Java-unit-1

11 May 2025 21:25

What is Java?

Java is a popular high-level,object-oriented programming language that was originally developed by Sun Microsystems and released in 1995. Currently, Java is owned by Oracle, and more than 3 billion devices run Java. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Java is a general-purpose programming language intended to let programmers ***Write Once, Run Anywhere (WORA)***. This means that compiled Java code can run on all platforms that support Java without the need to recompile.

The latest release of the Java Standard Edition is Java SE 23.
J2EE for Enterprise Applications,
J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME, respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is

1. Object-oriented
2. Platform Independent
3. Simple
4. Secure
5. Architecture-neutral
6. Portable
7. Robust
8. Interpreted
9. High Performance
10. Dynamic

What is JDK?

JDK is an abbreviation for **Java Development Kit** which includes all the tools, executables, and binaries required to compile, debug, and execute a Java Program.JDK is platform dependent i.e. there are separate installers for Windows, Mac, and Unix systems. JDK includes both JVM and JRE and is entirely responsible for code execution.

What is JRE?

JRE is a **Java Runtime Environment** which is the implementation of JVM i.e. the specifications that are defined in JVM are implemented and create a corresponding environment for the execution of code

What is JVM?

JVM is the abbreviation for **Java Virtual Machine** which is a specification that provides a runtime environment in which Java byte code can be executed.
JVM which is responsible for converting Byte code to machine-specific code.

Steps to Write, Save, and Run Hello World Program

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps –

1. Save the file with test.java
2. Compile the program # javac test.java
3. Run your program # java test

1. Public Main Class

```
public class MyFirstJavaProgram {
```

This line is creating a new class MyFirstJavaProgram and being public, this class is to be defined in the same name file as MyFirstJavaProgram.java. This convention helps Java compiler to identify the name of public class to be created before reading the file content.

3. Public Static Void Main

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

This line represents the main method that JVM calls when this program is loaded into memory. This method is used to

execute the program. Once this method is finished, program is finished in single threaded environment

4. Keywords Used

Let's check the purpose of each keyword in this line.

- **public** – defines the scope of the main method. Being public, this method can be called by external program like JVM.
- **static** – defines the state of the main method. Being static, this method can be called by external program like JVM without first creating the object of the class.
- **void** – defines the return type of the main method. Being void, this method is not returning any value.
- **main** – name of the method
- **String []args** – arguments passed on command line while executing the java command

5. System.out.println() Method

```
System.out.println("Hello World"); // prints Hello World
```

`System.out` represents the primary console and its **`println()`** method is taking "Hello World" as input and it prints the same to the console output.

Java Comments:

In Java, there are three types of comments:

- Single-line comments
- Multiline comments
- Documentation comments

1. Single Line Comment

The single-line comment is used to add a comment on only one line and can be written by using the two forward slashes (**`//`**).

```
// comment
```

2. Multiline Comment

The multiline (or, multiple-line) comments start with a forward slash followed by an asterisk (**`/*`**) and end with an asterisk followed by a forward slash (**`*/`**) and they are used to add comment on multiple lines.

```
/* Comment (line 1)
Comment (line 2)
...
*/
```

3. Documentation Comment

The documentation comments are used for writing the documentation of the source code. The documentation comments start with a forward slash followed by the two asterisks (**`/**`**), end with an asterisk followed by a backward slash (**`*/`**), and all lines between the start and end must start with an asterisk (**`*`**).

```
/**
* line 1
* line 2
...
*/
```

==========

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- **Methods** – A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** – Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** – Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names** – For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
  Example – *class MyFirstJavaClass*
- **Method Names** – All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
  Example – *public void myMethodName()*
- **Program File Name** – Name of the program file should exactly match the class name.
  When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).
  But please make a note that in case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.
  Example – Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*
- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.
In Java, there are several points to remember about identifiers. They are as follows –

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A keyword cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers –

- **Access Modifiers** – default, public , protected, private
- **Non-access Modifiers** – final, abstract, strictfp

Java Variables

Following are the types of variables in Java –

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

Java Arrays

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap.

Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.
With the use of enums it is possible to reduce the number of bugs in your code.

```
class FreshJuice {
  enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
  FreshJuiceSize size;
}

public class FreshJuiceTest {

  public static void main(String args[]) {
    FreshJuice juice = new FreshJuice();
    juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
    System.out.println("Size: " + juice.size);
  }
}
```

Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

Inheritance

In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

Java inheritance allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**

Interfaces

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

The Java data types are categorized into two main categories –

- Primitive Data Types
- Reference/Object Data Types

Java Primitive Data Types

Primitive data types are predefined by the language and named by a keyword. There are eight primitive data types supported by Java. Below is the list of the primitive data types:

- byte
- short
- int
- long
- float
- double
- boolean

byte Data Type

The **byte** data type is an 8-bit signed two's complement integer with a minimum value of -128 (-2^7) and a maximum value of 127 (inclusive) (2^7 -1).

```
byte a = 100;
byte b = -50;
```

short Data Type

The **short** data type is a 16-bit signed two's complement integer, which provides a range of values from -32,768 (-2^{15}) to 32,767 (inclusive) (2^{15} -1).

```
short s = 10000;
short r = -20000;
```

int Data Type

The **int** data type is a 32-bit signed two's complement integer, allowing for a wide range of values from -2,147,483,648 (-2^{31}) to 2,147,483,647 (inclusive) (2^{31} -1).

```
int a = 100000;
int b = -200000;
```

long Data Type

The **long** data type is a 64-bit signed two's complement integer, capable of representing a vast range of values from -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 (inclusive) (2^{63} -1).

```
long a = 100000L;
long b = -200000L;
```

float Data Type

The **float** data type is a single-precision 32-bit IEEE 754 floating-point representation.

```
float f1 = 234.5f;
```

double Data Type

The **double** data type is a double-precision 64-bit IEEE 754 floating-point representation,

```
double d1 = 123.4;
```

boolean Data Type

The **boolean** data type represents a single bit of information and can hold one of two possible values: true or false.

```
boolean one = true;
```

char Data Type

The **char** data type is a single 16-bit Unicode character, which represents a wide range of characters from different languages and symbols.

```
char letterA = 'A'
```

Java Non-Primitive (Reference/Object) Data Types

The non-primitive data types are not predefined. The reference data types are created using defined constructors of the classes. They are used to access objects.

The following are the non-primitive (reference/object) data types –

- **String**: The string is a class in Java, and it represents the sequences of characters.
- **Arrays**: Arrays are created with the help of primitive data types and store multiple values of the same type.
- **Classes**: The classes are the user-defined data types and consist of variables and methods.
- **Interfaces**: The interfaces are abstract types that are used to specify a set of methods.

Java Type Casting

Type casting is a technique that is used either by the compiler or a programmer to convert one data type to another in Java. **Type casting** is also known as **type conversion**. For example, converting int to double, double to int, short to int, etc.
There are two types of type casting allowed in Java programming:

- Widening type casting
- Narrowing type casting

Widening Type Casting

Widening type casting is also known as **implicit type casting** in which a smaller type is converted into a larger type, it is done by the compiler automatically.

Hierarchy

Here is the hierarchy of widening type casting in Java:

```
byte>short>char>int>long>float>double
```

Narrowing Type Casting

Narrowing type casting is also known as **explicit type casting** or **explicit type conversion** which is done by the programmer manually. In the narrowing type casting a larger type can be converted into a smaller type.

Syntax

Below is the syntax for narrowing type casting i.e., to manually type conversion:

```
double doubleNum = (double) num;
```

==========
Operators

Java operators are the symbols that are used to perform various operations on variables and values. By using these operators, we can perform operations like addition, subtraction, checking less than or greater than, etc.
There are different types of operators in Java, we have listed them below –

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Misc Operators

Java Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

Addition
Addition +
Subtraction -
Multiplication *
Division /
Modulus %
Increment ++
Decrement --

Java Assignment Operators

Assignment Operators are used to assign values to variables. These operators modify the value of a variable based on the operation performed.

=
+=
-+
/=
%=
<<=
>>=
&=
^=
|=

```
public class AssignmentExample {
   public static void main(String[] args) {
      int a = 10;

      // Assign and add
      a += 5;
      System.out.println("a += 5: " + a); // 15

      // Assign and subtract
      a -= 3;
      System.out.println("a -= 3: " + a); // 12

      // Assign and multiply
      a *= 2;
      System.out.println("a *= 2: " + a); // 24

      // Assign and divide
      a /= 4;
      System.out.println("a /= 4: " + a); // 6

      // Assign and modulus
      a %= 5;
      System.out.println("a %= 5: " + a); // 1
   }
}
```

Java Relational Operators

Relational operators are used to compare two values. These operators return a boolean result: true if the condition is met and false otherwise. Relational operators are commonly used in decision-making statements like if conditions and loops.

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

```
public class RelationalExample {
   public static void main(String[] args) {
      int A = 10, B = 5;

      System.out.println("A == B: " + (A == B)); // false
      System.out.println("A != B: " + (A != B)); // true
      System.out.println("A > B: " + (A > B));   // true
```

```
    System.out.println("A < B: " + (A < B));   // false
    System.out.println("A >= B: " + (A >= B)); // true
    System.out.println("A <= B: " + (A <= B)); // false
  }
}
```

Java Logical Operators

Logical operators are used to perform logical operations on boolean values. These operators are commonly used in decision-making statements such as if conditions and loops to control program flow.

```
&& (logical and)
|| (logical or)
! (logical not)

public class LogicalExample {
  public static void main(String[] args) {
    boolean A = true, B = false;

    System.out.println("A && B: " + (A && B)); // false
    System.out.println("A || B: " + (A || B)); // true
    System.out.println("!A: " + (!A)); // false
    System.out.println("!B: " + (!B)); // true
  }
}
```

Java Bitwise Operators

Bitwise operators are used to perform operations at the binary (bit) level. These operators work on individual bits of numbers. They are commonly used in low-level programming, encryption, and performance optimization.

```
& (bitwise and)
| (bitwise or)
^ (bitwise XOR)
~ (bitwise compliment)
<< (left shift)
>> (right shift)
>>> (zero fill right shift)


public class BitwiseExample {
  public static void main(String[] args) {
    int A = 60; // 0011 1100
    int B = 13; // 0000 1101

    System.out.println("A & B: " + (A & B)); // 12  (0000 1100)
    System.out.println("A | B: " + (A | B)); // 61  (0011 1101)
    System.out.println("A ^ B: " + (A ^ B)); // 49  (0011 0001)
    System.out.println("~A: " + (~A)); // -61  (1100 0011 in 2's complement)
    System.out.println("A << 2: " + (A << 2)); // 240 (1111 0000)
    System.out.println("A >> 2: " + (A >> 2)); // 15  (0000 1111)
    System.out.println("A >>> 2: " + (A >>> 2)); // 15  (0000 1111)
  }
}
```

Java Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

variable x = (expression) ? value if true : value if false

```
public class Test {

  public static void main(String args[]) {
    int a, b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );
```

```
    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

==========

Q3. Explain Control statements with example ?

Control Statements:

---

1. Decision Making
2. If Else Statement
3. Switch Statement
4. Loop Control
5. For Loop
6. For-Each Loop
7. While Loop
8. Do While Loop
9. Break Statement
10. Continue Statement

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

if statement
if...else statement **nested if statement** **switch statement**

2. If Else Statement

`if` statement is a conditional statement used to execute a block of code when a specified condition evaluates to **true**. If the condition is **false**, an optional **`else`** statement can be used to execute an alternative block of code.

i. If statement
ii. If-else statement
iii. Ladder if (if else if) statements
iv. Nested if-else Statement

```
if(Boolean_expression) {
  // Executes when the Boolean expression is true
}else {
  // Executes when the Boolean expression is false
}
```

```
public class Test {

  public static void main(String args[]) {
    int x = 30;

    if( x < 20 ) {
      System.out.print("This is if statement");
    }else {
      System.out.print("This is else statement");
    }
  }
}
```

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2) {
  // Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3) {
  // Executes when the Boolean expression 3 is true
}else {
  // Executes when the none of the above condition is true.
}
```

```
public class Test {

  public static void main(String args[]) {
    int x = 30;

    if( x == 10 ) {
      System.out.print("Value of X is 10");
    }else if( x == 20 ) {
      System.out.print("Value of X is 20");
    }else if( x == 30 ) {
      System.out.print("Value of X is 30");
    }else {
      System.out.print("This is else statement");
    }
  }
}
```

```
if(condition1){
        // code block
        if(condition2){
                //code block
        }
}
```

```
public class Test {

   public static void main(String[] args) {

      int x = 10, y = 20, z = 30;

      if(x >= y) {
         if(x >= z)
            System.out.println(x + " is the largest.");
         else
            System.out.println(z + " is the largest.");
      } else {
         if(y >= z)
            System.out.println(y + " is the largest.");
         else
            System.out.println(z + " is the largest.");
      }
   }
}
```

Java switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.
The **switch** statement can be used when multiple if-else statements are required. It can have multiple code blocks along with the case values and execute one of many code blocks based on the matched case value.

```
switch(expression) {
  case value :
    // Statements
    break; // optional

  case value :
    // Statements
    break; // optional

  // You can have any number of case statements.
  default : // Optional
    // Statements
}
```

Ex:

```
public class Test {

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

```
    char grade = 'C';

    switch(grade) {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
        break;
      default :
        System.out.println("Invalid grade");
    }
    System.out.println("Your grade is " + grade);
  }
}
```

Loop Statement

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages

Java Loops

Java programming language provides the following types of loops to handle the looping requirements:

1	**while loop** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** Like a while statement, except that it tests the condition at the end of the loop body.
4	**Enhanced for loop** As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

Loop Control Statements

1	**break statement** Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

In Java, a **for loop** is a repetition control structure used to execute a block of code **a specific number of times**
For loops are used in Java for tasks such as **iterating over arrays, performing calculations, and handling repetitive operations**.

The **Java for loop** is an *entry control loop*, meaning it checks the given condition before executing the loop body. Its structured syntax makes it more readable and concise compared to other loop structures like the while loop.

Syntax:

```
for(initialization; Boolean_expression; update) {
  // Statements
}
```

```
public class Test {
   public static void main(String[] args) {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Number: " + i);
      }
```

```
  }
}
```

1. Initialization

Defines and initializes the loop counter variable. This part executes only once at the start.

```
// Initialization example
int i = 1; // Loop variable declared and initialized
```

2. Boolean Expression (Condition)

Evaluates before each iteration. If the condition evaluates to **`true`**, the loop executes; if **`false`**, it terminates.

```
// Condition example
i <= 5; // Loop continues while i is less than or equal to 5
```

3. Body

Contains the statements that execute repeatedly as long as the Boolean condition remains **`true`**. This section may also include an update to the loop counter.

```
// Body of the loop
System.out.println("Iteration: " + i); // Executes in each iteration
```

4. Update

Modifies the loop counter at the end of each iteration.

```
// Update example
i++; // Increments i by 1 after each iteration
```

Example 1: Printing Numbers in a Range Using for Loop

Example 2: Printing Array Elements Using for Loop

Nested for Loops

A nested for loop is a for loop containing another for loop inside it.

Example

In this example, we are printing tables of the numbers from 1 to 10

```
public class Main {
  public static void main(String[] args) {
    // Implementing nested for loop
    // Initializing loop counters
    int num = 1;
    int i = 1;

    // outer for loop
    for (num = 1; num <= 10; num++) {
      //inner for loop
      System.out.print("Table of " + num + " is : ");
      for (i = 1; i <= 10; i++) {
        // printing table
        System.out.print(num * i + " ");
      }
      // printing a new line
      System.out.println();
    }
  }
}
```

Use Cases of Java for Loop

1. Iterating Over a Fixed Range of Numbers

```
public class ForLoopExample {
   public static void main(String[] args) {
      for(int i = 1; i <= 10; i++) {
         System.out.println("Number: " + i);
      }
   }
}
```

2. Traversing Arrays and Collections

```
public class ArrayTraversal {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40, 50};
      for(int i = 0; i < numbers.length; i++) {
         System.out.println("Element: " + numbers[i]);
      }
   }
}
```

3. Implementing Nested Loops

```
public class NestedLoopExample {
   public static void main(String[] args) {
      for(int i = 1; i <= 3; i++) {
         for(int j = 1; j <= 3; j++) {
            System.out.print(i + "," + j + " ");
         }
         System.out.println();
      }
   }
}
```

4. Repeating Tasks a Specific Number of Times

```
public class RepeatTaskExample {
   public static void main(String[] args) {
      for(int i = 0; i < 5; i++) {
         System.out.println("Task executed: " + (i+1));
      }
   }
}
```

5. Using Enhanced for Loop (for-each)

```
public class EnhancedForLoopExample {
   public static void main(String[] args) {
      String[] fruits = {"Apple", "Banana", "Cherry"};
      for(String fruit : fruits) {
         System.out.println(fruit);
      }
   }
}
```

Java for each Loop

A **for each** loop is a special repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for each** loop is useful even when you do not know how many times a task is to be repeated.

Syntax

Following is the syntax of enhanced for loop (or, for each loop) −

```
for(declaration : expression) {
  // Statements
}
```

Execution Process

- **Declaration** – The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression** – This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

```
import java.util.Arrays;
import java.util.List;

public class Test {

  public static void main(String args[]) {
    List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);

    for(Integer x : numbers ) {
      System.out.print( x );
      System.out.print(",");
    }
  }
}
```

Java while Loop

Java while loop statement repeatedly executes a code block as long as a given condition is true.
The **while** loop is an entry control loop, where conditions are checked before executing the loop's body.

Syntax of while Loop

The syntax of a while loop is –

```
while(Boolean_expression) {
  // Statements
}
```

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    while( x < 20 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
}
```

Java do while Loop

Java **do while** loop is similar to a while loop, except that a **do while** loop is guaranteed to execute at least one time.
The **do-while** loop is an exit control loop where the condition is checked after executing the loop's body.

Syntax of do while Loop

Following is the syntax of a do...while loop –

```
do {
  // Statements
}while(Boolean_expression);
```

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    do {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
}
```

The Java **break** statement is used to exit a loop immediately and transfer control to the next statement after the loop. It has two main usages: when encountered inside a loop, it terminates the loop instantly, and in a **switch** statement, it is used to exit a specific case (covered in the next chapter).

The **continue statement** can be used in any loop control structure to skip the current iteration and jump to the next one. In a **for** loop, it immediately transfers control to the update statement, while in a **while** or **do-while** loop, it jumps directly to the Boolean expression for the next iteration.

==========

Q1. What is class, object, constructors and methods.
Ans:
A **class** is a blueprint from which individual objects are created (or, we can say a class is a data type of an object type). In Java, everything is related to classes and objects. Each class has its methods and attributes that can be accessed and manipulated through the objects.

For example, if you want to create a class for *students*. In that case, "*Student*" will be a class, and student records (like *student1*, *student2*, etc) will be objects.

Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
An Object can be defined as an instance of a class.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.
A class can also be defined as a blueprint from which you can create an individual object. Class does not consume any space.

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Java Constructors

Java constructors are special types of methods that are used to initialize an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

==========

Q2. Explain oops concept with real life example

OOPs (Object-Oriented Programming)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently,

In Java, we use method overloading and method overriding to achieve polymorphism.

Abstraction
Hiding internal implementation and showing functionality only to the user is known as abstraction. For example, phone call, we do not know the internal processing.
In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A Java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

==========

Q4. What is an Array? Explain different types of array? (ID, 2D, 3D, Jagged Array)

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

We have three types of arrays:

1. Single Dimensional Array (1D)
2. Two Dimensional Array (2D)
3. Three Dimensional Array (3D)
4. Jagged Array

Single-Dimensional Array in Java

A single-dimensional array in Java is a linear collection of elements of the same data type. It is declared and instantiated using the following syntax:

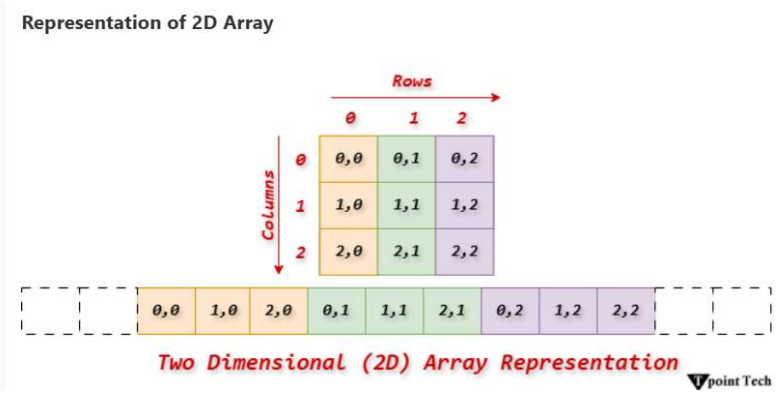
```
1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];
```

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
public class Main{
 public static void main(String args[]){
   //declaration, instantiation and initialization of an array
   int a[]={33,3,4,5};
   //traversing array
   for(int i=0;i<a.length;i++)//length is the property of array
      System.out.println(a[i]);
 }
}
```

Multidimensional Array in Java

A multidimensional array in Java is an array of arrays where each element can be an array itself. It is useful for storing data in row and column format.

It can be a two-dimensional (2D) or three-dimensional (3D) array.



Screen clipping taken: 15-05-2025 07:41

dataType[][] arrayRefVar;

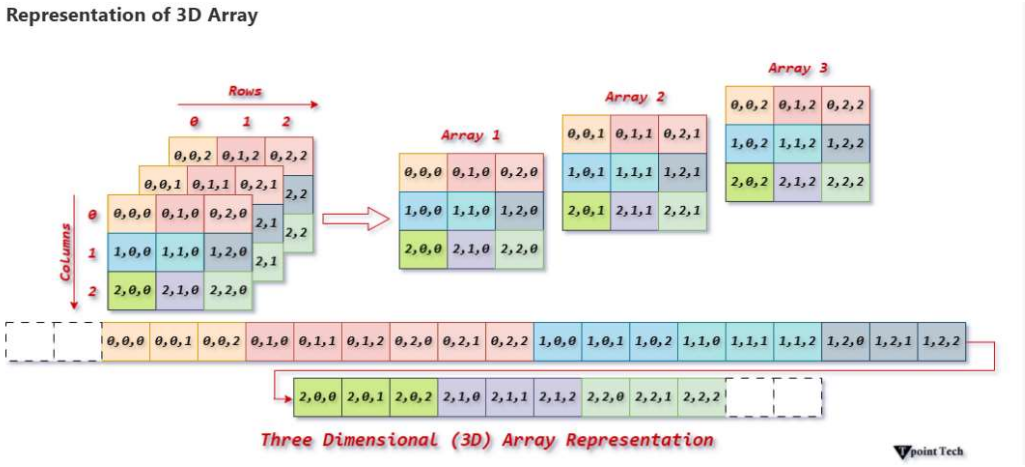
```
int[][] arr=new int[3][3];//3 row and 3 column
```

```
public class Main {
   public static void main(String args[]) {
      int arr[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // 3x3 matrix
      // Printing the 2D array
      for (int i = 0; i < 3; i++) {
         for (int j = 0; j < 3; j++) {
            System.out.print(arr[i][j] + " ");
         }
         System.out.println();
      }
   }
}
```

Three-Dimensional (3D) Array

It is a complex form of a 2D array. In other words, it is an array of a two-dimensional array.

datatype[][][] arrayName = **new** datatype[][][];



Screen clipping taken: 15-05-2025 07:44

```
public class Main {
   public static void main(String[] args)
   {
      //declaring and initializing three-dimensional array
      int[][][] threeDArray = {
         {
            {1, 2, 3},
            {4, 5, 6}
         },
         {
            {7, 8, 9},
            {10, 11, 12}
         }
      };
      //Print the elements of the 3D array
      System.out.println("Elements of the 3D Array:");
      for (int i = 0; i < threeDArray.length; i++) {
         for (int j = 0; j < threeDArray[i].length; j++) {
            for (int k = 0; k < threeDArray[i][j].length; k++) {
               System.out.print(threeDArray[i][j][k] + " ");
            }
            System.out.println(); // Move to the next line for better readability
         }
         System.out.println(); // Add a blank line between blocks
      }
   }
}
```

Jagged Arrays in Java

In Java, a jagged array is an array of arrays where each row of the array can have a different number of columns. This contrasts with a regular two-dimensional array, where each row has the same number of columns.

```
//Java Program to illustrate the jagged array
public class Main{
   public static void main(String[] args){
      //declaring a 2D array with odd columns
      int arr[][] = new int[3][];
      arr[0] = new int[3];
      arr[1] = new int[4];
      arr[2] = new int[2];
      //initializing a jagged array
      int count = 0;
      for (int i=0; i<arr.length; i++)
         for(int j=0; j<arr[i].length; j++)
            arr[i][j] = count++;

      //printing the data of a jagged array
      for (int i=0; i<arr.length; i++){
         for (int j=0; j<arr[i].length; j++){
            System.out.print(arr[i][j]+" ");
         }
         System.out.println();//new line
      }
   }
}
```

==========

Q5. what is Inheritance? Explain types of inheritance with suitable examples(programs

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism

Inheritance in Java enables a class to inherit properties and actions from another class, called a superclass or parent class. A class derived from a superclass is called a subclass or child group. Through inheritance, a subclass can access members of its superclass (fields and methods), enforce reuse rules, and encourage hierarchy.

Types of Inheritance in Java

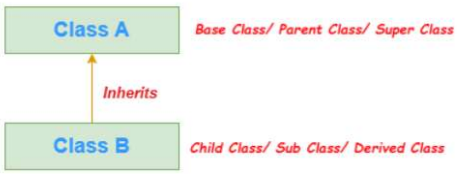
On the basis of class, there can be three types of inheritance in java:

1. single,
2. multilevel and
3. hierarchical.
4. Multiple
5. Hybrid

multiple and hybrid inheritance is supported through interface only.

Single Inheritance

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.



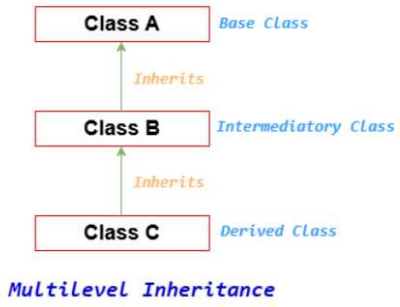
Single Inheritance

```
class Animal{
void eat(){System.out.println("eating...");}
}
```

```
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
public class Main{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Multilevel Inheritance

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

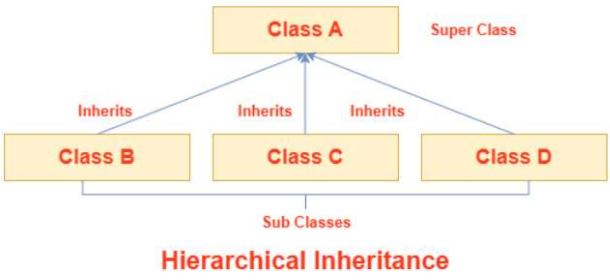


Multilevel Inheritance

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
public class Main{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Hierarchical Inheritance

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance



Hierarchical Inheritance

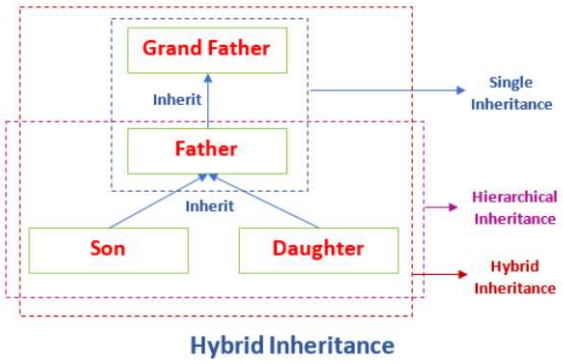
Screen clipping taken: 16-05-2025 05:58

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
```

```
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
public class Main{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Hybrid Inheritance in Java

The hybrid inheritance is the composition of two or more types of inheritance. The main purpose of using hybrid inheritance is to modularize the code into well-defined classes. It also provides the code reusability.



Screen clipping taken: 16-05-2025 06:06

The hybrid inheritance can be achieved by using the following combinations:

- Single and Multiple Inheritance (not supported but can be achieved through interface)
- Multilevel and Hierarchical Inheritance
- Hierarchical and Single Inheritance
- Multiple and Multilevel Inheritance

==========

Q.11 why Multiple Inheritance is not supported in java. Explain it with suitable example

To reduce the complexity and simplify the language, multiple inheritance is not supported in Java.
Suppose there are three classes A, B, and C. The C class inherits A and B classes. If A and B classes have the same method and we call it from child class object, there will be ambiguity to call the method of A or B class. It is called diamond problem.

```
class A {
void msg(){System.out.println("Hello");}
}
class B {
void msg(){System.out.println("Welcome");}
}
public class Main extends A,B {
 public static void main(String args[]){
   Main obj = new Main();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

==========

Q.17 how is multiple inheritance achieved in Java?

Java supports multiple inheritance through interfaces only, where a class can implement multiple interfaces. Multiple inheritance in Java is not possible by class, but it is possible through interfaces.

==========

Q.10 Explain forms of Inheritance, Benefits of Inheritance & costs of Inheritance

Advantages of Inheritance:

Code Reusability
Hierarchical Organization
Polymorphism
Easier Maintenance

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

==========

Q.6 what is polymorphism ? Explain types of it with examples.

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently,

In Java, we use method overloading and method overriding to achieve polymorphism

Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

Types of Polymorphism

There are two types of polymorphism in Java:

1. Compile-time Polymorphism
2. Runtime Polymorphism.

1) Compile-Time Polymorphism in Java

In Java, method overloading is used to achieve compile-time polymorphism. A class can have numerous methods with the same name but distinct parameter lists thanks to method overloading. The compiler uses the amount and kind of parameters provided to it during compilation to decide which method to call. This choice is made during compilation, which is why it's called "compile-time polymorphism."

Based on the inputs passed in during the method call, the compiler chooses the suitable overloaded method when a method is called.

```
6.  class Calculation {
7.      int add(int a, int b) {
8.          return a + b;
9.      }
10.     double add(double a, double b) {
11.         return a + b;
12.     }
13. }
14. public class Main {
15.     public static void main(String[] args) {
16.         Calculation calc = new Calculation();
17.         // Compile-time polymorphism: selecting the appropriate add method based on parameter types
18.         System.out.println("Sum of integers: " + calc.add(5, 3));
19.         System.out.println("Sum of doubles: " + calc.add(2.5, 3.7));
20.     }
```

```
}
```

2) Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

```
class Bank{
    float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
    float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
    float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
    float getRateOfInterest(){return 9.7f;}
}
public class Main{
 public static void main(String args[]){
  Bank b;
  b=new SBI();
  System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
  b=new ICICI();
  System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
  b=new AXIS();
  System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
 }
}
```

==========

Q.7 What is overloading? Explain Method overloading & constructor overloading with example

Constructor overloading in Java

In Java, we can overload constructors like methods.
The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.
Consider the following Java program, in which we have used different constructors in the class.

```
public class Student {
//instance variables of the class
int id;
String name;

Student(){
System.out.println("this a default constructor");
}

Student(int i, String n){
id = i;
name = n;
}

public static void main(String[] args) {
//object creation
Student s = new Student();
System.out.println("\nDefault Constructor values: \n");
System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);

System.out.println("\nParameterized Constructor values: \n");
Student student = new Student(10, "David");
System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.name);
```

```
}
}
```

Method Overloading

In Java, method overloading refers to the definition of numerous methods with distinct argument lists under the same class name. The type or quantity of parameters passed to the method when calling it influences which method the compiler calls.

```
public class MethodOverloading {
    public int adding(int a, int b) {
        return a + b;
    }
    public double adding(double x, double y) {
        return x + y;
    }
    public String adding(String s1, String s2) {
        return s1 + s2;
    }
    public static void main(String[] args) {
        MethodOverloading o1 = new MethodOverloading ();
        int r1 = o1.adding(2, 3);
        System.out.println("Value 1: " + r1);
        double r2 = o1.adding(0.2, 3.0);
        System.out.println("Value 2: " + r2);
        String r3 = o1.adding("Hello, ", "Everyone");
        System.out.println("Value 3: " + r3);
    }
}
```

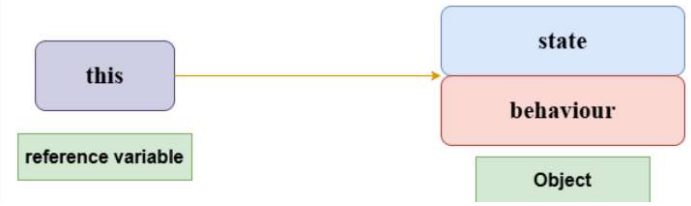
==========

Q.8 Explain the following Keywords with examples i)this ii)super iii) static iv) final

This:

this Keyword in Java

In Java, this is a **reference variable** that refers to the current object.



Screen clipping taken: 16-05-2025 13:14

Usage of Java this keyword

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
public class Main{
```

```
public static void main(String args[]){
Student s1=new Student(111,"Ankit",5000f);
Student s2=new Student(112,"Sumit",6000f);
s1.display();
s2.display();
}}
```

super Keyword in Java

The **super** keyword in Java is a reference variable that is used to refer to immediate parent class objects.

Uses of Java super Keyword

1. super can be used to refer to the immediate parent class instance variable.
2. super can be used to invoke the immediate parent class method.
3. super() can be used to invoke the immediate parent class constructor.

super is used to invoke the parent class constructor

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
    Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog(){
        super();  //calls the constructor of parent class
        System.out.println("dog is created");
    }
}
public class Main{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}
```

The static keyword in Java is primarily used for memory management. The static keyword can be used with variables, methods, blocks, and nested classes. The static keyword is associated with the class rather than an instance of the class.
The static can be as follows:

- Varying (also known as a class variable)
- Method (also known as a class method)
- Block
- Class hierarchy

final keyword in Java

The **final keyword in Java** is used to restrict the user. It is also known as a non-access modifier. We can use the final keyword with:

1. Variable
2. Method
3. Class
4. Parameter

1) Java final variable

When a variable is declared as final, it is known as a final variable. Its value cannot be changed once initialized. It behaves like a constant.
Syntax:

```
final datatype VARIABLE_NAME=VALUE;
```

2) Java final Method

A method declared as final is known as a final method. Subclasses cannot override the final method

3) Java final Class

A class declared with the final keyword is known as a final class. Note that the final class cannot be inherited.

```
class Main {
  final int SPEED_LIMIT=90;    //final variable
  void run() {
   SPEED_LIMIT=400;   //we cannot change the final variable
  }
  public static void main(String args[]) {
  Main obj=new  Main();
  obj.run();
  }
 }
```

==========

Q.9 what is abstraction? Explain abstract classes with suitable examples

Abstraction
Hiding internal implementation and showing functionality only to the user is known as abstraction. For example, phone call, we do not know the internal processing.
In Java, we use abstract class and interface to achieve abstraction.

ATM Machine: When we use an ATM, we interact with a simple interface-enter your PIN, select an option, and withdraw cash. But we do not see the complex backend processes like authentication, transaction validation, and communication with the bank servers. The internal workings are completely hidden from the user.

Achieving Abstraction

In Java, abstraction can be achieved in the following two ways:

1. Using Abstract Class (For Partial Abstraction)
2. Using Interface (For 100% abstraction)

1) Using Abstract Class

An **abstract class** is a class that cannot be instantiated directly. It serves as a blueprint for other classes. We define an abstract class using the **abstract** keyword

2) Using Interface

In Java, another way to implement abstraction is by using interfaces. Interfaces are useful for achieving 100% abstraction. In Java, as well as in other languages, interfaces include both variables and methods, but do not provide a method body.

```
interface Person {
    void display();
}
class Student implements Person {
    public void display() {
        System.out.println("This is the display method of the student class");
    }
}
class Lecturer implements Person {
    public void display() {
        System.out.println("This is the display method of the lecturer class");
    }
}
public class Main {
    public static void main(String[] args) {
        Person person1 = new Student();
        person1.display();
        Person person2 = new Lecturer();
        person2.display();
    }
}
```

==========

Q.12 What is the purpose of a package? Who explain the ways to define the package?

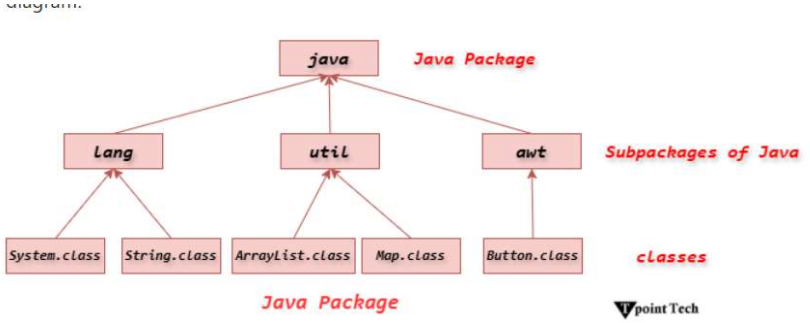
A **Java package** is a group of similar types of classes, interfaces and sub-packages.

Why use a package?

Avoid Naming Conflicts: Packages are used to avoid naming conflicts among class names.

Organise Classes: For organizing the class, interfaces and other components, packages are used

- **Control Access:** It is the package that controls the access of the protected and default members. Think what would be the use of default and protected access specifiers if packages were absent?
- **Reusability:** Java enhances the reusability management of code and does the promotion of data encapsulation with the help of packages.



Screen clipping taken: 17-05-2025 23:34

Types of Packages

There are two types of packages in Java.

1. Built-in Packages
2. User-defined Packages

Built-in Packages

Some of the built-in packages that are commonly used are:
java.lang: It contains those classes that are required for language support, such as classes for primitive data types and math operations. This package is automatically imported into Java.
java.io: It contains all those classes that are required for the input/output operations.
java.util: It contains utility classes that are needed for the data structures, such as Array, List, Map, etc.
java.applet: All those classes that are required for finding applets are found in this package.
java.awt: It contains all those classes that are required for creating components of the GUI (Graphical User Interface).
java.net: It includes classes that are used for doing tasks in the field of networking.

==========

Q.13 What is classpath? Explain the loads to set the classpath?

CLASSPATH: CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.
You need to set the CLASSPATH if:

- You need to load a class that is not present in the current directory or any sub-directories.
- You need to load a class that is not in a location specified by the extensions mechanism.

How to Set CLASSPATH in Windows Using Command Prompt

Type the following command in your Command Prompt and press enter.

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\jre1.8\rt.jar;
```

In the above command, The set is an internal DOS command that allows the user to change the variable value. CLASSPATH is a variable name. The variable enclosed in percentage sign (%) is an existing environment variable. The semicolon is a separator, and after the (;) there is the PATH of rt.jar file

==========

Q.14 what are different access specifiers. explain with Examples?

Access Modifiers / Access Specifiers
Access modifiers tells us the scope of the variable(ie where a variable can be accessed).
private: private members(variables & methods) can be accessed only with in the class.
default: these memebers can be accessed in any class with in the package.
protected: these members can be accessed in any class with in the package. Also if the class to which these members belong to has any derived classes in other packages, those classes also can

access.
public: these member can be accessed in any class.

==========

Q.15. What is an Interface. explain with an example program?

Java Interfaces

Java interface is a collection of abstract methods. The interface is used to achieve abstraction in which you can define methods without their implementations (without having the body of the methods). An interface is a reference type and is similar to the class.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

==========

Q.16 what is nested interface. Explain with an example program?

Q.16 what is nested interface. Explain with an example program?

Java Nested Interface

30 Mar 2025 | 2 min read

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

From <https://www.tpointtech.com/nested-interface>

==========

variables:
Variables are containers for storing data values.
The value depends on the data type of the variable.
The value can be altered during program exam execution.
There are three types of variables in java: local, instance and static.
Syntax: datatype variable_name[=default_value];
Example: int x=50;