Java Important answers

UNIT 1

1. **What are the main five Features of Java?**
   Platform Independence: Java is designed to be platform-independent. The compiled Java code (bytecode) can be executed on any platform with a Java Virtual Machine (JVM), making it highly portable.

   Object-Oriented: Java is a pure object-oriented programming language. It promotes the use of objects and classes, making it easier to structure and organize code. This approach encourages code reusability and maintainability.

   Security: Java provides several security features to protect systems from malicious code. It includes a robust security model, sandboxing, and built-in mechanisms to prevent unauthorized access and execution of potentially harmful code.

   Automatic Memory Management (Garbage Collection): Java incorporates automatic memory management through a garbage collector. This feature frees developers from manual memory management, helping to prevent memory leaks and reduce common programming errors.

   Rich Standard Library: Java comes with a rich set of class libraries and APIs that cover a wide range of functionality, from data structures to network communication. This extensive standard library simplifies software development by providing pre-built components and utilities.

2. **Explain briefly about JVM, JDK, JRE?**

   JVM (Java Virtual Machine):
   1.The JVM is an integral part of the Java platform and is responsible for executing Java bytecode.
   2.It provides platform independence by allowing Java programs to run on different operating systems without modification.
   3.The JVM takes care of memory management, garbage collection, and other low-level tasks.
   4.It translates bytecode into machine-specific instructions during runtime, making Java a "write once, run anywhere" language.

5.JVMs are available for various platforms, and different implementations of the JVM exist (e.g., Oracle HotSpot, OpenJ9, GraalVM).

JDK (Java Development Kit):
1.The JDK is a software package that includes the tools necessary for developing, compiling, and debugging Java applications.
2.It contains the Java compiler (javac) to convert Java source code into bytecode.
3.The JDK also includes a set of standard libraries, development utilities, and documentation.
4.Developers use the JDK to write and compile Java programs. It provides all the tools and resources for building Java applications.
JRE (Java Runtime Environment):
1.The JRE is a subset of the JDK and is needed to run Java applications.
2.It includes the JVM, necessary runtime libraries, and core classes that are required to execute Java programs.
3.Unlike the JDK, the JRE does not contain development tools like the Java compiler.
4.-users and systems that need to run Java applications should have the JRE installed.

3. What are the lexical issues in java, list out all the lexical issues?

In Java, lexical issues refer to problems related to the language's lexical structure or rules governing the formation of valid tokens (keywords, identifiers, literals, and operators). Here are some common lexical issues or considerations in Java:

Reserved Keywords: You cannot use Java's reserved keywords as identifiers. For example, int class = 5; is invalid because "class" is a reserved keyword.

Case Sensitivity: Java is case-sensitive, meaning that myVariable and myvariable are treated as distinct identifiers.

Identifier Rules: Identifiers (variable names, class names, etc.) must start with a letter, underscore, or dollar sign, followed by letters, digits, underscores, or dollar signs.

Special Characters: Identifiers cannot contain special characters like @, #, %, etc., except for underscores and dollar signs.

Whitespace: While whitespace (spaces, tabs, line breaks) is used to separate tokens, it has no meaning beyond that. Therefore, it's essential to use proper whitespace for code readability.

Literals: Literals are used to represent constant values in code. Java supports various types of literals, such as integers, floating-point numbers, characters, and strings. Be aware of the appropriate format for each.

4. **Explain about structure of java program in detail?**
   A Java program is a collection of classes and methods that are organized in a structured way to achieve a specific goal. Let's discuss the structure of a Java program in detail:

   Package Declaration (Optional):

   1.A Java program can begin with an optional package declaration, which is used to organize related classes into packages.
   2.It is declared using the package keyword followed by the package name.
   For example: package com.example.myprogram;
   Import Statements (Optional):
   1.Import statements are used to specify which classes or packages from external libraries should be accessible in the program.
   2.They come after the package declaration (if present) and before the class declaration.
   For example: import java.util.ArrayList;
   Class Declaration:
   1.Every Java program must contain at least one class.
   2.The class declaration is defined using the class keyword, followed by the class name.
   3.The class name should match the filename (with a .java extension) where the program is saved.
   For example: public class MyProgram {
   Main Method:
   1.In Java, the main entry point for a program is the main method.
   2.The main method has the following signature: public static void main(String[] args) { }
   3.It serves as the starting point for program execution, and the code within this method is executed.
   For example:
   public static void main(String[] args) {

```
    // Program logic goes here
}
```
Variables and Methods:

1.Within the class, you can define variables and methods. Variables are used to store data, and methods are used to define the behavior of the program.

2.Variables should be declared with a type and an optional access modifier (public, private, etc.).

3.Methods are declared with a return type, method name, and a set of parameters.

For example:
```
private int myVariable;

public void myMethod(int parameter) {
    // Method logic goes here
}
```
Comments:

1.Comments are used to add documentation to your code for understanding and future maintenance.

2.Java supports both single-line comments (//) and multi-line comments (/* ... */).

3.Comments should be used to explain the purpose of classes, methods, and complex code sections.

Braces and Blocks:

1.Java uses curly braces { } to define blocks of code, such as for classes, methods, loops, and conditional statements.

2.Proper indentation and alignment of braces are essential for code readability.

Terminator Semicolons:

Most statements in Java end with a semicolon ;. This is used to terminate statements and separate them.

Here's an example of a simple Java program structure:
```
package com.example.myprogram;

import java.util.ArrayList;

public class MyProgram {
    public static void main(String[] args) {
        // Program logic goes here
    }
    private int myVariable;

    public void myMethod(int parameter) {
        // Method logic goes here
    }
}
```

In this structure, we have a package declaration, import statements, a class declaration, the main method, some variables, and a method. This is a basic template for a Java program, and you can build upon it to create more complex and functional applications

UNIT 2

Q.1 What is variable?

In Java, a variable is a named container that stores data or values that can be manipulated or accessed in a program. Variables are a fundamental concept in programming and are used to store different types of data, including numbers, text, and objects. Variables allow you to work with and manipulate data in your Java programs.

Key characteristics of variables in Java include:

Data Type: Every variable in Java has a data type, which defines the kind of data it can hold. Common data types include int (integer), double (floating-point number), char (character), String (text), and many others.

Name: Variables are given names that follow certain rules, such as starting with a letter, underscore, or dollar sign, followed by letters, digits, underscores, or dollar signs.

Value: Variables store values of the specified data type. You can assign a value to a variable and later change it as needed.

Scope: Variables have a scope, which defines where they can be accessed in the code. Variables can have local scope (limited to a specific method or block) or class scope (accessible throughout the class).

Lifetime: Variables have a lifetime determined by their scope. Local variables typically exist only within the method or block in which they are defined. Class-level variables (also known as instance or static variables) live as long as the object or class they are associated with.

Q.2 Explain about types of variables?

In Java, there are several types of variables that you can use to store and work with data. These variables can be categorized into three main types:

Local Variables:

1.Local variables are declared inside a method, constructor, or block and are only accessible within that scope.
2.They have limited visibility and are typically used to store temporary or method-specific data.
3.Local variables must be initialized before use.
Example:

```
public void calculateSum() {
   int a = 5; // 'a' is a local variable
   int b = 10; // 'b' is a local variable
   int sum = a + b; // 'sum' is a local variable
   System.out.println("Sum is: " + sum);
}
```

Instance Variables (Non-Static Variables):

1.Instance variables are declared within a class but outside any method or constructor.
2.They are associated with an instance of the class (i.e., an object) and have class-level scope.
3.Instance variables are unique to each object of the class, and their values can vary from one object to another.
4.They are initialized with default values (e.g., 0 for numeric types, null for objects) when an object is created.
Example:

```
public class Student {
   String name; // 'name' is an instance variable
   int age;     // 'age' is an instance variable
}
```

Class Variables (Static Variables):

1.Class variables are declared with the static keyword within a class but outside any method or constructor.
2.They are associated with the class itself rather than individual objects and are shared among all instances (objects) of the class.
3.Class variables are initialized with default values when the class is loaded.
4.They can be accessed using the class name and are often used for constants or shared data.
Example:

```
public class MathUtils {
   public static final double PI = 3.14159265359; // 'PI' is a class variable
}
```

In summary, local variables are limited to a specific method or block, instance variables are associated with individual objects and have different values for each object, and class variables are associated with the class itself and are shared among all instances of the class. Choosing the appropriate type of variable depends on the scope and lifetime you need for the data in your Java program.

Q.7 what is the usage of final keyword?
In Java, the final keyword is used to denote that an element (a class, a method, or a variable) is not allowed to be changed or extended once it has been defined. Here's how the final keyword is used in different contexts:

Final Variables:

When applied to a variable, the final keyword indicates that the variable's value cannot be changed once it is initialized. This is often used for constants.

Example:

```
final int MAX_VALUE = 100;
// You can't reassign MAX_VALUE to a different value.
```

Final Methods:
1.When applied to a method, it means that the method cannot be overridden by subclasses.
2.This is often used to ensure that a method's behavior remains consistent in a class hierarchy.

Example:

```
class Parent {
   final void printMessage() {
      System.out.println("This is a final method.");
   }
}

class Child extends Parent {
   // You can't override printMessage() in the Child class.
}
```

Final Classes:
When applied to a class, it means that the class cannot be extended (i.e., you cannot create subclasses of a final class).

Example:
```
final class MyFinalClass {
   // This class cannot be extended.
}
```

Final Arguments:
1.When applied to method parameters, it means that the parameter value cannot be changed within the method.
2.This is useful for ensuring that a method does not modify its arguments.

Example:
```
void printInfo(final String message) {
   // You can't reassign 'message' within this method.
   System.out.println(message);
}
```
The final keyword helps make your code more robust and maintainable by preventing unintended modifications or extensions. It can be especially useful in scenarios where you want to ensure immutability, enforce design constraints, or improve security and predictability in your code.

Q.8 What are the types of data types and write the sizes of all data types?
In Java, data types can be categorized into two main categories: primitive data types and reference data types. Here are the commonly used primitive data types and their respective sizes in Java:

Integral Data Types:

byte: 8 bits, with a range of -128 to 127.
short: 16 bits, with a range of -32,768 to 32,767.
int: 32 bits, with a range of -2^31 to 2^31-1.
long: 64 bits, with a range of -2^63 to 2^63-1.

Floating-Point Data Types:
float: 32 bits, typically used for decimal numbers with single-precision.
double: 64 bits, typically used for decimal numbers with double-precision.
Character Data Type:
char: 16 bits, representing a single Unicode character.
Boolean Data Type:

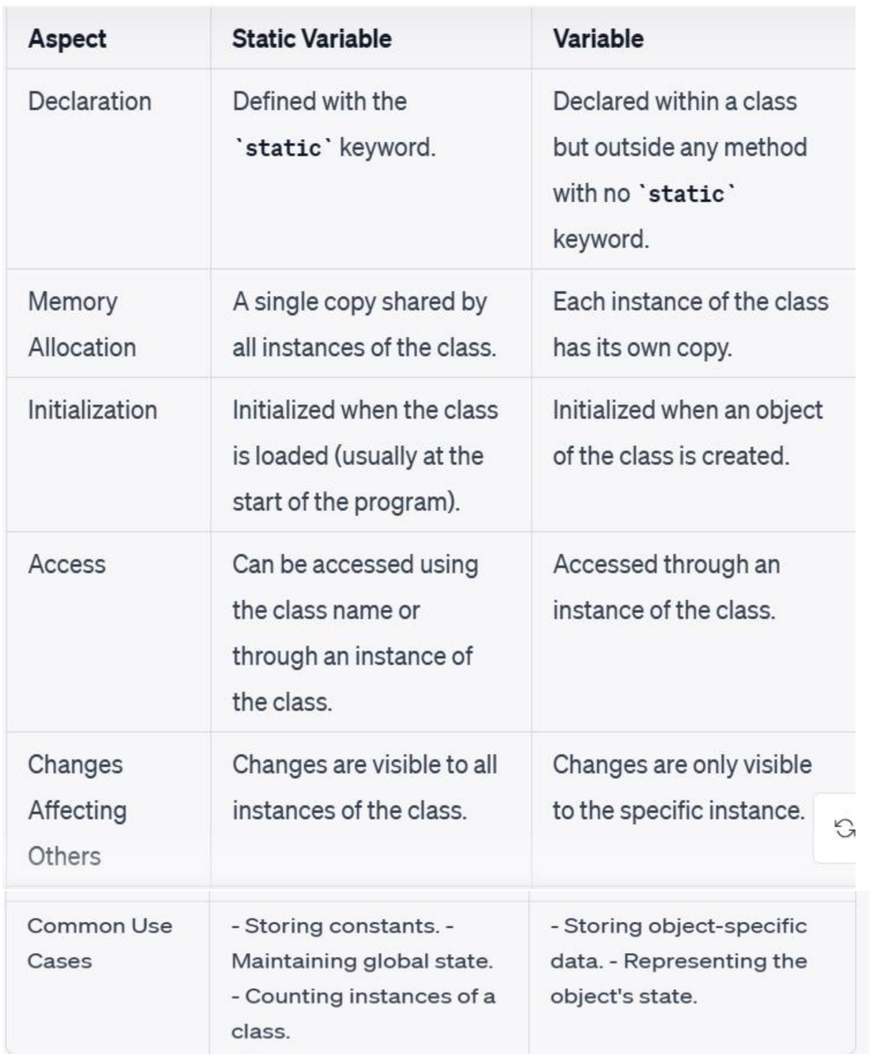
boolean: Represents a true or false value, but its size is not explicitly defined.

It's important to note that the actual size of these data types might vary depending on the platform and the Java Virtual Machine (JVM) being used. These sizes are based on the standard conventions, but they are not guaranteed to be the same on all systems.

In addition to primitive data types, Java also supports reference data types, which are used to store references (memory addresses) to objects. These include classes, interfaces, arrays, and user-defined types. The sizes of reference data types are not fixed and depend on the underlying architecture and the specific JVM implementation.

It's also worth noting that in Java, you can use the '**sizeof'** operator from the **'java.lang.instrument**' package to determine the size of objects at runtime, but this is not the same as the fixed sizes of primitive data types. The sizes of reference data types can vary based on the JVM's memory management and object layout.

Q. 9 Differentiate static and non-static variables in java?



Aspect	Static Variable	Variable
Declaration	Defined with the `static` keyword.	Declared within a class but outside any method with no `static` keyword.
Memory Allocation	A single copy shared by all instances of the class.	Each instance of the class has its own copy.
Initialization	Initialized when the class is loaded (usually at the start of the program).	Initialized when an object of the class is created.
Access	Can be accessed using the class name or through an instance of the class.	Accessed through an instance of the class.
Changes Affecting Others	Changes are visible to all instances of the class.	Changes are only visible to the specific instance.
Common Use Cases	- Storing constants. - Maintaining global state. - Counting instances of a class.	- Storing object-specific data. - Representing the object's state.

Q.10 What are the types of operators in java with one example program?

In Java, operators are symbols or keywords used to perform various operations on data. There are several types of operators, including:

Arithmetic Operators:

+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulo)

Example Program:

```
public class ArithmeticOperatorsExample {
   public static void main(String[] args) {
        int a = 10;
        int b = 3;
        int sum = a + b;
        int difference = a - b;
        int product = a * b;
        int quotient = a / b;
        int remainder = a % b;

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
        System.out.println("Product: " + product);
        System.out.println("Quotient: " + quotient);
        System.out.println("Remainder: " + remainder);
    }
}
```

Comparison Operators (Relational Operators):

== (equal to)
!= (not equal to)
< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)

Example Program:

```
public class ComparisonOperatorsExample {
    public static void main(String[] args) {
        int x = 5;
        int y = 7;

        boolean isEqual = x == y;
        boolean isNotEqual = x != y;
        boolean isLessThan = x < y;
        boolean isGreaterThan = x > y;

        System.out.println("Is equal: " + isEqual);
        System.out.println("Is not equal: " + isNotEqual);
        System.out.println("Is less than: " + isLessThan);
        System.out.println("Is greater than: " + isGreaterThan);
    }
}
```

Logical Operators:
&& (logical AND)
|| (logical OR)
! (logical NOT)

Example Program:
```
public class LogicalOperatorsExample {
   public static void main(String[] args) {
      boolean a = true;
      boolean b = false;

      boolean resultAnd = a && b;
      boolean resultOr = a || b;
      boolean resultNotA = !a;

      System.out.println("a && b: " + resultAnd);
      System.out.println("a || b: " + resultOr);
      System.out.println("!a: " + resultNotA);
   }
}
```
These are just a few examples of the types of operators in Java. There are also bitwise operators, assignment operators, and more, each with its specific purpose and use cases.

or

Q.11 Write all types operators in java and write a program for Bitwise Operators with output?

In Java, there are several types of operators, including arithmetic, comparison, logical, bitwise, assignment, and more. Here's a list of the primary operator types:

Arithmetic Operators:
+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulo)

Comparison Operators (Relational Operators):
== (equal to)
!= (not equal to)
< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)

Logical Operators:
&& (logical AND)
|| (logical OR)
! (logical NOT)

Bitwise Operators:
& (bitwise AND)
| (bitwise OR)
^ (bitwise XOR)
~ (bitwise NOT)
<< (left shift)
>> (right shift)
>>> (unsigned right shift)

Assignment Operators:
= (assignment)
+= (add and assign)
-= (subtract and assign)
*= (multiply and assign)
/= (divide and assign)
%= (modulo and assign)
&= (bitwise AND and assign)
|= (bitwise OR and assign)
^= (bitwise XOR and assign)
<<= (left shift and assign)
>>= (right shift and assign)
>>>= (unsigned right shift and assign)

Now, let's create a Java program that demonstrates the use of bitwise operators:

```
public class BitwiseOperatorsExample {
    public static void main(String[] args) {
        int a = 5;  // Binary: 0101
        int b = 3;  // Binary: 0011

        int bitwiseAnd = a & b;  // Binary AND: 0001 (Decimal 1)
        int bitwiseOr = a | b;   // Binary OR: 0111 (Decimal 7)
        int bitwiseXor = a ^ b;  // Binary XOR: 0110 (Decimal 6)
        int bitwiseNotA = ~a;
        System.out.println("Bitwise AND: " + bitwiseAnd);
        System.out.println("Bitwise OR: " + bitwiseOr);
        System.out.println("Bitwise XOR: " + bitwiseXor);
        System.out.println("Bitwise NOT of 'a': " + bitwiseNotA);
    }
}
```

Output of the program:

```
Bitwise AND: 1
Bitwise OR: 7
Bitwise XOR: 6
Bitwise NOT of 'a': -6
```

In the program, we use bitwise AND, OR, XOR, and NOT operators to manipulate the bits of two integer values (a and b). The results are presented in both binary and decimal forms.

Q.12 Define control statements and explain all the types of control statements?

Control statements in Java are used to control the flow of execution in a program. They determine the order in which statements are executed and help make decisions based on certain conditions. Java provides three main types of control statements:

Selection Statements:

Selection statements allow you to make decisions in your code based on specific conditions. In Java, the primary selection statements are:

1.if Statement:

The if statement is used to execute a block of code if a specified condition is true.

```
if (condition) {
   // Code to execute when the condition is true
}
```

2.if-else Statement:

The if-else statement is used to execute one block of code if a condition is true and another block if the condition is false.

```
if (condition) {
   // Code to execute when the condition is true
} else {
   // Code to execute when the condition is false
}
```

3.switch Statement:

The switch statement allows you to select one of many code blocks to be executed, based on a variable or an expression.

```
switch (variable) {
   case value1:
      // Code for value1
      break;
   case value2:
      // Code for value2
      break;
   default:
      // Code for all other cases
}
```

Iteration Statements (Loops):
Iteration statements are used to execute a block of code repeatedly. Java provides several loop structures, including:

1.for Loop:
The for loop is used to execute a block of code a specific number of times.
```
for (initialization; condition; update) {
   // Code to repeat
}
```
2.while Loop:
The while loop is used to execute a block of code as long as a condition is true.
```
while (condition) {
   // Code to repeat
}
```
3.do-while Loop:
The do-while loop is similar to the while loop but guarantees that the code block is executed at least once, as the condition is checked after the block.
```
do {
   // Code to repeat
} while (condition);
```
Jump Statements:
Jump statements are used to control the flow of execution by transferring control to other parts of the code. Java provides two primary jump statements:

1.break Statement:
The break statement is used to exit a loop or switch statement prematurely.
```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break; // Exit the loop when i is 5
   }
   // Code inside the loop
}
```
2.continue Statement:

The continue statement is used to skip the current iteration of a loop and continue to the next iteration.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // Skip the iteration when i is 5
    }
    // Code inside the loop
}
```

These control statements provide the necessary tools to make decisions, create loops, and manage the flow of your Java programs based on specific conditions and requirements.

Q.13 Explain about only conditional statements with one example program?

Conditional statements in Java are used to make decisions in your code based on specific conditions. These statements allow you to control the flow of your program by executing different blocks of code depending on whether certain conditions are met. In Java, there are primarily two types of conditional statements: the **if** statement and the **switch** statement.

if Statement:

The if statement allows you to execute a block of code if a specified condition is true. If the condition is false, the code block is skipped.

```
if (condition) {
    // Code to execute when the condition is true
} else {
    // Code to execute when the condition is false (optional)
}
```

Example Program:

```
import java.util.Scanner;

public class ConditionalStatementExample {
    public static void main(String[] args) {
        // Create a Scanner object for user input
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        if (number > 0) {
            System.out.println("The number is positive.");
        } else if (number < 0) {
            System.out.println("The number is negative.");
        } else {
            System.out.println("The number is zero.");
        }

        // Close the Scanner
        scanner.close();
    }
}
```

In this program, the if statement is used to check whether a user-input number is positive, negative, or zero. The program takes user input, evaluates the condition, and prints the appropriate message based on the result of the condition.

Q.14 **Explain about only looping statements with one example program?**

Looping statements, also known as iteration statements or loops, are used in Java to repeatedly execute a block of code as long as a specific condition is met. Java provides three primary looping statements**: for, while, and do-while**. These loops are essential for automating repetitive tasks and performing operations iteratively.

Let's look at an example program using the **for** loop:

```
public class LoopingStatementExample {
    public static void main(String[] args) {
        // Example 1: Using a for loop to print numbers from 1 to 10
        System.out.println("Example 1: Using a for loop");
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
        }
        System.out.println(); // Move to the next line

        // Example 2: Using a while loop to calculate the sum of numbers
from 1 to 100
        System.out.println("Example 2: Using a while loop");
        int sum = 0;
        int number = 1;
        while (number <= 100) {
           sum += number;
           number++;
        }
        System.out.println("Sum of numbers from 1 to 100: " + sum);

        // Example 3: Using a do-while loop to print the first 5 natural
numbers
        System.out.println("Example 3: Using a do-while loop");
        int count = 1;
        do {
           System.out.print(count + " ");
           count++;
        } while (count <= 5);
    }
}
```

In this program, we demonstrate the three types of loops:

for Loop (Example 1): We use a for loop to print numbers from 1 to 10. The loop initializes an integer variable i to 1, executes the loop as long as i is less than or equal to 10, and increments i in each iteration.

while Loop (Example 2): We use a while loop to calculate the sum of numbers from 1 to 100. The loop continues until the number is less than or equal to 100, and in each iteration, we add the number to the sum and increment the number.

do-while Loop (Example 3): We use a do-while loop to print the first 5 natural numbers. This loop guarantees that the code block is executed at least once, and then it continues as long as the condition is true.

Each of these loops serves a different purpose and allows you to control the flow of your program by repeating a block of code according to specific conditions.

Q.15 Explain about only jump or transfer statements with one example program?

Jump or transfer statements in Java are used to control the flow of program execution by transferring control to a different part of the code. The primary jump statements in Java are the **break** and **continue** statements.

Here's an example program that demonstrates their usage:

```
public class JumpStatementsExample {
   public static void main(String[] args) {
      // Example 1: Using the break statement to exit a loop
      System.out.println("Example 1: Using the break statement");
      for (int i = 1; i <= 5; i++) {
         if (i == 3) {
            System.out.println("Breaking out of the loop at i = 3");
            break; // Exit the loop
         }
         System.out.println("Inside the loop: i = " + i);
      }

      // Example 2: Using the continue statement to skip an iteration
      System.out.println("\nExample 2: Using the continue statement");
      for (int j = 1; j <= 5; j++) {
         if (j == 3) {
            System.out.println("Skipping iteration at j = 3");
            continue; // Skip this iteration
         }
         System.out.println("Inside the loop: j = " + j);
      }
   }
}
```

In this program, we demonstrate the use of jump statements:

break Statement (Example 1): The break statement is used to exit a loop prematurely. In this example, a for loop counts from 1 to 5. When i becomes 3, we use the break statement to exit the loop immediately. The program prints a message before breaking the loop.

continue Statement (Example 2): The continue statement is used to skip the current iteration of a loop and continue to the next iteration. In this example, a for loop counts from 1 to 5. When j becomes 3, we use the continue statement to skip that iteration. The program prints a message indicating the skipped iteration.

These jump statements are valuable for controlling the flow of your program within loops, allowing you to break out of a loop early or skip specific iterations as needed.

Q.16 Write Program to display the below pattern using Loops statement.

```
 *
 ***
 *****
 *******
```

```
public class StarPattern {
  public static void main(String[] args) {
    int rows = 4; // number of rows
    // outer loop for rows
    for (int i = 1; i <= rows; i++) {
      // inner loop for columns
      for (int j = 1; j <= 2 * i - 1; j++) {
        // print star
        System.out.print("*");
      }
      // move to next line
      System.out.println();
    }
  }
}
```

OUT PUT :

```
*
***
*****
*******
```

Q.17. Write Program to display the below pattern using Loops statement.

```
 *****
 ****
 **
 *
```

```
public class StarPattern {
  public static void main(String[] args) {
    int rows = 5; // number of rows
    // outer loop for rows
    for (int i = rows; i >= 1; i--) {
      // inner loop for columns
      for (int j = 1; j <= i; j++) {
        // print star
        System.out.print("*");
      }
      // move to next line
      System.out.println();
    }
  }
}
```

Out put :

```
*****
****
***
**
*
```

Q.18. Write Program to display the below pattern using Loops statement.

```
 *
 ***
 *****
```

```
public class StarPattern {
  public static void main(String[] args) {
    int rows = 3; // number of rows
    // outer loop for rows
    for (int i = 1; i <= rows; i++) {
      // inner loop for columns
      for (int j = 1; j <= 2 * i - 1; j++) {
        // print star
        System.out.print("*");
      }
      // move to next line
      System.out.println();
    }
  }
}
```

Out put :

```
*
***
*****
```

UNIT 4

Q.19 What is array write the syntax for array?

In Java, an array is a data structure that stores a fixed-size sequence of elements of the same data type. Arrays allow you to store and manipulate multiple values of the same type under a single name. Here is the syntax for declaring and using arrays in Java:

Array Declaration:

data_type[] array_name; // This declares an array variable

Array Initialization:

You can initialize an array in several ways:

1.Using an array initializer: Initialize the array when you declare it.

data_type[] array_name = {value1, value2, value3, ...};

Using the new keyword: Create an array object and then initialize its elements.

data_type[] array_name = new data_type[size];

Accessing Array Elements:

You can access individual elements of an array using square brackets ' **[]**' with the index (position) of the element you want to access. Arrays are zero-based, meaning the first element has an index of 0, the second element has an index of 1, and so on.

element = array_name[index];

Array Length:

You can obtain the length (number of elements) of an array using the **'length'** property:

int length = array_name.length;

Example: Here is an example of how to declare, initialize, and access elements in an array in Java:

```
public class ArrayExample {
   public static void main(String[] args) {
        // Declare and initialize an integer array
        int[] numbers = {1, 2, 3, 4, 5};

        // Access and print elements from the array
        System.out.println("Element at index 0: " + numbers[0]); // Prints 1
        System.out.println("Element at index 2: " + numbers[2]); // Prints 3

        // Find the length of the array
        int length = numbers.length;
        System.out.println("Array length: " + length); // Prints 5
    }
}
```

In this example, we declare and initialize an integer array named numbers, access and print elements at specific indices, and find the length of the array. The square brackets [] are used for element access, and the length property is used to determine the array's size.

Q.19 What is array and explain the types of arrays with syntax and examples?

In Java, an array is a data structure that stores a fixed-size sequence of elements of the same data type. Arrays are used to store and manipulate multiple values of the same type under a single name. They are an essential part of programming and are commonly used to manage collections of data.

Here are the main types of arrays in Java:

Single-Dimensional Array:

1.A single-dimensional array, also known as a one-dimensional array, is the simplest type of array.

2.It is used to store elements in a linear, sequential order.

Syntax for declaration and initialization:

```
data_type[] array_name = new data_type[size];
```

Example:

int[] numbers = new int[5]; // Declares an integer array of size 5

numbers[0] = 1; // Assigns values to array elements

numbers[1] = 2;

// ...

Multi-Dimensional Array:

1.Multi-dimensional arrays are used to store data in multiple dimensions, such as a table or a matrix.

2.They can be two-dimensional (2D), three-dimensional (3D), and so on.

Syntax for declaration and initialization:

data_type[][] array_name = new data_type[row_size][column_size];

Example:

int[][] matrix = new int[3][3]; // Declares a 2D integer array of size 3x3

matrix[0][0] = 1; // Assigns values to specific elements

matrix[1][2] = 3;

Dynamic Array (ArrayList):

A dynamic array in Java is implemented using the ArrayList class from the java.util package.

It is similar to a one-dimensional array but can dynamically resize itself as needed.

Syntax for creating an '**ArrayList':**

ArrayList<data_type> array_name = new ArrayList<data_type>();

Example:

import java.util.ArrayList;

ArrayList<String> names = new ArrayList<String>(); // Creates a dynamic array of strings

names.add("Alice"); // Adds elements to the ArrayList

names.add("Bob");

Jagged Array:

A jagged array is an array of arrays, where each element of the main array is an array.

The sub-arrays can have different sizes.

Syntax for declaration and initialization:

```
data_type[][] array_name = new data_type[][] {
    {row1_element1, row1_element2, ...},
    {row2_element1, row2_element2, ...},
    // ...
};
```

Example:

```
int[][] jaggedArray = new int[][] {
    {1, 2, 3},
    {4, 5},
    {6, 7, 8, 9}
};
```

These are the main types of arrays in Java. Depending on your requirements, you can choose the appropriate type of array for storing and manipulating data. Single-dimensional and multi-dimensional arrays are useful for fixed-size collections, while dynamic arrays (ArrayLists) are preferred for situations where you need a variable-size collection. Jagged arrays are useful when you have varying sizes of sub-arrays.

Q 20 . Create three arrays A,B and RES with 3x3 matrix. Input elements into A and B arrays, Store multiplication of A and B arrays into RES array and print all the arrays?

```
public class MatrixMultiplication {
    public static void main(String[] args) {
        // Create 3x3 arrays A, B, and RES
        int[][] A = new int[3][3];
        int[][] B = new int[3][3];
```

```
int[][] RES = new int[3][3];

// Input elements into arrays A and B
// You can modify these values as needed
int[][] inputA = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
int[][] inputB = {
    {9, 8, 7},
    {6, 5, 4},
    {3, 2, 1}
};

// Copy input values into arrays A and B
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        A[i][j] = inputA[i][j];
        B[i][j] = inputB[i][j];
    }
}

// Perform matrix multiplication and store the result in RES
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
                RES[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    // Print the arrays A, B, and RES
    System.out.println("Array A:");
    printArray(A);
    System.out.println("\nArray B:");
    printArray(B);
    System.out.println("\nArray RES (A * B):");
    printArray(RES);
  }

  // Helper method to print a 2D array
  public static void printArray(int[][] array) {
    for (int i = 0; i < array.length; i++) {
      for (int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

Q 21 Write program for adding two matrixes with 3x3 arrays?

```
public class MatrixAddition {
  public static void main(String[] args) {
    // Create 3x3 matrices A, B, and RES
    int[][] A = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    int[][] B = {
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
    };
    int[][] RES = new int[3][3];
    // Perform matrix addition and store the result in RES
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            RES[i][j] = A[i][j] + B[i][j];
        }
    }
    // Print the original matrices A and B
    System.out.println("Matrix A:");
    printMatrix(A);
    System.out.println("\nMatrix B:");
    printMatrix(B);
    // Print the result matrix RES
    System.out.println("\nMatrix RES (A + B):");
    printMatrix(RES);
}
```

```
    // Helper method to print a 3x3 matrix
    public static void printMatrix(int[][] matrix) {
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

UNIT 5

Q 22 What is object and write the syntax for object?

In Java, an object is an instance of a class. An object is a fundamental unit of object-oriented programming (OOP) and represents a real-world entity with attributes and behaviors. Objects are created from classes, which serve as blueprints or templates for defining the structure and behavior of objects.

Here's the syntax for creating an object in Java:

ClassName objectName = new ClassName();

ClassName is the name of the class from which you want to create an object.

objectName is a user-defined name you give to the object (can be any valid identifier).

new is a keyword used to allocate memory and initialize the object.

ClassName() is the constructor that is called to create and initialize the object. If you don't provide a constructor explicitly, a default constructor is used.

Example:

Let's say you have a class called Car, and you want to create an object of the Car class. Here's how you would do it:

Car myCar = new Car();

In this example**, Car** is the class, and **myCar** is the object. The **new** keyword allocates memory for the object, and the constructor **Car()** initializes it. You can then access the attributes and methods of th**e myCar** object using the dot notation, such as **myCar.setColor("Red")** to set the color of the car.

Q 23 What is abstraction in java?

In **Java**, **abstraction** refers to the practice of **hiding implementation details** and providing a **simplified view** of a system to its users. It allows us to focus on essential features while ignoring the underlying complexity. Abstraction is achieved through **abstract classes** and **interfaces**. Here’s how it works:

1. **Abstract Class**: An abstract class is declared using the abstract keyword. It can have both abstract methods (without implementation) and concrete methods (with implementation). Abstract classes serve as blueprints for other classes.
2. **Abstract Method**: An abstract method is declared without implementation in an abstract class. Subclasses that extend the abstract class must provide concrete implementations for these methods.
3. **Interfaces**: Interfaces define a contract that classes must adhere to. They contain only abstract method declarations (no implementation). A class can implement multiple interfaces.

Example: Consider a "shape" abstraction. We define an abstract class or interface with common behaviors (like calculating area) for various shapes (e.g., circle, rectangle). Concrete shape classes then provide specific implementations for these behaviors.

Q 24. Which Keyword is used to declare a constant?

A constant is a variable that cannot be changed after it is initialized. To declare a constant in Java, you can use the final keyword before the data type and the variable name, such as:

```
final String name = "John";
```

This means that the variable name will always have the value "John" and cannot be assigned a different value later. The final keyword makes the variable immutable or unchangeable[1].

You can also use the static keyword along with the final keyword to create a class constant, which is a constant that belongs to the class and not to any object of the class. For example:

```
static final double PI = 3.14;
```

Q 25. What is inheritance?

Inheritance is a concept of object-oriented programming that allows a class to inherit the features of another class. This helps to avoid code duplication and enhance reusability[1].

In Java, inheritance is achieved by using the extends keyword. For example, if you have a class called Animal that has some common attributes and methods for all animals, you can create a subclass called Dog that extends the Animal class and inherits its features. You can also add new attributes and methods to the Dog class that are specific to dogs.

```
class Animal {
  // common attributes and methods for all animals
}

class Dog extends Animal {
  // additional attributes and methods for dogs
}
```

In this example, the Dog class is a subclass or a child class of the Animal class, which is a superclass or a parent class. The relationship between the two classes is Dog IS-A Animal. This means that a dog is a type of animal and can access the fields and methods of the animal class.

Q26. Define Class, Types of Methods, and Types of Constructors with Example?

- A class is a blueprint or a template for creating objects. A class defines the attributes and behaviors of the objects of that type. For example, you can create a class called Student that has fields like name, age, grade, and methods like study, takeExam, displayInfo, etc. To create a class in Java, you can use the following syntax:

```
class ClassName {
  // declare fields
  // declare methods
}
```

- A method is a block of code that performs a specific task. A method can have parameters and return a value. A method can be invoked or called by using its name and passing arguments if required. For example, you can create a method called add that takes two integers as parameters and returns their sum. To create a method in Java, you can use the following syntax:

```
returnType methodName(parameterList) {
  // method body
  return value;
}
```

- A constructor is a special type of method that is used to initialize the objects of a class. A constructor has the same name as the class and does not have any return type. A constructor is called automatically when an object of the class is created using the new keyword. For example, you can create a constructor for the Student class that takes the name, age, and grade as parameters and assigns them to the fields of the object. To create a constructor in Java, you can use the following syntax:

```
ClassName(parameterList) {
  // constructor body
}
```

Q27. Write and Explain with example about Encapsulation, Inheritance, Abstraction and Polymorphism?

OOP stands for Object-Oriented Programming, which is a programming paradigm that uses objects and classes to represent and manipulate data. Objects are instances of classes, which define the properties (attributes) and behaviors (methods) of the objects. Classes can also inherit from other classes, which means they can share some or all of their attributes and methods with their parent classes. This allows for code reuse and modularity.

There are four main OOP concepts in Java: encapsulation, inheritance, abstraction, and polymorphism. Let me explain each one with an example.

- **Encapsulation**: This is the practice of hiding the internal details of an object from the outside world, and providing public methods to access and modify the object’s state. This way, the object’s data is protected from unauthorized or accidental changes, and the object’s behavior is consistent and predictable. For example, suppose we have a class called BankAccount that represents a bank account with a balance and an interest rate. We can use encapsulation to make the balance and interest rate private fields, and provide public methods to deposit, withdraw, and calculate interest on the account. This way, we can ensure that the balance is always positive, the interest rate is always valid, and the interest is calculated correctly.

```
public class BankAccount {
  // private fields
  private double balance;
  private double interestRate;

  // public constructor
  public BankAccount(double balance, double interestRate) {
    this.balance = balance;
    this.interestRate = interestRate;
  }

  // public methods
  public double getBalance() {
    return balance;
  }

  public void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
    }
  }

  public boolean withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
      balance -= amount;
      return true;
    } else {
      return false;
    }
  }

  public double calculateInterest(int years) {
    return balance * Math.pow(1 + interestRate / 100, years) - balance;
  }
}
```

- **Inheritance**: This is the mechanism of creating new classes from existing ones, by extending their attributes and methods. The new class is called a subclass or a child class, and the existing class is called a superclass or a parent class. The subclass inherits all the attributes and methods of the superclass, and can also add new ones or override existing ones. This allows for code reuse and specialization. For example, suppose we have a class called Animal that represents a generic animal with a name and an age. We can use inheritance to create subclasses of Animal, such as Dog and Cat, that inherit the name and age attributes from Animal, and also add new attributes and methods specific to each animal type.

```
public class Animal {
  // protected fields
  protected String name;
  protected int age;

  // public constructor
  public Animal(String name, int age) {
    this.name = name;
    this.age = age;
  }

  // public methods
  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public void makeSound() {
    System.out.println("Animal sound");
  }
}

public class Dog extends Animal {
  // private field
  private String breed;

  // public constructor
  public Dog(String name, int age, String breed) {
    super(name, age); // call the superclass constructor
    this.breed = breed;
  }

  // public method
  public String getBreed() {
    return breed;
  }

  // override the superclass method
  @Override
  public void makeSound() {
    System.out.println("Woof woof");
  }
}

public class Cat extends Animal {
  // private field
  private String color;

  // public constructor
  public Cat(String name, int age, String color) {
    super(name, age); // call the superclass constructor
    this.color = color;
  }

  // public method
  public String getColor() {
    return color;
  }

  // override the superclass method
  @Override
  public void makeSound() {
    System.out.println("Meow meow");
  }
}
```

- **Abstraction**: This is the process of hiding the implementation details of an object or a class from the user, and providing only the essential features or functionality. Abstraction helps to reduce complexity and increase readability of the code. One way to achieve abstraction in Java is by using abstract classes and interfaces. An abstract class is a class that cannot be instantiated, but can have abstract or concrete methods. An abstract method is a method that has no body, and must be implemented by a subclass. An interface is a collection of abstract methods that a class can implement to follow a certain behavior or contract. For example, suppose we have an abstract class called Shape that represents a generic shape with an area and a perimeter. We can use abstraction to define the abstract methods for calculating the area and the perimeter, and leave the implementation details to the subclasses that extend Shape, such as Circle and Rectangle. We can also use an interface called Drawable that defines an abstract method for drawing a shape on a canvas.

```
public abstract class Shape {
  // abstract methods
  public abstract double getArea();
  public abstract double getPerimeter();
}

public interface Drawable {
  // abstract method
  public void draw(Canvas canvas);
}

public class Circle extends Shape implements Drawable {
  // private field
  private double radius;

  // public constructor
  public Circle(double radius) {
    this.radius = radius;
  }

  // implement the abstract methods of the superclass and the interface
  @Override
  public double getArea() {
    return Math.PI * radius * radius;
  }

  @Override
  public double getPerimeter() {
    return 2 * Math.PI * radius;
  }

  @Override
  public void draw(Canvas canvas) {
    // draw a circle on the canvas using the radius
  }
}

public class Rectangle extends Shape implements Drawable {
  // private fields
  private double length;
  private double width;

  // public constructor
  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }

  // implement the abstract methods of the superclass and the interface
  @Override
  public double getArea() {
    return length * width;
  }

  @Override
  public double getPerimeter() {
    return 2 * (length + width);
  }

  @Override
  public void draw(Canvas canvas) {
    // draw a rectangle on the canvas using the length and width
  }
}
```

- **Polymorphism**: This is the ability of an object or a method to take different forms depending on the context. Polymorphism allows for flexibility and dynamic behavior of the code. There are two types of polymorphism in Java: compile-time polymorphism and run-time polymorphism. Compile-time polymorphism is achieved by method overloading, which means defining multiple methods with the same name but different parameters. Run-time polymorphism is achieved by method overriding, which means redefining a method in a subclass that was already defined in a superclass. For example, suppose we have a class called Calculator that has two overloaded methods for adding two numbers: one for integers and one for doubles. We can use compile-time polymorphism to call the appropriate method based on the type of the arguments. We can also use run-time polymorphism to override the add method in a subclass called ScientificCalculator that can add complex numbers.

```
public class Calculator {
  // compile-time polymorphism: method overloading
  public int add(int x, int y) {
    return x + y;
  }
  public double add(double x, double y) {
    return x + y;
  }
}

public class ScientificCalculator extends Calculator {
  // run-time polymorphism: method overriding
  @Override
  public ComplexNumber add(ComplexNumber x, ComplexNumber y) {
    return x.add(y);
  }
}
```

Q28. What are types of object creation in java?

Object creation is the process of allocating memory for an object and initializing its state. In Java, there are six different ways to create objects, each with its own advantages and disadvantages. Let me explain each one briefly.

- ***Using new keyword:*** This is the most common and basic way to create an object in Java. It invokes the constructor of the class and returns a reference to the newly created object. For example:

Student s1 = new Student ("John", 20); // creates a Student object with name John and age 20

- ***Using newInstance () method of Class class:*** This is a reflection-based method that creates an object of a class whose name is not known at compile time, but only at run time. It invokes the default (no-argument) constructor of the class and returns an object of type Object, which needs to be casted to the desired type. For example:

Class c = Class.forName ("Student"); // loads the Student class dynamically
Student s2 = (Student) c.newInstance (); // creates a Student object using the default constructor

- ***Using newInstance () method of Constructor class:*** This is another reflection-based method that creates an object of a class whose name and constructor parameters are not known at compile time, but only at run time. It takes an array of objects as arguments and invokes the matching constructor of the class and returns an object of type Object, which needs to be casted to the desired type. For example:

```
Class c = Class.forName ("Student"); // loads the Student class dynamically
Constructor cons = c.getConstructor (String.class, int.class); // gets the constructor with String and int parameters
Student s3 = (Student) cons.newInstance ("Mary", 19); // creates a Student object with name Mary and age 19
```

- ***Using clone () method:*** This is a method of Object class that creates a copy of an existing object and returns it. It does not invoke any constructor, but copies all the fields of the original object to the new one. The class must implement the Cloneable interface and override the clone () method to use this method. For example:

```
Student s4 = new Student ("Bob", 18); // creates a Student object with name Bob and age 18
Student s5 = (Student) s4.clone (); // creates a copy of s4 using clone () method
```

- ***Using deserialization:*** This is a process of converting a byte stream into an object. It is usually used to read an object from a file or a network. It does not invoke any constructor, but restores the state of the object as it was when it was serialized. The class must implement the Serializable interface to use this method. For example:

```
ObjectInputStream ois = new ObjectInputStream (new FileInputStream ("student.ser")); // creates an input stream from a file
Student s6 = (Student) ois.readObject (); // reads an object from the stream using deserialization
```

- ***Using factory method:*** This is a design pattern that provides a static method to create an object of a class without exposing its constructor. It can be used to create objects of different subclasses based on some logic or parameter. For example:

```
public abstract class Shape {
  public abstract void draw ();
  public static Shape getShape (String type) {
    if (type.equals ("circle")) {
      return new Circle ();
    } else if (type.equals ("rectangle")) {
      return new Rectangle ();
    } else {
      return null;
    }
  }
}
```

Shape s7 = Shape.getShape ("circle"); // creates a Circle object using factory method

Q29. What is the difference between super keyword and super method?

The super keyword in Java is a reference variable that is used to refer to the immediate parent class object of a subclass. The super keyword can be used in different ways, such as:

- To access the parent class's instance variables, if they are hidden by the subclass's variables with the same name.
- To invoke the parent class's methods, if they are overridden by the subclass's methods with the same name.
- To call the parent class's constructor from the subclass's constructor, using the super () syntax.

The super method in Java is not a separate concept, but rather a term used to refer to any method of the parent class that is invoked using the super keyword. For example, if we have a class Animal and a subclass Dog, and both classes have a method called eat (), then we can call the Animal's eat () method from the Dog class using super.eat (). This is an example of a super method.

30. What is polymorphism? And explain about method overloading with example?

Polymorphism is the ability of an object or a method to take different forms depending on the context. Polymorphism allows for flexibility and dynamic behavior of the code. There are two types of polymorphism in Java: compile-time polymorphism and run-time polymorphism[1].

Compile-time polymorphism is achieved by method overloading, which means defining multiple methods with the same name but different parameters within the same class. The compiler will decide which method to call based on the number and types of arguments passed to the method[2].

For example, suppose we have a class called Calculator that has two overloaded methods for adding two numbers: one for integers and one for doubles. We can use compile-time polymorphism to call the appropriate method based on the type of the arguments. Here is the code:

```
public class Calculator {
  // compile-time polymorphism: method overloading
  public int add(int x, int y) {
    return x + y;
  }

  public double add(double x, double y) {
    return x + y;
  }
}

public class Test {
  public static void main(String[] args) {
    Calculator c = new Calculator();
    System.out.println(c.add(2, 4)); // calls the int add method
    System.out.println(c.add(5.5, 6.3)); // calls the double add method
  }
}
```

The output of this code is:

```
6
11.8
```

Unit 6

31. What is package?

Package:

A package can be defined as a group of similar types of classes, interface, enumeration or sub-package.

Using package, it becomes easier to locate the related classes and it also provides a good structure for projects with hundreds of classes and other files.

A java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, .net, io, util, sql etc.

Additional points about package:

A package is always defined as a separate folder having the same name as the package name.

Store all the classes in that package folder.

All classes of the package which we wish to access outside the package must be declared public.

All classes within the package must have the package statement as its first line.

All classes of the package must be compiled before use (So that they

are error free)

The packages are classified into two types.

Built-in Packages

User-defined packages

32. What is the usage of this keyword?

this keyword in Java:

- In Java, this is a keyword which is used to refer current object of a class. We can it to refer any member of the class. It means we can access any instance variable and method by using this keyword.
- The main purpose of using this keyword is to solve the confusion when we have same variable name for instance and local variables.
- We can use this keyword for the following purpose.
- this keyword is used to refer to current object.
- this is always a reference to the object on which method was invoked.
- this can be used to invoke current class constructor.
- this can be passed as an argument to another method.

33. Explain any 5 in-built packages in java?

1.Built-in Packages: (Pre-defined packages)

These packages consist of a large number of classes which

are a part of Java API. Some of the commonly used built-in packages are:

1) java.lang:

Contains language support classes (e.g classed which defines

primitive data types, math operations). This package is automatically imported

by default.

2)java.io:

Contains classed for supporting input / output operations. It will

perform Read/Write operations.

3)java.util:

Contains utility classes which implement data structures like

Linked List, Dictionary and support; for Date / Time operations.

Ex: java.util.*;

ex: java.util.scanner class

4)java.applet:

Contains classes for creating Applets.

5)java.awt:

Contain classes for implementing the components for graphical

user interfaces (GUI) like button, menus etc..

6)java.net:

Contain classes for supporting networking operations. Like Client

Server programs and networking operation in java.

34. What is inner class and purpose of inner class?

An inner class is a special type of class that is defined within another class[12].
Inner classes are used for various purposes, such as:

- To logically group classes that are only used in one place[1].
- To increase encapsulation and hide the implementation details of the outer class[2].
- To access the members of the outer class, including private ones[1].
- To create an instance of an object that is associated with the outer class[3].

For example, here is a simple inner class in Java:

```
class OuterClass {
  private int x = 10; // a private field of the outer class

  class InnerClass {
    public void display() {
      System.out.println("x = " + x); // access the private field of the outer class
    }
  }
}
```

The inner class InnerClass can access the private field x of the outer class OuterClass. The inner class is only used by the outer class, so it makes sense to define it inside the outer class. This way, the inner class is hidden from the rest of the world and increases the encapsulation of the outer class.

There are different types of inner classes in Java, such as member inner classes, local inner classes, anonymous inner classes, and static nested classes. Each one has a somewhat different purpose and syntax.

35. Write one line about this keyword?

The this keyword in Java is a reference variable that refers to the current object of the class[12]. It can be used to access the members of the same class, such as fields, methods, and constructors[23].

36. What is the difference between interface and class?

Difference between Class and Interface:

Class

1.In class, you can instantiate variable

and create an object.

2.Class can contain concrete (with implementation) methods

3.The access specifiers used with classes are private, protected and public.

4.Constructors can be included in a class

5.A method body can exist in a class.

6.Classes do not support multiple inheritance.

Interface

1.In an interface, you can’t instantiate variable and create an object.

2.The interface cannot contain concrete (with implementation) methods

3.In Interface only one specifier is used- Public.

4.but not an inheritance.

5.but it cannot exist in an interface.

6.but it is supported by inheritance.

UNIT 7

37. Define string?

A string is a sequence of characters that can represent text, symbols, or numbers. In Java, a string is an object of the String class, which is immutable and cannot be changed once created. Strings can be created using double quotes, such as String s = "Hello";, or using the new keyword, such as String s = new String("Hello");. Strings have many methods to perform operations on them, such as length(), charAt(), concat(), equals(), substring(), and more. Strings are stored in a special memory area called the "string constant pool", which helps to save memory and improve performance

38. Define string and explain about five string predefined methods?

A string is a sequence of characters that can represent text, symbols, or numbers. In Java, a string is an object of the String class, which is immutable and cannot be changed once created. Strings can be created using double quotes, such as String s = "Hello";, or using the new keyword, such as String s = new String("Hello");. Strings have many methods to perform operations on them, such as length(), charAt(), concat(), equals(), substring(), and more. Strings are stored in a special memory area called the "string constant pool", which helps to save memory and improve performance[12].

Here are five examples of string predefined methods in Java:

- length() - This method returns the number of characters in the string. For example, "Hello".length() returns 5.
- charAt(int index) - This method returns the character at the specified index in the string. The index starts from 0. For example, "Hello".charAt(1) returns 'e'.
- concat(String str) - This method returns a new string that is the concatenation of this string and the argument string. For example, "Hello".concat(" World") returns "Hello World".
- equals(Object obj) - This method compares this string with the specified object for equality. It returns true if the object is also a string and has the

same sequence of characters as this string. For example, "Hello".equals("hello") returns false.

- substring(int beginIndex, int endIndex) - This method returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. For example, "Hello".substring(1, 4) returns “ell”.

39. Write a program for given input string is a palindrome string are not without using string reverse method?

```
// A method that checks if a string is a palindrome or not

public static boolean isPalindrome(String s) {

  // Convert the string to lowercase

  s = s.toLowerCase();

  // Initialize i and j

  int i = 0;

  int j = s.length() - 1;

  // Loop while i < j

  while (i < j) {

    // Compare the characters at i and j

    if (s.charAt(i) != s.charAt(j)) {

      // If they are not equal, return false

      return false;

    }
    // Increment i and decrement j
    i++;
    j--;
  }
  // If the loop ends without returning false, return true
  return true;
}

// A main method that tests some examples
public static void main(String[] args) {
  // Some example strings
  String[] examples = {"madam", "racecar", "hello", "level", "Bing"};
  // Loop through the examples
  for (String example : examples) {
    // Check if the example is a palindrome or not
    boolean result = isPalindrome(example);
    // Print the result
    System.out.println(example + " is a palindrome: " + result);
  }
}
```

The output of this program is:

```
madam is a palindrome: true
racecar is a palindrome: true
hello is a palindrome: false
level is a palindrome: true
Bing is a palindrome: false
```

40. What is the main difference between String class and StringBuffer Class?

String Vs StringBuffer Vs StringBuilder In Java

- String objects are immutable whereas StringBuffer and StringBuilder objects are mutable.
- StringBuffer and StringBuilder are same only difference is that StringBuilder is not synchronized whereas StringBuffer is. As StringBuilder is not synchronized, it is not thread safe but faster than StringBuffer.
- Use String, for a string which will be constant through out the application.
- Use StringBuffer, for a string which can change in multi-threaded applications.
- Use StringBuilder, for a string which can change in single-threaded applications.

UNIT 8

41. What is exception and list three exception types names?

Exception Handling:
An exception is an unexpected event that occurs during program execution.
It affects the flow of the program instructions which can cause the program to terminate abnormally.
An exception can occur for many reasons. Some of them are:
Major reasons why an exception Occurs:

- ✓ Invalid user input
- ✓ Device failure
- ✓ Loss of network connection
- ✓ Physical limitations (out of disk memory)
- ✓ Code errors
- ✓ Opening an unavailable file

Exceptions can be categorized in two ways:
1) Built-in Exceptions

- Checked Exception
- Unchecked Exception

1.Checked Exceptions:

- Checked exceptions are called compile-time exceptions
- Because these exceptions are checked at compile-time by the compiler.
- Checked exception is also called IO exceptions.

2.Unchecked Exceptions:

- The unchecked exceptions are just opposite to the checked exceptions.
- The compiler will not check these exceptions at compile time.
- In simple words, if a program throws an unchecked exception, and even
- if we didn’t handle or declare it,
- the program would not give a compilation error.

2).User-Defined Exceptions

- Built-in Exceptions:
- Built-in exceptions are the exceptions that are available in Java libraries.
- These exceptions are suitable to explain certain error situations.
- The built-in-exceptions are classified into two types Checked exception
- and unchecked exception.

42. What is an exception and types. Write a java program to implement any three exceptions in java

1.Checked Exceptions:

```
import java.io.*;
class SCE
{
public static void main(String args[])
{
FileInputStream CSE = new
FileInputStream("/Desktop/CSE.txt");

}
}
```

Output:

```
javac /tmp/ekGroR0BeB/CSE.java
/tmp/ekGroR0BeB/CSE.java:16: error: unreported exception
FileNotFoundException; must be caught or declared to be
thrown
= new FileInputStream("/Desktop/GFG.txt");
```

2.Unchecked Exceptions:

Example:

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo
{
public static void main(String args[])
{
File file = new File("E://file.txt");
FileReader fr = new FileReader(file);
}
}
```

Output:

```
javac /tmp/ekGroR0BeB/FilenotFound_Demo.java
/tmp/ekGroR0BeB/FilenotFound_Demo.java:8: error:
unreported exception
FileNotFoundException; must be caught or declared to be
thrown
FileReader fr = new FileReader(file);
^
1 error
```

43. What are the exception handling keywords and explain one by one with one example program?

Java Exception Keywords:
Java provides five keywords that are used to handle the exception.

Keyword Description

- try

  The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

- catch

  The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

- finally

  The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

- throw

  The "throw" keyword is used to throw an exception.

- Throws

  The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

EXAMPLE

```
class Main
{
public static void main(String[] args)
{
try
{
// code that generates exception
int divideByZero = 5 / 0;
}
catch (ArithmeticException e)
{
System.out.println("ArithmeticException => " +
e.getMessage());
}
finally
{
System.out.println("Finally Block Completed
Successfully");
}
}
}
```

Output:

```
ArithmeticException => / by zero
This is the finally block
```

UNIT 9

44. What is Multithreading?

Multithreading in Java:

- Multithreading in java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.Multiprocessing and multithreading, both are used to achieve multitasking.
- we use multithreading than multiprocessing because threads use a shared memory area.

- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

45. Define a thread, multi-threading. Explain life cycle of a thread. Write a java program using user defined threads?

THREAD:

A thread in Java is the direction or path that is taken while a program is being executed.Generally, all the programs have at least one thread, known as the main thread,that is provided by the JVM or Java Virtual Machine at the starting of theprogram's execution.

MULTITHREADING :

Multithreading in java is a process of executing multiple threadssimultaneously. A thread is a lightweight sub-process, the smallest unit of processing.Multiprocessing and multithreading, both are used to achieve multitasking. we use multithreading than multiprocessing because threads use a shared memory area.They don't allocate separate memory area so saves memory, and context- switching between the threads takes less time than process.

Life cycle of a Thread in Java:(Imp)

- The Life Cycle of a Thread in Java refers to the state transformations of a thread that begins with its birth and ends with its death.
- When a thread instance is generated and executed by calling the start() method of the Thread class,
- the thread enters the runnable state.
- When the sleep() or wait() methods of the Thread class are called, the thread enters a non-runnable mode.
- Thread returns from non-runnable state to runnable state and starts statement execution.
- The thread dies when it exits the run() process. In Java, these thread state transformations are referred to as the Thread life cycle.

There are basically 5 stages in the lifecycle of a thread, as given below:

1. New

2. Runnable

3. Running

4. Blocked (Non-runnable state)

5. Dead

1.New State: Born State:

- As we use the Thread class to construct a thread entity, the thread is born and is defined as being in the New state.
- That is, when a thread is created, it enters a new state, but the start() method on the instance has not yet been invoked.

2.Runnable State:

- A thread in the runnable state is prepared to execute the code.
- When a new thread's start() function is called, it enters a runnable state.
- In the runnable environment, the thread is ready for execution and is awaiting the processor's availability (CPU time).
- That is, the thread has entered the queue (line) of threads waiting for execution.

3.Running State:

- Running implies that the processor (CPU) has assigned a time slot to the thread for execution.
- When a thread from the runnable state is chosen for execution by the thread scheduler, it joins the running state.
- In the running state, the processor allots time to the thread for execution and runs its run procedure.
- This is the state in which the thread directly executes its operations.
- Only from the runnable state will a thread enter the running state.

4.Blocked State:

- When the thread is alive, i.e., the thread class object persists,
- but it cannot be selected for execution by the scheduler.
- It is now inactive.

5.Dead State:

- When a thread's run() function ends the execution of sentences,
- it automatically dies or enters the dead state.
- That is, when a thread exits the run() process, it is terminated or killed.
- When the stop() function is invoked, a thread will also go dead.

EXAMPLE

```
// A class that extends Thread class
class MyThread extends Thread {
   // Constructor
   public MyThread(String name) {
      super(name); // Set the name of the thread
   }

   // Override the run() method
   public void run() {
      System.out.println("Hello, I am " + this.getName());
   }
}

// A class that implements Runnable interface
class MyRunnable implements Runnable {
   // A reference to a Thread object
   private Thread t;
   // A name for the thread
   private String name;

   // Constructor
   public MyRunnable(String name) {
      this.name = name; // Set the name of the thread
   }

   // Override the run() method
   public void run() {
      System.out.println("Hello, I am " + name);
   }
```

```
    // A method to start the thread
    public void start() {
        if (t == null) { // If the thread is not created yet
            t = new Thread(this, name); // Create a new Thread
object
            t.start(); // Start the thread
        }
    }
}

// The main class
public class Main {
    public static void main(String[] args) {
        // Create two objects of MyThread class
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");

        // Start the threads
        t1.start();
        t2.start();

        // Create two objects of MyRunnable class
        MyRunnable r1 = new MyRunnable("Runnable 1");
        MyRunnable r2 = new MyRunnable("Runnable 2");

        // Start the threads
        r1.start();
        r2.start();
    }
}
```

OUT PUT

```
Hello, I am Thread 1
Hello, I am Runnable 1
Hello, I am Thread 2
```

```
Hello, I am Runnable 2
```

47. Explain Two ways of multithreading implementations in java classes with Example?

Multithreading in Java is a process of executing two or more threads simultaneously to maximize the utilization of CPU[1]. A thread is a lightweight sub-process, the smallest unit of processing[2]. Threads can be created by using two mechanisms in Java[1]:

- **Extending the Thread class**: We create a class that extends the java.lang.Thread class and overrides its run() method. This method defines the task of the thread. To start the thread, we call the start() method on the thread object. This method invokes the run() method on the thread object[13].
- **Implementing the Runnable interface**: We create a class that implements the java.lang.Runnable interface and overrides its run() method. This method defines the task of the thread. To start the thread, we create a Thread object and pass our class object as an argument to its constructor. Then we call the start() method on the Thread object. This method invokes the run() method on our class object[13].

Here are some examples of multithreading implementations in Java classes:

```
// Extending the Thread class
class MyThread extends Thread {
   // Constructor
   public MyThread(String name) {
      super(name); // Set the name of the thread
   }

   // Override the run() method
   public void run() {
      System.out.println("Hello, I am " + this.getName());
   }
}

// Implementing the Runnable interface
```

```
class MyRunnable implements Runnable {
    // A reference to a Thread object
    private Thread t;
    // A name for the thread
    private String name;

    // Constructor
    public MyRunnable(String name) {
        this.name = name; // Set the name of the thread
    }

    // Override the run() method
    public void run() {
        System.out.println("Hello, I am " + name);
    }

    // A method to start the thread
    public void start() {
        if (t == null) { // If the thread is not created yet
            t = new Thread(this, name); // Create a new Thread object
            t.start(); // Start the thread
        }
    }
}

// The main class
public class Main {
    public static void main(String[] args) {
        // Create two objects of MyThread class
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");

        // Start the threads
        t1.start();
        t2.start();

        // Create two objects of MyRunnable class
        MyRunnable r1 = new MyRunnable("Runnable 1");
        MyRunnable r2 = new MyRunnable("Runnable 2");

        // Start the threads
        r1.start();
        r2.start();
    }
}
```

The output of this program may vary depending on the CPU scheduling, but it will look something like this:

```
Hello, I am Thread 1
Hello, I am Runnable 1
Hello, I am Thread 2
Hello, I am Runnable 2
```

UNIT 10

48. What is a collection in java?

<u>Collection:</u>

Any group of individual objects are grouped into a single

object is known as Collection.

<u>Frame Work:</u>

Frame work is a set of classes and interface, which provides

ready-made Architecture in order to implement new feature of class.

<u>Collection of Frame Work:</u>

Collection of frame work defined in JDK1.2

in which holds collection of classes and interface in it. It is used to store

and manipulating a group of objects.

- in Java collection is a framework that provides an architecture to store andmanipulate the group of objects.
- A Collection in Java is an object which represents a group of objects, known as its elements.
- It is a single unit. They are used to standardize the way in which objectsare handled in the class.

49. What is collection in java and Explain about the hierarchy of collections?

<u>Collection:</u>

Any group of individual objects are grouped into a single object is known as Collection.

Frame Work:

Frame work is a set of classes and interface, which provides

ready-made Architecture in order to implement new feature of class.

Collection of Frame Work:

Collection of frame work defined in JDK1.2

in which holds collection of classes and interface in it. It is used to store

and manipulating a group of objects.

- in Java collection is a framework that provides an architecture to store andmanipulate the group of objects.
- A Collection in Java is an object which represents a group of objects, known as its elements.
- It is a single unit. They are used to standardize the way in which objectsare handled in the class.

Collection Hierarchy

- The Collection interface is the root interface of all collections in Java. It extends the Iterable interface, which allows iterating over the elements of a collection using an iterator object[2]. The Collection interface is extended by three sub-interfaces: List, Set, and Queue[2].
- The List interface represents an ordered collection of elements that allows duplicates. The List interface is implemented by classes such as ArrayList, LinkedList, Vector, and Stack[2].
- The Set interface represents an unordered collection of unique elements that does not allow duplicates. The Set interface is implemented by classes such as HashSet, LinkedHashSet, TreeSet, and EnumSet[2].
- The Queue interface represents a collection of elements that follows the first-in-first-out (FIFO) principle. The Queue interface is implemented by classes such as PriorityQueue, ArrayDeque, and LinkedList[2].

- The Map interface represents a collection of key-value pairs that maps each unique key to a value. The Map interface is not a sub-interface of the Collection interface, but it is part of the collection framework. The Map interface is implemented by classes such as HashMap, LinkedHashMap, TreeMap, Hashtable, and EnumMap[2].
- The SortedSet interface is a sub-interface of the Set interface that maintains a sorted order of its elements. The SortedSet interface is implemented by the TreeSet class[2].
- The SortedMap interface is a sub-interface of the Map interface that maintains a sorted order of its keys. The SortedMap interface is implemented by the TreeMap class[2].

50. What is collection in java and explain ArrayList with Exmple?

Array List:

- you must've figured it out from the name itself, that ArrayList is similar to Arrays.
- They are also called as dynamic arrays.
- That means, it does not have a fixed size.
- Its size can be increased or decreased if elements are added or removed.
- Since the ArrayList cannot be used for primitive data types like int, char etc. , we need to use a wrapper class.

Example:

```
import java.util.*;

public class ArrayListInterface

{

public static void main(String args[])

{
```

```
// create an ArrayList of String type
ArrayList<String> str = new ArrayList<String>();
// Initialize an ArrayList with add()
str.add("CSE");
str.add("IT");
str.add("CIVIl");
str.add("MECHANICAL");
str.add("BDA");
// to print the ArrayList
System.out.println("ArrayList is" + str);
}
}
```

Output:

```
ArrayList is [CSE, IT, CIVIl, MECHANICAL, BDA]
```

Important Logical Programming Questions

51. Write Program to count the number of digits from given input?

```
import java.util.Scanner;

public class CountDigits {

    public static void main(String[] args) {
        // Create a Scanner object to read the input
        Scanner sc = new Scanner(System.in);

        // Declare an int variable to store the input
        int num;

        // Prompt the user to enter the input
        System.out.print("Please enter a positive integer: ");

        // Read the input and validate it
        num = sc.nextInt();
        if (num < 0) {
            System.out.println("Invalid input. Please enter a positive integer.");
            return;
        }

        // Declare an int variable to store the count of digits
        int count = 0;

        // Use a for loop to divide the input by 10 and increment the count
        for (; num > 0; num = num / 10, count++);

        // Display the result
        System.out.println("The number of digits is: " + count);
    }
}
```

52. Write a program to swap two number without using method?

```
import java.util.Scanner;

public class SwapNumbers {

    public static void main(String[] args) {
        // Create a Scanner object to read the input
        Scanner sc = new Scanner(System.in);

        // Declare two int variables to store the input
        int x, y;
        // Prompt the user to enter the input
        System.out.print("Please enter two integers: ");

        // Read the input
        x = sc.nextInt();
        y = sc.nextInt();

        // Display the original values
        System.out.println("Before swapping: x = " + x + ", y = " + y);

        // Swap the values using bitwise XOR operator
        x = x ^ y; // x now holds x ^ y
        y = x ^ y; // y now holds x ^ y ^ y = x
        x = x ^ y; // x now holds x ^ y ^ x = y

        // Display the swapped values
        System.out.println("After swapping: x = " + x + ", y = " + y);
    }
}
```

Out put :

```
Please enter two integers: 10 20
Before swapping: x = 10, y = 20
After swapping: x = 20, y = 10
```

53. Write Program to display even numbers example?

```
public class DisplayEvenNumbers {
    public static void main(String[] args) {
        // Declare a variable to store the limit
        int limit = 100;

        // Print a message
        System.out.println("Even numbers from 1 to " + limit + ":");

        // Use a for loop to iterate from 1 to limit
        for (int i = 1; i <= limit; i++) {
            // Check if the number is divisible by 2
            if (i % 2 == 0) {
                // Print the number
                System.out.print(i + " ");
            }
        }
    }
}
```

Out put :

```
Even numbers from 1 to 100:
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72
```

74 76 78 80 82 84 86 88

54. Write a program given input number is palindrome number or not?

```
import java.util.Scanner;

public class PalindromeNumber {

    public static void main(String[] args) {
        // Create a Scanner object to read the input
        Scanner sc = new Scanner(System.in);

        // Declare an int variable to store the input
        int num;

        // Prompt the user to enter the input
        System.out.print("Please enter an integer: ");

        // Read the input and validate it
        num = sc.nextInt();
        if (num < 0) {
            System.out.println("Invalid input. Please enter a positive integer.");
            return;
        }

        // Declare an int variable to store the reversed number
        int reversed = 0;
        // Declare another int variable to store the original number
        int original = num;

        // Use a loop to reverse the input number
        while (num > 0) {
            // Extract the last digit of the input number
            int lastDigit = num % 10;

            // Add the last digit to the reversed number after multiplying it by 10
            reversed = reversed * 10 + lastDigit;

            // Remove the last digit of the input number by dividing it by 10
            num = num / 10;
        }

        // Compare the original number and the reversed number
        if (original == reversed) {
            System.out.println(original + " is a palindrome number.");
        } else {
            System.out.println(original + " is not a palindrome number.");
        }
    }
}
```

Out put :

```
Please enter an integer: 121
```

```
121 is a palindrome number.
```

55. Write a program for display the Armstrong number or not?

```
import java.util.Scanner;

public class ArmstrongNumber {

    public static void main(String[] args) {
        // Create a Scanner object to read the input
        Scanner sc = new Scanner(System.in);

        // Declare an int variable to store the input
        int num;

        // Prompt the user to enter the input
        System.out.print("Please enter a positive integer: ");

        // Read the input and validate it
        num = sc.nextInt();
        if (num < 0) {
            System.out.println("Invalid input. Please enter a positive integer.");
            return;
        }

        // Declare an int variable to store the original number
```

```
int original = num;

// Declare an int variable to store the number of digits
int digits = 0;

// Use a loop to count the number of digits
while (num > 0) {
    digits++; // Increment the digits by 1
    num = num / 10; // Remove the last digit of the input number
}

// Restore the value of num
num = original;

// Declare an int variable to store the sum of powers
int sum = 0;

// Use another loop to calculate the sum of powers
while (num > 0) {
    // Extract the last digit of the input number
    int lastDigit = num % 10;

    // Raise the last digit to the power of digits and add it to sum
    sum = sum + (int)Math.pow(lastDigit, digits);

    // Remove the last digit of the input number
```

```
            num = num / 10;
        }

        // Compare the original number and the sum of powers
        if (original == sum) {
            System.out.println(original + " is an Armstrong number.");
        } else {
            System.out.println(original + " is not an Armstrong number.");
        }
    }
}
```

Out put :

```
Please enter a positive integer: 371
371 is an Armstrong number.
```

56. Create a class named Flight with flightNumer, sourceLocation, destination, numberOfTickets, ticketFare, Define methods to input data using scanner class object, define another method to calculate ticketAmount, 18% tax and 5% discount on ticketAmount. Find the amount payable. Define another method to print the ticket details class Flight { Data members declaration inputData() method { ----- ---- } calculate() method { ----- ----- } putData() method { ----- ----- } main() method { ---- ---- } }

```
import java.util.Scanner;

// A class named Flight
class Flight {
    // Data members declaration
    private int flightNumber;
    private String sourceLocation;
    private String destination;
    private int numberOfTickets;
    private double ticketFare;

    // Constructor
    public Flight(int flightNumber, String sourceLocation, String destination, int
numberOfTickets, double ticketFare) {
        this.flightNumber = flightNumber;
        this.sourceLocation = sourceLocation;
        this.destination = destination;
        this.numberOfTickets = numberOfTickets;
        this.ticketFare = ticketFare;
    }

    // inputData() method
    public void inputData() {
        // Create a Scanner object to read the input
        Scanner sc = new Scanner(System.in);

        // Prompt the user to enter the data
        System.out.print("Enter flight number: ");
        flightNumber = sc.nextInt();
        System.out.print("Enter source location: ");
        sourceLocation = sc.next();
        System.out.print("Enter destination: ");
        destination = sc.next();
        System.out.print("Enter number of tickets: ");
        numberOfTickets = sc.nextInt();
        System.out.print("Enter ticket fare: ");
        ticketFare = sc.nextDouble();
    }

    // calculate() method
    public double calculate() {
        // Calculate the ticket amount
        double ticketAmount = numberOfTickets * ticketFare;

        // Add 18% tax
        ticketAmount = ticketAmount + (ticketAmount * 0.18);

        // Subtract 5% discount
        ticketAmount = ticketAmount - (ticketAmount * 0.05);

        // Return the final amount payable
        return ticketAmount;
    }

    // putData() method
    public void putData() {
        // Print the details of the flight
```

```
        System.out.println("Flight number: " + flightNumber);
        System.out.println("Source location: " + sourceLocation);
        System.out.println("Destination: " + destination);
        System.out.println("Number of tickets: " + numberOfTickets);
        System.out.println("Ticket fare: " + ticketFare);

        // Print the amount payable
        System.out.println("Amount payable: " + calculate());
    }

    // main() method
    public static void main(String[] args) {
        // Create an object of Flight class
        Flight f = new Flight(0, "", "", 0, 0.0);

        // Call inputData() method on it
        f.inputData();

        // Call putData() method on it
        f.putData();
    }
}
```

Out put :

```
Enter flight number: 101
Enter source location: Mumbai
Enter destination: Delhi
Enter number of tickets: 2
Enter ticket fare: 5000.0
Flight number: 101
Source location: Mumbai
Destination: Delhi
Number of tickets: 2
Ticket fare: 5000.0
Amount payable: 11830.0
```