ANNAMACHARYA UNIVERSITY

Estd. under Andhra Pradesh
Private Universities (Establishment and Regulation) Act, 2016
(University listed in UGC as per section 2(f) of the UGC Act, 1956)

LECTURE NOTES
ON
Object Oriented Programming using Java
(24ACSE32T)

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

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ANNAMACHARYA UNIVERSITY

Estd. under Andhra Pradesh
Private Universities (Establishment and Regulation) Act, 2016
(University listed in UGC as per section 2(f) of the UGC Act, 1956)

Title of the Course:	Object Oriented Programming using Java
Category:	Professional Core
Couse Code:	24ACSE32T
Year:	II B. Tech
Semester:	I Semester
Branch:	AI& DS, AI&ML, CSE, CSE(AI), CSE(AIML), CSE(DS) and CSE(IOTCSBT)

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	-	-	3

Course Objectives: This course will be able to
1.To understand the history, evolution, and core principles of Java and object-oriented programming.
2.To learn the use of data types, control structures, classes, objects, methods, and constructors.
3.To implement inheritance, access control, interfaces, and exception handling in Java applications.
4.To explore multithreading, generics, and synchronization for developing concurrent applications.
5.To apply lambda expressions and utilize the Java Collections Framework for efficient data handling.

Course Outcomes:
At the end of the course, the student will be able to
1. Use Java data types, operators, control structures, and arrays to write structured programs.
2. Design and implement classes, methods, constructors, and object-oriented features in Java.
3.Apply inheritance, access control, and exception handling to build robust Java applications.
4.Develop multithreaded programs and use generics for type-safe and reusable code.
5.Implement lambda expressions and work with the Java Collections Framework to manage data efficiently.

Unit 1	**Introduction to Java Programming**	10
The History and Evolution of Java, Magic: The Byte code, The Java Buzzwords, The Evolution of Java, Java SE 8. Object-Oriented Programming -Two Paradigms, Abstraction, The three OOP Principles, A First Simple Program-Entering the Program, Compiling the Program, Running the Program, Overview of Java, Data Types, Variables, Arrays, operators and control statements. Classes and Objects: Class Fundamentals, Declaration of Objects, Assigning Object Reference Variables, Introducing Methods, Adding a Method to the Class, Returning a Value, Adding a Method That Takes Parameters, Constructors, Parameterized Constructors, The this Keyword, Instance Variable Hiding, Garbage Collection, The finalize() Method, Overloading Methods, Overloading Constructors, Using Objects as Parameters, A Closer Look at Argument Passing, Returning Objects, Recursion.		

Unit 2	**Access Controls and Inheritance**	10
Introducing Access Control: Understanding static, Introducing final, Arrays Revisited, Introducing Nested and Inner Classes, Exploring the String Class. **Inheritance**: Inheritance Basics, Member Access and Inheritance, A Practical Example, Accessing super class members, Usage super key word, Creating a Multilevel Hierarchy, Accessing Constructors in inheritance, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Using final with Inheritance. Object Class.		

Unit 3	Packages, Interfaces and Exception Handling	10

Packages and Interfaces: Packages, Defining a Package, Finding Packages and CLASSPATH, A Short Package Example, Access Protection, an Access Example, Importing Packages.

Interfaces: Defining an Interface, Implementing Interfaces, Nested Interfaces, Applying Interfaces, Variables in Interfaces, Interfaces Can Be Extended, Default Interface Methods, Default Method Fundamentals, A More Practical Example, Multiple Inheritance Issues, Use static Methods in an Interface, Final Thoughts on Packages and Interfaces.

Exception Handling: Exception-Handling Fundamentals, Exception Types, Uncaught Exceptions, Using try and catch, Displaying a Description of an Exception, Multiple catch Clauses, Nested try Statements, throw, throws, finally, Built-in Exceptions, Creating Your Own Exception Subclasses.

Unit 4	Multithreaded Programming and Generics	10

Multithreaded Programming: The Java Thread Model, Thread Priorities, Synchronization, Messaging, The Thread Class and the Runnable Interface, The Main Thread, Creating a Thread, Implementing Runnable, Extending Thread, Choosing an Approach, Creating Multiple Threads, Using isAlive( ) and join( ), Thread Priorities, Synchronization Using Synchronized Methods, The synchronized Statement, Inter thread Communication.

Generics: What Are Generics, Generics Work Only with Reference Types, A Generic Class with Two Type Parameters, The General Form of a Generic Class, Bounded Types, Using Wildcard Arguments, Bounded Wildcards Creating a Generic Method, Generic Constructors, Generic Interfaces, Raw, Generic Class Hierarchies, Using a Generic super class, A Generic Subclass, Run-Time Type Comparisons Within a Generic Hierarchy, Casting, Overriding Methods in a Generic Class, Type Inference with Generics.

Unit 5	Lambda Expressions and The Collection of Framework	10

Lambda Expressions: Introducing Lambda Expressions, Lambda Expression Fundamentals, Functional Interfaces, Some Lambda Expression Examples, Block Lambda Expressions, Generic Functional Interfaces, Passing Lambda Expressions as Arguments, Lambda Expressions and Variable Capture.

java.util Package: The Collections Framework: Collections Overview, The Collection Interfaces: The Collection Interface, The List Interface; The Collection Classes: The ArrayList Class, The LinkedList Class, Accessing a Collection via an Iterator, Using an Iterator, The For-Each Alternative to Iterators, Storing User-Defined Classes in Collections, Working with Maps, The Map Interfaces, The Map Classes, The Collection Algorithms. Arrays, StringTokenizer.

Prescribed Textbook:

1. Herbert Schildt. Java. The complete reference, 11th Edition, Tata McGraw Hill

Reference Books:

1. J.Nino and F.A. Hosch, An Introduction to programming and OO design using Java, John Wiley&sons.
2. Y. Daniel Liang, Introduction to Java programming, Pearson Education. 6th Edition
3. R.A. Johnson- Thomson, An introduction to Java programming and object oriented application development,
4. Cay.S.Horstmann and Gary,Cornell, Core Java 2, Vol. 1, Fundamentals, Pearson Education. 7th Edition,
5. P. Radha Krishna, Object Oriented Programming through Java, University Press.

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.

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

1. JDK Alpha and Beta (1995)

2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (to be released by March 2022)

Since Java SE 8 release, the Oracle corporation follows a pattern in which every even version is release in March month and an odd version released in September month.

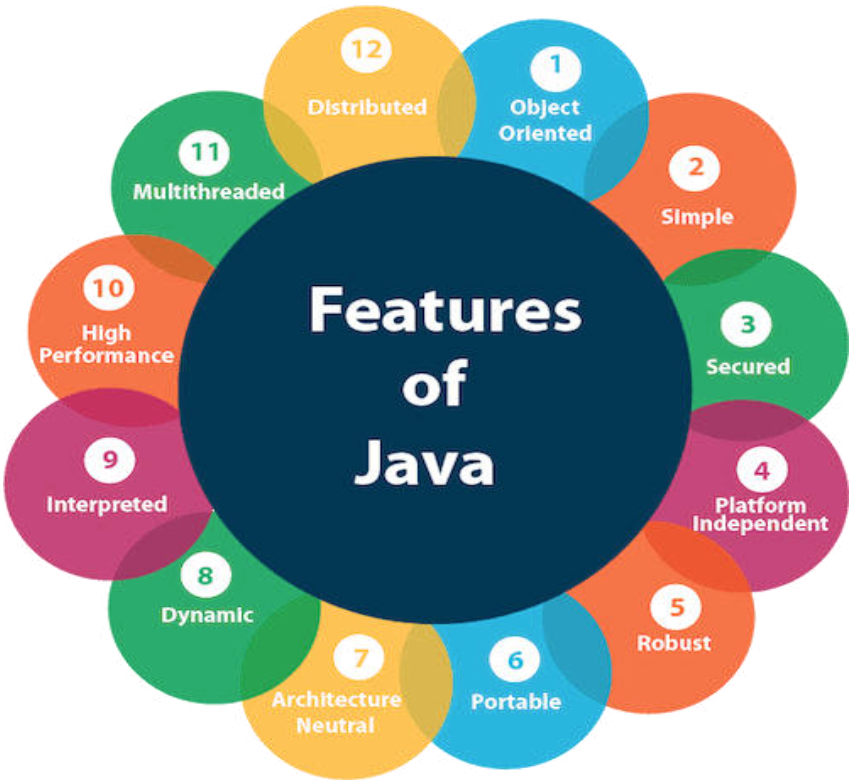
2. The features of Java

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

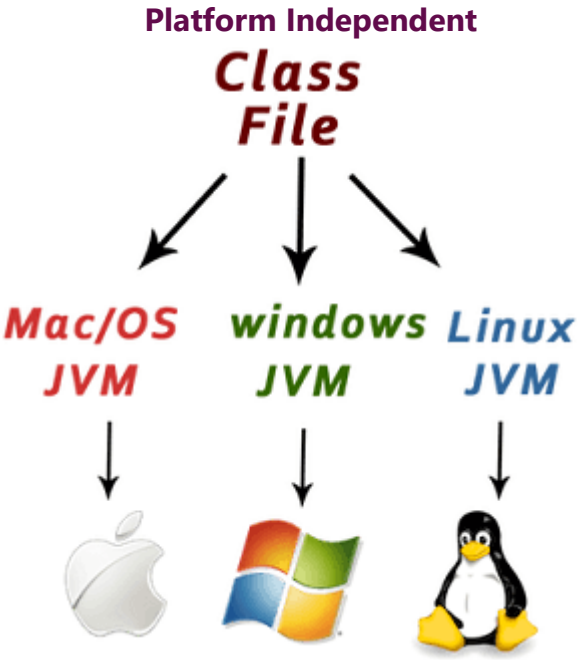
Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

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



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:

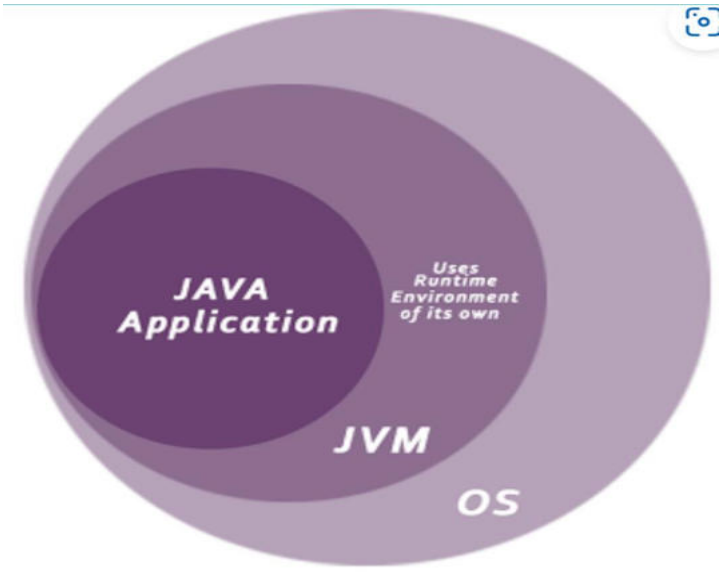
1. Runtime Environment
2. API(Application Programming Interface)

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. What is Object Oriented Paradigm in Java?

Ans:

The term Programming paradigm means the methodology for writing program codes. In general, two paradigms govern how you can construct a program. These two ways are:

1. a process-oriented model
2. an object-oriented model

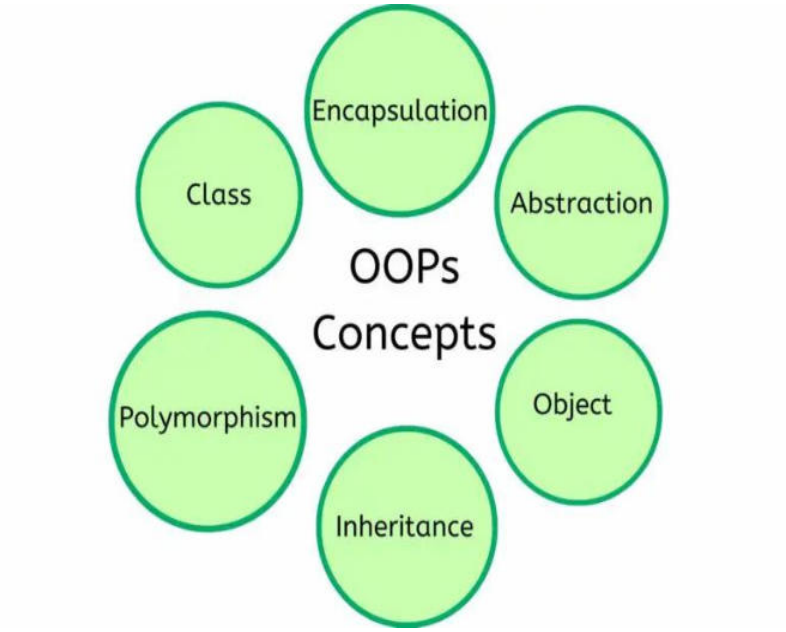
Many programming languages support both the paradigms like python. Python allows the users to code using both process-oriented and object-oriented methodologies. However, Java is exclusively object-oriented.

What is Object Oriented Paradigm

The **object-oriented programming paradigm (OOP)** has a completely different approach to problem-solving. It does not focus on the problem that needs to be solved but focuses on the objects that make up the system. You can compare objects with real-life entities like a car or a dog, and all these objects have a state and behavior

For example, a car state includes its name, colour, brand, and its behaviour includes moving, slowing down, and changing gears.

Therefore, the goal of an object oriented paradigm is to represent the real world while writing code.



Features of the Object Oriented Paradigm

- OOP breaks a problem into a number of entities called objects and then builds data and functions around them.
- It treats data as a critical element in the program development and therefore restricts the flow of data.
- OOP protects the data from accidental modification from outside functions.
- Objects of the different classes can interact easily through functions.
- The object-oriented paradigm follows a bottom-up approach.

4. 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.

5. 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();
}
    }
```

6. The Three OOP Principles

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)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

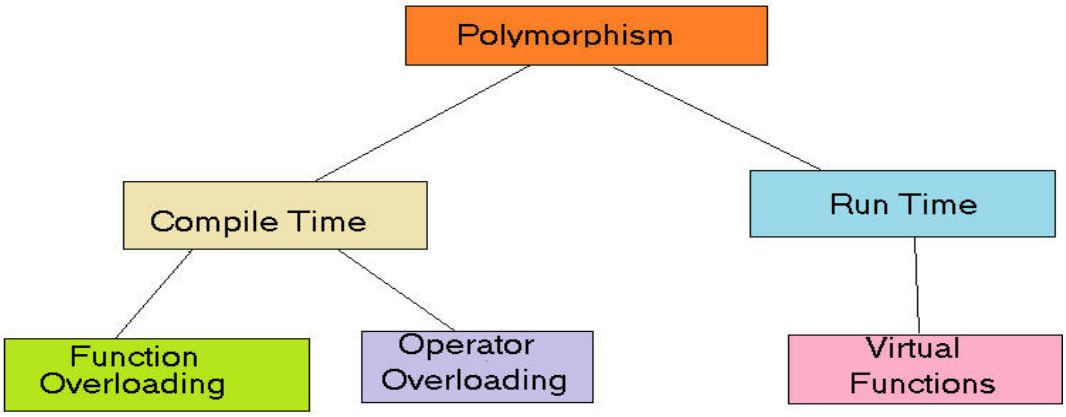
derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

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

7. Java simple programs (or)

What is program structure in java

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.

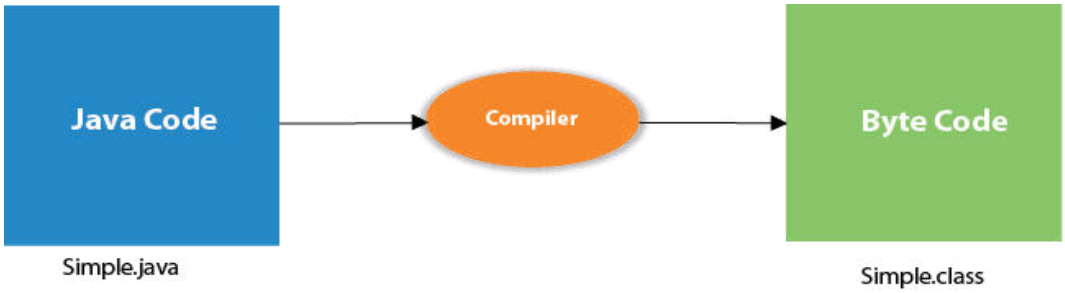
To compile: javac Simple.java

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.

8. 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.

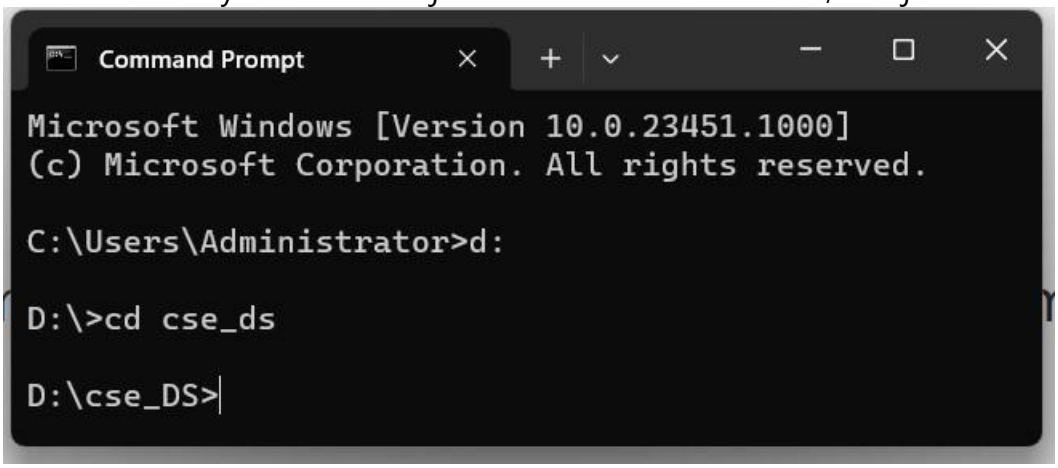
```
class Demo
{
public static void main(String args[])
{
System.out.println("Hello!");
System.out.println("Java");
}
}
```

Step 2:

Open Command Prompt.

Step 3:

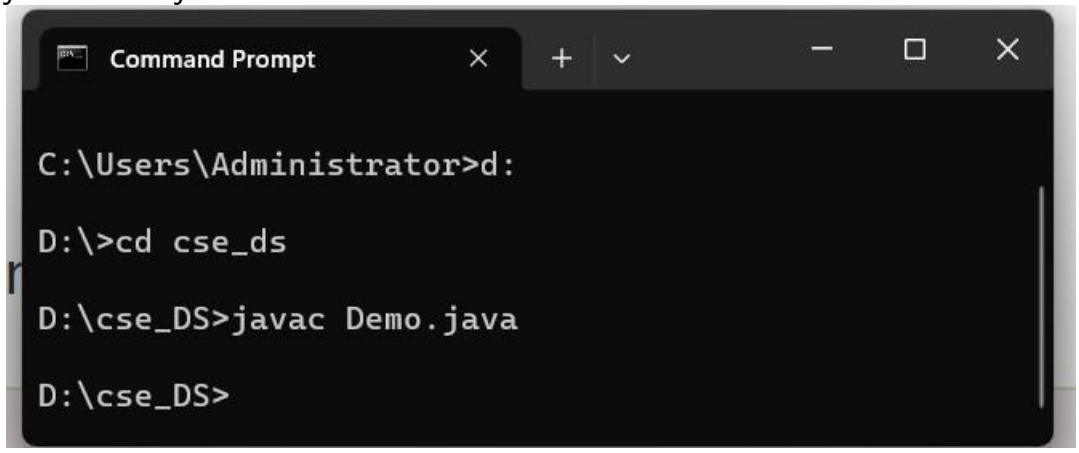
Set the directory in which the .java file is saved. In our case, the .java file is saved in D:\cse_DS>Demo



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.

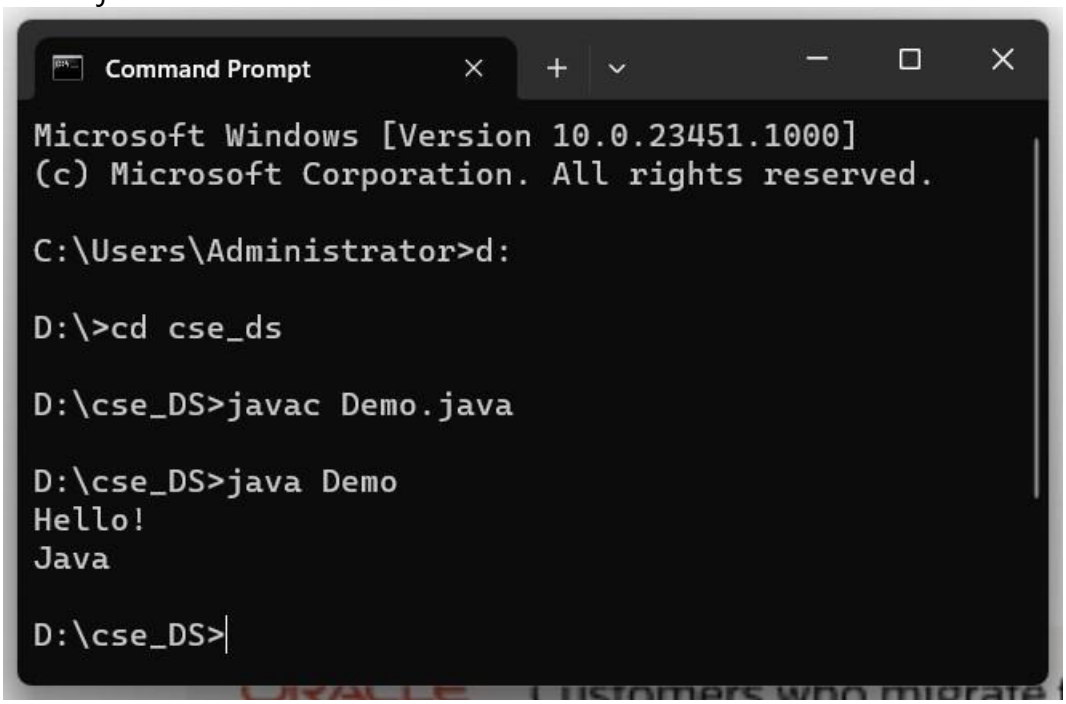
javac Demo.java



Step 5:

Use the following command to run the Java program:

java Demo

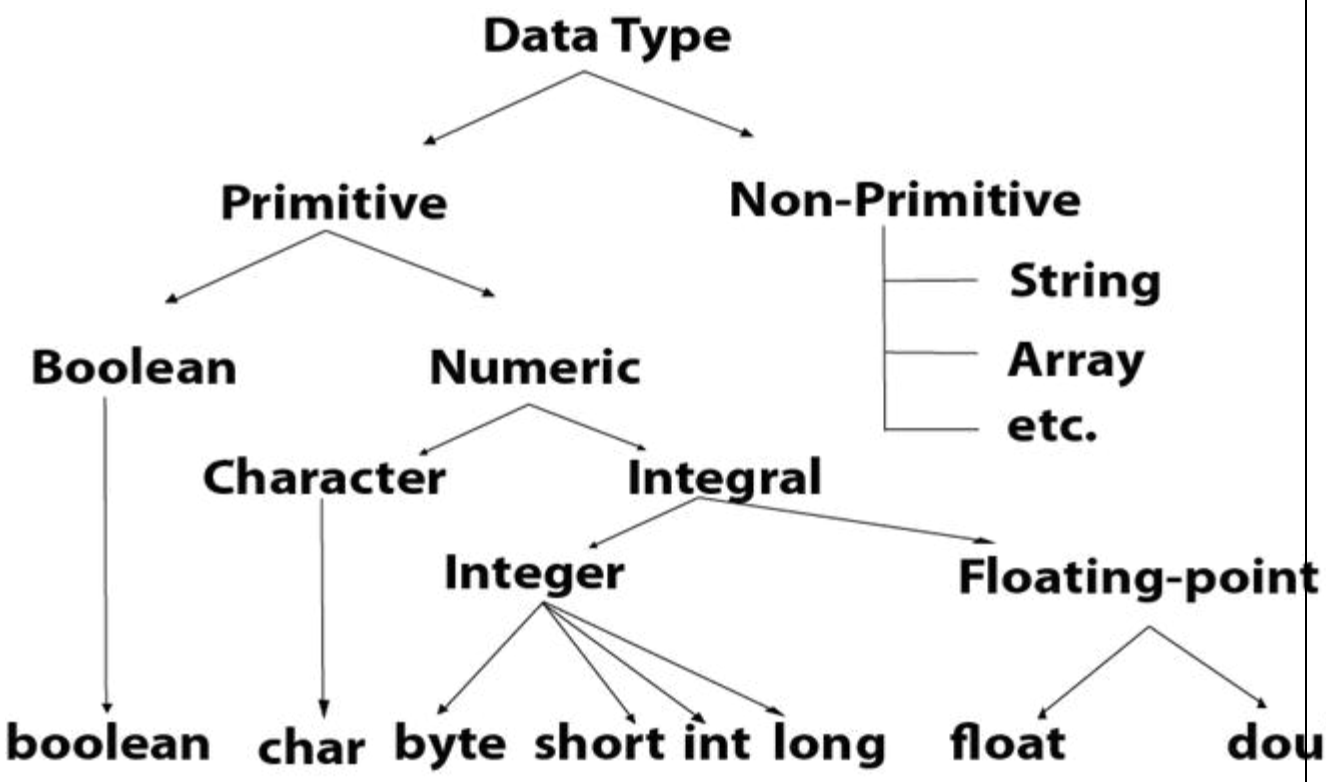


9. Data Types in Java

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.hffse 12
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
        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.

10. What is variable and explain different types of variables

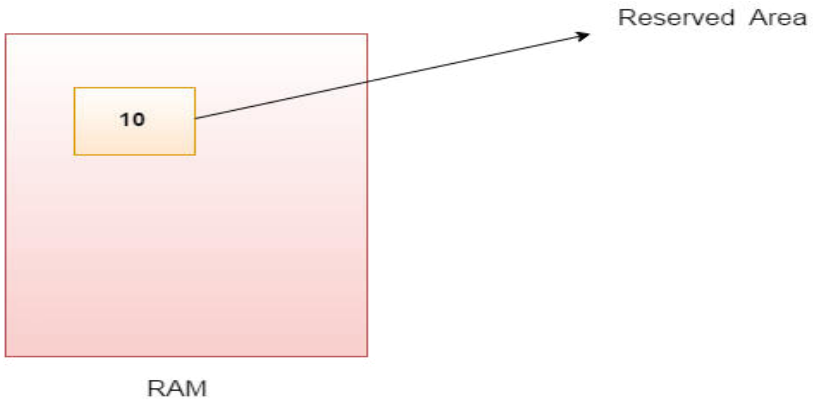
Ans:

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.

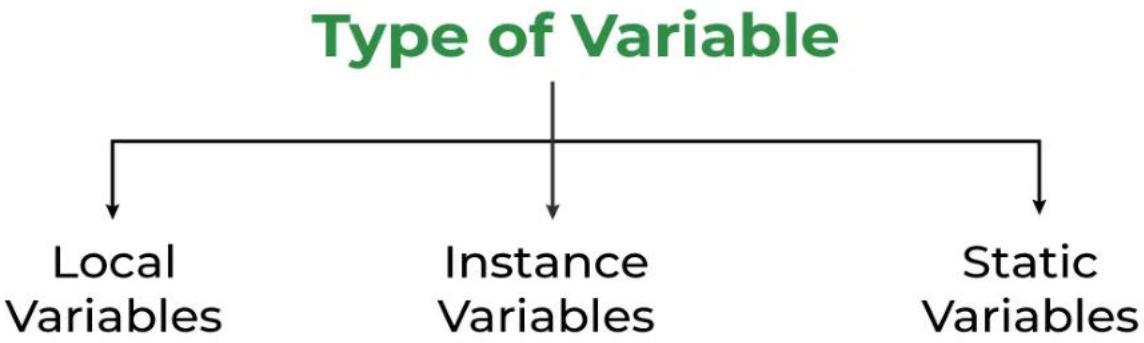



```
int data=50;//Here data is variable
```

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
```

11. Java Arrays

Ans:

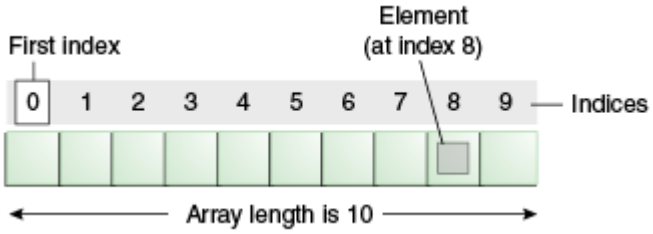
an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure

where we store similar elements. We can store only a fixed set of elements in a Java array.

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

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array

Output:

```
10
```

```
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Example:

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
```

```
    //printing array using for-each loop
    for(int i:arr)
    System.out.println(i);
    }}
```

Output:

```
33
3
4
5
```

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

```
    //Java Program to demonstrate the way of passing an array
    //to method.
    class Testarray2{
    //creating a method which receives an array as a parameter
    static void min(int arr[]){
    int min=arr[0];
    for(int i=1;i<arr.length;i++)
     if(min>arr[i])
      min=arr[i];

    System.out.println(min);
    }

    public static void main(String args[]){
    int a[]={33,3,4,5};//declaring and initializing an array
    min(a);//passing array to method
    }}
```

Output:

```
3
```

Returning Array from the Method

We can also return an array from the method in Java.

```
//Java Program to return an array from the method
class TestReturnArray{
```

```
//creating method which returns an array
static int[] get(){
return new int[]{10,30,50,90,60};
}

public static void main(String args[]){
//calling method which returns an array
int arr[]=get();
//printing the values of an array
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}}
```

Output:

```
10
30
50
90
60
```

2)Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
```

```
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

12. 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

13. 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
```

14.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;
 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
```

15. What is class

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
```

16. **method**

ans:

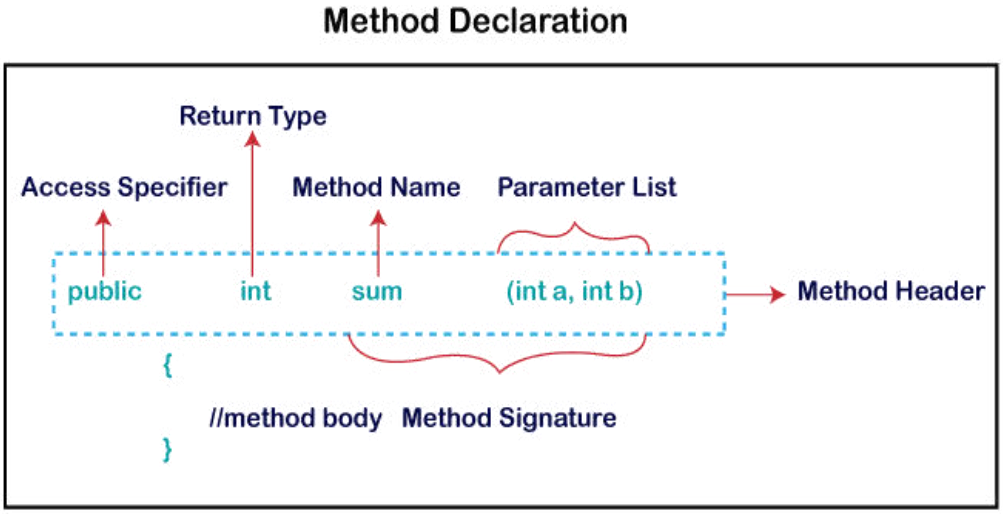
A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.

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");
}
```

17. Adding a Method to the Class

Ans:

In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the Box class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the Box class rather than the BoxDemo class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the Box class compute it.

To do this, you must add a method to Box, as shown here:

```
// This program includes a method inside the box class.
```

```
class Box {
double width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

18. Returning a Value

Ans:

While the implementation of volume( ) does move the computation of a box's volume inside the Box class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better

way to implement volume( ) is to have it compute the volume of the box and return the result
to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, when volume( ) is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the value returned by volume( ). Thus, after

```
vol = mybox1.volume()
```

19. 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.

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.

```
// This program uses a parameterized method.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
```

```
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, the setDim( ) method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

Passing Object as Parameter in Function

```
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);
  }
}
```

20. 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:

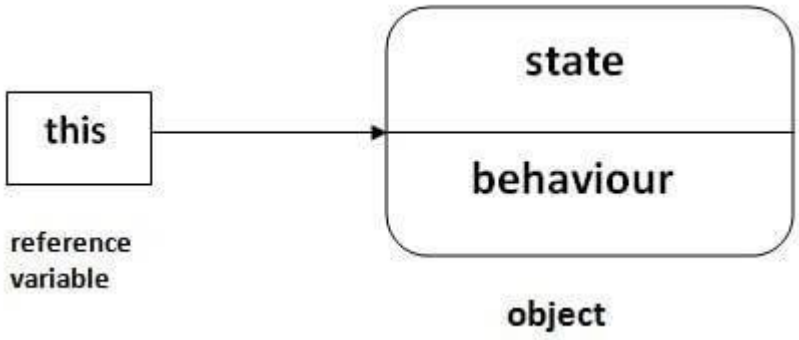
```
111 Karan
222 Aryan
```

21. 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

22. Instance Variable Hiding

Ans:

Instance variable hiding refers to a state when instance variables of the same name are present in superclass and subclass. Now if we try to access using subclass object then instance variable of subclass hides instance variable of superclass irrespective of its return types.

In Java, if there is a local variable in a method with the same name as the instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of this reference.

```
// Java Program to Illustrate Instance Variable Hiding

// Class 1
// Helper class
class Test
{
    // Instance variable or member variable
   private int value = 10;

   // Method
   void method()
 {
       // This local variable hides instance variable
```

```
    int value = 40;

    // Note: this keyword refers to the current instance

    // Printing the value of instance variable
    System.out.println("Value of Instance variable : "
                + this.value);
     // Printing the value of local variable
    System.out.println("Value of Local variable : "
                + value);
  }
}
// Class 2
// Main class
class simple {

  // Main driver method
  public static void main(String args[])
 {
     // Creating object of current instance
    // inside main() method
    Test obj1 = new Test();

    // Calling method of above class
    obj1.method();
  }
}
```

23. Garbage Collection

Ans:

garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

1. By anonymous object: Anonymous objects are those objects in Java which are created without any reference variable. As a result, after creation, we have no way to access the anonymous object.
   new Student();
   In the above code, even though an object of Student class is created, it has no reference variable and cannot be used after creation, but it acquires memory.
2. By nulling reference: When an object is referenced to null value, it is considered an unreferenced object as it holds only null value.
   Student s = new Student();
   s = null;

Here, s is an unreferenced object that holds null value. It is considered garbage by JVM.

3. By assigning a reference to another object: When one object is assigned to another, first object is of no use and can be considered as garbage.
   Student s1 = new Student();
   Student s2 = new Student();
   s1 = s2;
   After the last statement s1 = s2 is executed, the first object becomes unreferenced and can be considered for garbage collection.

Advantage of Garbage Collection

It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.

It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

24. 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");
  }
}
```

25. 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

26. 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
```

27. Using Objects as Parameters

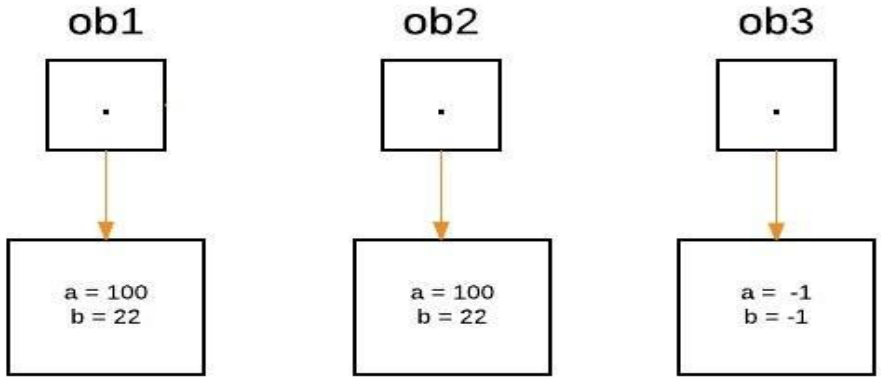
Ans:

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);
```




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

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

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);
  }
}
```

Output

Sum of a and b :13

28. Returning Objects

Ans:

A method can return any type of data, including class types that you create. For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test
{
        int a;
        Test(int i)
        {
    a = i;
        }
        Test incrByTen()
        {
    Test temp = new Test(a+10);  // temp is object
    return temp; // return object
        }
        class RetOb
        {
    public static void main(String args[]) {
    Test ob1 = new Test(2);
    Test ob2;
    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);
    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "+ ob2.a);
        }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
```

ob2.a after second increase: 22

As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using new, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

29. 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( )

30. A Closer Look at Argument Passing

Ans:

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed.

In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test
 {
                void meth(int i, int j)
                {
          i *= 2;
          j /= 2;
                }
}
class CallByValue
 {
```

```
        public static void main(String args[])
         {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a and b before call: " +
    a + " " + b);
    ob.meth(a, b);
    System.out.println("a and b after call: " +
    a + " " + b);
        }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

UNIT-2

1. Access Modifiers in Java
(Or)access control
Ans:

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,

```
// Animal.java file
// public class
public class Animal
{
        // public variable
        public int legCount;

        // public method
        public void display()
        {
            System.out.println("I am an animal.");
            System.out.println("I have " + legCount + " legs.");
        }
}

// Main.java
public class Main
{
    public static void main( String[] args )
    {
        // accessing the public class
        Animal animal = new Animal();

        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }
}
```

2. 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.

3. 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();
      }
}
```

4. arrays revisited

ans:

Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its length instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.


class Length {

public static void main(String args[]) {
int a1[] = new int[10];

int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is  + a1.length);
System.out.println("length of a2 is + a2.length);
System.out.println("length of a3 is + a3.length);
}
}
```

This program displays the following output:

length of a1 is 10 length of a2 is 8 length of a3 is 4

As you can see, the size of each array is displayed. Keep in mind that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the length member to good use in many situations. For example, here is an improved version of the Stack class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you

create stacks of any size. The value of stck.length is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.


class Stack {


private int stck[];
private int tos;


  allocate and initialize stack Stack(int size) {

stck = new int[size]; tos = -1;

}

  Push an item onto the stack void push(int item) {

if(tos==stck.length-1) // use length member System.out.println("Stack is full.");
else

stck[++tos] = item;

}

  Pop an item from the stack


int pop() { if(tos < 0) {


System.out.println("Stack underflow.");
return 0;


}


else
 return stck[tos--];
 }
 }


class TestStack2 {


public static void main(String args[]) {
Stack mystack1 = new Stack(5);
```

```
Stack mystack2 = new Stack(8);


   push some numbers onto the stack for(int i=0; i<5; i++)
mystack1.push(i); for(int i=0; i<8; i++) mystack2.push(i);

   pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)

System.out.println(mystack1.pop());


System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
 System.out.println(mystack2.pop());


}
 }
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

5. What is string and explain string classes

Ans:

What is String in Java

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

An array of characters works same as Java string. For example:

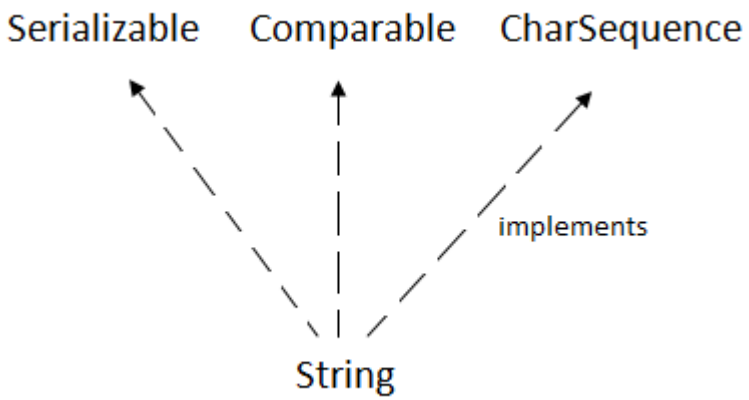
```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
    String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

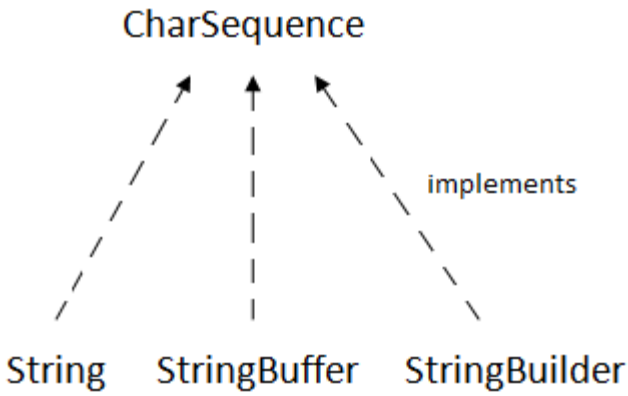
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

There are two ways to create String object:

1. By string literal
2. By new keyword

    1) By string literal

    Java String literal is created by using double quotes. For Example:

        String s="welcome";

        Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled

instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

2) By new keyword

```
String s=new String("Welcome");
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

```
public class StringExample
{
public static void main(String args[])
{
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}
}
```

Output:

```
java
strings
example
```

6. Inheritance basics

Ans:

The process of creating new class from existing class is know as inheritance .

Inheritance in Java is a concept that acquires the properties from one class to other classes; for example, the relationship between father and son. Inheritance in Java is a process of acquiring all the behaviors of a parent object.

The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class).

The **extends** keyword is used to perform inheritance in Java.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Example.

```
// A simple example of inheritance.
// Create a superclass.
class A
{
        int i, j;
        void showij()
        {
                System.out.println("i and j: " + i + " " + j);
        }
}
// Create a subclass by extending class A.
class B extends A
{
                        int k;
                void showk()
                {
                        System.out.println("k: " + k);
                }
                void sum()
                        {
                                System.out.println("i+j+k: " + (i+j+k));
                        }
}
class SimpleInheritance
 {
        public static void main(String args[])
        {
                A superOb = new A();
                        B subOb = new B();
                        // The superclass may be used by itself.
                        superOb.i = 10;
                        superOb.j = 20;
```

```
                System.out.println("Contents of superOb: ");
                superOb.showij();
                System.out.println();
                /* The subclass has access to all public members of
                its superclass. */
                subOb.i = 7;
                subOb.j = 8;
                    subOb.k = 9;
            System.out.println("Contents of subOb: ");
                subOb.showij();
                subOb.showk();
                System.out.println();
                System.out.println("Sum of i, j and k in subOb:");
                subOb.sum();
                }
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

7. Member Access and Inheritance

Ans:

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
private to their class.
This program contains an error and will not
compile.
*/
// Create a superclass.
class A
{
        int i; // public by default
        private int j; // private to A
        void setij(int x, int y)
        {
```

```
                i = x;
                        j = y;
        }
}
// A's j is not accessible here.
class B extends A
 {
        int total;
        void sum()
                 {
                        total = i + j; // ERROR, j is not accessible here
                }
}
class Access
 {
        public static void main(String args[])
        {
                B subOb = new B();
                        subOb.setij(10, 12);
                subOb.sum();
                        System.out.println("Total is " + subOb.total);
        }
}
```

This program will not compile because the reference to j inside the sum( ) method of B causes an access violation. Since j is declared as private, it is only accessible by other members of its own class. Subclasses have no access to it.

8. A More Practical Example

Ans:

```
class Vehicle
 {
          protected String brand = "Ford";        // Vehicle attribute
  public void honk()
{
 // Vehicle method
                System.out.println("Tuut, tuut!");
        }
}

class Car extends Vehicle
 {
        private String modelName = "Mustang";    // Car attribute
```

```
    public static void main(String[] args)
    {
        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (from the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand attribute (from the Vehicle class) and the value of the
        modelName from the Car class
        System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

9. Accessing super class members

Or

A Superclass Variable Can Reference a Subclass Object

Or

accessing parent class variable using java super keyword:

Ans:

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

You will find this aspect of inheritance quite useful in a variety of situations.

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

```
// Access Super Class Methods and Instance
// Variables With Super Keyword in Java
import java.io.*;
// super class
class helloworld
{

    // instance variable
    String name = "helloworld is the name";
    void print()
    {
        System.out.println("This is the helloworld class");
    }
}
```

```
// derived class
class GFG1 extends helloworld
{

    // invoking the instance variable of parent class
    String name = super.name;
    void print()
    {
        // calling the overridden method
        super.print();
        System.out.println("This is the GFG1 class");

        // printing the name
        System.out.println(name);
    }
}
class GFG {
    public static void main(String[] args)
    {
        // instance of the derived class
        GFG1 ob = new GFG1();

        // calling the unoverridden method print
        ob.print();
    }
}
```

Usage of Java super Keyword

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

10. Usage super key word

ANS:

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.

2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal
{
        String color="white";
}
class Dog extends Animal
{
        String color="black";
        void printColor()
        {
                System.out.println(color);//prints color of Dog class
                System.out.println(super.color);//prints color of Animal class
        }
}
class TestSuper1
{
        public static void main(String args[]){
        Dog d=new Dog();
        d.printColor();
        }
}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method.

It should be used if subclass contains the same method as parent class.

In other words, it is used if method is overridden.

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

```
        }
    }
    class Dog extends Animal
    {
        void eat()
        {
                System.out.println("eating bread...");
        }
        void bark()
        {
                System.out.println("barking...");
        }
        void work()
        {
                super.eat();
                bark();
        }
    }
    class TestSuper2
    {
        public static void main(String args[])
        {
                Dog d=new Dog();
            d.work();
        }
}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

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

```
        class Animal
        {
                Animal()
                {
                        System.out.println("animal is created");
```

```
        }
    }
    class Dog extends Animal
    {
        Dog()
        {
            super();
            System.out.println("dog is created");
        }
    }
    class TestSuper3
    {
        public static void main(String args[])
        {
            Dog d=new Dog();
        }
    }
```

Output:
animal is created
dog is created

11. what is inheritance and explain types of inheritance

ans:

Inheritance is a mechanism of driving a new class from an existing class. The existing (old) class is known as base class or super class or parent class. The new class is known as a derived class or sub class or child class. It allows us to use the properties and behavior of one class (parent) in another class (child).

A class whose properties are inherited is known as parent class and a class that inherits the properties of the parent class is known as child class. Thus, it establishes a relationship between parent and child class that is known as parent-child or Is-a relationship.

Suppose, there are two classes named Father and Child and we want to inherit the properties of the Father class in the Child class. We can achieve this by using the extends keyword.

Syntax:

```
//inherits the properties of the Father class
class Child extends Father
{
//functionality
}
```

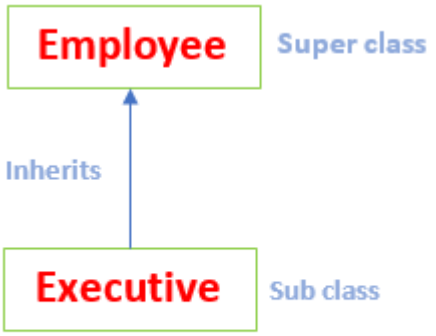
Types of Inheritance

Java supports the following four types of inheritance:

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as simple inheritance.



Single Inheritance

In the above figure, Employee is a parent class and Executive is a child class. The Executive class inherits all the properties of the Employee class.

Let's implement the single inheritance mechanism in a Java program.

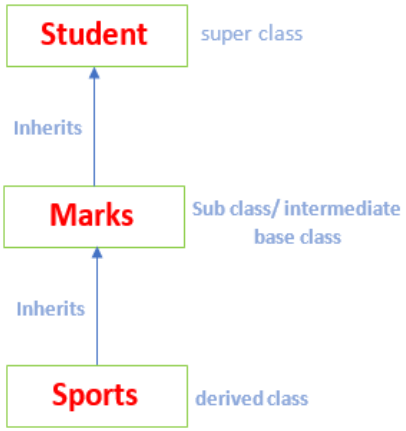
```
class Employee
{
    float salary=40000;
}
class Programmer extends Employee
{
     int bonus=10000;
     public static void main(String args[])
    {
            Programmer p=new Programmer();
            System.out.println("Programmer salary is:"+p.salary);
             System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
```

OUTPUT:

Programmer salary is:40000.0
Bonus of programmer is:10000

Multi-level Inheritance

In multi-level inheritance, a class is derived from a class which is also derived from another class is called multi-level inheritance. In simple words, we can say that a class that has more than one parent class is called multi-level inheritance. Note that the classes must be at different levels. Hence, there exists a single base class and single derived class but multiple intermediate base classes.



Multi-level Inheritance

```
//super class
class Student
{
int reg_no;
void getNo(int no)
{
reg_no=no;
}
void putNo()
{
System.out.println("registration number= "+reg_no);
}
}
//intermediate sub class
class Marks extends Student
{
float marks;
void getMarks(float m)
{
marks=m;
}
```

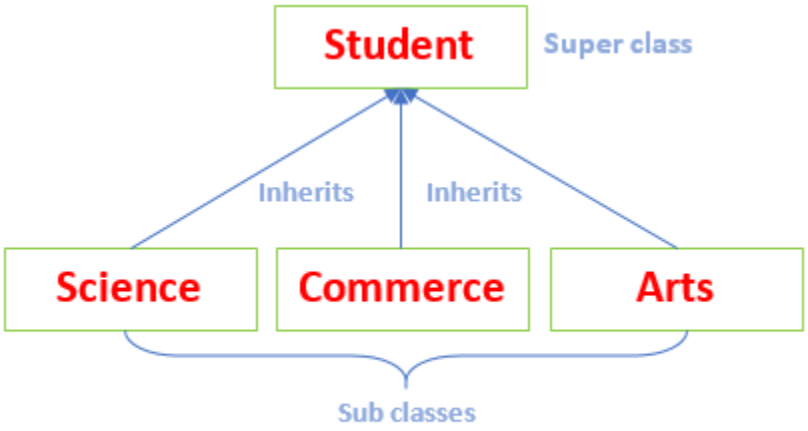
```
void putMarks()
{
System.out.println("marks= "+marks);
}
}
//derived class
class Sports extends Marks
{
float score;
void getScore(float scr)
{
score=scr;
}
void putScore()
{
System.out.println("score= "+score);
}
}
public class MultilevelInheritanceExample
{
public static void main(String args[])
{
Sports ob=new Sports();
ob.getNo(0987);
ob.putNo();
ob.getMarks(78);
ob.putMarks();
ob.getScore(68.7);
ob.putScore();
}
}
```

Output:

registration number= 0987
marks= 78.0
score= 68.7

Hierarchical Inheritance

If a number of classes are derived from a single base class, it is called hierarchical inheritance.



Hierarchical Inheritance

In the above figure, the classes Science, Commerce, and Arts inherit a single parent class named Student.

Let's implement the hierarchical inheritance mechanism in a Java program.

```
//parent class
class Student
{
	public void methodStudent()
	{
		System.out.println("The method of the class Student invoked.");
	}
}
class Science extends Student
{
	public void methodScience()
	{
	System.out.println("The method of the class Science invoked.");
	}
}
class Commerce extends Student
{
	public void methodCommerce()
	{
	System.out.println("The method of the class Commerce invoked.");
	}
}
class Arts extends Student
{
		public void methodArts()
```

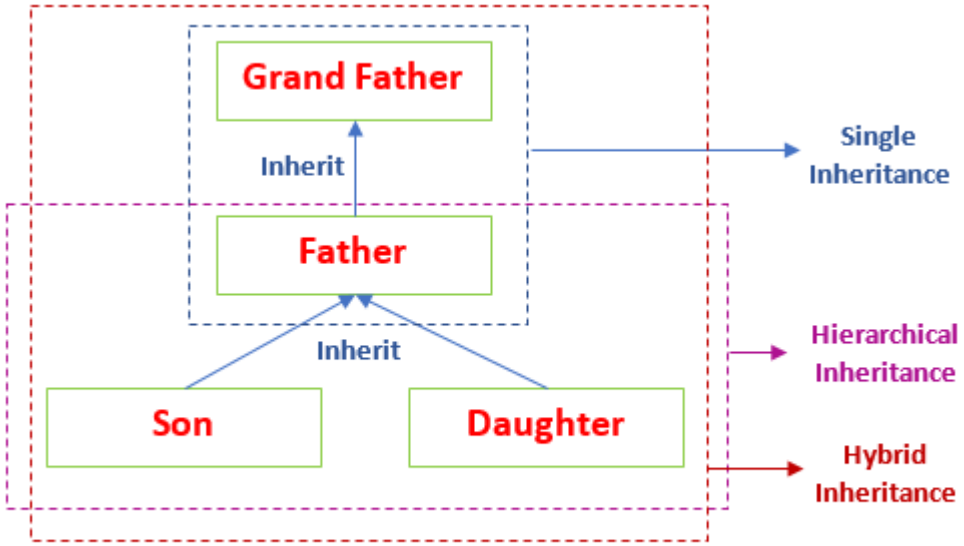
```
                {
                    System.out.println("The method of the class Arts invoked.");
            }
}
public class HierarchicalInheritanceExample
    {
            public static void main(String args[])
                {
                    Science sci = new Science();
            Commerce comm = new Commerce();
                    Arts art = new Arts();
            //all the sub classes can access the method of super class
                    sci.methodStudent();
            comm.methodStudent();
                    art.methodStudent();
    }
}
```

Output:

The method of the class Student invoked.
The method of the class Student invoked.
The method of the class Student invoked**.**

Hybrid Inheritance

Hybrid means consist of more than one. Hybrid inheritance is the combination of two or more types of inheritance.



Hybrid Inheritance

In the above figure, GrandFather is a super class. The Father class inherits the properties of the GrandFather class. Since Father and GrandFather represents single inheritance. Further, the Father

class is inherited by the Son and Daughter class. Thus, the Father becomes the parent class for Son and Daughter. These classes represent the hierarchical inheritance. Combinedly, it denotes the hybrid inheritance.

Let's implement the hybrid inheritance mechanism in a Java program.

```
//parent class
class GrandFather
{
        public void show()
        {
                System.out.println("I am grandfather.");
        }
}
//inherits GrandFather properties
class Father extends GrandFather
{
        public void show()
        {
                System.out.println("I am father.");
        }
}
//inherits Father properties
class Son extends Father
{
        public void show()
        {
                System.out.println("I am son.");
        }
}
//inherits Father properties
public class Daughter extends Father
{
        public void show()
        {
                System.out.println("I am a daughter.");
        }
        public static void main(String args[])
        {
                Daughter obj = new Daughter();
                obj.show();
        }
}
```

Output:

I am daughter.

Multiple Inheritance (not supported)

Java does not support multiple inheritances due to ambiguity. For example, consider the following Java program.

```
class Wishes
{
    void message()
    {
        System.out.println("Best of Luck!!");
    }
}
class Birthday
{
    void message()
    {
        System.out.println("Happy Birthday!!");
    }
}
public class Demo extends Wishes, Birthday  //considering a scenario
{
        public static void main(String args[])
        {
            Demo obj=new Demo();
            //can't decide which classes' message() method will be invoked
            obj.message();
        }
}
```

Output:

The above code gives error because the compiler cannot decide which message() method is to be invoked.

Due to this reason, Java does not support multiple inheritances at the class level but can be achieved through an interface.

12. Accessing constructor in inheritance

Ans:

A constructor in Java is similar to a method with a few differences. Constructor has the same name as the class name. A constructor doesn't have a return type.
A Java program will automatically create a constructor if it is not already defined in the program. It is executed when an instance of the class is created.
While implementing inheritance in a Java program, every class has its own constructor. Therefore the execution of the constructors starts after the object initialization. It follows a certain sequence according to the class hierarchy. There can be different orders of execution depending on the type of inheritance.
For example, given a subclass called B and a superclass called A, is A's constructor called before B's, or vice versa? The answer is that in a class hierarchy, constructorsare called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( )is used. If super( ) is not , then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
        A()
        {
                System.out.println("Inside A's constructor.");
        }
}
// Create a subclass by extending class A.
class B extends A
{
        B()
        {
                System.out.println("Inside B's constructor.");
        }
}
// Create another subclass by extending B.
class C extends B
{
        C()
        {
                System.out.println("Inside C's constructor.");
        }
}
class CallingCons
{
        public static void main(String args[])
```

```
        {
            C obj = new C();
        }
    }
```

The output :

Inside A's constructor
Inside B's constructor
Inside C's constructor

As you can see, the constructors are called in order of derivation.

If you think about it, it makes sense that constructors are executed in order of derivation.

Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first

13. Method Overriding

Ans:

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle
{
  //defining a method
    void run()
    {
        System.out.println("Vehicle is running");}
    }
//Creating a child class
class Bike2 extends Vehicle
{
    //defining the same method as in the parent class
    void run()
    {
        System.out.println("Bike is running safely");
    }
```

```
        public static void main(String args[])
        {
                Bike2 obj = new Bike2();//creating object
               obj.run();//calling method
        }
}
```

Output:

Bike is running safely

14. **Runtime polymorphism**

Or

Dynamic Method Dispatch

Ans:

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

a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable)

```
// Dynamic Method Dispatch
class A
{
        void callme()
        {
                System.out.println("Inside A's callme method");
        }
}
class B extends A
{
                // override callme()
                        void callme()
                        {
                       System.out.println("Inside B's callme method");
                }
}
class C extends A
{
        // override callme()
        void callme()
```

```
        {
                System.out.println("Inside C's callme method");
        }
}
class Dispatch
{
                public static void main(String args[])
                 {
                        A a = new A(); // object of type A
                        B b = new B(); // object of type B
                        C c = new C(); // object of type C
                        A r; // obtain a reference of type A
                        r = a; // r refers to an A object
                        r.callme(); // calls A's version of callme
                        r = b; // r refers to a B object
                        r.callme(); // calls B's version of callme
                        r = c; // r refers to a C object
                        r.callme(); // calls C's version of callme
                }
        }
```

The output from the program is shown here:

Inside A's callme method
Inside B's callme method
Inside C's callme method

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme( ) method

15. Abstract class

Ans:

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

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

abstract type name(parameter-list);

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is,

an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
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();
    }
}
```

Output:

running safely

16. Using final with Inheritance

Or

Final keyword

Ans:

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The

blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9
{
    final int speedlimit=90;//final variable
    void run()
    {
        speedlimit=400;
    }
    public static void main(String args[])
    {
        Bike9 obj=new  Bike9();
        obj.run();
    }
}//end of class
```

Output:

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike
{
    final void run()
    {
        System.out.println("running");
    }
}

class Honda extends Bike
{
    void run()
    {
        System.out.println("running safely with 100kmph");
    }
```

```
    public static void main(String args[])
    {
        Honda honda= new Honda();
        honda.run();
    }
}
```

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike
{
}

class Honda1 extends Bike
{
    void run()
    {
        System.out.println("running safely with 100kmph");
    }

    public static void main(String args[])
    {
        Honda1 honda= new Honda1();
         honda.run();
     }
}
```

Output:Compile Time Error