Local Variable in Java: A local variable is a variable which has value within a particular method or a function. Outside the scope of the function the program has no idea about the variable.

For Example:

```
class DataFlair {
int a = 9,
b = 10;
void LearnJava {
int local_j = 45; // A local variable
String s = "DataFlair Training"; //A local variable
}
}
```

A local variable cannot be accessed directly outside a function.

2. Java Instance Variable: Instance variables are those which are declared inside the body of a class but not within any methods. These differ with each instance of the class created although they have the same identity.

Example:

```
import java.io. * ;
class Person {
int height,
weight; // Instance Variables
```

```
Person(int h, int w) {
this.height = h;
this.weight = w;
}
void run() {
System.out.println("Huff Puff");
}
void print() {
System.out.println("Now my weight is" + this.weight);
}
public static void main(String[] args) throws IOException {
Person A = new Person(170, 65);
A.run();
A.print();
}
}
```

Output:

```
Huff Puff
Now my weight is 65.
```

The variables height and weight are the instance variables, i,e, they are unique for every instance created. These cannot be initialized with a static keyword. However one instance of the Person class cannot share the details of the other instance of the same class.

3. Static Variables in Java::What if we want a variable that is shared across all the instances of a class? That's where static variables come in. These are class- level variables which are the same for all the instances generated . As soon a class loads, the static variables get their initialization.

Static Variables are declared in the following way:
static <variable_name> =<variable_value>

For Example:

```
import java.io. * ;
class DataFlair {
```

```
static int studentCount;
DataFlair() {
studentCount = 15;
}
void addStudent() {
studentCount++;
}
public static void main(String[] args) throws IOException {
DataFlair java = new DataFlair();
DataFlair python = new DataFlair();
java.addStudent();
python.addStudent();
System.out.println("Total Students " + studentCount);
}}
```

Output:

Total Students 17

Q10) Following are some ways in which you can create objects in Java:

1) **Using new Keyword :** Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).

   Example:

```
public class A
{
String str="hello";
public static void main(String args[])
{
A obj=new A();  //creating object using new keyword
System.out.println(obj.str);
}  }
```

2) **Using New Instance :** If we know the name of the class & if it has a public default constructor we can create an object –**Class.forName**. We can use it to create the Object of a Class. Class.forName

actually loads the Class in Java but doesn't create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

Example:

```
public class NewInstanceExample
{
String str="hello";
public static void main(String args[])
{
try
{
NewInstanceExample obj= NewInstanceExample.class.newInstance();

System.out.println(obj.str);
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

3) Using clone() method: Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.
To use clone() method on an object we need to implement Cloneable and define the clone() method in it.

Example:

```
public class CloneExample implements Cloneable
{
//creates and returns a copy of this object
protected Object clone() throws CloneNotSupportedException
{
return super.clone();
}
```

```
String name = "Microprocessor";
public static void main(String[] args)
{
CloneExample obj1 = new CloneExample();     //creating object of class

try
{
CloneExample obj2 = (CloneExample) obj1.clone();
System.out.println(obj1.name);
}
catch (Exception e)
{
e.printStackTrace();
}
}
}
```

4) Using newInstance() method of Constructor class : This is similar to the newInstance() method of a class. There is one newInstance() method in the **java.lang.reflect.Constructor** class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.
Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.

Example:

```
import java.lang.reflect.Constructor;
public class NewInstanceExample1
{
String str="hello";
public static void main(String args[])
{
try
{
```

```
Constructor<NewInstanceExample1> obj =NewInstanceExample1.class.getConstructor();
NewInstanceExample1 obj1 = obj.newInstance();
System.out.println(obj1.str);
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

5.Using deserialization : Whenever we serialize and then deserialize an object, JVM creates a separate object. In deserialization, JVM doesn't use any constructor to create the object.To deserialize an object we need to implement the Serializable interface in the class.

Example

In the following example we have first serialized the object and then deserialized the object.

```
import java.io.*;
class Demo implements Serializable
{
public int i;
public String s;
public Demo(int i, String s) //default constructor
{
this.i = i;
this.s = s;
}
}
public class DeserializationExample
{
```

```
public static void main(String[] args)
{
Demo object = new Demo(8, "javatpoint");
String filename = "Demofile.ser";      //specified file name (must have extension .ser)
/*-----------------Serialization----------*/
try
{
FileOutputStream file = new FileOutputStream(filename);  //Saving of object in the file
ObjectOutputStream out = new ObjectOutputStream(file);
out.writeObject(object);          //serialize object
out.close();                 //closes the ObjectOutputStream
file.close();                //closes the file
System.out.println("Object serialized");
}
catch(IOException e)
{
e.printStackTrace();
}
Demo obj = null;
/*-----------------Deserialization--------*/
try
{
FileInputStream file = new FileInputStream(filename); // reading an o bject from a file
ObjectInputStream is = new ObjectInputStream(file);
obj = (Demo)is.readObject();       //deserialize object
is.close();                  //closes the ObjectInputStream
file.close();                //closes the file
System.out.println("Object deserialized ");
System.out.println("number = " + obj.i);
System.out.println("string = " + obj.s);
}
catch(IOException e)
```

```
{
System.out.println("IOException is caught");
}
catch(ClassNotFoundException e)
{
System.out.println("ClassNotFoundException is caught");
}
}
}
```

Q11)

```
1.  import java.util.Scanner;
2.  public class JavaLoopExcercise
3.  {
4.      public static void main(String[] args)
5.      {
6.          Scanner console = new Scanner(System.in);
7.
8.          int number;
9.          int max = Integer.MIN_VALUE;
10.             int min = Integer.MAX_VALUE;
11.             char choice;
12.             do
13.             {
14.                 System.out.print("Enter the number ");
15.                 number = console.nextInt();
16.                 if(number > max)
17.                 {
18.                     max = number;
19.                 }
20.
21.                 if(number < min)
22.                 {
23.                     min = number;
24.                 }
```

```
25.
26.                 System.out.print("Do you want to continue y/n? ");
27.                 choice = console.next().charAt(0);
28.
29.             }while(choice=='y' || choice == 'Y');
30.
31.             System.out.println("Largest number: " + max);
32.             System.out.println("Smallest number: " + min);
33.         }
34.     }
```

OUTPUT:

Enter the number 5

Do you want to continue y/n? y

Enter the number 2

Do you want to continue y/n? y

Enter the number 10

Do you want to continue y/n? n

Largest number: 10

Smallest number: 2

PART-C

Q1) A *widening conversion* changes a value to a data type that can allow for any possible value of the original data. Widening conversions preserve the source value but can change its representation. This occurs if you convert from an integral type to Decimal, or from Char to String.

A *narrowing conversion* changes a value to a data type that might not be able to hold some of the possible values. For example, a fractional value is rounded when it is converted to an integral type, and a

numeric type being converted to Boolean is reduced to either True or False.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other..

Widening or Automatic Conversion

Example:

```
class Test
{
   public static void main(String[] args)
   {
      int i = 100;

      // automatic type conversion
      long l = i;

      // automatic type conversion
      float f = l;
      System.out.println("Int value "+i);
      System.out.println("Long value "+l);
```

```
        System.out.println("Float value "+f);
    }
}
```

Output:

```
Int value 100

Long value 100

Float value 100.0
```

Narrowing or Explicit Conversion

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

This is useful for incompatible data types where automatic conversion cannot be done.

Here, target-type specifies the desired type to convert the specified value to.

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

Example:

```
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);
```

```
        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:

```
Double value 100.04

Long value 100

Int value 100
```

Q2) Primitive data types: A primitive data type specifies the size and type of variable values, and it has no additional methods. There are eight primitive data types in Java. The primitive data types include boolean, char, byte, short, int, long, float and double.

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0 **Example–** byte a = 100, byte b = -50

.short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Default value is 0 **Example–** short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Integer is generally used as the default data type for integral values unless there is a concern about memory.

- The default value is 0
  Example– int a = 100000, int b = -200000

long

- Long data type is a 64-bit signed two's complement integer
- This type is used when a wider range than int is needed
- Default value is 0L
  Example– long a = 100000L, long b = -200000L

float

- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
  Example– float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Default value is 0.0d
  Example– double d1 = 123.4

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- Default value is false
  Example– boolean one = true

char

- char data type is a single 16-bit Unicode character
- Char data type is used to store any character
  Example – char letterA = 'A'

program to print first 100 Fibonacci numbers

```
import java.util.Scanner;
public class Fibonacci
{
   public static void main(String[] args)
   {
      int n, a = 0, b = 0, c = 1;
      System.out.print("Fibonacci Series:");
      for(int i = 1; i <= 100; i++)
      {
         a = b;
         b = c;
         c = a + b;
         System.out.print(a+" ");
      }
   }
}
```

Q3) Output :101

Explanation:

As instance variables, b1 and b2 are initialized to false. The if tests on lines 7 and 9 are successful so b1 is set to true and x is incremented. The next if test to succeed is on line 19(else if (b2 = true)) (note that the code is not testing to see if b2 is true, it is setting b2 to be true). Since line 19 was successful, subsequent else-if's (line 21)(else if (b1 | b2) will be skipped.

Q5) output: Compilation fails

Main.java:5: error: illegal start of expression

static int i = 0;

^

1 error

Explanation:
Compilation failed because static was an illegal start of expression - method variables do not have a modifier (they are always considered local).

Q6) Output: j=2

Explanation:
Because there are no break statements, As there is no case 1 it becomes the default case and adds 2 to j. The result is j = 2.

Q7)

```
java.util.Scanner;
class Power
{
  public static void main(String arg[])
  {
    long n,p,r=1;
    Scanner sc=new Scanner(System.in);
    System.out.println("enter number");
    n=sc.nextLong();
    System.out.println("enter power");
    p=sc.nextLong();
    if(n>=0&&p==0)
    {
      r =1;
    }
    else if(n==0&&p>=1)
      {
        r=0;
      }
    else
      {
```

```
            for(int i=1;i<=p;i++)
              {
                r=r *n;
                 }
          }
           System.out.println(n+"^"+p+"="+r);
     }
 }
```

Output: enter numberenter number 6

enter power6

6^6=46656

Q8)

```
import java.util.Scanner;

public class SumAgain
{
   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);

      int number1, number2;
      char choice;

      do
      {
         System.out.print("Enter the first number ");
         number1 = console.nextInt();

         System.out.print("Enter the second number ");
         number2 = console.nextInt();

         int sum = number1 + number2;
         System.out.println("Sum of numbers: " + sum);
```

```
            System.out.print("Do you want to continue y/n? ");
            choice = console.next().charAt(0);

            System.out.println();

        }while(choice=='y' || choice == 'Y');
    }

}
```

Q9)

```
import java.util.Scanner;

public class GuessMyNumber
{
    public static void main(String[] args)
    {
        Scanner console = new Scanner(System.in);

        int number, // To hold the random number
            guess,  // To hold the number guessed by user
            tries = 0; // To hold number of tries

        number = (int) (Math.random() * 100) + 1; // get random number
between 1 and 100

        System.out.println("Guess My Number Game");
        System.out.println();
        do
        {
            System.out.print("Enter a guess between 1 and 100 : ");
            guess = console.nextInt();
            tries++
        if (guess > number)
        {
```

```
            System.out.println("Too high! Try Again");
        }
        else if (guess < number)
        {
            System.out.println("Too low! Try Again");
        }
        else
        {
             System.out.println("Correct! You got it in " + tries + "
guesses!");
        }
        }while (guess != number);
    }
 }
```

Q10)

```
import java.util.Scanner;
public class P31 {

 public static void main(String[] args) {
   Scanner cs=new Scanner(System.in);
        System.out.println("Enter the row size:");

        int row_size,out,in1,in2;
        int np=1;
        row_size=cs.nextInt();

       for(out=0;out<row_size;out++)
       {
       for(in1=row_size-1;in1>out;in1--)
       {
          System.out.print(" ");
       }
       for(in2=0;in2<np;in2++)
       {
```

```
            System.out.print(np-out);
        }
        np+=2;
        System.out.println();
        }
      cs.close();
} }
```