Java

- Java is a class-based, object-oriented programming language developed by James Gosling at Sun Microsystems in the year 1995.
- Java is not a fully object-oriented programming language as it supports primitive datatypes like int, float, etc., which are not objects.
- Java is based on the Write Once, and Run Anywhere (WORA) principle, meaning that the compiled Java code can run on all machines that support Java without the need for recompilation.
- Java is owned by Oracle and is used for:
    1. Mobile applications (Especially Android apps)
    2. Web applications
    3. Games
    4. Database Connections
    5. And much, much more!

History of Java

- Java is a programming language created in 1991 by **James Gosling, Mike Sheridon and Patrick Naughton**, a team of Sun engineers known as the **Green Team**.
- First public implementation of Java was released in1996 as **Java 1.0**

Java Version	Release Date	Major Features
Java SE 1.0	January 23, 1996	Initial release
Java SE 1.1	February 19, 1997	Inner classes, JavaBeans, JDBC, RMI
Java SE 1.2	December 8, 1998	Swing, Collections Framework, JIT Compiler
Java SE 1.3	May 8, 2000	HotSpot JVM, RMI Custom Socket Factories, Java Sound API
Java SE 1.4	February 6, 2002	assert keyword, Regular Expressions, Exception Chaining, NIO
Java SE 5.0	September 30, 2004	Generics, Enhanced for loop, Autoboxing/Unboxing, Enumerated Types, Metadata (Annotations), Varargs

Java SE 6	December 11, 2006	Scripting Language Support, Web Services, Compiler API, Improved GUI, JAX-WS, JAXB 2.0
Java SE 7	July 28, 2011	Fork/Join Framework, Diamond Operator, Strings in switch, try-with-resources, NIO.2, G1 Garbage Collector
Java SE 8	March 18, 2014	Lambda Expressions, Stream API, Date and Time API (JSR 310), Nashorn JavaScript Engine, Optional
Java SE 9	September 21, 2017	Module System (Project Jigsaw), JShell, HTTP/2 Client, Multi-Release JAR files
Java SE 10	March 20, 2018	Local-Variable Type Inference (var), Experimental JIT Compiler
Java SE 11	September 25, 2018	Long-Term Support (LTS), HTTP Client (Standard), Flight Recorder, Local-Variable Syntax for Lambda Parameters
Java SE 12	March 19, 2019	Switch Expressions (preview), JVM Constants API, Microbenchmark Suite
Java SE 13	September 17, 2019	Text Blocks (preview), Switch Expressions (preview), Reimplement the Legacy Socket API
Java SE 14	March 17, 2020	Switch Expressions, Pattern Matching for instanceof (preview), Records (preview), Helpful NullPointerExceptions
Java SE 15	September 15, 2020	Text Blocks, Hidden Classes, Pattern Matching for instanceof, Records (second preview), Sealed Classes (preview)
Java SE 16	March 16, 2021	Records, Pattern Matching for instanceof, Sealed Classes (second preview), Foreign-Memory Access API (incubator)
Java SE 17	September 14, 2021	Long-Term Support (LTS), Sealed Classes, Pattern Matching for switch (preview), Strong encapsulation
Java SE 18	March 22, 2022	UTF-8 by Default, Simple Web Server, Code Snippets in Java API Documentation
Java SE 19	September 20, 2022	Virtual Threads (preview), Structured Concurrency (incubator), Record Patterns (preview)
Java SE 20	March 21, 2023	Second Preview of Virtual Threads, Structured Concurrency (second incubator), Pattern Matching for switch (second preview)

The first version of Java is Java 1.0 which was released in 1996 and the latest version is Java 22 which is released in 2024.

Why Java is named as Java

- James Gosling and his team initiated a project to develop a language for digital devices such as set-top boxes, television, etc., and called this project **Greentalk** and its file extension was .**gt** and later become to known as **OAK**.
- The name OAK was used by Gosling after an OAK tree that remained outside his office and OAK was also a national tree of so many nations like USA, France, Germany, etc. But later they had to rename it as it was already a trademark of **OAK Technologies**.
- Gosling and his team did a brainstorm session after which they came up several names out of which **JAVA** was decided after much discussion.
- **Java** is the name of island in Indonesia where the **first coffee** (named as Java) was produced, and this name was chosen by Gosling while having coffee near his office.

Key Terminology

Before learning Java, one must be familiar with the following terms of Java:

- **JVM (Java Virtual Machine)**
- **Bytecode**
- **JDK (Java Development Kit)**
- **JRE (Java Runtime Environment) or Java RTE**
- **Garbage Collector**
- **Classpath**

JVM (Java Virtual Machine):

- JVM acts as a **run-time** engine to run Java applications. JVM is the one that calls the **main** method present in Java code. JVM is a part of **JRE**.
- The compilation phase of a Java program is done by **JAVAC** compiler which is a primary Java compiler included in the Java Development Kit (JDK). It takes the program as input and generates bytecode as output.

- In the running phase of a program, JVM executes the bytecode generated by compiler.
- The main purpose of JVM is to execute the bytecode produced by the JAVAC compiler. Every Operating System has a different JVM but the output they produce after the execution of byte is same across all the systems. This is why Java is also known as a **platform-independent language.**

Bytecode:

- The JAVAC compiler of JDK compiles source code to bytecode so that it can be executed by JVM.
- This bytecode is saved as .**class** file by the compiler. To view the bytecode, a disassembler like **javap** is required.

JDK (Java Development Kit):

- It is a **complete kit** that includes everything including **compiler, JRE, Java debugger (JDB), Java docs, etc.**
- For a program to execute in Java, we need to **install JDK** on our computer to **create, run and compile the Java program**.

JRE (Java Runtime Environment):

- JRE is a part of JDK which allows a Java program to run soon after the installation.
- JRE works as a translator and a facilitator between a Java program and an operating system. It is made up of multiple elements which are:
    - **JVM**
    - **Java class libraries**
    - **Java class loaders**

Garbage Collector:

- **Garbage collection** in Java is a process by which Java programs perform **automatic memory management**.
- When Java programs run on the JVM objects are created on the heap, which is a portion of memory dedicated to the program. Eventually some objects will no longer be needed. The **garbage collector** finds these **unused objects** and **deletes** them to **free up memory**.
- **Java garbage collection** is an **automatic process** of looking at heap memory, identifying which objects are in use and which are not, and **deleting** the **unused objects**.
- An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program.

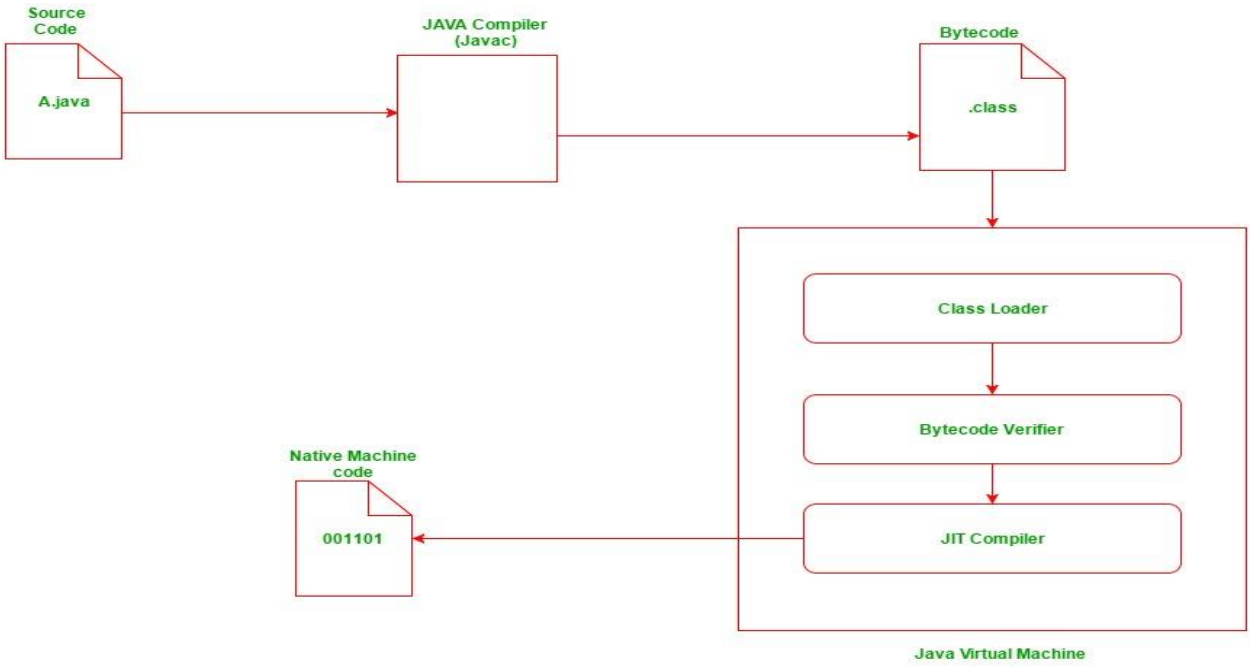
Classpath:

The classpath is the **file path** where the **Java runtime** and **Java compiler** look for **.class files** to **load**. By default, JDK provides many libraries. If you want to include external libraries, they should be added to the classpath.

Features of Java

1. Java is **platform-independent**, which means that code written in Java can run on any platform that has a Java Virtual Machine (JVM) installed.
2. Java is known for its "**write once, run anywhere**" philosophy, which makes it a popular choice for cross-platform development.
3. Java provides **automatic memory management** through garbage collection, which makes it easier to write and maintain code.
4. Java is a **strongly typed language**, which means that every variable and expression has a specific type that must be declared before use.
5. Java supports **multithreading**, which makes it possible to write programs that can perform multiple tasks simultaneously.

Execution of a Java program



This diagram illustrates the process of executing a Java program. Here's a summary and explanation of each component:

- **Source Code (A.java):** The Java source code written by the programmer.
- **Java Compiler (Javac):** Compiles the source code into bytecode.
- **Bytecode (.class):** The compiled intermediate code that is platform independent.
- **Class Loader:** Loads the .class files into the Java Virtual Machine (JVM).
- **Bytecode Verifier:** Checks the bytecode for security and correctness.
- **JIT Compiler:** Just-In-Time compiler converts bytecode into native machine code during runtime.
- **Native Machine Code (001101):** The platform-specific machine code executed by the CPU.
- **Java Virtual Machine (JVM):** The environment that loads, verifies, and executes Java bytecode.

Why Only public static void main(String[] args)

- The execution of every Java program begins from a particular standard method called **main().** The Java Virtual Machine (JVM) looks for the main() method and treats it as the entry point. Any program without main() cannot be executed by the JVM.
- The main() method in Java has the following syntax: **public static void main(String[] args).** If we change this syntax, we will get errors and our program will not run, as this specific syntax is required to execute a program.

Breakdown of *public static void main(String[] args):*

public:

- The method must be public so that the Java runtime can access it.
- If it were not public, it would not be accessible from outside the class, including by the JVM.

static:

- The method must be static so that it can be called without creating an instance of the class.
- The JVM needs to call this method without creating an object of the class. Since it is static, it belongs to the class itself rather than any specific instance of the class.
- The methods declared using **static** keyword has following restrictions:
    1. They must access only static data,
    2. They can only call other static methods.
    3. They cannot refer to this or super.

void:

- The method returns no value as it is the starting point for the program, and there's no need for the JVM to expect any return value from this method.
- The void keyword is used when we expect nothing in return by function.

main:

- This is the name that the JVM looks for as the entry point of a Java application.

String[] args:

- This is an array of String objects that allows the Java application to accept command-line arguments.
- When a Java program is run, any command-line arguments are passed to the main method through this parameter.

Note

- The **parameter** of **main()** must be a **String[] type** only, even though we do not want to pass any arguments through command line.
- We can declare the same **String[] as String…** also and it's not mandatory to name the **parameter** as **args only**, we can change it as per our wish.

Useful Links

How to Download and Install Java for 64-bit machine?

Setting up the environment in Java

How to Download and Install Eclipse on Windows?

Primitive and Non-Primitive Data types in Java

Every variable in Java has a data type. Data types specify the size and type of values that can be stored in an identifier.

In Java Data types are classified into two categories:

- **Primitive data type or intrinsic or built-in data type**
- **Non-Primitive data type or derived or reference data type**

Primitive Data Types:

- In Java, the Primitive Data Types are the **predefined data types**. They specify the size and type of any standard value.
- Java has 8 primitive data types:
    1. **byte (1 byte – 8 bits) (-128 to 127) (default value: 0)**
    2. **short (2 bytes – 16 bits) (-32768 to 32767) (default value: 0)**
    3. **int (4 bytes – 32 bits) (-2 ^ 31 to + 2 ^ 31- 1) (default value: 0)**
    4. **long (8 bytes – 64 bits) (-2 ^ 63 to 2 ^ 63 - 1) (default value: 0)**
    5. **float (4 bytes – 32 bits) (default value: 0.0f)**
    6. **double (8 bytes - 64 bits) (default value: 0.0d)**
    7. **char (2 bytes – 16 bits) (0 to 65,535 unicode characters)**
    8. **boolean (1 bit) (true or false) (default value: false)**

Note

Use float for **memory efficiency** when **precision** (around 6-7 decimal places) is sufficient. **Use double** for **high precision** (around 15-16 decimal places) when **memory is not a major concern**.

Non-Primitive Data Types:

- Non-Primitive Data Types are also called **Object Data Types** or **Referenced Data Types** because **they refer to any object**.
- Unlike the Primitive Data Types, the Non-Primitive Data Types are created by the users in Java.
- Java has 5 Non-Primitive Data Types:
  Arrays
    - Array is a **linear data structure** where all elements are arranged sequentially. It is a collection of elements of the **same data type** stored at contiguous memory locations.

Classes

- A **class** is a user-defined blueprint or prototype from which objects are created.

Strings

- String is a **collection of characters** surrounded by double quotes which are used to store texts.
- In Java, string objects are **immutable**. Immutable simply means unmodifiable or unchangeable. Once a string object is created its data or state can't be changed but a new string object is created.

Interfaces

- An **Interface** in Java programming language is defined as an abstract type used to specify the behavior of a class.

Enums

- It is just **like a class** but can have **only constants** that are **public, static, final** which are **unchangeable**, cannot be overridden. An **enum** cannot be used to create objects or **cannot extend** any class but can extend interfaces.

Note

- The **JVM initializes default values** to **class-level variables** when we forget to initialize them, but coming to **local variables** the **default values will not be initialized by JVM**, if we don't initialize local variables we will get compilation errors.
- **Primitive values** are **stored** on the **stack**. Copying a primitive variable creates a separate copy; changes to the copy do not affect the original.
- **Reference variables** are **stored** on the **stack**, but **original objects** are **stored** on the **heap**. Copying a reference variable creates another reference to the same object; changes to the object are reflected across all references.

Java Identifiers & Rules

- In Java, **Identifiers** are used for **Identification purposes,** and they can be **class names, variable names, method names.**
- Identifiers define the way we should name variables, classes, methods, etc, in our program.
- **Allowed Characters:** Identifiers can use alphanumeric characters (A-Z, a-z, 0-9), $ (dollar sign), and _ (underscore). Special characters like @ are not allowed.
- **Starting Character:** Identifiers should not start with digits (0-9).
- **Case Sensitivity:** Identifiers are case-sensitive.
- **Length:** No limit on identifier length, but 4 -15 characters is recommended.
- **Reserved Words:** Reserved words cannot be used as identifiers.

Reserved Words in Java

- In Java a keyword is a reserved word that have a predefined meaning
- There are 68 reserved words in Java as of 2024 and they are as follows:
  abstract, assert, boolean, break, byte, case, catch, char, class, const (reserved but not used), continue, default, do, double, else, enum, exports, extends, final, finally, float, for, goto (reserved but not used), if, implements, import, instanceof, int, interface, long, module, native, new, null, opens, package, private, protected, provides, public, requires, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, to, transient, try, uses, void, volatile, while, with, yield, var, record, sealed, permits, and non-sealed.
- Refer this link for all keywords: Keywords in Java

Java Variables & Variable Scope

- In Java, Variables are the data containers that store data values during Java program execution.
- Every Variable in Java is assigned a data type that designates the type and quantity of value it can hold. A variable is a memory location name for the data.
- Variables are classified into 3 categories:
    1. **Local Variables**
    2. **Instance Variables**
    3. **Static Variables**
- **Scope of a variable** is the part of the program where the variable is accessible.
- Variables have the following scopes:
    1. **Member Variables (Class Level Scope)**
    2. **Local Variables (Method Level Scope)**
    3. **Loop Variables (Block Scope)**

Wrapper Classes

- A Wrapper class in Java is a class whose object wraps or **contains primitive data types**. When we create an object to a wrapper class, it contains a field and, in this field, we can store primitive data types.
- Wrapper Classes convert primitive data types into Objects. Objects are needed to modify the arguments passed into a method because primitive data types are **passed by value**.
- **Data structures** in the **Collection framework** such as **ArrayList**, **HashSet** store only objects (reference types) and not primitive data types.
- The process of converting primitive types to objects of their corresponding wrapper classes is called **autoboxing**.
- **Example**: *char c = 'a'; Character d = c;*
- The process of converting an object of a wrapper class to its corresponding primitive data type is called **unboxing**.
- **Example:** *Character c = 'a'; char d = c;*

Output Formatting using printf

In programming it is essential to print the output in a given **format**.
Most of us are familiar with **prinntf( )** in C to **format output** but we can do the same using **printf( )** in Java.

Example:

```
import java.util.Date;
public class MyClass {
    public static void main(String args[]) {
      int semFee=100000;
      double cgpa = 9.4200;
      boolean isPlaced = false;
      String name = "Kalyan";
      char gender = 'm';
      Date time = new Date();
      System.out.printf("%,d\n",semFee);
      System.out.printf("%.2f\n",cgpa);
      System.out.printf("%b\t%B\n",isPlaced,isPlaced);
      System.out.printf("%s\t%S\n",name,name);
      System.out.printf("%c\t%C\n",gender,gender);
      System.out.printf("Current Time: %tT\n", time);
      System.out.printf("Hours: %tH  Minutes: %tM Seconds: %tS\n", time,time, time);

    }
}
```

Output:

```
100,000
9.42
false   FALSE
Kalyan  KALYAN
m   M
Current Time: 17:06:14
Hours: 17  Minutes: 06 Seconds: 14
```

BufferedReader Class

- BufferedReader is a simple class that is used to read a sequence of characters. It has a simple function read 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 which expects a stream of characters. BufferedReader can throw checked Exceptions.

StringBuffer Class

- StringBuffer is a class in Java that represents a **mutable sequence** of characters. It provides an **alternative** to **immutable String class**, allowing us to modify the contents of a string class without creating new objects.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters pre allocated than needed, to allow room for growth.
- The **default capacity** of **StringBuffer** is **16 characters**, we can specify size when creating a StringBuffer.
- StringBuffers are **thread safe** and doesn't support **synchronization**.
- **Thread safety** means that multiple threads can access it concurrently.
- Some methods of **StringBuffer**:
  append( ), insert( ), reverse( ), delete( ), replace( )

StringBuilder Class

- The function of StringBuilder is very similar to the **StringBuffer** class, as both provide an alternative to String Class by making a mutable sequence of characters and it include same methods and functionalities.
- StringBuilders are not **thread safe** but supports **synchronization**.
- **Synchronization** is used to make sure that some synchronization methods are accessed by single thread only.

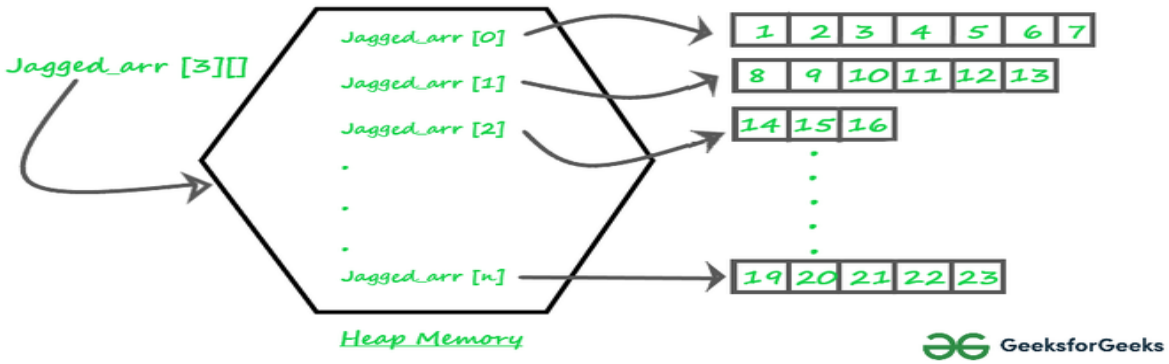
Note

***StringBuffers** are **thread-safe** but **doesn't support synchronization** whereas **StringBuilders** are **not thread-safe** but **supports synchronization**.*

Jagged Arrays

- A [jagged array](#) is an **array of arrays** such that member arrays can be of different sizes i.e. we can create a 2-D array with **variable number of columns** in each row.
- A **2D array** in Java has a **uniform structure** where all rows have the same number of columns, while a **jagged array** has a non-uniform structure consisting of rows that can have **varying lengths**.

Pictorial representation of Jagged array in Memory:



OOPs

- [Object-Oriented Programming (OOP)](#) is a methodology or paradigm to design a program using **Classes** and **Objects**.
- **OOP** organizes code into **classes** which acts as **blueprints** to create **objects**. These **objects** are defined by their attributes and properties in a class and contain **encapsulated data**.
- **OOP** is mainly based on the following **four pillars**:
    1. **Encapsulation**
    2. **Inheritance**
    3. **Polymorphism**
    4. **Abstraction**

Advantages of OOPs:

1. Code Reusability
2. Reduced Redundancy
3. Modularity

Disadvantages of OOPs:

1. Lengthy Programs leading to slow execution.
2. Challenges in Debugging and Testing.

Key Terminology of OOPs

Below is the list of Key Terminologies commonly used in java:

1. **Classes**
2. **Objects**
3. **Constructors**
4. **Constructor Chaining**
5. **This keyword**
6. **Interfaces**
7. **Access Modifiers**

Classes:

- A **class** is a user-defined blueprint or prototype from which objects are created.
- A class can also be defined as collection of data members and member functions.
- Classes do not occupy any memory until an object is instantiated.
- Whenever an object is created using a new keyword, the actual object is stored in the heap and the pointing address is stored in the stack.

Objects:

- An object is a basic unit of OOP that represents real-life entities.
- It is an instance of a class.

Constructors:

- In Java, constructor is a **special method** which is invoked automatically at the time of **object creation**. It is used to initialize the data members of new objects.
- **Constructors** have the **same name** as class name, and they do not have any **return type** (not even void).
- Constructors can be **overloaded** based on the number of arguments and the type of arguments passed during object creation.
- Constructors are only called once, during object creation and if we do not create any constructor Java creates a default constructor by itself where we cannot initialize values as it is default one.
- Constructors are of three types:
    1. **Non - Parameterized or Default Constructors**
    2. **Parameterized Constructors**
    3. **Copy Constructors**

Non - Parameterized or Default Constructors:

- A constructor that is created by the programmer and has no parameters is called **Non – Parameterized Constructor**.
- When do not create a constructor, a constructor without any parameters will be automatically created by Java which is called **Default Constructor.**

Parameterized Constructors:

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**.

Copy Constructors:

Unlike other constructors, the **copy constructor** is passed with another object which copies the data available from the passed object to the newly created object.

Constructor Chaining:

- Constructor chaining is the process of calling one constructor from another constructor with respect to the current object.
- Constructor chaining in the same class can be done using **this( )** keyword for constructors in the same class.
- Constructor chaining from the base class can be done 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 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.

This Keyword:

- In Java **this keyword** is used to refer to the current instance of the class.
- It is used to pass the current objects as a parameter to another object.
- It is also used to refer to the current class instance variable.

Interfaces:

- An **Interface** in Java programming language is defined as an abstract type used to specify the behavior of a class.
- The interface in Java is a mechanism to achieve **abstraction** which also represents **IS-A** relationship.
- There can be **only abstract methods** which means all the methods in an interface are declared with an **empty body** and all fields are **public, static, and final by default**. It is used to achieve abstraction and multiple inheritance in Java.
- To declare an interface, use the **interface keyword.**
- To implement the interface for class, use the **implements keyword**.
- In an interface, you **can't instantiate variables** and create an object.

Access Modifiers:

Access Modifiers defines the access type of the method, class, variable, i.e. from where it can be accessed in your application. In Java, there are 4 types of access specifiers:

- **public**: Accessible in all classes in your application.
- **protected**: Accessible within the package in which it is defined and, in its subclass, (including subclasses declared outside the package).
- **private**: Accessible only within the class in which it is defined.
- **default** (declared/defined without using any modifier): Accessible within the same class and package within which its class is defined.

Encapsulation

- Encapsulation is one of the core pillars of OOP where the internal details of an object are hidden, only specific functions are provided to interact with that object's data.
- In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively.
- 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.

Example:

- Imagine a **capsule** you take as **medicine**. The capsule contains medicine inside it, but you don't see or touch the medicine directly. The capsule ensures that the medicine reaches your stomach safely without being tampered with by your hands, air, or anything else.
- This is like encapsulation in programming:

1. **The Capsule**: Represents the class in OOP.
2. **The Medicine Inside**: Represents the data and methods in the class.
3. **The Shell of the Capsule**: Represents the protective barrier that hides the inner workings (data and methods) from the outside world.

- Just like the capsule hides and protects the medicine, encapsulation in programming hides the internal state and functionality of an object, exposing only what is necessary for other parts of the program to use.
- **Advantages**: Data Hiding.
- **Disadvantages**: Increased Complexity.

Inheritance

- Inheritance in OOP is the mechanism by which one class is allowed to **inherit (acquire)** the **features (fields and methods)** of another class.
- A class that inherits (acquires) from another class can reuse the methods and fields of that class with or without adding new fields and methods.
- The class whose features are inherited using **extends keyword** is known as **super class** or **parent class** or **base class**.
- The class which **inherits (acquires)** the fields or methods is called **sub class** or **child class** or **derived class**.
- **Advantages**: Code reusability, abstraction.
- **Disadvantages**: We cannot restrict to required properties acquisition; Changes made in the super class will be affecting the sub class.

Note:

1. **Constructors** and **private data members without getters and setters** cannot be inherited using inheritance.
2. A super class can have **any number of sub classes**, but a sub class can have only **one super class**.
3. Java doesn't support **multiple inheritance** for which we can use **Interfaces**.

Types of Inheritance

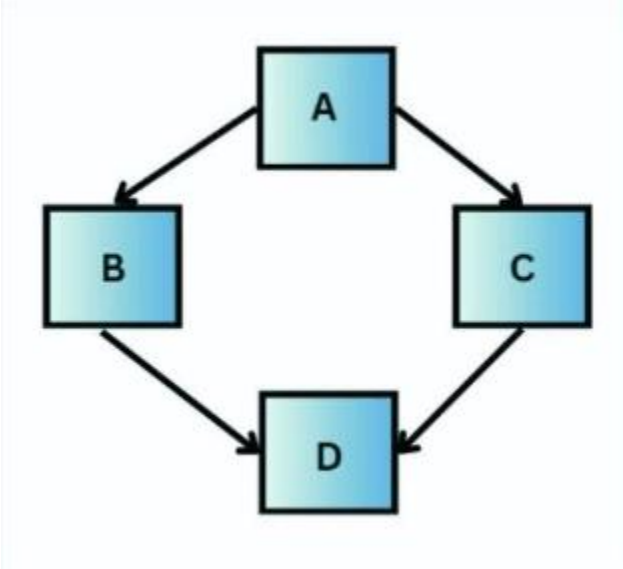
Single Inheritance: When one sub class inherits from one super class, we call this inheritance as Single Inheritance.

Hierarchical Inheritance: When more than one sub class is inherited from same super class, we call this type of inheritance as Hierarchical Inheritance.

Multilevel Inheritance: When a derived class will be inheriting a base class, and as well as the same derived class also acts as the base class for other classes, we call this type of inheritance as Multilevel inheritance.

Hybrid Inheritance:

- Hybrid Inheritance is a combination of two or more types of Inheritance.
- It refers to the ability of a class to inherit properties and behaviors from multiple sources, combining different types of inheritance.
- Since Java does not support multiple inheritance, hybrid inheritance is implemented using a combination of class inheritance and interface implementation.



Common combinations of inheritance that can be considered as hybrid inheritance in Java:

- Single Inheritance with Interfaces.
- Multilevel Inheritance with Hierarchical Structure.
- Hierarchical Inheritance Along a Single Inheritance Path.
- Combining Interfaces with Multilevel Inheritance.

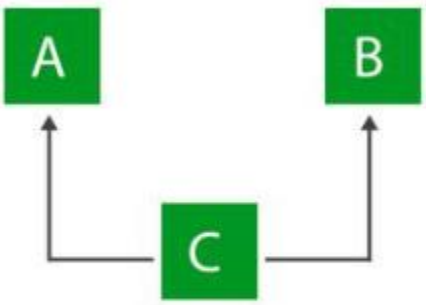
In the diagram above,

- A could be a superclass or an interface that B and C are either extending or implementing.
- B and C are subclasses or interfaces. If A is a class, B and C are subclasses extending A. If A is an interface, B and C could be interfaces extending A or classes implementing A.
- D is a subclass that is extending class B and extending or implementing C. If B and C are both interfaces, D can implement both (multiple inheritance of interfaces is allowed). If B is a class and C is an interface, D is extending B and implementing C (a combination of class inheritance and interface implementation).
- The diagram could represent D as a class that inherits from class B and implements the interface C, while both B and C extend or implement A. This creates a hybrid structure by combining interface implementation with class inheritance.

Multiple Inheritance:

- When one class can have **more than one super class** and can inherit features from all super classes, we call this type of inheritance as Multiple inheritance.
- Java **doesn't support** Multiple Inheritance with **classes**. In Java we can achieve multiple inheritance only through **interfaces**.

In the Image class C is derived from interface A and interface B



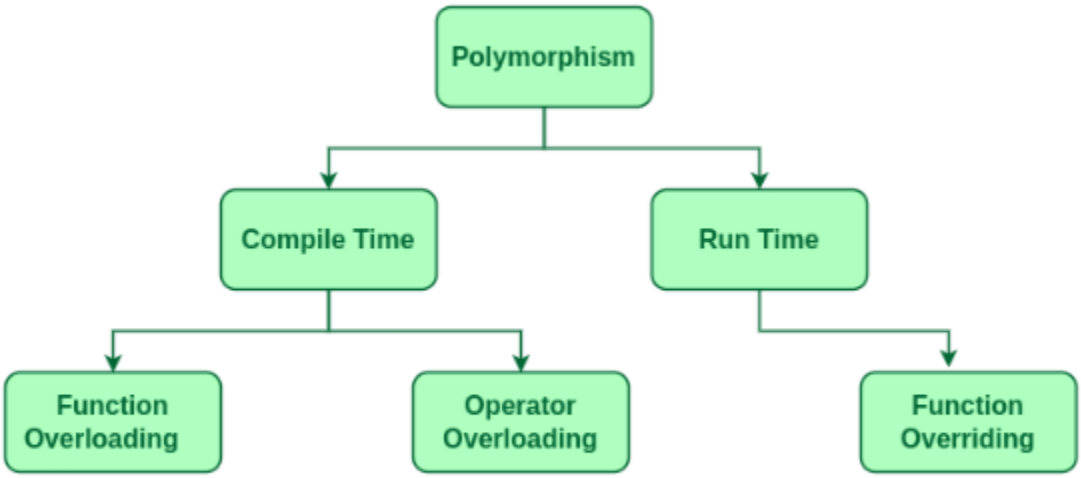
Multiple Inheritance

Polymorphism

Polymorphism is the ability of any data to be processed in more than one form. The word itself indicates the meaning as **poly means many** and **morphism means types**.

Polymorphism allows us to perform a single action in different ways.

Real-life Illustration of Polymorphism in Java: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee.



Polymorphism

Compile Time Polymorphism:

- It is also known as **static polymorphism**, and it takes during compile time of a program.
- This type of polymorphism is achieved by **function overloading** or **operator overloading**. **Java doesn't support operator overloading**.
- **Function overloading** is a technique which allows you to have more than one function with the same function name but with different functionality.
- **Function overloading** can be possible on the following basis:
    1. Same Function name with different number of parameters.
    2. Same Function name with different types of parameters.

Run Time Polymorphism:

- It is also known as **dynamic polymorphism**, and it takes place during the run time of a program.
- This type of polymorphism is achieved by **function overriding**.
- **Function overriding** is a process where the child class or sub class contain the same methods as declared in the parent or super class.
- In this process, the call to an **overridden method** is resolved **dynamically** at **runtime** rather than at compile-time.

Abstraction

- Abstraction in Java is the concept of **hiding** the complex implementation details and showing only the essential features of the object.
- In Java abstraction is achieved by interfaces and **abstract classes**. We can achieve 100% abstraction using interfaces.
- **Consider a real-life example of a man driving a car**: The man only knows that pressing the **accelerators** will increase the **speed** of a car or applying **brakes** will **stop** the car, but he does not know how on pressing the accelerator the speed is increasing, he does not know about the **inner mechanism** of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

Encapsulation vs Abstraction

Encapsulation is focused on how data is protected and accessed within an object, while **abstraction** is focused on simplifying the interaction with the object by exposing only the necessary parts.

Abstract Class & Methods

- Any class declared using **abstract keyword** is called as **abstract class** and it can consist of both **abstract methods** and **concrete methods**.

- Any class that contains one or more **abstract methods** should be defined as **abstract class** and a method defined as abstract must be **redefined** in its **subclass** making **overriding** compulsory.
- An abstract method is a method declared without any implementation.
- There can be **no object** for abstract class which means that an abstract class cannot be instantiated using new keyword.
- An abstract class can have **parameterized constructors** and the **default constructor** is always present in an abstract class.

Interfaces vs Abstract Classes

- **Abstract classes** can have both **abstract** (without implementation) and **concrete** (with implementation) methods whereas **Interfaces** can only have abstract methods (until Java 8, which introduced default and static methods).
- **Abstract classes** support **single inheritance** (a class can inherit only one abstract class) but **Interfaces** support **multiple inheritance** (a class can implement multiple interfaces).
- **Abstract classes** can have **member variables** whereas **Interfaces** cannot have member variables (except static final constants).

Lambda Expressions

- **Lambda Expressions, Functional Interfaces, Method Reference, Streams, Comparable & Comparator, Date/Time API** are added in **Java 8**.
- A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are like methods, but they do not need a name and they can be implemented right in the body of a method.
- Lambda expressions provide a way to represent **instances of functional interfaces** (interfaces with a single abstract method) in a more concise way.

- A Lambda expression consists of three parts:

Argument List	Arrow Token	Body
(int x, int y)	->	x + y

- Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.
- **Functional Interfaces** are the interfaces that contain only **one abstract method** and there is no restriction for **default** and **static methods**.
- **Method References** provide a way to refer to methods directly without invoking them. It is of the following three types:
  1. Reference to a static method
     Syntax: (ContainingClass::staticMethodName)
  2. Reference to instance method of a particular object
     Syntax: (containingObject::instanceMethodName)
  3. Reference to instance method of an arbitrary object of a particular type
     Syntax: (ContainingType::methodName)
  4. Reference to a constructor
     Syntax: (ClassName::new)
- **Stream** is an API which is used to **process group of Objects**. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- **Collections** Framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of the details of their representation. It includes interfaces, implementations, and algorithms for storing and manipulating collections of objects.
- **Collection** is the **root interface** for so many **Interfaces** like (Set, List, Queue, Dequeue) and **Classes** like (ArrayList, LinkedList, HashSet).

Java Comparable Interface

- Comparable is used to **order** the **objects** of a **user-defined class** and it is found **java.lang** package.
- It contains only one method named **compareTo** (object) and provides a **single sorting sequence** only, i.e., you can sort the elements based on single data member only.

Java Comparator Interface

- Comparator is used to order the objects of a user-defined class and it is found in **java.util** package.
- It contains two methods **compare**(Object obj1,Object obj2) and **equals**(Object element) and provides **multiple sorting sequences**, i.e., you can sort the elements on the basis of any data member.

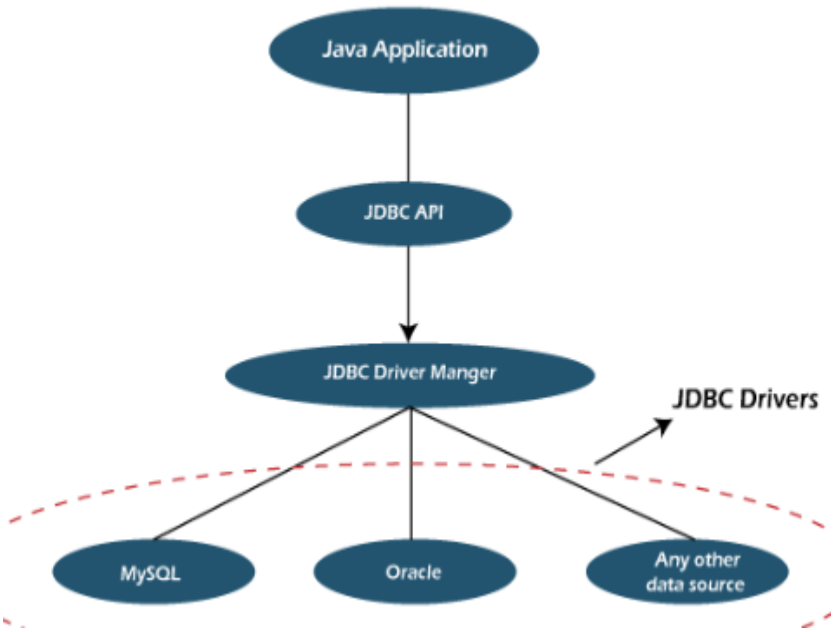
Comparable vs Comparator

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

JDBC

JDBC stands for **Java Database Connectivity**.

JDBC is a **Java API** (Application Programmable Interface) which is used to **connect** and **execute queries** on **databases**.



DriverManager: It uses some database-specific **drivers** to effectively connect enterprise applications to databases.

JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

Steps to Connect to a Database Using JDBC

1.Import JDBC Packages:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
```

2. Load and Register the JDBC Driver:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

3. Establish a Connection:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/demo", "root", "your_password");
```

4. Create a Statement Object:

```
Statement stmt = conn.createStatement();
```

5. Execute SQL Queries:

```
// Create a table
stmt.execute("CREATE TABLE IF NOT EXISTS student(sid INT, sname VARCHAR(20), scgpa FI

// Insert data into the table
stmt.execute("INSERT INTO student(sid, sname, scgpa) VALUES(1, 'John', 3.5)");
```

6. Process the Results:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM student");
while (rs.next()) {
    int id = rs.getInt(1);
    String name = rs.getString(2);
    float cgpa = rs.getFloat(3);
    System.out.println(id + " " + name + " " + cgpa);
}
```

7. Close the Connection:

```
conn.close();
```

Note

Key points on Changing Driver Loading URL and Connection URL for Different Databases:

Introduction:

When working with JDBC (Java Database Connectivity), the driver loading URL and connection URL need to be tailored to the specific database you are using. Additionally, you'll need to specify the database name, username, and password to establish a connection.

Driver Loading URL:

The driver loading URL is the fully qualified name of the JDBC driver class. It varies depending on the database vendor.

```
Example for commonly used databases:
- MySQL: `com.mysql.cj.jdbc.Driver`
- PostgreSQL: `org.postgresql.Driver`
- Oracle: `oracle.jdbc.driver.OracleDriver`
- SQL Server: `com.microsoft.sqlserver.jdbc.SQLServerDriver`
```

Connection URL:

The connection URL is the address used to connect to the database. It includes the protocol, the database vendor, the server address, the port, and the database name.

```
Example format for commonly used databases:
- MySQL: `jdbc:mysql://<server>:<port>/<database>`
- PostgreSQL: `jdbc:postgresql://<server>:<port>/<database>`
- Oracle: `jdbc:oracle:thin:@<server>:<port>:<SID>`
- SQL Server: `jdbc:sqlserver://<server>:<port>;databaseName=<database>`
```

Database Name, Username, and Password:

These credentials are used to authenticate and access the database and need to be modified by us.

Disclaimer

- The following notes are general summaries and overviews of the topics discussed.
- These notes are not exhaustive and do not cover all aspects of the subject matter.
- The information provided herein is intended for educational purposes only and should not be used as a substitute for professional advice, detailed study, or official course materials.
- **In the notes, implementation codes are added as URLs linked to each topic, which you can see as blue colored links. To access all the codes, visit the following link: All Implementation Codes.**

References

For more detailed information, please refer to the following resources:

Reference 1: Complete Java Tutorial

Reference 2: Complete Java Tutorial

Reference 3: Java Cheat Sheet

Reference 4: Java Most Asked Questions

Reference 5: Top Java Interview Questions

Prepared By: ***Reddy Venkat Kalyan***
