* **History of JAVA** => https://en.wikipedia.org/wiki/Java_%28programming_language%29

The Complete History of Java Programming Language => Java is an Object-Oriented programming language developed by James Gosling in the early 1990s. The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc. Originally C++ was considered to be used in the project but the idea was rejected for several reasons (For instance C++ required more memory). Gosling endeavored to alter and expand C++ however before long surrendered that for making another stage called Green. James Gosling and his team called their project "Green talk" and its file extension was .gt and later became known as "OAK".

Why "Oak"? => The name Oak was used by Gosling after an oak tree that remained outside his office. Also, Oak is an image of solidarity. But they had to later rename it as "JAVA" as it was already a trademark by Oak Technologies. "JAVA" Gosling and his team did a brainstorm session and after the session, they came up with several names such as JAVA, DNA, SILK, RUBY, etc. Java name was decided after much discussion since it was so unique. Java was created on the principles like Robust, Portable, Platform Independent, High Performance, Multithread, etc. and was called one of the Ten Best Products of 1995 by the TIME MAGAZINE. JDK 1.0 was introduced in 1996. The first public implementation of JAVA 1.0 was in 1995, in 2006 sun released JVM as a free and open-source software in same year J2SE was renamed as java standard edition i.e. se. Major release was java se 6 in 2006 and se 8 in 2014. The Java language has experienced a few changes since JDK 1.0 just as various augmentations of classes and packages to the standard library. In Addition to the language changes, considerably more sensational changes have been made to the Java Class Library throughout the years, which has developed from a couple of hundred classes in JDK 1.0 to more than three thousand in J2SE 5. Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

*** Features of JAVA** => https://www.javatpoint.com/features-of-java (Can use this link for whole java knowledge)

Primary/Main Features of Java =>

1. Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of the bytecode. That is why we call java a platform-independent language.

2. Object-Oriented Programming Language: Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

The four main concepts of Object-Oriented programming are:

- Inheritance
- Abstraction
- Encapsulation
- Polymorphism

3. Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation.

4. Robust: Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

5. Secure: In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows ArrayIndexOutOfBound Exception if we try to do so. That's why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the os(operating system) environment which makes java programs more secure.

6. Distributed: We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

7. Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.

8. Portable: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

9. High Performance: Java architecture is defined in such a way that it reduces overhead during the runtime and at sometimes java uses Just In Time (JIT) compiler. where the compiler compiles code on-demand basics, i.e. it only compiles those methods that are called making applications to execute faster.

10. Dynamic flexibility: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes, and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

11. Sandbox Execution: Java programs run in a separate space that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier. Bytecode verifier also provides additional security as its role is to check the code for any violation of access.

12. Write Once Run Anywhere: As discussed above java application generates a '.class' file that corresponds to our applications(program) but contains code in binary format. It provides ease to architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

13. Power of compilation and interpretation: Most languages are designed with the purpose of either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

* Advantages of Java:

1. Platform independent: Java code can run on any platform that has a Java Virtual Machine (JVM) installed, which means that applications can be written once and run on any device.

2. Object-Oriented: Java is an object-oriented programming language, which means that it follows the principles of encapsulation, inheritance, and polymorphism.

3. Security: Java has built-in security features that make it a secure platform for developing applications, such as automatic memory management and type checking.

4. Large community: Java has a large and active community of developers, which means that there is a lot of support available for learning and using the language.

5. Enterprise-level applications: Java is widely used for developing enterprise-level applications, such as web applications, e-commerce systems, and database systems.

* Disadvantages of Java:

1. Performance: Java can be slower compared to other programming languages, such as C++, due to its use of a virtual machine and automatic memory management.

2. Memory management: Java's automatic memory management can lead to slower performance and increased memory usage, which can be a drawback for some applications.

* Package in java:

In Java, a package is a namespace that organizes a set of related classes and interfaces. Packages help in organizing and managing Java code by providing a hierarchical structure for class files. They also help in avoiding naming conflicts and provide better modularity and reusability of code.

* Java Classes:

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/attributes. It is a user-defined template from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

* Properties of Java Classes:

1) Class is not a real-world entity. It is just a template from which objects are created.
2) Class does not occupy memory.
3) Class is a group of variables of different data types and a group of methods.
4) A Class in Java can contain:
    * Data member
    * Method
    * Constructor
    * Nested Class
    * Interface

* Variables:

In Java, Variables are the data containers that save the data values during Java program execution. Every Variable in Java is assigned to a data type that designates the type and quantity of value it can hold. A variable is a memory location name for the data value.

Java variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location. In Java, all variables must be declared before use.

1. Member Variables (Class Level Scope) => These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class.
2. Local Variables (Method Level Scope) => Variables declared inside a method have method level scope and can't be accessed outside the method.

* Java Objects:

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class.

A typical Java program creates many objects, which interact by invoking methods. An object consists of:

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

* Java Methods:

The method in Java or Methods of Java is a collection of statements that perform some specific task and
return the result to the caller.

Java Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.

1. A method is like a function i.e. used to expose the behavior of an object.
2. It is a set of codes that perform a particular task.

* Keywords in JAVA:

In Java, keywords are reserved words that have predefined meanings and are used to define the syntax and structure of the Java programming language. You cannot use keywords as identifiers (such as variable names, class names, or method names) in your Java code.

Here's a list of Java keywords:

1. abstract: Used to declare abstract classes and methods.
2. assert: Used to perform assertion testing.

3. boolean: Used to declare boolean variables and literals (`true` or `false`).
4. break: Used to terminate a loop or switch statement.
5. byte: Used to declare byte variables.
6. case: Used within switch statements to define different cases.
7. catch: Used to catch exceptions in exception handling.
8. char: Used to declare char variables.
9. class: Used to declare a class.
10. const: Reserved for future use. Not currently used in Java.
11. continue: Used to skip the current iteration of a loop and continue with the next iteration.
12. default: Used within switch statements to define the default case.
13. do: Used to start a do-while loop.
14. double: Used to declare double variables.
15. else: Used in if-else conditional statements.
16. enum: Used to define enumerated types.
17. extends: Used to indicate inheritance in class declarations.
18. final: Used to declare constants, make methods non-overridable, or make classes non-inheritable.
19. finally: Used in exception handling to specify a block of code that is always executed, regardless of whether an exception occurs or not.
20. float: Used to declare float variables.
21. for: Used to start a for loop.
22. if: Used to start an if statement.
23. implements: Used to indicate that a class implements an interface.
24. import: Used to import packages or classes.
25. instanceof: Used to test if an object is an instance of a particular class or interface.
26. int: Used to declare int variables.
27. interface: Used to declare an interface.
28. long: Used to declare long variables.
29. native: Used to declare native methods, which are implemented in platform-dependent code.
30. new: Used to create new objects.
31. null: Used to represent a null reference.
32. package: Used to define a package.
33. private: Used to specify that a class member is accessible only within the same class.
34. protected: Used to specify that a class member is accessible within the same package or by subclasses.
35. public: Used to specify that a class member is accessible from any other class.
36. return: Used to return a value from a method.
37. short: Used to declare short variables.
38. static: Used to declare static fields, methods, and nested classes.
39. strictfp: Used to restrict floating-point calculations to ensure portability.
40. super: Used to refer to the superclass or to call superclass constructors and methods.
41. switch: Used to start a switch statement.
42. synchronized: Used to specify that a method can be accessed by only one thread at a time.

43. this: Used to refer to the current object.
44. throw: Used to explicitly throw an exception.
45. throws: Used in method declarations to indicate the exceptions that may be thrown by the method.
46. transient: Used to specify that a field should not be serialized.
47. try: Used to start a try-catch block for exception handling.
48. void: Used to declare that a method does not return any value.
49. volatile: Used to indicate that a variable may be modified asynchronously.
50. while: Used to start a while loop.

These keywords are an integral part of Java syntax and are used to define the structure, control flow,
and behavior of Java programs.

* Decision Making in Java:

Decision Making in programming is similar to decision-making in real life. In programming also, we face some situations where we want a certain block of code to be executed when some condition is fulfilled. A programming language uses control statements to control the flow of execution of a program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

* Java's Selection statements:

1. if
2. if-else
3. nested-if
4. If ladder
5. switch-case
6. jump – break, continue, return

1. if: if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax:

```
if(condition) {
  // Statements to execute if
  // condition is true
}
```

2. if-else: The if statement alone tells us that if a condition is true, it will execute a block of statements and if the condition is false, it won't. But what if we want to do something else if the condition is false?

Here comes the else statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition) {
   // Executes this block if
   // condition is true
} else {
   // Executes this block if
   // condition is false
}
```

3. nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

```
Syntax:
if (condition1) {
  // Executes when condition1 is true
  if (condition2) {
    // Executes when condition2 is true
  }
}
```

4. if-else-if ladder: Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that 'if' is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. There can be as many as 'else if' blocks associated with one 'if' block but only one 'else' block is allowed with one 'if' block.

```
Syntax:
if (condition){
   statement;
}else if (condition){
   statement;
}else{
   statement;}
```

5. switch-case: The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

```
Syntax:
switch (expression) {
 case value1:
   statement1;
   break;
 case value2:
   statement2;
   break;
 case valueN:
   statementN;
```

```
    break;
  default:
    statementDefault;
}
```

6. jump: Java supports three jump statements: break, continue and return. These three statements transfer control to another part of the program.

* Break: In Java, a break is majorly used to Terminate a sequence in a switch statement (discussed above). To exit a loop, and Used as a "civilized" form of goto.

* Continue: Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

* Return: The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

* Loops in Java:

1. while loop: A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

   Syntax:

   ```
   while (boolean condition) {
     loop statements...
   }
   ```

2. for loop: for loop provides a concise way of writing the loop structure. Unlike a while loop, a for loop's statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

   Syntax:

   ```
   for (initialization condition; testing condition; increment/decrement) {
     statement(s);
   }
   ```

3. do while: do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of Exit Control Loop.

   Syntax:

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

4. Infinite loop: One of the most common mistakes while implementing any sort of looping is that it may not ever exit, that is the loop runs for infinite time. This happens when the condition fails for some reason.
5. Nested Loop: Nested loop means a loop statement inside another loop statement. There are different

combinations of loop using for loop, while loop, do-while loop.

6. Continue Statement: Continue statement is often used inside in programming languages inside loops control structures. Inside the loop, when a continue statement is encountered the control directly jumps to the beginning of the loop for the next iteration instead of executing the statements of the current iteration. The continue statement is used when we want to skip a particular condition and continue the rest execution. Java continue statement is used for all type of loops but it is generally used in for, while, and do-while loops. In the case of for loop, the continue keyword force control to jump immediately to the update statement. Whereas in the case of a while loop or do-while loop, control immediately jumps to the Boolean expression.
   Syntax: continue keyword along with a semicolon
   continue;
7. Break statement: Break Statement is a loop control statement that is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.
   Syntax:
   break;

* JAVA Data Types:

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated.

I) Primitive: Primitive data are only single values and have no special capabilities.

There are 8 primitive data types.

a) Byte -> [1 byte (8 bits)] The byte data type is an 8-bit signed two's complement integer. The byte data type is useful for saving memory in large arrays.[-128 to 127 no can be stored]
b) Short -> (2 byte) 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.
c) int -> (4 byte) It is a 32-bit signed two's complement integer.
d) long -> (8 byte) 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.
e) float -> (4 byte) The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point no.
f) double -> (8 byte) The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice.
g) char -> (2 byte) The char data type is a single 16-bit Unicode character with the size of 16 bits.

h) boolean -> (1 bit) Boolean data type represents only one bit of information either [True/False]

*** Why char uses 2 bytes in Java and what is \u0000?
-> Char uses 2 bytes in java because it uses the Unicode system rather than the ASCII system.
"\u000" is the lowest range of the Unicode system.

II) Non-Primitive/ Reference: 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.

a) Strings -> can add characters, Strings are immutable(i.e we cannot change the original string)
Substring -> if there is a line and we want only single world we can use substring

b) 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.

c) 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

d) 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).

e) Arrays -> Arrays are non-primitive data type. [] we use this bracket for array. if we don't give value to the variable java will initiate it with 0 value in case of int, for boolean case it will print true or false. For finding the length of array i.e. how much we have stored we can check it by length function. For sorting we have to use Arrays.sort function. We can also use 2D arrays for containing info of 2 different things.

String => 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.

* Methods to Take Input in Java:

There are two ways by which we can take Java input from the user or from a file

1. BufferedReader Class
2. Scanner Class
3. Console Class

1. Using BufferedReader Class for String Input in Java:

It is a simple class that is used to read a sequence of characters. It has a simple function that reads a character another read which reads, an array of characters, and a readLine() function which reads a line.

InputStreamReader() is a function that converts the input stream of bytes into a stream of characters so that
it can be read as BufferedReader expects a stream of characters. BufferedReader can throw checked Exceptions.

Syntax => BufferedReader bfn = new BufferedReader(new InputStreamReader(System.in));

2. Using Scanner Class for Taking Input in Java:

It is an advanced version of BufferedReader which was added in later versions of Java. The scanner can read formatted input. It has different functions for different types of data types. The scanner is much easier to read as we don't have to write throws as there is no exception thrown by it. It was added in later versions of Java. It contains predefined functions to read an Integer, Character, and other data types as well.

Syntax => Scanner sc = new Scanner(System.in);

3. Using Console class (for console-based applications):

The Console class provides methods to read input from the console in console-based applications. This class is typically used in terminal or command-line applications.

* Major Differences between Scanner and BufferedReader Class in Java:

1.BufferedReader is synchronous while Scanner is not.
2.BufferedReader should be used if we are working with multiple threads.
3.BufferedReader has a significantly larger buffer memory than Scanner.
4.The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
5.BufferedReader is a bit faster as compared to Scanner because the Scanner does the parsing of input data and BufferedReader simply reads a sequence of characters.

Parts of System.out.println():

The statement can be broken into 3 parts which can be understood separately:

*System: It is a final class defined in the java.lang package.

*out: This is an instance of PrintStream type, which is a public and static member field of the System class.

*println(): As all instances of the PrintStream class have a public method println(), we can invoke the same on out as well. This is an upgraded version of print(). It prints any argument passed to it and adds a new line to the output. We can assume that System.out represents the Standard Output Stream.

* Access Modifiers in Java:

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc. to the user, depending upon the access modifier used with the element.

* Types of Access Modifiers in Java:

There are four types of access modifiers available in Java:

*Default –> (No keyword required) When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default. Default access

modifiers are accessible only within the same package.

*Private -> The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared. Any other class of the same package will not be able to access these members. Top-level classes or interfaces cannot be declared as private/protected because it means only visible within the enclosing class i.e. nested classes.

*Protected -> The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

*Public -> The public access modifier is specified using the keyword public. The public access modifier has the widest scope among all other access modifiers. Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

* What are the 12 modifiers in Java?

=> 12 Modifiers in Java are public, private, protected, default, final, synchronized, abstract, native, strictfp, transient, and volatile.

* Casting => Converting form one type to other type of data type

I) Implicit -> Storing int data in double or small data in large type.

II) Explicit -> Storing double type in int type, large in small.

* Combinations in JAVA:

I) No return type no arguments -> Use 'Void' keyword or at the end of program write return 0. It does not return anything with no arguments.

II) No return type with Arguments -> return type method name (parameter list), this is the syntax. The parameters can be given anything.

III) Return type but not taking any argument -> Whenever method is having return type other than void then method must have return statement. The both data types should be match.

IV) Return type with arguments -> both the data type and argument is available.

* Pillars of java or oops concept:

In object-oriented programming (OOP), there are four main pillars, also known as principles or concepts, that define the foundation of OOP languages like Java. These pillars encapsulate the key ideas and best practices for designing and implementing object-oriented systems.

The four pillars of OOP are:

1. Inheritance: Inheritance is the mechanism by which a new class, known as a subclass or derived class, can inherit properties and behaviors (attributes and methods) from an existing class, known as a superclass or base class. This allows for code reuse and promotes the creation of hierarchical relationships between classes. In Java, inheritance is achieved using the extends keyword to indicate that one class extends another class.

2. Abstraction: Abstraction is the process of simplifying complex systems by focusing on the essential properties and ignoring irrelevant details. In Java, abstraction is achieved through abstract classes and interfaces. An abstract class is a class that cannot be instantiated and may contain abstract methods, which are declared without implementation. An interface is a reference type that specifies a set of methods that a class must implement. Abstraction allows for modeling complex systems at higher levels of abstraction, making it easier to understand and manage large codebases.

3. Encapsulation: Encapsulation refers to the bundling of data (attributes or properties) and methods (functions or procedures) that operate on that data into a single unit, known as a class. Encapsulation helps in hiding the internal state of an object and only exposing the necessary functionality through well-defined interfaces. In Java, encapsulation is achieved through access modifiers (public, private, protected, default) and getter and setter methods.

4. Polymorphism: Polymorphism means the ability of an object to take on multiple forms or behaviors. In Java, polymorphism can be achieved through method overriding and method overloading. Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. Method overloading allows multiple methods with the same name but different parameter lists to coexist within the same class. Polymorphism enables flexibility, extensibility, and code reusability in object-oriented systems.

These four pillars of object-oriented programming—encapsulation, inheritance, polymorphism, and abstraction—forms the core principles that guide the design and implementation of Java programs and other object-oriented systems. They help in creating modular, maintainable, and scalable software solutions by promoting code reusability, flexibility, and extensibility.

* JAVA Inheritance:

Inheritance is inheriting the properties of the parent class into the child class. Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. You can also add new methods and fields in your current class. Inheritance represents the IS-A relationship which is also known as the parent-child relationship.

1. Single Inheritance => When a class inherits another class, it is known as a single inheritance.
2. Multilevel Inheritance => When there is a chain of inheritance, it is known as multilevel inheritance.
3. Hierarchical Inheritance => When two or more classes inherit a single class, it is known as hierarchical inheritance.
4. Hybrid Inheritance => It is the combination of any two methods above.
5. Multiple Inheritance => When there are more than one parent class for a single child class, it is known as multiple inheritance.

* Why Multiple Inheritance is not supported in java?
=> Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more

than one parent class. The problem occurs when there exist methods with the same signature in both the
super classes and subclass. On calling the method, the compiler cannot determine which class method to be
called and even on calling which class method gets the priority, creating an ambiguity for the compiler. This problem is also known as Diamond problem.

* What is the Need of Inheritance in Java?

=> 1. Code reusability -> Code is a supper class to all the subclass.
2. Method Overriding -> Is achieved. It is one of ways by which java achieve runtime polymorphism.
3. Abstraction -> Concept of abstract where we do not have to provide all details is achieved through inheritance.

* What is Java Compiler | How it works:

A compiler in Java is a computer program that is used for compiling Java programs. It is not platform-independent. It converts (translates) source code (.java file) into bytecode (.class file). A bytecode is a binary code that is understood and interpreted by Java Virtual Machine (JVM) on the underlying operating system. It is not similar to machine code.

* Responsible tasks for Compiler:

Apart from compiling a source program, Java compiler is responsible for the following tasks that are as follows:

a) Java compiler checks the syntaxial error (Syntax error). It also generates a list of all error messages when
it finds errors in the program and does not produce an object code unless the errors are rectified reported by
the Java compiler.

b) It converts source code into byte code with the help of Java Virtual Machine (JVM).

c) It also adds the additional code to your program if required.

{more info on => https://www.scientecheasy.com/2021/03/java-compiler.html/}

* Interpreter in Java:

Interpreter in Java is a computer program (software) that implements Java Virtual Machine (JVM) and runs Java applications (programs). It translates (converts) bytecode to machine code (native code) line by line during runtime. In other words, Java interpreter is a computer program that converts the high-level
program into assembly language (machine language). It is used for executing Java programs. Interpreter takes
bytecode as an input and executes that code by converting it to machine code. It is recognized by “Java.exe”
command.

Some of the Java interpreter command options are as follows:

1. -version: display interpreter version.
2. -verbose: display interpreter information.

3. -help: display interpreter options.

The interpreter options are case-sensitive.

* Functions of Java Interpreter =>

Java interpreter is responsible for the following functions that are as follows:

1. The main function of interpreter is to convert the bytecode instruction to machine code line by line at runtime, without changing the sequence.
2. It performs all the activities of Java runtime system.
3. It runs application programs by reading and interpreting bytecode files.

* Interpreter vs Compiler: How is Interpreter different from a compiler in Java?

=> Here, we have listed the key differences between an interpreter and a compiler. They are as follows:

1. An interpreter translates program line by line whereas, a compiler translates the entire program together.
2. Execution process of Interpreter is slower whereas, execution process of the compiler is faster.
3. Interpreter takes less compile-time whereas compiler takes more time.
4. It will not produce intermediate object code. Compiler produces intermediate object code.
5. Interpreter compiles the program until an error is found. Compiler shows all the errors once at the end of the compilation.
6. Python, PHP, Ruby, and Perl use an interpreter. Java, C++, Scala, and C use a compiler.

* Java JDK, JRE and JVM:

1. What is JVM?

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program. When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent machine-language. It's because when you write Java code, it's ultimately written for JVM but not your physical computer. Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.

2. What is JRE?

JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications. JRE is the superset of JVM.

JRE consists of the following main components that are as follows:

1. Java API (Application Programming Interface)
2. Class Loader

3. Bytecode verifier
4. Java Virtual Machine (Interpreter)

3. What is JDK?

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it. In addition to JRE, JDK also contains a number of development tools (compilers, Javadoc, Java Debugger, etc.).

* What is Java API (Application Interface Programming)?

Java Application Programming Interface (API) is a very large collection of prepackaged, ready-made software components that provides the core functionality of the Java programming language. In simple words, Java API is a large collection of already defined classes, interfaces, and methods in the form of Java packages. It provides many useful capabilities, such as Graphical User Interface (GUI), Date, Time, and Calendar capabilities to programmers. Java API is grouped into libraries of related classes and interfaces along with their fields, constructors, and methods. These libraries are provided in the form of packages. It provides extra programming features built on the core java platform. It means that the basic features of Java programming language do not change when a new version of it is released. Since Java API is flexible, it can be opened to add new packages or libraries into it. Some popular libraries and their functionality from the Java API, in short, are as follows:

a) Java.lang: It is a package that provides fundamental classes to design the Java programming language.
The Java.lang package is dynamically imported (i.e. loaded) in a Java program. It does not need to import explicitly. The most commonly used classes from Java.lang package are Double, Float, Integer, String, StringBuffer, System, and Math.
b) Java.io: It supports Input/Output through the file system, keyboard, network, etc. The java.io package contains several classes to perform input and output operations. The most commonly used classes from java.io are File class, InputStreams, OutputStreams, Readers, Writers, and RandomAccessFile.
c) Java.util: It supports various programming utilities. The java.util package provides legacy collection classes, event model, collections framework, date and time capabilities, and other utility classes such as string tokenizer.
d) Java.math: It is used to support mathematical operations.
e) Java.security: It supports security functions.
f) Java.awt: It supports creating graphical user interface (GUI), painting graphics, and images.
g) Java.sql: It support for accessing relational databases through SQL.

h) Java.beans: It supports creating java beans.

i) Java.net: The java.net package provides classes that are used for implementing networking in java programs.

j) Java.imageIO: It support for image input/output.

These predefined Java API provide a tremendous amount of core functionality to a programmer. A programmer should be aware of these Java APIs. He should know how to use these Java APIs. These are examples of some important libraries from Java API.

* What is Class Loader in Java?

The Java ClassLoader is a part of the JRE that dynamically loads Java classes (.class files) into the JVM. When we write a program in java, the program is placed in memory by the class loader before it can be executed. Java Class loader takes .class file containing bytecode and transfers it to the memory. It loads the .class file from a disk on your system or over a network. After loading of class, it is passed to the bytecode verifier.

There are basically three subcomponents of Class loader in Java. They are as follows:

a. Bootstrap class loader

b. Extensions class loader

c. System class loader

a) Bootstrap class loader: Bootstrap ClassLoader loads classes from the location rt.jar.

Bootstrap ClassLoader doesn't have any parent ClassLoader. It is also called as the Primordial ClassLoader.

The bootstrap class loader loads the core Java libraries located in C:\Program Files\Java\jre1.8.0_181\lib.

b) Extensions Class Loader: The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from jre/lib/ext directory or any other directory pointed by the system property java.ext.dirs.

The extensions class loader loads the classes from the extensions directory C:\Program Files\Java\jre1.8.0_181\lib\ext.

c) System Class Loader: An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable CLASSPATH, -classpath or -cp command line option. The Application ClassLoader is a child class of Extension ClassLoader. The system class loader loads the code

from the location specified in the CLASSPATH environment variable which is defined by the operating system.

Principles of functionality of a JAVA ClassLoader:

1. Delegation Model -> The Java Virtual Machine and the Java ClassLoader use an algorithm called the Delegation

Hierarchy Algorithm to Load the classes into the Java memory. The ClassLoader works based on a set of

operations given by the delegation model.

They are:

* ClassLoader always follows the Delegation Hierarchy Principle.

* Whenever JVM comes across a class, it checks whether that class is already loaded in the method area.
* If the Class is already loaded in the method area then the JVM proceeds with execution.
* If the class is not present in the method area then the JVM asks the Java ClassLoader to load that particular class, then ClassLoader hands over the control to Application ClassLoader.
* Application ClassLoader then delegates the request to Extension ClassLoader and the Extension ClassLoader in turn delegates the request to Bootstrap ClassLoader.
* Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
* Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
* Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a ClassNotFoundException exception is generated.

2. Visibility Principle -> The Visibility Principle states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoader. Suppose a class GEEKS.class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception java.lang.ClassNotFoundException.

3. Uniqueness Property -> The Uniqueness Property ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn't able to find the class, only then the current instance would attempt to do so itself.

**Methods of Java.lang.ClassLoader ->

After the JVM requests for the class, a few steps are to be followed in order to load a class. The Classes are loaded as per the delegation model but there are a few important Methods or Functions that play a vital role in loading a Class.

1. loadClass (String name, boolean resolve): This method is used to load the classes which are referenced by the JVM. It takes the name of the class as a parameter. This is of type loadClass(String, boolean).
2. defineClass(): The defineClass() method is a final method and cannot be overriden. This method is used to define a array of bytes as an instance of class. If the class is invalid then it throws ClassFormatError.
3. findLoadedClass(String name): This method is used to verify whether the Class referenced by the JVM was previously loaded or not.
4. Class.forName(String name, boolean initialize, ClassLoader loader): This method is used to load

the class as well as initialize the class. This method also gives the option to choose any one of the ClassLoaders. If the ClassLoader parameter is NULL then Bootstrap ClassLoader is used.

5. findClass(String name): This method is used to find a specified class. This method only finds but doesn't load the class.

Abstraction:

Abstraction is the process by which we show only essential part and hide implementation details. In Java, we can achieve abstraction using abstract class and interface.

1. Abstract Keyword: It is an access modifier, it is applicable for classes, and methods but not variables. It is used to achieve abstraction which is one of the pillars of Object-Oriented Programming (OOP). We use "abstract" keyword to achieve abstraction in java.

Types of abstract methods:

1. Concrete or non-abstract method:

Syntax:

```
accesss_modifier return_type method(parameter list){
  // logic or code implementation;
  }
```

2.Abstract or non-concrete method:

Syntax:

```
accesss_modifier abstract return_type method(parameter list);
```

Rules for Abstraction:

1. It is not mandatory to have abstract method in abstract class.
2. If we define any abstract method in a class, then that class must be abstract class.
3. We cannot create an object of abstract class.
4. If one abstract class extends another abstract class, then it is not mandatory for that child class to provide
   method implementation for abstract method of supper class.
5. If concrete or non-abstract class extends abstract class, then it's mandatory for that child class to provide
   method implementation for abstract method for supper class.

** whole purpose of abstract method is that, child class should provide implementation for abstract method.
** We cannot use public to abstract method when we are using multilevel inheritance.
** We cannot use super keyword in static method.

* Why we need to use Abstraction:

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its

subclasses, leaving it to each subclass to fill in the details.

* When to Use Abstract Method in Java?

There are the following uses of abstract method in Java. They are as follows:

1. An abstract method can be used when the same method has to perform different tasks depending on the object calling it.

2. A method can be used as abstract when you need to be overridden in its non-abstract subclasses.

*** Advantage of Abstract class in Java:**

The main advantages of using abstract class in Java application are as follows:

1. Abstract class makes programming better and more flexible by giving the scope of implementing abstract methods.
2. Programmers can implement an abstract method to perform different tasks depending on the need.
3. We can easily manage code.

2. Interface: An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. A Java interface contains static constants and abstract methods. It is achieved using 'interface' keyword. Methods in interface are by default 'public and abstract', also the variables declared are public, static and final.

- One class can extends only one class, but one class can implements multiple interfaces.
- One interface can extends one or more interface. 'implements' keyword is used to indicate that a class is implementing an interface.
- When we want to achieve 100% abstraction (it must have only abstract method) you can go with interface.
- Rules for Interface and abstract are same.
- It is used to achieve abstraction and multiple inheritances in Java using Interface.
- When we decide on a type of entity by its behavior and not via attribute, we should define it as an interface.
- There is a contract between interface and class, class must provide implementation to the interface method.

* Syntax for Java Interfaces:

```
interface {
   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

* Uses of Interfaces in Java are mentioned below:

1. It is used to achieve total abstraction.
2. Since java does not support multiple inheritances in the case of class, by using an interface it can

achieve multiple inheritances.
3. Any class can extend only 1 class, but any class implement an infinite number of interfaces.
4. It is also used to achieve loose coupling.
5. Interfaces are used to implement abstraction.

** @FunctionalInterface annotation => It is not mandatory to define a Functional interface with @FunctionalInterface annotation. Java does not impose this rule.
But, if we define an interface with @FunctionalInterface annotation, Java Compiler will give an error
in case we define more than one abstract method within that interface.

* Java Constructors:

A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. Every time an object is created using the new() keyword, at least one constructor is called. It can be used to set initial values for object attributes.

** How Java Constructors are Different From Java Methods?*
* Constructors must have the same name as the class within which it is defined it is not necessary for
the method in Java.
* Constructors do not return any type while method(s) have the return type or void if does not return any value.
* Constructors are called only once at the time of Object creation while method(s) can be called any number of times.
* There are no "return value" statements in the constructor, but the constructor returns the current class instance. We can write 'return' inside a constructor.

* *Need* -> constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

* Types of Constructors in Java:
1. Default Constructor -> A constructor that has no parameters is known as default the constructor. A default constructor is invisible, and provided by JVM itself. And if we write a constructor with no arguments, the compiler does not create a default constructor.

2. Parameterized Constructor -> A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.
3. Copy Constructor -> Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

* Constructor Chaining in Java:
Constructor chaining is the process of calling one

constructor from another constructor with respect to current object. One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

* Constructor chaining can be done in two ways:
1. Within same class: It can be done using this() keyword for constructors in the same class
2. From base class: by using super() keyword to call the constructor from the base class.

Constructor chaining occurs through inheritance. A sub-class constructor's task is to call super class's default constructor first. This ensures that the creation of sub class's object starts with the initialization of the data members of the superclass. There could be any number of classes in the inheritance chain. Every constructor calls up the chain till the class at the top is reached.

* Why do we need constructor chaining?
=> This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

* Rules of constructor chaining :
- The this() or super() statement must always be the first line of the constructor.
- There should be at-least one constructor without the this() / super() keyword.
- Constructor chaining can be achieved in any order.

* Alternative method : using Init block :

When we want certain common resources to be executed with every constructor, we can put the code in the init block. Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

Types of variables:

1. Local Variables:
   - A variable that is declared and used inside the body of methods, constructors, or blocks is called local variable in java. We cannot access these local variables outside the method.
   - We must assign a local variable with a value at the time of creating. If you use a local variable without initializing a value, you will get a compile-time error like "variable x not have been initialized".
   - We cannot use access modifiers with local variables.
   - The local variables are visible only within the declared constructors, methods, or blocks.
   - A local variable is not equivalent to an instance variable.
   - A local variable cannot be static.
   - Local variables are stored in stack memory.

2. Instance Variables:

- A variable that is declared inside the class but outside the body of the methods, constructors, or any blocks is called instance variable in Java.

- They are available for the entire class methods, constructors, and blocks. It is also called non-static variable because it is not declared as static.

- Instance variables get memory when an object is created using the keyword 'new' and destroyed when the object is destroyed.

- We can also use access modifiers with instance variables. If we do not specify any modifiers, the default access modifiers will be used which can be accessed in the same package only.

- It is not necessary to initialize the instance variable.

- All instance variables are stored in PermGen space (Permanent Generation space) of heap memory. If the variables are primitive type, then variable and its value both are stored as a name-value pair in the permgen. But if the variable is user-defined (object) then its reference is stored in PermGen but actually, it is stored in Young/old generation of heap memory.

3. Static Variables:

- A variable which is declared with a static keyword is called static variable in Java. A static variable is also called class variable because it is associated with the class.

Static:

- Static is a non-access modifier and is used to create methods or variables that can be accessed without creating an object of a class

- Static variables are always declared inside the class but outside of any methods, constructors, or blocks.

- We can call static methods in static method without creating an object, but we cannot access non-static method in static without creating an object.

- Whatever is static, it belongs to whole class and not any specific object.

- Using static variables, we can reduce the memory uses or we can optimize memory

- Static variable will get the memory only once. If anyone changes the value of the static variable using the class name, it will replace the previous value and display the changed value. This is because it is constant for every object created.

- Memory allocation for static variables happens only once when the class is loaded into the memory and it is destroyed when class unloaded from the memory. All the static variables are stored in PermGen space of heap memory.

* Features of Static Keyword in Java:

There are several important features of static keyword in java that must keep in mind. They are as follows:

1. Static keyword in Java can be applied with variables, methods, inner classes, and blocks.

2. We cannot declare a class with static keyword but the inner class can be declared as static.

3. It belongs to the class than an instance of the class.

4. One basic rule of working with static keyword is that we cannot directly call instance members within the static area because the static members are linked with the class.

5. Static members get memory once when the class is loaded into the memory. But instance members get the memory after the object creation of the class.

* Use of Static Keyword in Java:

There are mainly two uses of java static keyword that are as follows:

1. The main purpose of using static keyword is that we can access the data, method, or block of the class without any object creation. Let's understand it with a simple example. As you know that the main method is static in Java because the object is not required to call the static method. If it is a non-static method then JVM will create an object first and then it will call the main() method which creates the problem of an extra memory location.

2. It is used to make the programs more memory efficient.

* Characteristics of static keyword:

1. Shared memory allocation: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.

2. Accessible without object instantiation: Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.

3. Associated with class, not objects: Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.

4. Cannot access non-static members: Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.

5. Can be overloaded, but not overridden: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

* Advantages of static keyword:

1. Memory efficiency: Static members are allocated memory only once during the execution of the program, which can result in significant memory savings for large programs.

2. Improved performance: Because static members are associated with the class rather than with individual instances, they can be accessed more quickly and efficiently than non-static members.

3. Global accessibility: Static members can be accessed from anywhere in the program, regardless of whether an instance of the class has been created.

4. Encapsulation of utility methods: Static methods can be used to encapsulate utility functions that don't require any state information from an object. This can improve code organization and make it easier to reuse utility functions across multiple classes.

5. Constants: Static final variables can be used to define constants that are shared across the entire program.

6. Class-level functionality: Static methods can be used to define class-level functionality that does not require any state information from an object, such as factory methods or helper functions.

* Difference between local, instance and static variable:

· Use static variables for shared data across all instances of a class or for constants.

· Use instance variables for attributes specific to each object (instance) of a class.

· Use local variables for temporary storage within methods or blocks.

* What are Blocks in Java?

- Blocks are the special type of containers in java language. It acts as a container for executable java statements.

- Blocks mainly perform special type of operations. e.g. Database connectivity, Server Configuration etc.

- Blocks in java are / a statement's which may be inside or outside of a method. It usually treats those statement as single line. Even before the execution of main method the blocks get executed. We need not to call blocks in a main () method.

Blocks are mainly classified into two types:

1. Static Block -> We have to declare static blocks by using static keywords.

We use static block mainly to develop business logic where outcome of the logic is same for all the users and it executes only once in an application lifecycle. If java class contains static block and main method then static block executes before the main method at the time of class loading. It is possible to create multiple static blocks inside the single java class. We can initialize static variables inside the static block.

Syntax:
```
class StaticInit {
    static int z;
// TIllMay2024.Static Initialization Block
    static {
        z = 10;
    }
}
```

2. Non static Block -> We declare non static blocks without using the static keyword.

We use non static blocks to develop business logic which is common for all objects also, which executes multiple times in an application lifecycle. Non static blocks execute at the time of object creation. It is possible to declare multiple non static blocks inside the single java class. We can initialize non static as well as static members inside the non-static block.

Syntax:
```
class InstanceInit {
    int x;
// Instance Initialization Block
    {
        x = 5;
    }
}
```

- When we create non-static & static blocks in a class, we'll get the static blocks printed first and then non-static blocks and then starts execution of main in method.

- When we have multiple static blocks in a class then they'll follow the order in which they are defined.

- The scope of variables declared in a block statement can only be used within that block ie. they have Local scope. The non-static block executes only after creating an object of that class but static block does not need an object creation of class.

- Instance block also follows order in which they are declared. Instance block gets executed every time when Object of a class are created.

* Difference between Methods and Blocks in Java:

1. Methods in Java are responsible to perform normal business logic operation whereas, blocks perform special operations.

2. Developer is responsible to call the method explicitly for execution while blocks will be automatically executed by JVM.

3. Every method is having an identity but blocks do not have any identity.

4. We can pass parameters by using methods but it is not possible to pass input parameters for blocks.

5. It is possible to return value from method whereas you cannot return a value from block.

* Encapsulation in Java:

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Difference between Abstraction and Encapsulation:

1. Purpose:

- Abstraction focuses on hiding unnecessary implementation details and exposing only essential features of an object. It simplifies complexity by modeling real-world entities as objects with well-defined behaviors and characteristics.

- Encapsulation, on the other hand, emphasizes bundling data (attributes or properties) and methods (functions or procedures) that operate on the data into a single unit, called a class. It restricts direct access to the internal state of objects and provides controlled access through well-defined interfaces.

2. Scope:

- Abstraction is a broader concept that encompasses the process of modeling real-world entities as objects with simplified representations. It involves identifying essential characteristics and behaviors of objects and hiding irrelevant details.

- Encapsulation is a specific technique within abstraction that focuses on bundling data and methods together within a class and controlling access to them. It ensures that the internal state of objects is accessed and modified only through well-defined interfaces (getter and setter methods).

3. Level of Detail:

- Abstraction involves abstracting away unnecessary details to create a simplified view of objects and their interactions. It focuses on defining what an object does without specifying how it does it.

- Encapsulation involves encapsulating data and methods within a class and hiding the implementation details from the outside world. It focuses on defining the interface through which the internal state of objects can be accessed and modified.

4. Implementation:

- Abstraction is typically achieved through the use of abstract classes, interfaces, and inheritance. It allows for the creation of generalized, reusable components that can be extended and specialized by subclasses.

- Encapsulation is implemented using access modifiers (such as public, private, protected) to control the visibility of members (variables and methods) of a class. It ensures that the internal state of objects is protected from unauthorized access and modification.

In summary, while abstraction and encapsulation are related concepts in OOP, they serve different purposes and operate at different levels of abstraction. Abstraction focuses on simplifying the representation of objects, while encapsulation focuses on bundling data and methods together and controlling access to them. Together, they contribute to creating well-designed, modular, and maintainable software systems.

* POJO:

POJO in Java stands for Plain Old Java Object. Generally, a POJO class contains variables and their Getters and Setters. It is not tied to any Java Framework, any Java Program can use it. It increases the readability & re-usability of a Java program.

* Advantages of Encapsulation:

1. Data Hiding: It is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.

2. Increased Flexibility: We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables write-only then we have to omit the setter methods like setName(), setAge(), etc. or if we wish to make the variables read-only then we have to omit the get methods like getName(), getAge(), etc.

3. Reusability: Encapsulation also improves the re-usability and is easy to change with new requirements.

4. Testing code is easy: Encapsulated code is easy to test for unit testing.

5. Freedom to programmer in implementing the details of the system: This is one of the major advantages of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

* Disadvantages of Encapsulation in Java =>

1. Can lead to increased complexity, especially if not used properly.

2. Can make it more difficult to understand how the system works.

3. May limit the flexibility of the implementation.

* Why Encapsulation?

Ans -> 1. Better control of class attributes and methods

2. Class attributes can be made read-only (if you only use the get method), or write-only (if you only use the set method)

3. Flexible: the programmer can change one part of the code without affecting other parts

4. Increased security of data

*Polymorphism:

Polymorphism means the ability of an object to take on multiple forms or behaviors. In Java, polymorphism can be achieved through method overriding and method overloading. Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. Method overloading allows multiple methods with the same name but different parameter lists to coexist within the same class. Polymorphism enables flexibility, extensibility, and code reusability in object-oriented systems

- Java does not support operator overloading or polymorphism, it only support method polymorphism.

- Java allows us to use same method name with different parameter list.

* Method Signature:

In Java, a method signature is part of the method declaration. It's the combination of the method name and the parameter list. The reason for the emphasis on just the method name and parameter list is because of overloading. It's the ability to write methods that have the same name but accept different parameters.

1. Method Overloading or compile time or static binding or early binding:

- When we have multiple methods with same name but with different parameters in the same class is known as method overloading.

- Method overloading in Java is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding.

- In Method overloading compared to the parent argument, the child argument will get the highest priority.

- We can also overload static method, main method and constructors.

- Java compiler differentiates overloaded methods with their signatures.

- Different Ways of Method Overloading in Java:

Changing the Number of Parameters.

Changing Data Types of the Arguments.

Changing the Order of the Parameters of Methods

- Rules of method overloading:

1. Method overloading happens within same class.
2. Method signature must be different.
3. Access modifiers can be anything or different.
4. Return type can be anything or different.
5. Exception thrown can be anything.

6. The implementation does not matter in this case. A method can have any kind of logic.

Operator overloading:

Operator overloading is the ability to redefine the functionality of the operators. Programming languages like c++ supports operator overloading. you can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

* Java does not support operator overloading due to the following reasons:

1. Makes code complex − In case of operator overloading the compiler and interpreter (JVM) in Java need to put an extra effort to know the actual functionality of the operator used in a statement.

2. Programming error − Custom definition for the operators creates confusion for the programmers especially the new developers. Moreover, while working with programming languages that support operator overloading the program error rate is high compared to others.

3. Easy to develop tools like IDEs − Removal of operator overloading concept keeps the language simple for handling and process leading to a number of Integrated development environment in Java.

4. Method overloading − The functionality of operator overloading can be achieved in Java using method overloading in Java in a simple, error free and clear manner.

* Advantages of Method Overloading:

Method overloading improves the Readability and reusability of the program.

Method overloading reduces the complexity of the program.

Using method overloading, programmers can perform a task efficiently and effectively.

Using method overloading, it is possible to access methods performing related functions with slightly different arguments and types.

Objects of a class can also be initialized in different ways using the constructors.

* Variable Argument (Varargs):

The varargs allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many arguments we will have to pass in the method, varargs is the better approach.

* Advantage of Varargs:

We don't have to provide overloaded methods so less code.

* Syntax of varargs:

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

```
return_type method_name(data_type... variableName){}
```

* Que: What is the difference between Overloading and Overriding?

Answer: Overloading is about the same function having different signatures. Overriding is about the same function, and same signature but different classes connected through inheritance.

2. Method overriding or Run time or Dynamic Binding or Late Binding:

When we provide different implementation to parent class method, is called as method overriding.

* When should we use method overriding?

Ans: When we want to change the implementation provided in parent class method, we override that method in child class and provide different implementation.

* Rules for method overriding:

1) In java, a method can only be Overridden in Subclass, not in same class. (It should have parent child relationship)

2) The argument list/method signature should be exactly the same as that of the overridden/parent class method.

3a) Return type must be same for both parent and child class methods. -> Primitive data types

3b) The return type should be the same or a subtype of the return type declared in the original

overridden method in the super class. Covariant return types. -> Object

4) The access level cannot be more restrictive than the overridden method's access level.

For example: if the super class method is declared default then the over-ridding method in the sub class be either protected or public.

5) A method declared final cannot be overridden.

6) A method declared static cannot be overridden but can be re-declared.

7) If a method cannot be inherited then it cannot be overridden. Ex: private methods in super class.

8) A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.

9) A subclass in a different package can only override the non-final methods declared public or protected.

10) Constructors cannot be overridden.

11) An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.

(Example: Wedding card => Name => Father but Child can attend the marriage)

* What is the real life/practical examples of a Java overloading and overriding?

Ans:

overriding: suppose there is a method getInterestRate() which returns the interest rate of a bank. RBI is the superclass and it returns 7 for getInterestRate(). There are various banks like sbi, axis, icici, etc which extend RBI class and override the getInterestRate() method to return 7.5, 8, 8.5, etc respectively.

overloading: The payment option on any ecommerce website has several options like net_banking, COD, credit card, etc. That means, a payment method is overloaded several times to perform single payment function in various ways.

* Introduction to Java Casting:

In Java, casting is a technique that converts an object of a class into another. Casting allows you to access fields and methods that are specific to a target class. Technically, you should only use casting when there is an actual relationship between the classes in the class hierarchy. It means that you can only cast an object of classes that have a superclass-subclass relationship. If you attempt to cast unrelated classes, you will an error runtime. More specifically, you'll get an ClassCastException.

Java supports two types of object castings: upcasting and down-casting.

1. Upcasting: Upcasting converts an object of a subclass to its superclass:

Syntax: Superclass object = new Subclass(); // upcasting

In this syntax, we create a new object of a subclass and assign it to a reference variable with the type of Superclass. The upcasting is always safe and doesn't require an explicit cast. In the following syntax, we assign an instance of a subclass to a reference of a superclass:

Superclass object = subclassObject; // upcasting

2. Down-casting: Down-casting is the casting of an object of a superclass to a subclass:

Syntax: Subclass subObj = (Subclass) superObj; // Down-casting

Note that you can only perform a down-casting if the original object is of the subclass type. And you need to use parentheses and the subclass type. For safety, you can use the instanceof operator to check if the object being down-casted is an instance of the Subclass:

```
if (obj instanceof Subclass) {
   Subclass subObj = (Subclass) obj; // Downcast only if safe
}
```

* Why java does not fully support down casting:

Java prioritizes compile-time type safety, which down-casting cannot fully verify. Down-casting can lead to runtime exceptions like ClassCastException, making it a potential source of errors. Java's design favors composition over inheritance, discouraging the overuse of down-casting.

* Summary:

1.Use Java casting to convert an object of one class to another, within the class hierarchy.

2.Use down-casting to convert an object of a superclass to its subclass.

3.Use upcasting to convert an object of a subclass to its superclass.

* Typecasting in Java:

Typecasting in Java is the process of converting one data type to another data type using the casting operator. When you assign a value from one primitive data type to another type, this is known as type casting. To enable the use of a variable in a specific manner, this method requires explicitly instructing the Java compiler to treat a variable of one data type as a variable of another data type.

Syntax:

<datatype> variableName = (<datatype>) value;

* Types of Type Casting:

There are two types of Type Casting in java:

1. Widening Type Casting: A lower data type is transformed into a higher one by a process known as widening type casting. Implicit type casting and casting down are some names for it. It occurs naturally. Since there is no chance of data loss, it is secure.

   Widening Type casting occurs when: The target type must be larger than the source type. Both data types must be compatible with each other.

Syntax: larger_data_type variable_name = smaller_data_type_variable;

2. Narrow Type Casting: The process of downsizing a bigger data type into a smaller one is known as narrowing type casting. Casting up or explicit type casting are other names for it. It does not just happen by itself. If we don't explicitly do that, a compile-time error will occur. Narrowing type casting is unsafe because data loss might happen due to the lower data type's smaller range of permitted values. A cast operator assists in the process of explicit casting.

   Syntax: smaller_data_type variable_name = (smaller_data_type) larger_data_type_variable;

* Types of Explicit Casting:

Mainly there are two types of Explicit Casting:

1. Explicit Upcasting: Upcasting is the process of casting a subtype to a supertype in the inheritance tree's upward direction. When a sub-class object is referenced by a superclass reference variable, an automatic process is triggered without any further effort.

2. Explicit Down-casting: When a subclass type refers to an object of the parent class, the process is referred to as down-casting. If it is done manually, the compiler issues a runtime ClassCastException error. It can only be done by using the instanceof operator. Only the downcast of an object that has already been upcast is possible.