**Lecture 1: Introduction to JAVA & JVM**

\*\*When a java program is successfully compiled,a class file is created which has binary data.This class file can run on any platform,that's why java is platform independent.\*\*

**JVM: JVM has three components:**

**1-Class Loader Subsystem:**

a) Bootstrap class loader, b) Extension class loader, c) Application class loader

Application Class Loader: The Application class loader in the Java Virtual Machine (JVM) is responsible for loading Java classes specified on the classpath when a Java application is run. It searches for classes in directories and JAR files specified by the classpath, typically including application-specific classes and libraries. Following a delegation model, it first delegates class loading requests to its parent class loader (such as the Extension class loader and Bootstrap class loader) before attempting to load classes itself. This ensures that core Java classes are loaded from standard locations first.

Extension Class Loader: The Extension class loader in the Java Virtual Machine (JVM) is responsible for loading classes from the extensions directory (**`$JAVA\_HOME/lib/ext`** by default) or any other directory specified by the **`java.ext.dirs`** system property. The Extension class loader follows a delegation model where, before attempting to load a class itself, it delegates the class loading request to the Bootstrap class loader. This hierarchical approach ensures that core Java classes are always loaded from the bootstrap classpath, maintaining the integrity of the Java platform.

Bootstrap Class Loader: The Bootstrap class loader in the Java Virtual Machine (JVM) is the first class loader to load. It is responsible for loading crucial Java classes that are part of the core Java runtime environment. These classes include the classes from the **`java.lang`** package, fundamental runtime classes ( **`java.util`, `java.io`,** etc.), and other runtime components that are essential for the JVM to function properly. The Bootstrap class loader is the parent of all other class loaders in the JVM, including the Extension class loader and the Application class loader. It operates in a hierarchical manner where it attempts to load classes first before delegating the class loading request to its child class loaders.

\*\*Once the class is loaded using Class Loader Subsystem,that class file needs to be stored in memory area\*\*

**2-Memory Area:**

a)Class/Method Area , b)Heap Area , c)Stack Area ,d)PC Register, e)Native Method Area

Class/Method Area: In the Java Virtual Machine (JVM), the **Method Area** (also known as the **Class Area**) is a part of memory where metadata about loaded classes is stored. This includes information about classes, interfaces, methods, and constants.

Heap Area: The **Heap Area** in the Java Virtual Machine (JVM) is a runtime data area where objects and arrays created by a Java program are allocated. The heap is essential for the dynamic allocation of memory for Java objects at runtime. The size of the heap can be specified when starting the JVM or managed dynamically by the JVM itself based on application requirements and available system resources.

Stack Area: The **Stack Area** in the Java Virtual Machine (JVM) is where each thread in a Java application allocates memory for method invocations and local variables. The stack operates in a last-in, first-out (LIFO) manner and stores primitive local variables and references to objects in the heap. Each method invocation in Java results in a corresponding frame being pushed onto the stack, which contains parameters, local variables, return values, and other information necessary for the method's execution.

Heap Area: The **Program Counter (PC) Register** in the Java Virtual Machine (JVM) is a special-purpose register that stores the memory address of the JVM instruction currently being executed. It plays a crucial role in the JVM's execution model by keeping track of the next instruction to be executed for each thread.

Native Method Area: Native Method Area (NMA) is a part of the Java Virtual Machine (JVM) where native methods are stored and executed. Native methods are methods written in languages other than Java, like C or C++, and are used when Java code needs to interact with system libraries or perform operations that are not directly supported by Java itself.

**3-Execution Engine:**

Execution Engine is a crucial component of the Java Virtual Machine (JVM) responsible for executing Java bytecode. When you compile Java source code, it gets translated into bytecode, which is a platform-independent intermediate representation of the program. The Execution Engine takes this bytecode and translates it into instructions that can be understood by the host system's hardware.

Interpreter: When you compile Java source code, it gets translated into bytecode.The Interpreter reads this bytecode line by line and executes corresponding native instructions.

JIT Compiler: JIT (Just-In-Time) compiler is a key component of the Java Virtual Machine (JVM) that enhances the performance of Java applications. The bytecode is then executed by the JVM. Here’s where the JIT compiler comes in: instead of interpreting bytecode line by line like the Interpreter, the JIT compiler analyzes and compiles parts of the bytecode that are frequently executed into native machine code, which is specific to the underlying hardware. This compiled native code is then executed directly by the CPU, which typically results in significant performance improvements over pure interpretation.

**Lecture 2: Access Modifiers**

**Types of Access Modifiers:**

In Java, access modifiers are keywords used to define the accessibility of classes, variables, methods, and constructors.

**Public:** Classes, methods, and variables declared as public can be accessed from everywhere be it any other class or package.

**Protected:** Allows access within the same package and by subclasses even if they are in different packages. This means that protected members are accessible by the class itself, its subclasses, and classes in the same package.

**Default:** If no access modifier is specified, the default access level is applied. Classes, methods, and variables with default access are accessible only within the same package.

**Private:** Private members are accessible only within the same class. They cannot be accessed from outside the class, not even from subclasses.

**Variables:** Variables are named storage locations used to hold data during program execution. They are classified based on the type of data they can store, such as integers int, floating-point numbers double, characters char, and more complex types like objects String, custom classes. Variables must be declared with a specific data type before they can be used, and they can be initialized with an initial value.

**Identifiers:** Identifiers in Java are names given to variables, methods, classes, packages, and other elements in the code.

**Data Members:** These are variables declared within a class that hold data associated with objects created from that class. They represent the state or properties of objects and define what kind of data an object can store.

**Member Functions:** These are functions defined within a class that perform actions or provide functionality related to the objects of the class. Member functions can manipulate the data members of the class and perform operations that are relevant to the objects' behavior.

**Lecture 3:Keywords**

**Keywords:** Keywords are reserved words that have predefined meanings and cannot be used as identifiers (such as variable names or method names) in your code.

**Different types of Data types:**

**1.Integer type:** Used for whole numbers without decimal points, such as int (for typical integers) and long (for larger integers).

**2.Floating point type:** Used for numbers with decimal points, such as float (for single-precision floating-point numbers) and double (for double-precision floating-point numbers, commonly used).

**3.Boolean Type:** Represents true or false values with the boolean type.

**4.Character type:** Represents single characters, such as letters or symbols, with the char type.

**5.Class types:** Defined by classes in Java, which can be instantiated to create objects. Example: String, ArrayList, Scanner.

**6.Array types:** Used to hold a fixed number of values of a single type sequentially. Example: int[], String[].

**Control flow statements:** Control flow statements in Java are commands that dictate the order in which instructions are executed in a program.

**Types of Control flow statements:**

**1.Conditional Statements:** These statements alter the flow of execution based on whether a specified condition evaluates to true or false.

* **if:** Executes a block of code if a specified condition is true.
* **else if:** Allows you to specify a new condition if the previous if condition is false.
* **else:** Specifies a block of code to be executed if the condition in the if statement is false.

**2.Looping Statements:** These statements execute a block of code repeatedly until a specified condition is met.

 **for:** Executes a sequence of statements a fixed number of times.

 **while:** Repeats a statement or block of statements while a given condition is true.

 **do-while:** Similar to while, but ensures that the block of code is executed at least once, regardless of whether the condition is true or false.

**3.Branching Statements:** These statements alter the normal flow of control within loops or switch statements.

  **break:** Terminates the loop or switch statement it is in and transfers control to the statement immediately following the loop or switch.

 **continue:** Skips the current iteration of a loop and proceeds with the next iteration.

 **return:** Exits from the current method and optionally returns a value.

**4.Switch Statement:** Provides a means of handling multiple potential execution paths based on the value of an expression. It simplifies the readability of multiple if-else conditions when the value of a variable or expression matches specific cases.

**OOP’s Keywords:**

**class**: Used to declare a new class in Java. A class is a blueprint or template for creating objects that define its properties (fields) and behaviors (methods).

**new**: Used to create new objects from a class. It allocates memory for the object and initializes its fields.

**interface**: Defines a contract for classes to implement. It contains abstract methods (methods without a body) that classes implementing the interface must define. Interfaces support multiple inheritance in Java.

**extends**: Indicates that a class is inheriting the properties and behaviors of another class. It establishes an "is-a" relationship between classes, where the subclass (child class) inherits from the superclass (parent class).

**implements**: Used by a class to implement an interface. It means the class agrees to provide concrete implementations for all the abstract methods defined in the interface it implements.

**this**: Refers to the current instance of a class. It is used to access instance variables, invoke methods, and to explicitly pass the current object as a parameter to other methods or constructors within the same class.

**super**: Refers to the superclass of the current object or invokes the superclass constructor. It is used to access superclass methods and constructors, especially in cases where a subclass overrides a superclass method but still wants to access the superclass version.

**abstract**: Used to declare a class that cannot be instantiated on its own and may contain abstract methods that subclasses must implement. Abstract classes provide a way to define common behavior for related classes while allowing subclasses to provide specific implementations.

**Modifiers:**

 **static**: Used to declare members (variables and methods) that belong to the class rather than instances of the class. Static members are shared among all instances of the class and can be accessed directly using the class name.

 **final**: When applied to variables, it indicates that the variable's value cannot be changed once initialized. When applied to methods, it prevents subclasses from overriding the method. When applied to classes, it prevents inheritance (the class cannot be extended).

 **synchronized**: Used to control access to methods or blocks of code by multiple threads. It ensures that only one thread can execute a synchronized method or block at a time, preventing concurrent access to shared resources and avoiding data inconsistency.

 **transient**: Used with variables to indicate that they should not be serialized when the object containing them is serialized. This is typically used for variables that hold temporary or non-persistent data.

 **strictfp**: Ensures consistent floating-point arithmetic across different platforms by restricting floating-point calculations to strict IEEE 754 standards. Methods and classes declared with strictfp will produce identical results on all platforms for floating-point operations.

 **void**: Used as a return type for methods to indicate that the method does not return any value. Void methods are typically used for performing actions or operations without returning a result.

 **return**: Used inside methods to explicitly return a value to the caller. It exits the method and optionally passes back a value of the specified return type if the method has a non-void return type.

 **native**: Indicates that a method is implemented in platform-dependent code (such as C or C++) and not in Java. Native methods are used when interfacing with system-level functionality or when performance is critical and Java's capabilities are insufficient.

 **assert**: Used for debugging purposes to test assumptions about the program's state. If an assert statement evaluates to false, an AssertionError is thrown at runtime, helping to identify logical errors during development.

**volatile**: Used with variables that are shared among multiple threads to ensure visibility and ordering of updates. It prevents threads from caching the variable's value locally, ensuring that changes made by one thread are immediately visible to other threads.

**Exception Handling:**

**try**: The try block is used to enclose code that might throw an exception. It's followed by one or more catch blocks and an optional finally block. If an exception occurs within the try block, control is passed to the appropriate catch block.

**catch**: catch blocks follow a try block and are used to handle specific types of exceptions that might be thrown within the try block. Each catch block specifies the type of exception it can handle and provides code to handle the exception, allowing the program to recover gracefully from errors.

**finally**: The finally block, if present, is used to execute code that needs to be run regardless of whether an exception was thrown or not. It ensures that cleanup or finalization tasks (such as closing files or releasing resources) are performed, making it useful for maintaining program stability.

**throw**: The throw keyword is used to explicitly throw an exception. It can be used within methods to manually generate an exception based on specific conditions or criteria defined by the programmer.

**throws**: The throws keyword is used in method declarations to indicate that the method may throw one or more types of exceptions. It specifies the exceptions that the method might throw but does not handle them within the method itself. Instead, it signals to the caller of the method that they need to handle potential exceptions.

**Identifiers:** Identifiers in Java are names given to entities such as variables, methods, classes, packages, etc.

**Rules of Identifiers:**

1)They must begin with a letter (A to Z or a to z), currency character ($), or an underscore (\_).  
2)Subsequent characters may be letters, currency characters, underscores, or digits (0 to 9).  
3)Java identifiers are case-sensitive.  
4)They cannot be Java keywords or reserved words.

**Lecture 4: OOP’s Concepts**

**Class & Object:** A class serves as a blueprint or template that defines the attributes (data fields) and behaviors (methods) that objects of that class will have whereas object is an instance of a class, created using the new keyword.

**Encapsulation:** Encapsulation in Java refers to bundling the data (attributes) and methods (behaviors) that operate on the data into a single unit, called a class. It protects the internal state of an object from outside interference and manipulation, promoting data integrity and security.

**Abstraction:** Hiding complex implementation details and exposing only the essential features of an object. This is achieved through abstract classes and interfaces, allowing programmers to focus on what an object does rather than how it does it, facilitating easier maintenance and reusability of code.

**Constructor:** A constructor in Java is a special method that is automatically invoked when an object of a class is created. Its purpose is to initialize the object's state by assigning initial values to the object's instance variables. Constructors have the same name as the class and do not have a return type, not even void. They can be overloaded to provide different ways of initializing objects with varying sets of parameters, ensuring flexibility in object creation and initialization.

**Abstract class & methods**: An abstract class in Java is a class that cannot be instantiated on its own and is typically used as a superclass for other classes. It may contain abstract methods, which are declared but not implemented in the abstract class itself. Subclasses extending an abstract class must provide concrete implementations for all inherited abstract methods, ensuring consistency and enforcing specific behavior across related classes. Abstract classes can also have non-abstract methods and fields, making them useful for defining common behavior and characteristics shared among multiple subclasses.

**Inheritance:** Inheritance is a fundamental concept in object-oriented programming (OOP) where a class (known as a subclass or derived class) can inherit attributes and behaviors from another class (known as a superclass or base class). This relationship allows the subclass to reuse the code defined in the superclass, promoting code reusability and reducing redundancy. In Java, inheritance is achieved using the extends keyword, where a subclass inherits members (fields and methods) from its superclass.

**Why Multiple Inheritance not Supported in Java**: Java does not support multiple inheritance because it can lead to ambiguity and complexity in the inheritance hierarchy. Multiple inheritance occurs when a class inherits from more than one superclass, potentially inheriting conflicting or ambiguous behaviors from each superclass.When super classes have same method then child class calls which method is where ambiguous situation comes.

**Polymorphism:** Polymorphism in Java refers to the ability of objects to take on multiple forms or to exhibit different behaviors based on their data types or classes. It allows methods to be invoked on objects of different classes in a unified way, achieved through method overriding (in subclasses) and method overloading (within the same class).

**Method Overloading:** Method overloading in Java occurs when a class has multiple methods with the same name but different parameter lists (number, type, or order of parameters). It allows different methods to perform similar tasks with variations in how they are invoked or the types of data they operate on. Overloaded methods must differ in their parameter lists or return types, enabling developers to create cleaner and more readable code by using the same method name for logically related functionalities with different input scenarios.

**Lecture 5: Exception Handling**

**Errors:** Error indicate serious problems that typically cannot be handled programmatically because they stem from issues outside the control of the application itself. These include critical events like OutOfMemoryError or StackOverflowError, which often result from system-level constraints or environmental issues.

**Exception:** Exceptions are events or conditions that disrupt the normal flow of a program but are generally recoverable. They can be caused by user actions, programming errors, or unexpected conditions like NullPointerException or FileNotFoundException.

**Checked Exception:** **Checked Exceptions** are those that the Java compiler mandates to be handled explicitly either by catching them with try-catch blocks or by declaring them in the method's throws clause. These exceptions typically represent conditions that a program can anticipate and possibly recover from, such as file not found (FileNotFoundException) or database access issues (SQLException).

**Unchecked Exception**: Unchecked Exception do not require explicit handling or declaration by the programmer. They arise primarily from programming errors and are not enforced by the compiler. Unchecked exceptions extend from RuntimeException and its subclasses, such as NullPointerException or ArrayIndexOutOfBoundsException. These exceptions typically indicate bugs in the code, such as attempting to access an array element beyond its bounds.

**IOException:** IOException is a checked exception in Java that occurs during input/output operations, such as reading from or writing to files. This exception covers a wide range of possible errors, including situations where the file might not exist (FileNotFoundException), permission issues, hardware problems, or network connectivity issues during file operations.

**SQL Exception:** SQLException is a checked exception specific to Java Database Connectivity (JDBC) operations, indicating issues with database access or SQL queries. It arises when there are errors in SQL syntax, database connection failures, or problems related to the database schema. Handling SQLException is crucial in JDBC programming to manage transaction rollbacks, retry database operations, or inform users about issues with data access.

**ClassNotFoundException:** ClassNotFoundException is a checked exception that occurs when the JVM attempts to load a class dynamically using its name but fails to find the corresponding class file in the classpath. This exception typically arises due to missing dependencies or incorrect class names specified in the code.

**FileNotFoundException:** FileNotFoundException is a subclass of IOException that specifically indicates the failure to find a specified file at the given path during file input/output operations. This checked exception is thrown when attempting to access a file that does not exist, either due to incorrect file paths, permissions, or file deletion before access.

**Runtime Exceptions:**

**NullPointerException:** **NullPointerException** occurs when an application tries to use an object reference that has not been initialized or is set to null. This typically happens when invoking methods or accessing fields on null objects, leading to a runtime error that needs careful validation and defensive programming to avoid.

**ArithmeticException: ArithmeticException** arises when an arithmetic operation encounters an exceptional condition, such as division by zero. This exception highlights errors in mathematical calculations and requires preemptive checks to ensure divisor values are non-zero, preventing unexpected program crashes.

**IndexOutOfBoundException**: **IndexOutOfBoundsException** occurs when trying to access an element from an invalid index in arrays, lists, or other data structures. This exception signals attempts to retrieve or modify elements beyond the valid range, necessitating boundary checks to ensure data integrity and avoid runtime failures.

**IllegalArgumentException**: It indicates that a method has been passed an illegal or inappropriate argument. This exception commonly occurs when method parameters fail validation checks, such as negative values for operations expecting positive inputs, requiring robust input validation to enforce correct usage and prevent erroneous program states.

**Different types of Errors:**

**OutOfMemoryError:** OutOfMemoryError is a type of error in Java that occurs when the Java Virtual Machine (JVM) cannot allocate enough memory to support the demands of an application. This error typically arises when an application or its components attempt to allocate memory beyond the JVM's available resources, leading to exhaustion of heap space. Common causes include memory leaks, excessive data processing, or inadequate JVM heap size configuration. Handling OutOfMemoryError involves diagnosing memory usage patterns, optimizing data structures and algorithms to reduce memory consumption, and configuring JVM parameters such as heap size (-Xmx) appropriately to ensure sufficient memory allocation for the application's requirements.

**StackOverflowError:** StackOverflowError is another type of error in Java that occurs when a program exhausts its call stack due to excessive recursion. Each time a method is called in Java, a new frame is pushed onto the call stack to store method parameters, local variables, and return addresses. If recursion does not have a proper termination condition or recurses too deeply without returning, it consumes all available stack space, resulting in a StackOverflowError.

**Assertion Error:** AssertionError is an error in Java that occurs when an assertion statement fails. Assertions are typically used to verify assumptions made by the programmer about the program's state during development or testing phases. If an assertion condition evaluates to false at runtime, Java throws an AssertionError.

**Lecture 5: Exception Handling**

**Synchronization:** Synchronization in Java is used to control access to shared resources or critical sections of code to ensure that only one thread can access them at a time. This prevents data corruption and ensures thread safety.

**Block Level Lock:** Block-level synchronization is achieved using the synchronized keyword with a block of code. This allows synchronized access to a specific block of code rather than an entire method, reducing the scope of lock contention and improving performance in scenarios where not all code within a method requires synchronization.

**Method Level Lock:** Method-level synchronization involves declaring a method with the synchronized keyword. This ensures that only one thread can execute the synchronized method on a particular instance of the class at any given time. It simplifies synchronization but may lead to decreased performance if multiple threads contend for the same lock.

I**nterface:** In Java, an interface defines a contract for classes to implement. It declares methods that implementing classes must define. Interfaces facilitate code reuse, multiple inheritance, and polymorphism.

**Runnable Interface:**Runnable is a functional interface in Java that represents a task that can be executed concurrently by a thread. It contains a single abstract method run() that must be overridden to define the task's behavior. Implementing Runnable separates the thread's behavior from its lifecycle, promoting better code organization and reusability compared to extending the Thread class.

**Which is better to implement multithreading, Thread class or Runnable interface, and why?**

Implementing the Runnable interface is generally preferred over extending the Thread class for multithreading in Java. This is because:

**More Flexibility:** Implementing Runnable allows the same runnable object to be executed by multiple threads or submitted to thread pools, promoting better resource management and scalability.

**Avoids Inheritance Limitations:** Java does not support multiple inheritance, so extending the Thread class limits the flexibility of the class hierarchy.

**Deadlock during Multithreading:**

**Deadlock** occurs when two or more threads are blocked forever, each waiting on a resource held by the other threads. It typically happens due to improper synchronization and resource allocation strategies.

To Avoid Deadlock:

**Resource Ordering:** Ensure threads acquire locks on resources in the same order to prevent circular dependencies.

**Timeouts and Detection:** Use timeouts or deadlock detection mechanisms to identify and recover from deadlock situations.

**Avoid Nested Locks:** Minimize the use of nested locks and ensure that locks are acquired and released in a consistent manner.

**Executor Framework:**

The **Executor Framework** in Java provides a higher-level abstraction for managing and executing threads asynchronously. It includes several types of thread pools:

**FixedThreadPool:** Executes tasks using a fixed number of threads. Useful when a limited number of concurrent threads are required.

**SingleThreadPool:** Uses a single thread to execute tasks sequentially. Ensures tasks are executed in the order they are submitted.

**CachedThreadPool:** Dynamically adjusts the number of threads based on the workload. Suitable for handling a large number of short-lived tasks.

**ScheduledThreadPool:** Executes tasks at a specified time or with a fixed delay between executions. Useful for periodic tasks or tasks requiring scheduling.

**Serialization:** Serialization in Java refers to the process of converting an object into a stream of bytes, which can be easily stored in a file or transmitted over a network. The main purpose of serialization is to persist the state of an object so that it can be recreated later if needed, or to transmit it across a network to another location.

**Need of Serialization:**

**Persistence:** Serialized objects can be saved to a file system and later deserialized to recreate the original object. This is useful for storing application state or for data persistence.

**Networking:** Objects can be serialized and sent over a network to another system, where they can be deserialized and used. This is commonly used in client-server applications.

**Caching:** Serialized objects can be stored in memory or on disk for caching purposes, allowing for quicker access and reduced computation time.

**How it Works?**

**Serializable Interface:** In Java, to make an object serializable, the class must implement the java.io.Serializable interface. This interface acts as a marker or tag to indicate that the objects of this class can be serialized. It does not have any methods and serves as a signal to the Java runtime environment.

 When an object of a serializable class is to be serialized, Java converts the object into a stream of bytes. This process includes all data members (fields) of the object and their types.

 Any object references within the object being serialized are also serialized recursively.

 Non-serializable fields (marked with transient keyword) are not included in the serialization process unless custom serialization methods are implemented.