PRATIK KUMAR

- **Assignment Date: June 7, 2024**

Report: JVM Architecture and Its Flow

JVM Architecture and Its Flow

Java Virtual Machine (JVM) serves as the cornerstone of the Java platform, facilitating the execution of Java bytecode across different hardware and operating systems. Understanding its architecture and flow is fundamental for Java developers to optimize application performance and troubleshoot issues effectively.

1. Overview of JVM Architecture

The JVM architecture consists of several key components, each playing a crucial role in the execution of Java programs:

- **Class Loader Subsystem:** Responsible for loading class files into memory dynamically during runtime. It comprises three distinct class loaders: Bootstrap, Extension, and Application class loaders.
- **Runtime Data Area:** Memory allocation within the JVM for storing various data structures during program execution. Key components include:
  - **Method Area:** Stores class-level data like class metadata, constant pool, and static fields.
  - **Heap:** Memory area shared among all threads for object allocation. It supports garbage collection for automatic memory management.
  - **Stack:** Each thread has its own stack, which stores method-specific data (local variables, partial results, etc.). It also manages method invocation and returns.
  - **PC Registers:** Each thread has its own Program Counter (PC) register, which holds the address of the currently executing bytecode instruction.
- **Execution Engine:** Interprets bytecode instructions and executes them. It consists of two components:
  - **Interpreter:** Reads bytecode and executes the corresponding native code instructions.
  - **Just-In-Time (JIT) Compiler:** Optimizes frequently executed bytecode segments into native machine code to improve performance.

2. Flow of Execution in JVM

Understanding the sequence of operations within the JVM is crucial for developers to comprehend how Java programs are executed:

- **Loading:** Class Loader subsystem loads class files into memory. It involves locating and reading bytecode files from the file system or network.
- **Verification:** Bytecode verifier checks bytecode for illegal operations that can violate JVM security.
- **Preparation:** Memory is allocated for class variables and static fields. Default values are assigned to them.
- **Resolution:** Symbolic references in the bytecode are replaced with direct references to memory addresses.
- **Initialization:** Static variables and initializers are initialized to their designated values.
- **Execution:** JVM executes bytecode instructions line by line, leveraging the execution engine for interpretation or JIT compilation.

➢ Assignment Date: June 10, 2024

1. Java Keywords with Brief Definitions:

Java keywords are reserved words that have predefined meanings and cannot be used for naming variables, classes, or methods.

- **abstract**: Used to declare a class or method that does not have full implementation.
- **assert**: Used for debugging purposes to make assertions about the program's state.
- **boolean**: Defines a variable that can hold true or false values.
- **break**: Exits from the loop or switch statement.
- **byte**: Defines an 8-bit integer variable.
- **case**: Specifies a branch in switch statements.
- **catch**: Catches exceptions generated by try statements.
- **char**: Defines a 16-bit Unicode character.
- **class**: Declares a class.
- **const**: Not used (reserved for future use).
- **continue**: Skips the current iteration of a loop and proceeds with the next iteration.
- **default**: Specifies the default case in a switch statement.
- **do**: Starts a do-while loop.
- **double**: Defines a double-precision floating-point variable.
- **else**: Specifies the block of code that executes if the condition in the if statement is false.
- **enum**: Declares an enumerated (unchangeable) data type.
- **extends**: Indicates that a class is inheriting the properties of another class.
- **final**: Prevents the inheritance of a class, overriding of a method, or modification of a variable.
- **finally**: Executes a block of code regardless of whether an exception is thrown or not.
- **float**: Defines a single-precision floating-point variable.
- **for**: Initiates a for loop.
- **goto**: Not used (reserved for future use).
- **if**: Executes a block of code if a specified condition is true.
- **implements**: Implements an interface.
- **import**: Imports other Java packages or classes.
- **instanceof**: Tests whether an object is an instance of a specific class or interface.
- **int**: Defines an integer variable.
- **interface**: Declares an interface.
- **long**: Defines a long integer variable.
- **native**: Specifies that a method is implemented in native code using JNI (Java Native Interface).
- **new**: Creates new objects.
- **null**: Refers to an un-initialized or unknown object.
- **package**: Declares a package.
- **private**: Restricts access to variables, methods, or constructors within the same class.
- **protected**: Provides access to variables, methods, or constructors within the same package or in subclasses.
- **public**: Provides unrestricted access to variables, methods, or constructors.
- **return**: Exits from a method and optionally returns a value.
- **short**: Defines a short integer variable.
- **static**: Allows methods, variables, and blocks to be accessed without creating an instance of the class.
- **strictfp**: Restricts floating-point calculations to ensure portability.
- **super**: Refers to the superclass (parent class) of the current object.
- **switch**: Evaluates a variable and executes the corresponding block of code based on the value.
- **synchronized**: Ensures that a method or block of code can be accessed by only one thread at a time.

- **this**: Refers to the current instance of a class.
- **throw**: Throws an exception.
- **throws**: Specifies the exceptions that a method can throw.
- **transient**: Prevents serialization of a variable.
- **try**: Starts a block of code that will be tested for exceptions.
- **void**: Specifies that a method does not return any value.
- **volatile**: Indicates that a variable may be changed unexpectedly.
- **while**: Initiates a while loop.

2. Java Identifiers with Brief Definitions:

Identifiers are names given to variables, methods, classes, and labels in Java.

- **Variable Identifier**: A name given to a variable that holds data temporarily during program execution.
- **Method Identifier**: A name given to a block of code that performs a specific task.
- **Class Identifier**: A name given to a blueprint for creating objects, defining methods, and variables.
- **Package Identifier**: A name given to a namespace that organizes a set of related classes and interfaces.
- **Label Identifier**: A name used to mark a position in code for use with break or continue statements.

3. Various Java Access Modifiers with Brief Definitions:

Access modifiers control the visibility and accessibility of classes, methods, and variables in Java.

- **private**:
  - **Definition**: The most restrictive access level. Variables, methods, and constructors that are declared as private can only be accessed within the same class.
  - **Usage**: Use private access modifier to hide the internal implementation details of a class and ensure encapsulation. It prevents direct access from outside classes, thereby enhancing security and maintaining code integrity.

- **default (no modifier)**:
  - **Definition**: If no access modifier is specified, the default access level is applied. Variables, methods, and constructors with default access can only be accessed by classes in the same package.
  - **Usage**: It is useful when you want members to be accessible within a specific package but not outside it. This promotes package-level encapsulation and helps organize related classes and functionalities.

- **protected**:
  - **Definition**: Variables, methods, and constructors declared as protected can be accessed by classes in the same package and by subclasses even if they are in different packages.
  - **Usage**: Use protected access modifier when you want to allow subclasses to access certain members for inheritance purposes while still restricting access to unrelated classes outside the package. It facilitates controlled sharing of members between related classes and promotes code reusability.

- **public**:
  - **Definition**: The least restrictive access level. Variables, methods, and constructors declared as public can be accessed from any other class.

- **Usage**: Use public access modifier when you want a member to be widely accessible across different classes and packages. It is commonly used for methods that form part of a class's interface or API, allowing them to be freely invoked and utilized by other parts of the program.

➢ Assignment Date: June 11, 2024

1. Java Keywords:

Java keywords are reserved words that have predefined meanings and cannot be used for naming variables, classes, or methods.

- Examples of Java Keywords:
  - **abstract**: Used to declare a class that cannot be instantiated or a method that must be implemented by subclasses.
  - **boolean**: Defines a variable that can hold true or false values.
  - **break**: Exits from the loop or switch statement.
  - **class**: Declares a class.
  - **final**: Prevents inheritance, method overriding, or modification of variables.
  - **if**: Executes a block of code based on a specified condition.
  - **new**: Creates new objects.
  - **public**: Provides unrestricted access to variables, methods, or classes.
  - **return**: Exits from a method and optionally returns a value.
  - **static**: Allows methods, variables, and blocks to be accessed without creating an instance of the class.
  - **void**: Specifies that a method does not return any value.
  - **while**: Initiates a while loop, executing a block of code repeatedly as long as a condition is true.

2. Data Types ASCII Words:

In Java, ASCII (American Standard Code for Information Interchange) characters are integral to programming, especially in the context of data types and control statements.

- **Data Types ASCII Words:**
  - **char**: Represents a single 16-bit Unicode character.
  - **byte**: Represents an 8-bit signed integer.
  - **short**: Represents a 16-bit signed integer.
  - **int**: Represents a 32-bit signed integer.
  - **long**: Represents a 64-bit signed integer.
  - **float**: Represents a single-precision 32-bit floating point.
  - **double**: Represents a double-precision 64-bit floating point.

3. Control Statements as Keywords:

Control statements are used to manage the flow of execution within a program. While not exclusively keywords, they are crucial in programming and often work in conjunction with keywords like "if", "else", "for", "while", and "switch".

4. Modifier as Keywords:

Modifiers in Java alter the state or behavior of variables, methods, and classes. Examples include access modifiers (public, private, protected) and non-access modifiers (static, final, abstract).

5. Access Modifiers:

Access modifiers control the visibility and accessibility of classes, methods, and variables within Java programs. They include:

- **private**: Restricts access to the same class.
- **protected**: Allows access within the same package and by subclasses.
- **public**: Permits access from anywhere.
- **default (no modifier)**: Allows access within the same package.

6. OOPs Keywords:

In Object-Oriented Programming (OOP), certain keywords are fundamental to defining classes, objects, inheritance, polymorphism, and encapsulation. Examples include "class", "interface", "extends", "implements", "super", "this", and "new".

7. Identifiers and Rules:

Identifiers in Java are names given to classes, variables, methods, and labels. They must adhere to specific rules:

- Must begin with a letter (A to Z or a to z), currency character ($) or underscore (_).
- Subsequent characters may be letters, digits, currency characters, or underscores.
- Cannot be a Java keyword.
- Case-sensitive (MyVariable and myVariable are different).

➢ Assignment Date: June 12, 2024

1. OOPs Concepts with Object:

Object-Oriented Programming (OOP) is centered around the concept of objects, which are instances of classes. Each object encapsulates data and behavior. Key OOPs concepts include:

- **Classes**: Blueprints for creating objects. They define attributes (fields) and behaviors (methods) shared by objects of the same type.
- **Polymorphism**: The ability of objects to respond to the same message (method call) in different ways. It includes method overloading and method overriding.
- **Abstraction**: Hides complex implementation details and shows only essential features of an object. Abstract classes and interfaces facilitate abstraction.
- **Encapsulation**: Bundles data (attributes) and methods (behaviors) that operate on the data into a single unit (class). Access to the data is controlled through access modifiers.
- **Inheritance**: Enables a new class (subclass or derived class) to inherit attributes and methods from an existing class (superclass or base class).

2. Classes:

- **Definition**: Classes are templates or blueprints for creating objects. They define the attributes (fields) and behaviors (methods) that objects of the class will have.
- **Purpose**: Encapsulate data for the object and define methods to operate on that data. Classes provide a structure for organizing code and promote reusability.

3. Polymorphism:

- **Definition**: Polymorphism refers to the ability of different objects to respond to the same message (method call) in different ways. It can be achieved through method overloading and method overriding.
- **Types**:
    - **Method Overloading**: Having multiple methods with the same name but different parameters within the same class. The compiler determines which method to execute based on the method's signature.
    - **Method Overriding**: Inheritance-based polymorphism where a subclass provides a specific implementation of a method that is already provided by its superclass.

4. Abstraction:

- **Definition**: Abstraction focuses on hiding the complex implementation details of a class or method and exposing only the essential features or functionalities to the user.
- **Uses**: Abstract classes and interfaces are key tools for achieving abstraction in Java. They provide a blueprint that subclasses can follow while allowing methods to be defined without implementations (in interfaces) or partially implemented (in abstract classes).

5. Encapsulation:

- **Definition**: Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on the data into a single unit (class). It restricts direct access to some of an object's components.
- **Purpose**: Protects the internal state of an object from unwanted changes and ensures that the object's state can only be modified through well-defined methods.

6. Inheritance:

- **Definition**: Inheritance allows one class (subclass or derived class) to inherit attributes and methods from another class (superclass or base class). It promotes code reuse and supports the concept of hierarchical classification.
- **Types**: Java supports single inheritance (where a subclass extends only one superclass) and multiple levels of inheritance (where classes can be derived from other classes).

7. Key Points of Abstract Classes:

- **Definition**: Abstract classes cannot be instantiated directly and can contain abstract methods (methods without a body) that must be implemented by subclasses.
- **Uses**: They serve as blueprints for other classes and provide a way to define common behaviors for subclasses while enforcing rules for implementation.

8. Method Overloading and Method Overriding:

- **Method Overloading**: Involves creating multiple methods in the same class with the same name but different parameters. The compiler determines which method to execute based on the method's signature (number and types of parameters).

- **Method Overriding**: Occurs in a subclass that provides a specific implementation of a method that is already provided by its superclass. It allows a subclass to provide a specialized implementation of a method that is already provided by its superclass.

➢ Assignment Date: June 13, 2024

Hierarchy of Exception classes

Exception classes in Java extend from `Throwable`. They are categorized into Checked Exceptions (`Exception`) and Unchecked Exceptions (`RuntimeException`). Both are subclasses of `Throwable`.

Types of Exception

Exceptions in Java include Checked Exceptions (e.g., `IOException`), Unchecked Exceptions (e.g., `NullPointerException`), and Errors (e.g., `OutOfMemoryError`).

try-catch-finally

- **try:** Encloses code that may throw exceptions.
- **catch:** Handles exceptions thrown within the `try` block.
- **finally:** Executes code regardless of whether an exception is thrown or not, used for cleanup tasks.

throw

`throw` keyword is used to explicitly throw an exception within a method or block.

throws

`throws` keyword specifies that a method can throw exceptions of a particular type.

Multiple catch block

Allows handling different types of exceptions separately in `try-catch` blocks.

Exception Handling with method Overriding

Subclasses can override methods that throw exceptions, but the overridden method must not throw broader checked exceptions than the superclass method.

Java Collection Framework

Hierarchy of Collection Framework

- **Interfaces:** `Collection` (extends `Iterable`) and `Map`.
- **Classes:** Implementations include `List`, `Set`, `Queue`, `Deque`, and `Map`.

Collection interface

Root interface of collections framework. Extends `Iterable`, providing basic operations.

Iterator interface

Enables iteration over collections via `hasNext()` and `next()` methods.

Set, List, Queue, Map

- **Set:** Unordered collection of unique elements (`HashSet`, `LinkedHashSet`, `TreeSet`).
- **List:** Ordered collection (`ArrayList`, `LinkedList`, `Vector`).
- **Queue:** Ordered collection for FIFO operations (`PriorityQueue`).
- **Map:** Key-value pairs (`HashMap`, `LinkedHashMap`, `TreeMap`, `ConcurrentHashMap`).

Comparator, Comparable interfaces

- **Comparator:** Custom sorting logic for objects.
- **Comparable:** Default sorting logic for objects.

ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, HashMap, ConcurrentHashMap classes

Various implementations of collection interfaces with specific characteristics (e.g., `ArrayList` for resizable arrays, `HashMap` for hash table-based mapping).

Multithreading

Lifecycle of a Thread

States include New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated.

Thread Priority in Multithreading

Prioritizes threads for scheduling by the JVM (MIN_PRIORITY to MAX_PRIORITY).

Runnable interface in Java

Functional interface representing a task runnable by a thread.

start() function in multithreading

Begins execution of a thread, invoking its `run()` method.

Thread.sleep() Method in Java

Pauses current thread execution for a specified time.

Thread.run() in Java

Defines thread's task to execute when started.

Deadlock in Java

Occurs when threads are blocked indefinitely waiting for each other.

Synchronization in Java

Controls access to shared resources, preventing data corruption.

Method level lock

Synchronizes access to instance methods.

Block level lock

Synchronizes specific blocks of code.

Executor Framework Java

Provides higher-level API for managing thread execution asynchronously.

Callable Interface in Java

Similar to `Runnable` but returns a result and can throw exceptions.

- **Assignment Date: June 14, 2024**

  Thread: In Java, a thread represents a separate flow of control. Threads allow concurrent execution, enabling applications to perform multiple tasks simultaneously.

Multithreading: Multithreading refers to the ability of a CPU (or a single core in a multi-core processor) to provide concurrent execution of multiple sequences or threads of instructions.

Lifecycle of a Thread: A thread in Java undergoes several states: New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated. These states dictate how a thread progresses from creation to completion.

Thread Priority in Multithreading: Threads can be assigned priorities ranging from MIN_PRIORITY (1) to MAX_PRIORITY (10), affecting their scheduling by the JVM.

Runnable Interface in Java: The `Runnable` interface is used to define a task that can be executed by a thread. It represents a thread that can be started, run, and stopped.

start() function in Multithreading: The `start()` method initiates the execution of a thread. It allocates system resources and invokes the thread's `run()` method.

Thread.sleep() Method in Java: `Thread.sleep()` pauses the execution of the current thread for a specified amount of time, allowing other threads to execute during the sleep period.

Thread.run() in Java: The `run()` method contains the code executed by the thread when `start()` is called. It defines the thread's task or job.

Deadlock in Java: Deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources they need.

Synchronization in Java: Synchronization ensures that only one thread at a time can access a shared resource, preventing data corruption and maintaining consistency.

Method Level Lock: Method level locks in Java synchronize access to instance methods of an object, ensuring only one thread can execute them at a time per instance.

Block Level Lock: Block level locks synchronize specific blocks of code within methods, allowing more fine-grained control over thread access to shared resources.

Executor Framework Java: The Executor framework provides a higher-level abstraction for managing and executing threads asynchronously, simplifying concurrent programming.

Callable Interface in Java: The `Callable` interface is similar to `Runnable` but allows threads to return a result and throw exceptions. It is used with ExecutorService to manage concurrent tasks.

Java 8 Features

Functional Interfaces: Functional interfaces have exactly one abstract method and can be used as lambda expressions. They enable functional programming paradigms in Java.

Optional: `Optional` is a container object used to represent a nullable value, helping to avoid `NullPointerException` and improving code clarity.

Default Methods: Default methods in interfaces allow adding new methods to existing interfaces without breaking existing implementations.

Stream API: The Stream API enables functional-style operations on streams of elements, facilitating parallel processing and concise code.

Java Time API: The Java Time API provides a comprehensive set of classes for date and time manipulation, offering improved consistency and functionality over the old `java.util.Date` and `java.util.Calendar`.

Lambda Expressions: Lambda expressions introduce a concise syntax for defining anonymous functions and enable functional programming features in Java.

Method Reference: Method references provide a shorthand syntax for invoking a method as a lambda expression, improving code readability and reducing boilerplate code.

Meta Space: The Meta Space is the replacement for the Permanent Generation (PermGen) in Java 8 and above. It stores metadata related to classes and methods.