ANNAMACHARA INSTITUTE OF TECHNOLOGY & SCIENCES, RAJAMPET
(AUTONOMOUS)

LECTURE NOTES ON Java programming (23A055ET)

B.TECH III Year - I Sem (2025-26)

DEPARTMENT OF MECHANICAL ENGINEERING

ANNAMACHARYA INSTITUTE OF TECHNOLOGY & SCIENCES

(An Autonomous Institution)

Title of the Course:	**JAVA PROGRAMMING**
Category:	**Open Elective-1**
Couse Code:	23A055ET
Year:	III B. Tech
Semester:	I Semester
Branch:	CE,EEE,ME,ECE

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	-	-	3

Course Objectives: This course will be able to

1. Identify Java language components and how they work together in applications
2. Learn the fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries.
3. Learn how to extend Java classes with inheritance and dynamic binding and how to use exception
4. Understand how to design applications with threads in Java
5. Understand how to use Java apis for program development

Course Outcomes:

At the end of the course, the student will be able to

CO1: Analyze problems, design solutions using OOP principles, and implement them efficiently in Java. (L4)

CO2: Design and implement classes to model real-world entities, with a focus on attributes, behaviours, and relationships between objects (L4)

CO3: Demonstrate an understanding of inheritance hierarchies and polymorphic behaviour, including method overriding and dynamic method dispatch. (L3)

CO4: Apply Competence in handling exceptions and errors to write robust and fault-tolerant code. (L3)

CO5: Perform file input/output operations, including reading from and writing to files using Java I/O classes, graphical user interface (GUI) programming using JavaFX. (L3)

CO6: Choose appropriate data structure of Java to solve a problem (L6)

Unit 1

Object Oriented Programming: Basic concepts, Principles, Program Structure in Java: Introduction, Writing Simple Java Programs, Elements or Tokens in Java Programs, Java Statements, Command Line Arguments, User Input to Programs, Escape Sequences Comments, Programming Style. Data Types, Variables, and Operators :Introduction, Data Types in Java, Declaration of Variables, Data Types, Type Casting, Scope of Variable Identifier, Literal Constants, Symbolic Constants, Formatted Output with printf() Method, Static Variables and Methods, Attribute Final, Introduction to Operators, Precedence and Associativity of Operators, Assignment Operator (=), Basic Arithmetic Operators, Increment (++) and Decrement (- -) Operators, Ternary Operator, Relational Operators, Boolean Logical Operators, Bitwise Logical Operators. Control Statements: Introduction, if Expression, Nested if Expressions, if–else Expressions, Ternary Operator?:, Switch Statement, Iteration Statements, while Expression, do–while Loop, for Loop, Nested for Loop, For–Each for Loop, Break Statement, Continue Statement.

Unit 2

Classes and Objects: Introduction, Class Declaration and Modifiers, Class Members, Declaration of Class Objects, Assigning One Object to Another, Access Control for Class Members, Accessing Private Members of Class, Constructor Methods for Class, Overloaded Constructor Methods, Nested Classes, Final Class and Methods, Passing Arguments by Value and by Reference, Keyword this.

Methods: Introduction, Defining Methods, Overloaded Methods, Overloaded Constructor Methods, Class Objects as Parameters in Methods, Access Control, Recursive Methods, Nesting of Methods, Overriding Methods, Attributes Final and Static.

Unit 3

Arrays: Introduction, Declaration and Initialization of Arrays, Storage of Array in Computer Memory, Accessing Elements of Arrays, Operations on Array Elements, Assigning Array to Another Array, Dynamic Change of Array Size, Sorting of Arrays, Search for Values in Arrays, Class Arrays, Two dimensional Arrays, Arrays of Varying Lengths, Three-dimensional Arrays, Arrays as Vectors.

Inheritance: Introduction, Process of Inheritance, Types of Inheritances, Universal Super Class Object Class, Inhibiting Inheritance of Class Using Final, Access Control and Inheritance, Multilevel Inheritance, Application of Keyword Super, Constructor Method and Inheritance, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Interfaces and Inheritance.

Interfaces: Introduction, Declaration of Interface, Implementation of Interface, Multiple Interfaces, Nested Interfaces, Inheritance of Interfaces, Default Methods in Interfaces, Static Methods in Interface, Functional Interfaces, Annotations.

Unit 4

Packages and Java Library : Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java. lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto un boxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java. .Instant), Formatting for Date/Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throw able, Unchecked Exceptions, Checked Exceptions.

Java I/O and File: Java I/O API, standard I/O streams, types, Byte streams, Character streams, Scanner class, Files in Java(Text Book 2)

Unit 5

String Handling in Java: Introduction, Interface Char Sequence, Class String, Methods for Extracting Characters from Strings, Comparison, Modifying, Searching; Class String Buffer. **Multithreaded Programming:** Introduction, Need for Multiple Threads Multithreaded Programming for Multi-core Processor, Thread Class, Main Thread Creation of New Threads, Thread States, Thread Priority-Synchronization, Deadlock and Race Situations, Inter thread Communication - Suspending, Resuming, and Stopping of Threads. **Java Database Connectivity:** Introduction, JDBC Architecture, Installing My SQL and My SQL Connector/J, JDBC Environment Setup, Establishing JDBC Database Connections, Result Set Interface Java FX GUI: Java FX Scene Builder, Java FX App Window Structure, displaying text and image, event handling, laying out nodes in scene graph, mouse events (Text Book 3)

Learning Resources:

Textbooks:

1. JAVA one step ahead, Anitha Seth, B.L. Juneja, Oxford.
2. Joy with JAVA, Fundamentals of Object Oriented Programming, Debasis Samanta,
   Monalisa Sarma, Cambridge, 2023.
3. JAVA 9 for Programmers, Paul Deitel, Harvey Deitel, 4th Edition, Pearson.

Reference Books:

1. The complete Reference Java, 11thedition, Herbert Schildt,TMH
2. Introduction to Java programming, 7th Edition, Y Daniel Liang, Pearson

Online Learning Resources:

1. https://nptel.ac.in/courses/106/105/106105191/
2. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_01288046454761881 6347 _shared/overview

E Ramesh
Assistant professor ,cse

UNIT-I

1. The history and evaluation of java
   Ans:

Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.
The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.
The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".
Java was developed by James Gosling, who is known as the father of Java,in1995.
James Gosling and his team members started the project in the early '90s.
Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

1) **James Gosling,** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.
3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
4) After that, it was called **Oak** and was developed as a part of the Green project.
5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.
7) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
8) Notice that Java is just a name, not an acronym.
9) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
10) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
11) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

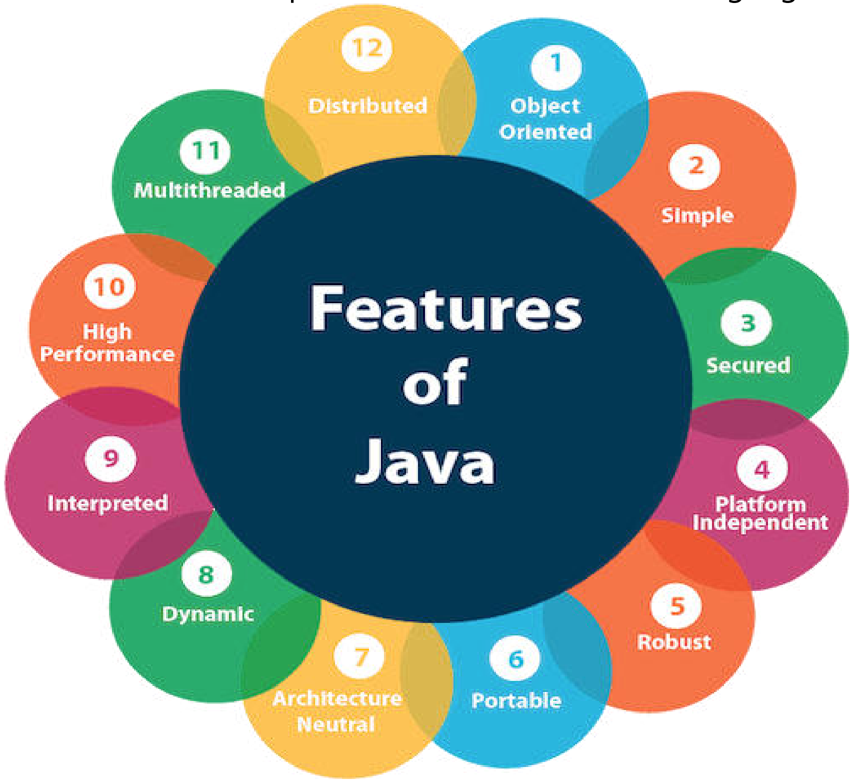
2. The features of Java

Or

Buzzwords of java

Ans:

Java is a general-purpose, class-based, object-oriented programming language.
The principles for creating Java programming were "Simple, Robust, Portable, Platform independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object Oriented, Interpreted, and Dynamic".
The features of Java are also known as Java buzzwords.
A list of the most important features of the Java language is given below.



Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

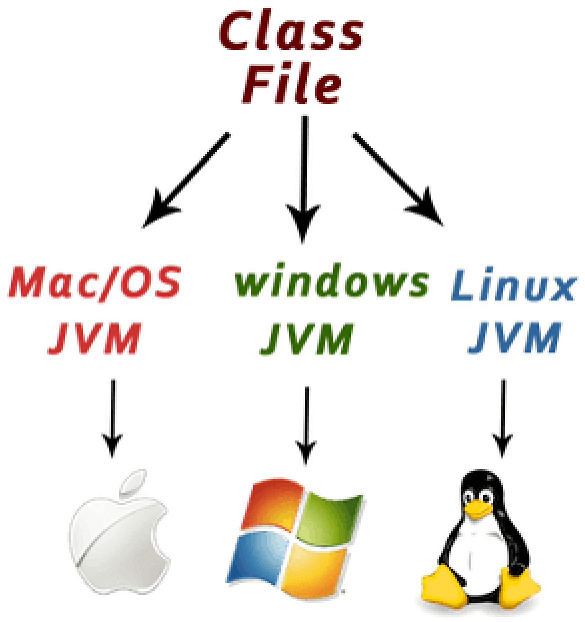
- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.
Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.
Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent



Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.
There are two types of platforms software-based and hardware-based. Java provides a software-based platform.
The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

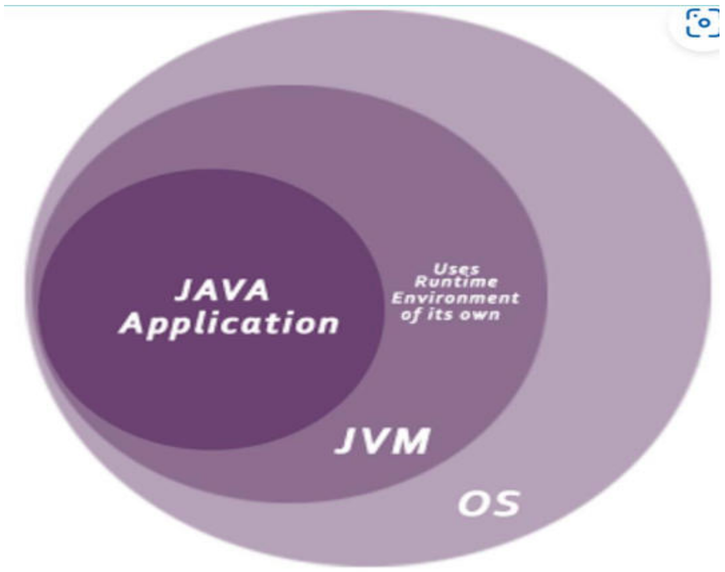
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside a virtual machine sandbox



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Robust

The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.

- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

3. Comparison between Procedure-Oriented and Object-Oriented Approach

Ans:

1. In the case of POP, the program is divided into small parts based on the functions. On the other hand, in OOP, the program is divided into objects, which are instances of classes.
2. In procedure-oriented programming, functions are the highest priority, and data is the lowest priority. Whereas in object-oriented programming, the data is a critical element.
3. The procedure-oriented approach is less secure in comparison to the object-oriented approach. In OOP, due to abstraction data hiding is possible, which makes it more secure.

4. Abstraction

Ans:

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is a non-access modifier, used for classes and methods:

Abstract class:

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method

Example of abstract class

```
abstract class A
{
}
```

Abstract Method in Java:

A method which is declared as abstract and does not have implementation is known as an abstract method.

```
abstract void printStatus();
```

Example of Abstract class that has an abstract method

```
abstract class Bike
{
  abstract void run();
}
class Honda4 extends Bike
{
void run()
{
System.out.println("running safely");
}
public static void main(String args[])
{
 Bike obj = new Honda4();
 obj.run();
}
     }
```

5. The OOP Principles
   (Or) java concepts (or) oops concepts
   Ans:

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

1)class
2) object
3)inheritance
4) encapsulation
5)polymorphism
6)data abstraction

1)Encapsulation:

Encapsulation is a process of wrapping code and data together into a single unit.
Encapsulation helps with data security, allowing you to protect the data stored in a class from system-wide access. As the name suggests, it safeguards the internal contents of a class like a capsule.

Encapsulation in Java:

- Restricts direct access to data members (fields) of a class
- Fields(data member) are set to private
- Each field has a getter and setter method
- Getter methods return the field
- Setter methods let us change the value of the field

2)Inheritance:

Process of creating new class from existing class is known as inheritance .

one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

Let us discuss some frequently used important terminologies:

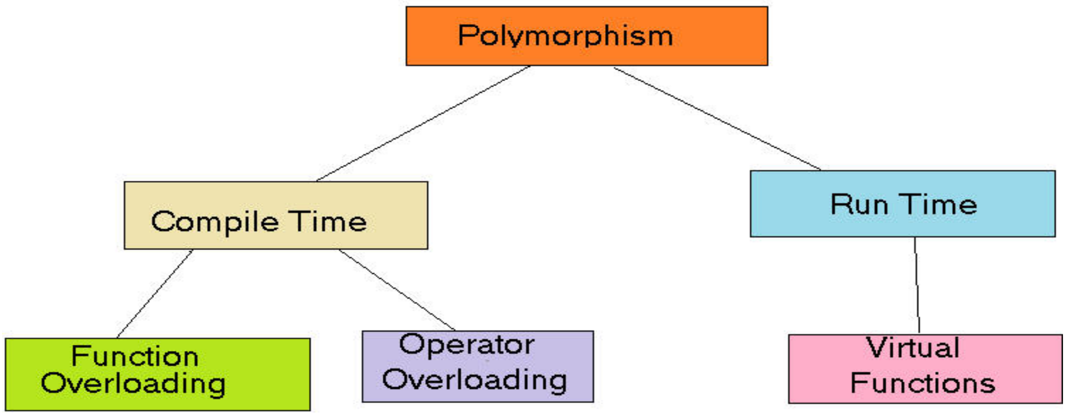
- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).
- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

3) polymorphism:

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
In Java, we use method overloading and method overriding to achieve polymorphism.
Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism



Compile-Time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Method Overloading

When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Polymorphism in Java:

- The same method name is used several times
- Different methods of the same name can be called from an object

- All Java objects can be considered polymorphic (at the minimum, they are of their own type and instances of the Object class)
- Static polymorphism in Java is implemented by method overloading
- Dynamic polymorphism in Java is implemented by method overriding

6. Java simple programs (or)
   What is program structure in java
   Or
   Programming Style
   Ans:
   we will learn how to write the simple program of Java. We can write a simple hello Java program easily after installing the JDK.
   To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.
   Let's create the hello java program:

```
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

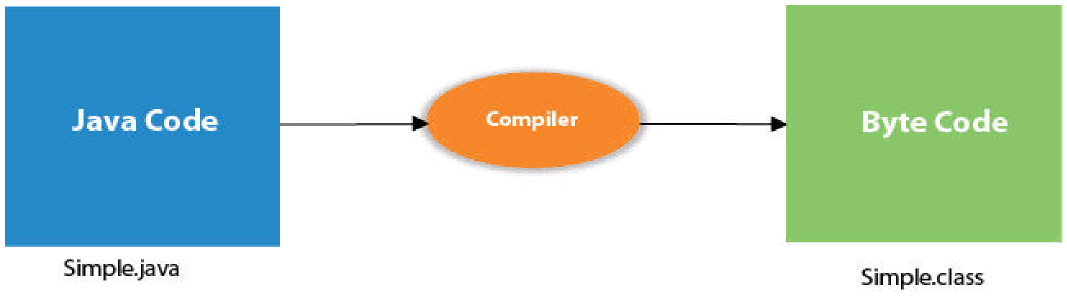
Save the above file as Simple.java.

To compile:	javac Simple.java
To execute:	java Simple

Output:

```
Hello Java
```

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method.

The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.

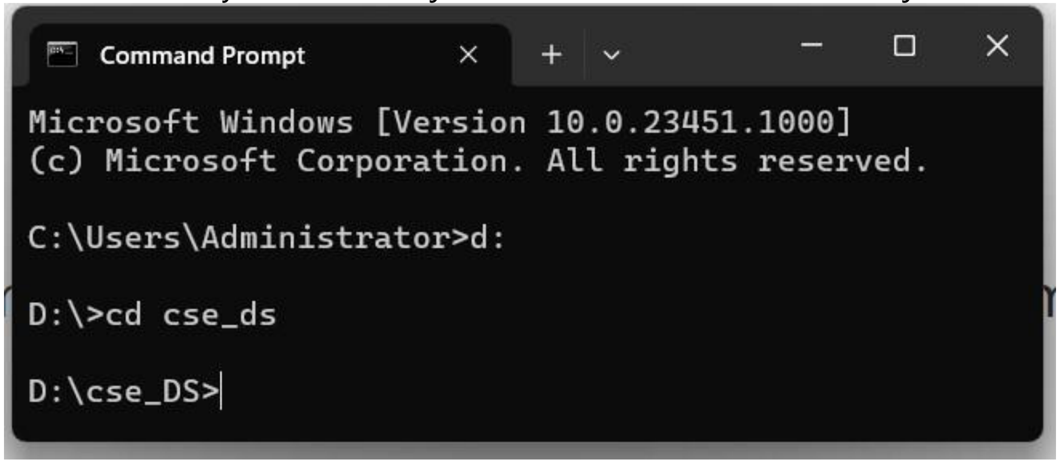
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

7. How to Compile and Run Java Program And:

how to compile and run java program step by step.
Step 1:
Write a program on the notepad and save it with **.java** (for example, DemoFile.java) extension.

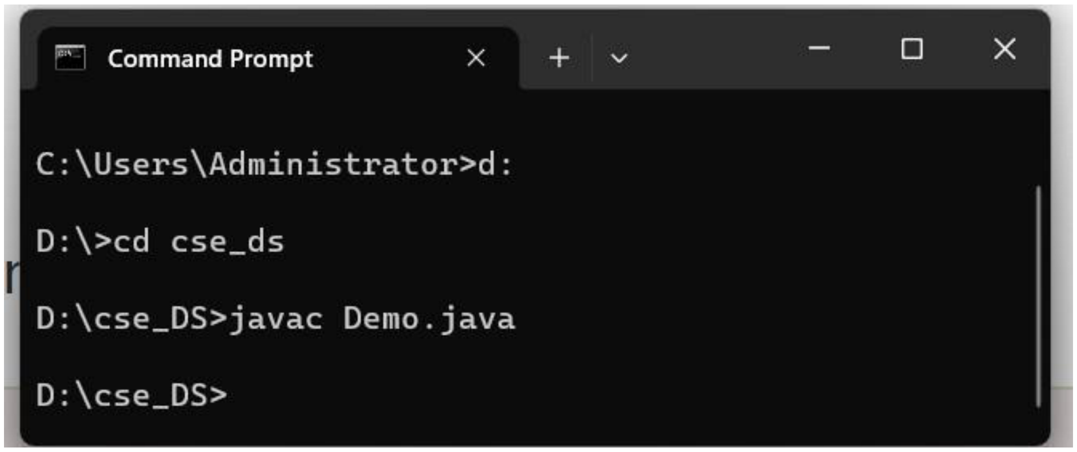
```
class Demo
{
public static void main(String args[])
{
System.out.println("Hello!");
System.out.println("Java");
}
}
```

Step 2:
Open Command Prompt.
Step 3:
Set the directory in which the .java file is saved. In our case, the .java file is saved in D:\cse_DS>Demo



```
Microsoft Windows [Version 10.0.23451.1000]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>d:

D:\>cd cse_ds

D:\cse_DS>
```

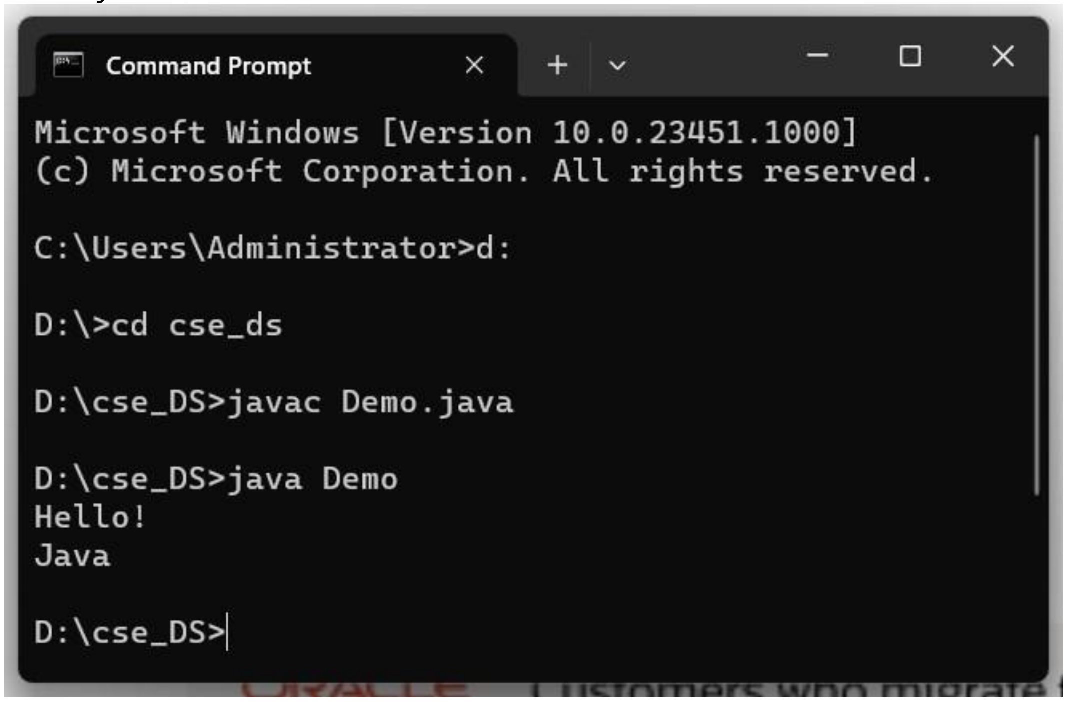
Step 4:
Use the following command to compile the Java program. It generates a .class file in the same folder. It also shows an error if any.
javac Demo.java



```
C:\Users\Administrator>d:

D:\>cd cse_ds

D:\cse_DS>javac Demo.java

D:\cse_DS>
```

Step 5:
Use the following command to run the Java program:

java Demo



```
Microsoft Windows [Version 10.0.23451.1000]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>d:

D:\>cd cse_ds

D:\cse_DS>javac Demo.java

D:\cse_DS>java Demo
Hello!
Java

D:\cse_DS>
```

8. Elements or Tokens in Java Program

Ans:

In Java, a **token** is the smallest individual unit(small part of code) in a program that the compiler recognizes. These include keywords, identifiers, literals, operators, and separators. Tokens can be classified as follows

1. **Keywords**: public, class, static, void, int
2. **Identifiers**: Example, main, args, a
3. **Literals**: 5, "Value of a: "
4. **Operators**: =, +
5. **Separators**: {, }, (,), ;, [], .

1. Keyword

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. **Java** language supports the following keywords:

abstract	**assert**	**boolean**
break	**byte**	**case**
catch	**char**	**class**
const	**continue**	**default**
do	**double**	**else**
enum	**exports**	**extends**
final	**finally**	**float**
for	**goto**	**if**
implements	**import**	**instanceof**
int	**interface**	**long**
module	**native**	**new**
open	**opens**	**package**
private	**protected**	**provides**
public	**requires**	**return**
short	**static**	**strictfp**
super	**switch**	**synchronized**
this	**throw**	**throws**
to	**transient**	**transitive**
try	**uses**	**void**
volatile	**while**	**with**

2. Identifiers

A method name, class name, variable name, or label is an identifier in Java. The user typically defines these. The identifier names cannot be the same as any reserved keyword. Let's see an example to understand identifiers:

```
public class Test
{
    public static void main(String[] args)
    {
        int num = 10;  // identifies
    }
}
```

There are rules for naming identifiers.

- The characters allowed are **[A-Z], [a-z], [0-9], _ and $.**
- Identifiers are case-sensitive. That is, "ninja" is not the same as "NINJA".
- Identifier names should not start with a digit. For example, "007IamNinja" is an invalid identifier.
- Whitespace is not allowed inside an identifier.
- Keywords can't be used as an identifier.

3. Constants/Literals

Literals represent fixed values in a source code. These are similar to standard variables with the difference that these are constant. These can be classified as an integer literal, a string literal, a boolean etc. The user defines these mainly to define constants.
Syntax to define literals:
final data_type variable_name;
There are five types of literals in Java:

- Integer
- Floating Point
- Boolean
- Character
- String

4. **Operators**
Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-

Arithmetic Operators
Unary Operators
Assignment Operator
Relational Operators
Logical Operators
Ternary Operator
Bitwise Operators
Shift Operators
instance of operator
Precedence and Associativity

5. Separators
Separators are used to separate different parts of the codes. It tells the compiler about completion of a statement in the program. The most commonly and frequently used separator in java is semicolon **(;).**

int variable;   //here the semicolon (;) ends the declaration of the variable

9. Java Statements

Ans:

In Java, each statement is a complete unit of execution. For example,
int score = 9*5;
Here, we have a statement. The complete execution of this statement involves multiplying integers 9 and 5 and then assigning the result to the variable score.
In the above statement, we have an expression 9 * 5. In Java, expressions are part of statements.

Expression statements

We can convert an expression into a statement by terminating the expression with a ;. These are known as expression statements. For example,

```
// expression
number = 10
// statement
number = 10;
```

In the above example, we have an expression number = 10. Here, by adding a semicolon (;), we have converted the expression into a statement (number = 10;).

Consider another example,

```
// expression
++number
// statement
++number;
```

10. Command Line Arguments

Ans:

The **command-line arguments** in Java allow us to pass arguments during the execution of the program.
In Java, **command line arguments** is a way to pass inputs to the java program during application execution. **Command line arguments** can be passed by multiple ways to Java application or program.

Program

```
 import java.io.*;
class commandlineargument

 {
        public static void main(String[] args)
        {
                int a=Integer.parseInt(args[0]); // commandline argument
                int b=Integer.parseInt(args[1]);
```

```
                    int c=a+b;
                    System.out.println("Sum of two variables "+ c);
                }
        }
        Output:
```
Compile:

```
        javac commandlineargument.java
```
Run:

```
        java commandlineargument 10 20
        Sum of two variables 30
```

11. User Input to Programs

Java provides different ways to get input from the user
There are two ways to use user input
1) Command line argument
2) Scanner class

The **command-line arguments** in Java allow us to pass arguments during the execution of the program.

In Java, **command line arguments** is a way to pass inputs to the java program during application execution. **Command line arguments** can be passed by multiple ways to Java application or program.

Program

```
        import java.io.*;
class commandlineargument
 {
    public static void main(String[] args)
    {
        // take input from the user
         int a=Integer.parseInt(args[0]); // commandline argument
         int b=Integer.parseInt(args[1]);
        int c=a+b;
        System.out.println("Sum of two variables "+ c);
    }
}
Output:
```
Compile:

```
javac commandlineargument.java
```
Run:

```
        java commandlineargument 10 20
Sum of two variables 30
```

12. Escape Sequences Comments

Ans:

tring that are preceded by a backslash (). These characters are referred to as Escape Sequences or Escape Characters, and they play a crucial role in various programming languages, including Java, Python, and C++.

Escape sequences in Java aim to represent those characters in a string that are difficult to represent directly. We can achieve certain formatting effects and can print certain effects using escape sequences in Java. The escape sequences must be enclosed in double quotes (""). There are a total of 8 escape sequences in Java.

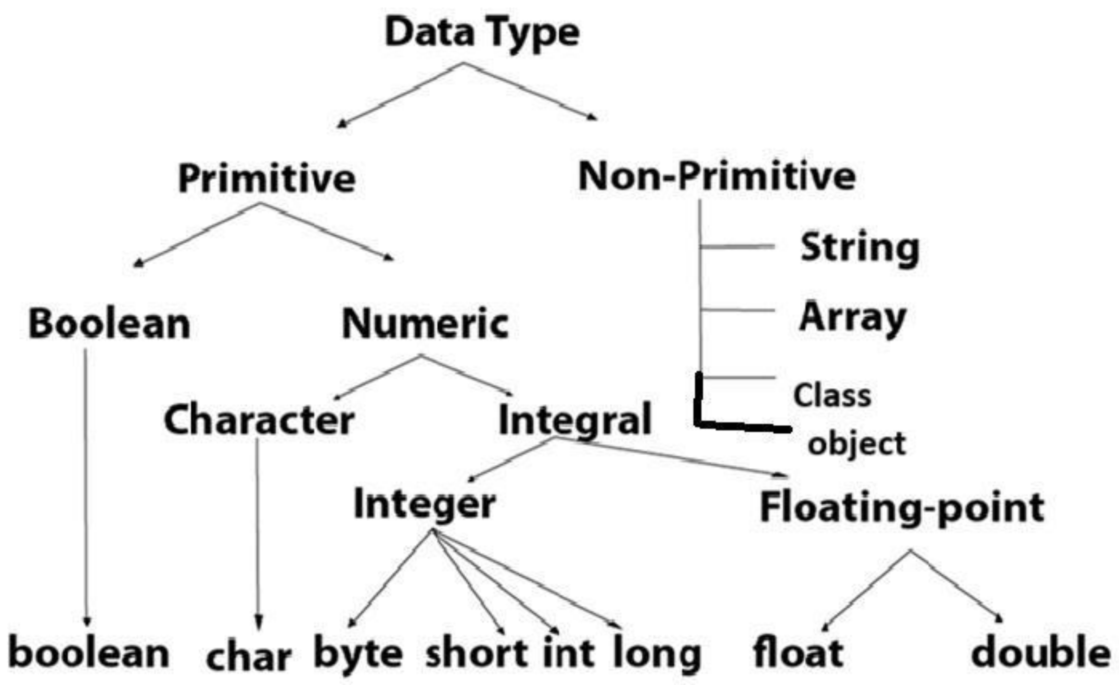
Escape Sequences	**Purpose**
\n	It is used to insert a new line.
\t	It is used to insert a tab.
\r	It is used to represent a carriage return.
\'	It is used to insert a single quotation mark.
\"	It is used to insert a double quotation mark.
\\	It is used to insert a backslash.
\f	It is used to insert a form feed.
\b	It is used to represent a backspace character.

13. Data Types in Java

Ans:

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays, string, object
3.



Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte

int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

boolean type

The boolean data type has two possible values, either true or false.
Boolean data type represents only one bit of information **either true or false** which is intended to represent the two truth values of logic and Boolean algebra
Default value: false.
They are usually used for true/false conditions.
The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.
Syntax:

```
boolean booleanVar;
```

```
example:
```

```
class Main

{

  public static void main(String[] args)

{

    boolean flag = true;

    System.out.println(flag);    // prints true

  }

}
```

2)Byte Data Type:

The byte data type can have values from -128 to 127 (8-bit signed two's complement integer).

If it's certain that the value of a variable will be within -128 to 127, then it is used instead of int to save memory.

Default value: 0

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Syntax:

```
byte byteVar;
```

```
class Main
```

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


    byte range;

    range = 124;

    System.out.println(range);    // prints 124

  }

}
```

3**. short type**:

The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.
The short data type in Java can have values from -32768 to 32767 (16-bit signed two's complement integer).
Default value: 0
The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Syntax:

```
short shortVar;
```

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

    short temperature;
    temperature = -200;
    System.out.println(temperature);  // prints -200
  }
}
```

4.Int Data Type:

It is a 32-bit signed two's complement integer.

If you are using Java 8 or later, you can use an unsigned 32-bit integer. This will have a minimum value of 0 and a maximum value of 232-1. To learn more, visit How to use the unsigned integer in java 8?

Default value: 0

Syntax:

```
int intVar;
```

example:

```
class Main {

  public static void main(String[] args) {

        int range = -4250000;

    System.out.println(range);  // print -4250000

  }

}
```

5.long type

The range of a long is quite large. The long data type is a 64-bit two’s complement integer and is useful for those occasions where an int type is not large enough to hold the desired value. The size of the Long Datatype is 8 bytes (64 bits).
Syntax:

```
long longVar;
```

example:

```
class LongExample {
  public static void main(String[] args) {
              long range = -42332200000L;
    System.out.println(range);    // prints -42332200000
  }
}
```

6. double type:

The double data type is a double-precision 64-bit floating-point.
It should never be used for precise values such as currency.
Default value: 0.0 (0.0d)

```
class Main {

  public static void main(String[] args) {

                double number = -42.3;

    System.out.println(number);  // prints -42.3

  }

}
```

7. float type:

The float data type is a single-precision 32-bit floating-point. Learn more about single-precision and double-precision floating-point if you are interested.
It should never be used for precise values such as currency.
Default value: 0.0 (0.0f)

```
class Main {
  public static void main(String[] args) {
                float number = 42.3f;
    System.out.println(number);  // prints 42.3
  }
}
```

8. char type

It's a 16-bit Unicode character.
The minimum value of the char data type is '\u0000' (0) and the maximum value of the is '\uffff'.
Default value: '\u0000'

```
class Main {
  public static void main(String[] args) {
        char letter = '\u0051';
    System.out.println(letter);  // prints Q
  }
}
```

(or)

```
class Main {
  public static void main(String[] args) {
             char letter1 = '9';
    System.out.println(letter1);  // prints 9
             char letter2 = 65;
    System.out.println(letter2);  // prints A

  }
}
```

Single program:

```
// Java Program to Demonstrate Char Primitive Data Type

// Class
class GFG {

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

        // Creating and initializing custom character
        char a = 'G';

        // Integer data type is generally
        // used for numeric values
```

```java
        int i = 89;

        // use byte and short
        // if memory is a constraint
        byte b = 4;

        // this will give error as number is
        // larger than byte range
        // byte b1 = 7888888955;

        short s = 56;

        // this will give error as number is
        // larger than short range
        // short s1 = 87878787878;

        // by default fraction value
        // is double in java
        double d = 4.355453532;

        // for float use 'f' as suffix as standard
        float f = 4.7333434f;

        // need to hold big range of numbers then we need
        // this data type
        long l = 12121;

        System.out.println("char: " + a);
        System.out.println("integer: " + i);
        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
        System.out.println("long: " + l);
    }
}
```
Output

```
char: G

integer: 89

byte: 4

short: 56

float: 4.7333436

double: 4.355453532

long: 12121
```

Non-Primitive Data Type or Reference Data Types
The **Reference Data Types** will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, objects, arrays, etc.
1. Strings
Strings are defined as an array of characters. The difference between a character array and a string in Java is, that the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char-type entities. Unlike C/C++, Java strings are not terminated with a null character.
Syntax: Declaring a string
<String_Type> <string_variable> = "<sequence_of_string>";

Example:
```
// Declare String without using new operator

String s = "GeeksforGeeks";



// Declare String using new operator

String s1 = new String("GeeksforGeeks");
```

2. Class
A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access. Refer to access specifiers for classes or interfaces in Java
2. **Class name:** The name should begin with an initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body is surrounded by braces, { }.

3. Object
An Object is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

4. Interface
Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.

- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

5. Array

An Array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays.

- In Java, all arrays are dynamically allocated. (discussed below)
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using size.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each has an index beginning with 0.
- Java array can also be used as a static field, a local variable, or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

14. Type Casting

Ans:

Type Casting

The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting.

In Java, there are 13 types of type conversion. However, in this tutorial, we will only focus on the major 2 types.

1. Widening Type Casting
2. Narrowing Type Casting

Widening Type Casting

In **Widening Type Casting**, Java automatically converts one data type to another data type.

Example: Converting int to double

```
class Main {
  public static void main(String[] args) {
    // create int type variable
    int num = 10;
    System.out.println("The integer value: " + num);

    // convert into double type
    double data = num;
    System.out.println("The double value: " + data);
  }
}
```

Output

```
he integer value: 10
The double value: 10.0
```

Narrowing Type Casting

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

Example: Converting double into an int

```
class Main {
  public static void main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);

    // convert into int type
    int data = (int)num;
    System.out.println("The integer value: " + data);
  }
}
```

Output

```
The double value: 10.99
The integer value: 10
```

15. What is variable and explain different types of variables

(or)

Scope of Variable Identifier

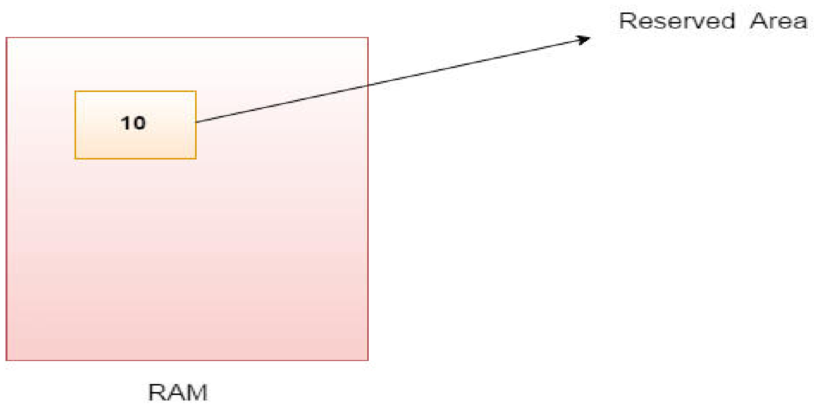
Ans:

Variable is name that can hold a data

A variable is a container which holds the value while the Java program is executed.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

A variable is assigned with a data type.

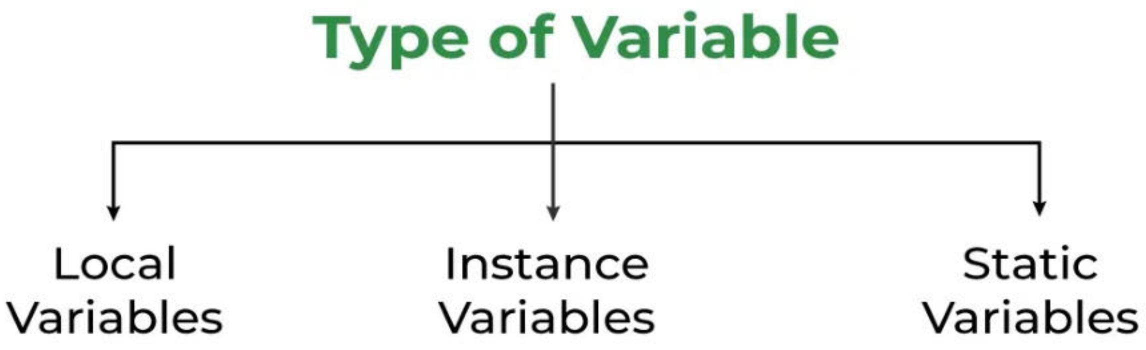


int data=50;//Here data is variable

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable



1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

```
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        // Declared a Local Variable
        int var = 10;

        // This variable is local to this main method only
        System.out.println("Local Variable: " + var);
    }
}
```

Output:

```
Local Variable: 10
```

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local.

You can create a single copy of the static variable and share it among all the instances of the class.

Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
public class A
{
   static int m=100;//static variable
   void method()
   {
      int n=90;//local variable
   }
   public static void main(String args[])
   {
      int data=50;//instance variable
   }
}//end of class
```

16. Literal Constants

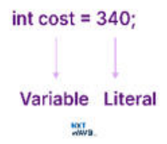
Ans:

Literals in Java

Java, a Literal is value of boolean, numeric, character, or string data. Any constant value that can be assigned to the variable is called a literal.

// Here 100 is a constant/literal.

int x = 100;



Types of Literals in Java

Java supports the following types of literals:

- Integral Literals
- Floating-Point Literals
- Char Literals
- String Literals
- Boolean Literals

Integral Literals in Java

For Integral data types (byte, short, int, long), we can specify literals in four ways, which are listed below:

1. Decimal literals (Base 10): In this form, the allowed digits are 0-9.

int x = 101;

2. Octal literals (Base 8): In this form, the allowed digits are 0-7.

// The octal number should be prefix with 0.

int x = 0146;

3. Hexadecimal literals (Base 16): In this form, the allowed digits are 0-9, and characters are a-f. We can use both uppercase and lowercase characters, as we know that Java is a case-sensitive programming language, but here Java is not case-sensitive.

// The hexa-decimal number should be prefix

// with 0X or 0x.

int x = 0X123Face;

4. Binary literals: From 1.7 onward, we can specify literal value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.

int x = 0b1111;

Example:

```
public class Geeks {
  public static void main(String[] args)
  {
     // decimal-form literal
    int a = 101;
     // octal-form literal
    int b = 0100;
    // Hexa-decimal form literal
    int c = 0xFace;
     // Binary literal
    int d = 0b1111;

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
```

```
        }
    }
```
Floating-Point Literal in Java

For Floating-point data types, we can specify literals in only decimal form, and we can not specify in octal and Hexadecimal forms.

1. Decimal literals(Base 10): In this form, the allowed digits are 0-9.

double d = 123.456;

Example:

Char Literals in Java

For char data types, we can specify literals in four ways which are listed below:

1. Single quote: We can specify literal to a char data type as a single character within the single quote.

char ch = 'a';

2. Char literal as Integral literal: we can specify char literal as integral literal, which represents the Unicode value of the character, and that integral literal can be specified either in Decimal, Octal, and Hexadecimal forms. But the allowed range is 0 to 65535.

char ch = 062; // Octal literal representing character with Unicode code 50 (which is '2')

Note: If invalid digits are used for octal (means digit other than 0=7), it will it will cause a compile-time error, for example:

char b = 0789; // Invalid octal literal due to digits 8 and 9 — causes compile-time error

3. Unicode Representation: We can specify char literals in Unicode representation '\uxxxx'. Here xxxx represents 4 hexadecimal numbers.

char ch = '\u0061';// Here /u0061 represent a.

4. Escape Sequence: Every escape character can be specified as char literals.

char ch = '\n';

String Literals in Java

Any sequence of characters within double quotes is treated as String literals.

String s = "Hello";

String literals may not contain unescaped newline or linefeed characters. However, the Java compiler will evaluate compile-time expressions, so the following String expression results in a string with three lines of text.

Example:

```
String text = "This is a String literal\n"
+ "which spans not one and not two\n"
+ "but three lines of text.\n";
```

Boolean Literals in Java

Only two values are allowed for Boolean literals, i.e., true and false.

boolean b = true;

boolean c = false;

17. Symbolic Constants

Ans:

Java - symbolic constants
In Java, a symbolic constant is a named constant value defined once and used throughout a program. Symbolic constants are declared using the final keyword.

- Which indicates that the value cannot be changed once it is initialized.
- The naming convention for symbolic constants is to use all capital letters with underscores separating words.

Syntax of Symbolic Constants
final data_type CONSTANT_NAME = value;

- **final**: The final keyword indicates that the value of the constant cannot be changed once it is initialized.
- **data_type**: The data type of the constant such as int, double, boolean, or String.
- **CONSTANT_NAME**: The name of the constant which should be written in all capital letters with underscores separating words.
- **value:** The initial value of the constant must be of the same data type as the constant.

Initializing a symbolic constant:
final double PI = 3.14159;

18. Formatted Output with printf() Method

Ans:

Print some formatted text to the console.
The %s character is a placeholder for the string "World":
System.out.printf("Hello %s!", "World");
The printf() method outputs a formatted string.
Data from the additional arguments is formatted and written into placeholders in the formatted string, which are marked by a % symbol. The way in which arguments are formatted depends on the sequence of characters that follows the % symbol.

19. **Understanding static (or) static keyword**

Ans:

if we want to access class members, we must first create an instance of the class. But there will be situations where we want to access class members without creating any variables.

In those situations, we can use the static keyword in Java. If we want to access class members without creating an instance of the class, we need to declare the class members static.
The Math class in Java has almost all of its members static. So, we can access its members without creating instances of the Math class. For example,

The static can be:

1. Variable
2. Method
3. Block

4. Nested class

1) Java static variable:

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student
{
            int rollno;//instance variable
             String name;
             static String college ="ITS";//static variable
              //constructor
              Student(int r, String n)
           {
                         rollno = r;
                      name = n;
            }
           //method to display the values
            void display ()
           {
                      System.out.println(rollno+" "+name+" "+college);
           }
}
//Test class to show the values of objects
public class TestStaticVariable1
{
            public static void main(String args[])
           {
                       Student s1 = new Student(111,"Karan");
                       Student s2 = new Student(222,"Aryan");
                       //we can change the college of all objects by the single line of code
                       //Student.college="BBDIT";
                       s1.display();
                       s2.display();
            }
}
```

2) Java static method:

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student
{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change()
         {
            college = "BBDIT";
            }
         //constructor to initialize the variable
           Student(int r, String n)
         {
                     rollno = r;
                    name = n;
          }
    //method to display values
    void display()
         {
                   System.out.println(rollno+" "+name+" "+college);
         }
}
//Test class to create and display the values of object
public class TestStaticMethod
{
         public static void main(String args[])
         {
            Student.change();//calling change method
            //creating objects
            Student s1 = new Student(111,"Karan");
           Student s2 = new Student(222,"Aryan");
            Student s3 = new Student(333,"Sonoo");
          //calling display method
          s1.display();
            s2.display();
            s3.display();
          }
}
```

3) **Java static block**:

Is used to initialize the static data member.

It is executed before the main method at the time of classloading.

```
class A2
{
            Static
                   {
                            System.out.println("static block is invoked");
                   }
           public static void main(String args[])
          {
            System.out.println("Hello main");
          }
}
```

Output:static block is invoked

Hello main

4) **static class**

A class can be made static only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

```
import java.io.*;
 public class test
{
    private static String str = "GeeksforGeeks";
    // Static class
   static class MyNestedClass
          {
                    // non-static method
                    public void disp()
                   {
                   System.out.println(str);
                    }
           }
     public static void main(String args[])
          {
                    test.MyNestedClass obj    = new test.MyNestedClass();
                   obj.disp();
          }
}
```

Output: GeeksforGeeks

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

20. **Introducing final . (or) final keyword**

Ans:

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes. Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

```
class Main
 {
  public static void main(String[] args)
{
            // create a final variable
            final int AGE = 32;
             // try to change the final variable
              AGE = 45;
             System.out.println("Age: " + AGE);
    }
}
```

2. Java final Method

Before you learn about final methods and final classes, make sure you know about the Java Inheritance.

In Java, the final method cannot be overridden by the child class. For example,

```
class FinalDemo
{
        // create a final method
         public final void display()
{
                System.out.println("This is a final method.");
         }
}

class Main extends FinalDemo
{
        // try to override final method
        public final void display()
{
         System.out.println("The final method is overridden.");
  }

  public static void main(String[] args)
{
   Main obj = new Main();
                         obj.display();
 }
}
```

3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass
{
  public void display()
 {
                        System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass
 {
  public  void display()
{
                                       System.out.println("The final method is overridden.");
  }

  public static void main(String[] args)
{
   Main obj = new Main();
                        obj.display();
             }
}
```

21. operator

Ans:

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:
incrementing/decrementing a value by one.
negating an expression.
inverting the value of a Boolean.

Example:

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

```
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Arithmetic Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15

5

50

2

0
```

Arithmetic Operator Example:

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Example

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

Output:

```
true
true
true
10
true
11
```

Ternary Operator:

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Example

```
public class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Example:

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

22. Control statement (or) control flow statement

Ans:

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement

- continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement

```
if(condition) {
statement 1; //executes when condition is true
}
```

Example:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Example:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
1.  }
2.  }
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
```

```
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";
    if(address.endsWith("India")) {
```

```
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
   case value1:
    statement1;
    break;
    .
```

```
        .
        .
        case valueN:
         statementN;
         break;
        default:
         default statement;
    }
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

Example:

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.
It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){
//looping statements
}
```

Example

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

```
Printing the list of first 10 even numbers


0
2
4
6
8
10
```

Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do
{
//statements
} while (condition);
```

Example

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.
The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.
The break statement example with for loop
Consider the following example in which we have used the break statement with the for loop.
BreakExample.java

```
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
    for(int i = 0; i<= 10; i++) {
    System.out.println(i);
    if(i==6) {
    break;
    }
    }
    }
    }
```

Output:

```
0
1
```

```
2
3
4
5
6
```

continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample
{
        public static void main(String[] args)
         {
                // TODO Auto-generated method stub
                for(int i = 0; i<= 2; i++)
                 {
                        for (int j = i; j<=5; j++)
                        {

                                if(j == 4)
                                {
                                        continue;
                                }
                                System.out.println(j);
                        }
                }
        }
 }
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

UNIT-II

23.what is object

ans:

In object-oriented programming technique, we design a program using objects and classes.
An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:
State: represents the data (value) of an object.
Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.
An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
Object Definitions:

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behavior.
- The object is an instance of a class.

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference
Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

```
class Student{
 int rollno;  // class members
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation()
{System.out.println(rollno+" "+name);
}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

```
111 Karan
222 Aryan
```

2) **Object and Class Example: Initialization through a constructor**

An constructor is similar to method

Syntx of constructor : class name with parenthesis

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();  // Employee() is constructor
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

24. What is class

(or)

Declaration of Class Objects

(0r)

Class Members

Ans:

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
```

}

Instance variable

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method:

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

25. Assigning One Object to Another

Ans:

It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of another object.

```
 class GFG {
   int x, y;

   // Constructor to initialize
   // object fields
   GFG() {
      x = 10;
      y = 20;
   }
}

public class Main1 {
  public static void main(String[] args)
   {
      GFG o1 = new GFG();

      // Cloning obj1 into obj2
      GFG o2 = new GFG();
              o2=o1;

      System.out.println("o1: " + o1.x + " " + o1.y);
      System.out.println("o2: " + o2.x + " " + o2.y);
   }
}
```

output:

```
D:\cse>javac Main1.java

D:\cse>java Main1
o1: 10 20
o2: 10 20
```

26. Access Modifiers in Java

(Or)access control

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.
access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods.
There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

1) Private:
   The private access modifier is accessible only within the class.

we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}

public class Simple
{
         public static void main(String args[])
        {
                 A obj=new A();
                System.out.println(obj.data);//Compile Time Error
                 obj.msg();//Compile Time Error
         }
}
```

2) Default:
If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered.
If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

```
//save by A.java
package pack;
class A
{
         void msg()
        {
                System.out.println("Hello");
        }
}

//save by B.java
package mypack;
import pack.*;
class B
{
         public static void main(String args[])
        {
                A obj = new A();//Compile Time Error
                obj.msg();//Compile Time Error
         }
}
```

3) Protected:
The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

```
class Animal
{
   // protected method
   protected void display()
  {
       System.out.println("I am an animal");
   }
}

class Dog extends Animal
 {
          public static void main(String[] args)
         {

             // create an object of Dog class
          Dog dog = new Dog();
          // access protected method
          dog.display();
   }
}
```

4) Public:

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction. For example,

```
// A.java file
// public class
class A
{
         public int data=40;
         public void msg()
         {
                  System.out.println("Hello java");
         }
}

public class Simple
{
          public static void main(String args[])
         {
                   A obj=new A();
                  System.out.println(obj.data);
                   obj.msg();
          }
```

}

27. **method**

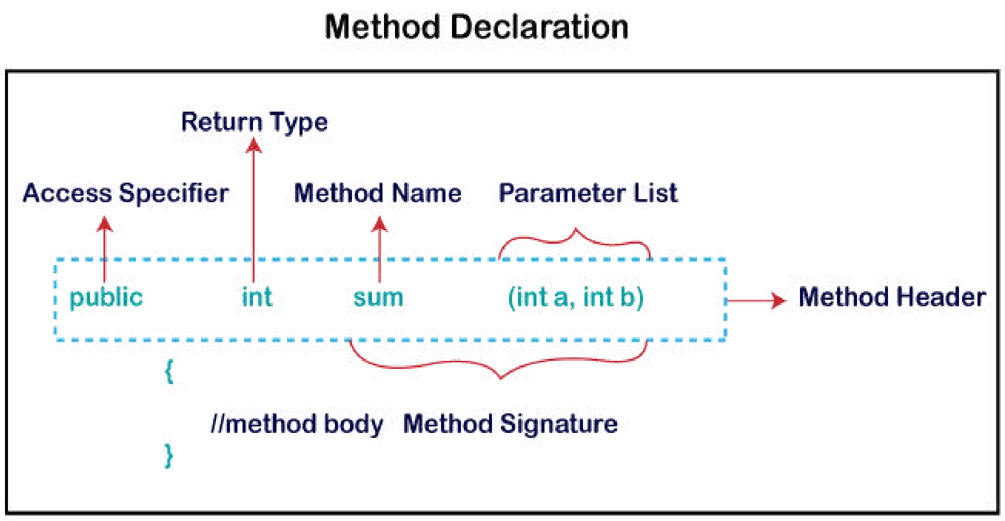
ans:

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.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as method header, as we have shown in the following figure.



Syntax:

type name(parameter-list)

{ // body of method }

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Types of Method
There are two types of methods in Java

- Predefined Method
- User-defined Method

Predefined Method:
n Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.
Each and every predefined method is defined inside a class. Such as print() method is defined in the java.io.PrintStream class. It prints the statement that we write inside the method. For example, print("Java"), it prints Java on the console.

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

User-defined Method
The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.
How to Create a User-defined Method
Let's create a user defined method that checks the number is even or odd. First, we will define the method

```
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

28. Passing Arguments by Value and by Reference

Or

Adding a Method That Takes Parameters

Ans:

Parameter passing in Java refers to the mechanism of transferring data between methods or functions. Java supports two types of parameters passing techniques

1. Call-by-value
2. Call-by-reference.

Understanding these techniques is essential for effectively utilizing method parameters in Java.

Types of Parameters:

1. Formal Parameter:

A variable and its corresponding data type are referred to as formal parameters when they exist in the definition or prototype of a function or method. As soon as the function or method is called and it serves as a placeholder for an argument that will be supplied. The function or method performs calculations or actions using the formal parameter.

Syntax:

```
returnType functionName(dataType parameterName)

  {
     // Function body
     // Use the parameterName within the function
  }
```

In the above syntax:

- returnType represents the return type of the function.
- functionName represents the name of the function.
- dataType represents the data type of the formal parameter.
- parameterName represents the name of the formal parameter.

2. Actual Parameter:

The value or expression that corresponds to a formal parameter and is supplied to a function or method during a function or method call is referred to as an actual parameter is also known as an argument. It offers the real information or value that the method or function will work with.

Syntax:

functionName(argument)

In the above syntax:

- functionName represents the name of the function or method.
- argument represents the actual value or expression being passed as an argument to the function or method.

1. Call-by-Value:

In Call-by-value the copy of the value of the actual parameter is passed to the formal parameter of the method. Any of the modifications made to the formal parameter within the method do not affect the actual parameter.

```
class Student{
 int rollno;  // class members
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation()
{System.out.println(rollno+" "+name);
}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");  // 111, karan are call by values
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

```
111 Karan
```

```
222 Aryan
```

Call-by-Reference:

call by reference" is a method of passing arguments to functions or methods where the memory address (or reference) of the variable is passed rather than the value itself. This means that changes made to the formal parameter within the function affect the actual parameter in the calling environment.

In "call by reference," when a reference to a variable is passed, any modifications made to the parameter inside the function are transmitted back to the caller. This is because the formal parameter receives a reference (or pointer) to the actual data.

```java
class Student{
 int rollno;  // class members
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation()
{System.out.println(rollno+" "+name);
}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();  // s1 is reference object of student class
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

```
111 Karan
222 Aryan
```

29. Constructors

Ans:

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
It is a special type of method which is used to initialize the object.
Every time an object is created using the new() keyword, at least one constructor is called.
It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor
There are two rules defined for the constructor.

Constructor name must be the same as its class name
A Constructor must have no explicit return type
A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default constructor (no-arg constructor)

```
//Let us see another example of default constructor
//which displays the default values
class Student3
{
    int id;
    String name;
    //method to display the value of id and name
    void display()
    {
            System.out.println(id+" "+name);
    }

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

```
        {
                //creating objects
                Student3 s1=new Student3();
                Student3 s2=new Student3();
            //displaying values of the object
            s1.display();
            s2.display();
        }
    }
```

Output:

```
0 null
0 null
```

Parameterized Constructors

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4
{
  int id;
  String name;
  //creating a parameterized constructor
  Student4(int i,String n)
{
          id = i;
          name = n;
         }
  //method to display the values
  void display(){System.out.println(id+" "+name);
}

  public static void main(String args[])
{
                //creating objects and passing values
                Student4 s1 = new Student4(111,"Karan");
                 Student4 s2 = new Student4(222,"Aryan");
                 //calling method to display the values of object
                s1.display();
                 s2.display();
  }
}
```

Output:

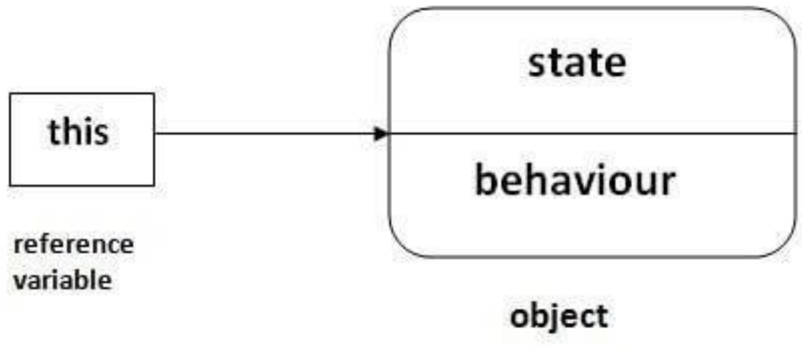
```
111 Karan
222 Aryan
```

30. The this Keyword

Ans:

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.



Usage of Java this keyword
Here is given the 6 usage of java this keyword.
We can use this keyword for the following purpose.

1) this keyword is used to refer to current object.
2) this is always a reference to the object on which method was invoked.
3) this can be used to invoke current class constructor.
4) this can be passed as an argument to another method.

1) this keyword is used to refer to current object.
   The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
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);}
}

class TestThis2
{
    public static void main(String args[])
    {
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example

```
    class A
    {
        void m()
        {
        System.out.println("hello m");
        }
        void n()
        {
                System.out.println("hello n");
            //m();//same as this.m()
            this.m();
        }
    }
    class TestThis4
    {
        public static void main(String args[])
        {
                A a=new A();
                a.n();
        }
    }
```

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
class A
{
        A()
        {
                System.out.println("hello a");
        }
        A(int x)
        {
                this();
                System.out.println(x);
        }
}
class TestThis5
{
        public static void main(String args[])
        {
                A a=new A(10);
        }
}
```

Output:

```
hello a
10
```

31. Finalize() method

Ans:

finalize() method in Java is a method of the Object class that is used to perform cleanup activity before destroying any object. It is called by Garbage collector before destroying the objects from memory.
finalize() method is called by default for every object before its deletion. This method helps Garbage Collector to close all the resources used by the object and helps JVM in-memory optimization.

Syntax

```
protected void finalize() throws Throwable
```

Throw

Throwable - the Exception is raised by this method

Example 1

```
public class JavafinalizeExample1
{
    public static void main(String[] args)
   {
      JavafinalizeExample1 obj = new JavafinalizeExample1();
      System.out.println(obj.hashCode());
      obj = null;
      // calling garbage collector
      System.gc();
      System.out.println("end of garbage collection");

   }
   @Override
   protected void finalize()
   {
      System.out.println("finalize method called");
   }
}
```

32. Overloading Methods

Ans:

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading.**

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.
So, we perform method overloading to figure out the program quickly.

Advantage of method overloading
Method overloading increases the readability of the program.

Different ways to overload the method
There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments
In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder
{
        static int add(int a,int b)
        {
                return a+b;
        }
        static int add(int a,int b,int c)
        {
                return a+b+c;
}
}
class TestOverloading1
{
        public static void main(String[] args)
        {
                System.out.println(Adder.add(11,11));
                System.out.println(Adder.add(11,11,11));
        }
}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments
In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder
{
        static int add(int a, int b)
        {
                return a+b;
        }
        static double add(double a, double b)
        {
                return a+b;
        }
}
class TestOverloading2
{
        public static void main(String[] args)
        {
                System.out.println(Adder.add(11,11));
                System.out.println(Adder.add(12.3,12.6));
        }
}
Output:
22
24.9
```

33. Overloading Constructors
    ans:

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.

Example

```
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);
               }
          }
```

Output:

```
this a default constructor

Default Constructor values:

Student Id : 0
Student Name : null

Parameterized Constructor values:

Student Id : 10
Student Name : David
```

34. Using Objects as Parameters

Or

Class Objects as Parameters in Methods

Ans:

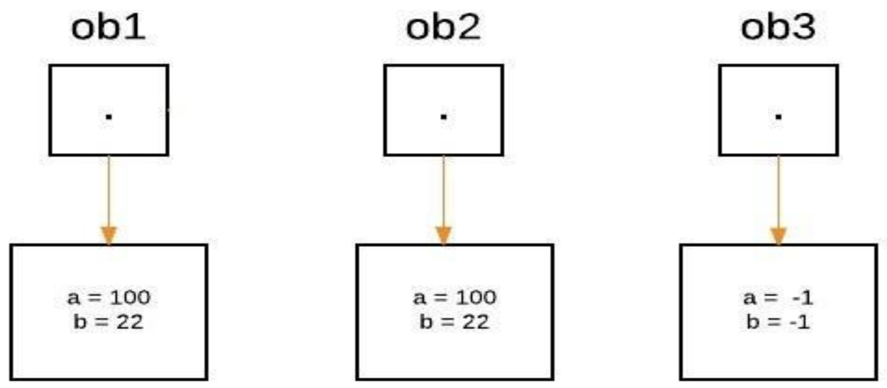
Although Java is strictly passed by value, the precise effect differs between whether a primitive type or a reference type is passed. When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.

Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect the object used as an argument.

Illustration: Let us suppose three objects 'ob1' , 'ob2' and 'ob3' are created:

```
ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
```



```java
// Class
// Helper class
class ObjectPassDemo {
    int a, b;

    // Constructor
    ObjectPassDemo(int i, int j)
    {
        a = i;
        b = j;
    }

    // Method
    boolean equalTo(ObjectPassDemo o)
    {
        // Returns true if o is equal to the invoking
        // object notice an object is passed as an
        // argument to method
        return (o.a == a && o.b == b);
    }
}

// Main class
public class GFG {
    // MAin driver method
    public static void main(String args[])
    {
        // Creating object of above class inside main()
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);

        // Checking whether object are equal as custom
        // values
        // above passed and printing corresponding boolean
        // value
        System.out.println("ob1 == ob2: "
                           + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: "
                           + ob1.equalTo(ob3));
    }
}
```

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

anther Example

```
class Add {
  int a;
  int b;

  Add(int x, int y) // parametrized constructor
  {
    a = x;
    b = y;
  }
  void sum(Add A1) // object  'A1' passed as parameter in function 'sum'
  {
    int sum1 = A1.a + A1.b;
    System.out.println("Sum of a and b :" + sum1);
  }
}

public class Main {
  public static void main(String arg[]) {
    Add A = new Add(5, 8);
    /* Calls  the parametrized constructor
    with set of parameters*/
    A.sum(A);
  }
}
```

35. Recursion

Ans:

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial
{
        // this is a recursive method
        int fact(int n)
         {
                int result;
                if(n==1) return 1;
                result = fact(n-1) * n;
                return result;
        }
}
class Recursion
{
        public static void main(String args[])
         {
                Factorial f = new Factorial();
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of fact( ) may seem a bit confusing. Here is how it works. When fact( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of fact(n–1)*n. To evaluate this expression, fact( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning

To better understand how the fact( ) method works, let's go through a short example.

When you compute the factorial of 3, the first call to fact( ) will cause a second call to be made with an argument of 2.

This invocation will cause fact( ) to be called a third time with an argument of 1.

This call will return 1, which is then multiplied by 2 (the value of n in the second invocation).

This result (which is 2) is then returned to the original invocation of fact( ) and multiplied by 3 (the original value of n). This yields the answer, 6.

You might find it interesting to insert println( ) statements into fact( )

36. Nested class

Ans:

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

Example

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```

37. Nesting of Methods

Ans:

If declare method inside another method is know nested method

Syntax:

```
class Main
{
    method1(){

        // statements
    }

    method2()
    {
       // statements

      // calling method1() from method2()
       method1();
    }
    method3()
    {
      // statements

      // calling of method2() from method3()
      method2();
    }
}
```

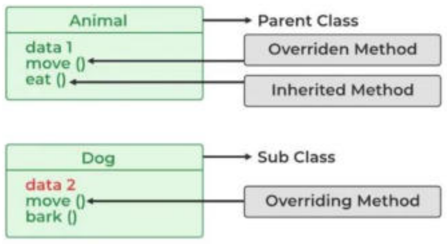
38. Overriding Methods

Ans:

Overriding in Java occurs when a **subclass or child class implements a method that is already defined in the superclass or base class**. When a subclass provides its own version of a method that is already defined in its superclass, we call it **method overriding.** The subclass method must match the parent class method's **name, parameters, and return type.**

```
// Example of Overriding in Java
class Animal {
   // Base class
   void move() { System.out.println(
     "Animal is moving."); }
   void eat() { System.out.println(
     "Animal is eating."); }
}

class Dog extends Animal {
   @Override void move()
   { // move method from Base class is overriden in this
     // method
       System.out.println("Dog is running.");
   }
   void bark() { System.out.println("Dog is barking."); }
}

public class Geeks {
   public static void main(String[] args)
   {
       Dog d = new Dog();
       d.move(); // Output: Dog is running.
       d.eat(); // Output: Animal is eating.
       d.bark(); // Output: Dog is barking.
   }
}
```
Output

```
Dog is running.
Animal is eating.
Dog is barking.
```
Explanation: The Animal class defines base functionalities like **move()** and **eat()**. The Dog class inherits from Animal and **overrides** the move() method to provide a specific behavior **Dog is running.** Both classes can access their own methods. When creating a Dog object, calling move() executes the overridden method.



39. Understanding static (or) static keyword

Ans:

if we want to access class members, we must first create an instance of the class. But there will be situations where we want to access class members without creating any variables.

In those situations, we can use the static keyword in Java. If we want to access class members without creating an instance of the class, we need to declare the class members static.

The Math class in Java has almost all of its members static. So, we can access its members without creating instances of the Math class. For example,

The static can be:

5. Variable
6. Method
7. Block
8. Nested class

1) Java static variable:

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student
{
	 int rollno;//instance variable
	  String name;
	  static String college ="ITS";//static variable
	   //constructor
	   Student(int r, String n)
	{
		   rollno = r;
		name = n;
	 }
	//method to display the values
	 void display ()
	{
		System.out.println(rollno+" "+name+" "+college);
	}
}
//Test class to show the values of objects
public class TestStaticVariable1
{
	 public static void main(String args[])
	{
		 Student s1 = new Student(111,"Karan");
		 Student s2 = new Student(222,"Aryan");
		 //we can change the college of all objects by the single line of code
		 //Student.college="BBDIT";
		 s1.display();
```

```
                s2.display();
        }
}
```

2) Java static method:

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student
{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change()
        {
           college = "BBDIT";
           }
        //constructor to initialize the variable
          Student(int r, String n)
        {
                    rollno = r;
                   name = n;
         }
    //method to display values
    void display()
        {
                 System.out.println(rollno+" "+name+" "+college);
        }
}
//Test class to create and display the values of object
public class TestStaticMethod
{
        public static void main(String args[])
        {
           Student.change();//calling change method
           //creating objects
           Student s1 = new Student(111,"Karan");
```

```
             Student s2 = new Student(222,"Aryan");
              Student s3 = new Student(333,"Sonoo");
            //calling display method
            s1.display();
              s2.display();
              s3.display();
             }
}
```

3) **Java static block**:

Is used to initialize the static data member.

It is executed before the main method at the time of classloading.

```
class A2
{
            Static
                    {
                              System.out.println("static block is invoked");
                    }
           public static void main(String args[])
          {
             System.out.println("Hello main");
            }
}
```

```
Output:static block is invoked
       Hello main
```

4) static class

A class can be made static only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

```
import java.io.*;
 public class test
{
    private static String str = "GeeksforGeeks";
    // Static class
   static class MyNestedClass
          {
                      // non-static method
                      public void disp()
                     {
                     System.out.println(str);
                       }
           }
     public static void main(String args[])
          {
                      test.MyNestedClass obj    = new test.MyNestedClass();
                     obj.disp();
          }
}
Output: GeeksforGeeks
```

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

40. Introducing final . (or) final keyword

Ans;

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes. Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

```
class Main
 {
  public static void main(String[] args)
{
                    // create a final variable
                    final int AGE = 32;
                     // try to change the final variable
                       AGE = 45;
                     System.out.println("Age: " + AGE);
        }
}
```

2. Java final Method

Before you learn about final methods and final classes, make sure you know about the Java Inheritance.

In Java, the final method cannot be overridden by the child class. For example,

```
class FinalDemo
{
       // create a final method
        public final void display()
      {
                      System.out.println("This is a final method.");
        }
}

class Main extends FinalDemo
{
       // try to override final method
       public final void display()
      {
```

```
                    System.out.println("The final method is overridden.");
        }

        public static void main(String[] args)
                {
                   Main obj = new Main();
                obj.display();
              }
}
```

3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass
{
        public void display()
       {
                  System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass
 {
        public  void display()
                  {
                                 System.out.println("The final method is overridden.");
                    }
              public static void main(String[] args)
            {
                      Main obj = new Main();
                  obj.display();
       }
}
```