01

Java

WHAT YOU WILL LEARN

* JVM Architecture
* Packages
* Default Classes
* Encapsulation
* Class And Object
* Abstraction
* Polymorphism
* Exception Handling
* Collections

JVM Architecture

What is JVM (Java Virtual Machine)?

A blue circle with black text

Description automatically generated with low confidence

* Java is developed with the concept of WORA (Write Once Run Anywhere).
* JVM is an application-based VM, which is a part of the Java Run Time Environment (JRE) and is responsible to load and run java class files.
* JVM is the one which calls the main method of a java program.
* Architectural Diagram of JVM:

Graphical user interface, application

Description automatically generated

**JVM have 3 layers of architecture**

1. Class Loader Subsystem
2. Loading

Types of class loaders:

* **Bootstrap class loader**: This is responsible to load core java API classes(such as string class, string buffer etc.) i.e the classes present in rt.jar.The path of rt.jar is jdk/jre/lib/rt.jar. This path (jdk/jre/lib/) is called Bootstrap classpath.
* **Extension class loader:** This is the child class of Bootstrap class loader.This class loader is responsible to load classes from extension class path (jdk/jre/lib/ext/ .jar) etc.
* **Application class loader/ System class loader:** This is the child class of extension class loader. This class loader is responsible to load classes from the application classpath. It internally uses the environment variable classpath.

Diagram

Description automatically generated

* In the loading phase, the class loader reads each .class file and store corresponding binary data in the **Method area**.
* Below data are stored in the method area:
  + Fully qualified name of the loaded class and its immediate parent class.
  + .class file information such as the file is related to class or interface or enum, methods information, variable information, constructor information, modifiers information, constant pool information, etc.
* After loading the .class file, immediately JVM creates an object for that loaded class on the heap memory of type java.lang.class.
* These loaded class “Class object” can be used by programmers, to get class-level information such as Method information or variable information, constructor information, etc. For every loaded type, only one class object will be created even though we are using the class multiple times in our program as shown in the below image.

Diagram

Description automatically generated

1. Linking - Linking consists of 3 activities Verify, Prepare, Resolve.
   1. It is the process of ensuring that, the binary representation of a class is structurally correct or not, i.e inside JVM, the Byte-code verifier will check whether the .class file is generated by a valid compiler or not, whether .class file is properly formatted or not.
   2. Prepare/Preparation: In this phase, JVM will allocate memory for class level static variables and assign default values.
   3. Resolve/Resolution: It is the process of replacing symbolic memory references/names in our program with the original memory references from the Method area.
2. Initializing - In this, all static variables are assigned with original values and static blocks will be executed from parent to child and from top to bottom.

Note: While loading, linking and initialization if any error occurs, then we will get a run time exception saying java.lang.linkageError.

1. Memory Area Subsystem
2. Method area: For every JVM, one method area will be available. Inside the method area, class level binary data including static variables will be stored. Constant pools of a class will be stored inside the method area. Method area can be accessed by multiple threads simultaneously. Hence, method area data will not be continuous.
3. Heap Area: For every JVM, one heap area is available. Heap area will be created at the time of JVM startup. Objects and corresponding instance variables will be stored in the heap area. The Heap area can be accessed by multiple threads hence, the data stored in the heap memory is not thread-safe. The heap area need not be continuous.
4. **Stack Area:** For every thread, JVM will create a separate stack at the time of thread creation. Each and every method call performed by that thread will be stored in the stack including local variables too. After completing a method the corresponding entry will be removed. After completing all method calls, the stack will become empty. The empty stack will be destroyed by JVM just before terminating the thread. Each entry in the stack is called a stack frame or activation record. The data stored in the stack is available only for the corresponding thread and not available to the remaining threads. Hence this data is thread safe. Each stack frame contains Local Variable Arrays, Operand Stack, Framed Data.
5. PC registers: For every thread, separate PC(Program counter) registers will be created at the time of thread creation. PC registers contain the address of the currently executing instruction, once instruction execution completes automatically PC register will be incremented to hold the address of the next instruction.
6. Native method stacks: For every thread, JVM will create a separate native method stack. All native method calls invoked by the thread will be stored in the corresponding Native method stack.
7. Execution Engine Subsystem - This is the central component of JVM and is responsible to execute java class files. It mainly contains three components:
8. Interpreter: This is responsible for reading byte code and interpret it into sample machine code (native code) and executes the machine code line by line. The drawback of the interpreter is, it interprets every time, even if the same method is invoked multiple times, which reduces the performance of the system.
9. JIT (Just In Time) Compiler: The main purpose of the JIT compiler is to improve performance. Internally JIT compiler maintains a separate count for every method. Whenever JVM comes across any method call, first that method will be interpreted normally by the interpreter, and the JIT compiler increments the corresponding count variable. This process will continue for every method. Once if any method count reaches the threshold value then the JIT compiler identifies that as a repeatedly used method(Hot spot). Immediately, the JIT compiler compiles that method and generates the corresponding native code. Next time JVM comes across that method call, then JVM uses native code directly and executes it instead of interpreting it once again so that performance of the system will be improved. The threshold count varies from JVM to JVM. JIT compilation is applicable only for repeatedly required methods.
10. Garbage Collector: GC Collects and removes unreferenced objects.

Packages

Diagram, schematic

Description automatically generated

* **java.lang** : Contains classes and interfaces that are fundamental to the design of Java programming language. Classes like *String, StringBuffer, System, Math, Integer*etc are part of this package.
* **java.io** : Provides classes for system input/output operations. Classes like *BufferedReader, BufferedWriter, File, InputStream, OutputStream, PrintStream, Serializable*etc are part of this package.
* **java.util** : Contains the collections framework, some internationalization support classes, properties, random number generation classes. Classes like *ArrayList, LinkedList, HashMap, Calendar, Date, TimeZone* etc are part of this package.
  + **Java.util.function**
    - Functional interfaces available for java8
  + **Java.util.stream**
    - Stream operations done

Default Classes

* Object - The Object class is the parent class of all the classes in java by default.
  + Object Methods.

Table

Description automatically generated

Table

Description automatically generated

* String –

Why string is immutable

* The key benefits of keeping this class as immutable are caching, security, synchronization, and performance.
  1. The String is the most widely used data structure. Caching the String literals and reusing them saves a lot of heap space because different String variables refer to the same object in the String pool.
  2. The String is widely used in Java applications to store sensitive pieces of information like usernames, passwords, connection URLs, network connections, etc. It's also used extensively by JVM class loaders while loading classes.
  3. Being immutable automatically makes the String thread safe since they won't be changed when accessed from multiple threads.
  4. String pool exists because Strings are immutable. In turn, it enhances the performance by saving heap memory and faster access of hash implementations when operated with Strings.

String methods

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | It returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | It returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns a formatted string. |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns formatted string with given locale. |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index. |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | It returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | It checks the equality of string with the given object. |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | It checks if string is empty. |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | It concatenates the specified string. |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified char value. |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified CharSequence. |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | It compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex. |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex and limit. |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | It returns an interned string. |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index. |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index starting with given index. |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index. |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index starting with given index. |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | It removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | It converts given type into string. It is an overloaded method. |

* Wrapper
  + Why wrapper classes are uses and what are?

Encapsulation

Difference between Final class and Immutable class

|  |  |  |
| --- | --- | --- |
| Sr.N | Final | Immutability |
| 1 | Final is related to reference variable | Immutability is related object |
| 2 | Can’t Reassign the object but can change its values | Can reassign the object but cant change values |
| 3 |  | Rules –  1)Declare the class as final so it can’t be extended.  2)Make all fields private so that direct access is not allowed.  3)Don’t provide setter methods for variables.  Make all mutable fields final so that its value can be assigned only once.  4) Make all mutable fields final so that its value can be assigned only once.  5) Initialize all the fields via a constructor performing deep copy.  6) Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference. |
|  |  |  |

Or

Principle of Immutable classes

Or

How to Make custom immutable class

or

How to make custom final class

Or

How to make singleton class - a class must ensure that only single instance should be created and single object can be used by all other classes

There are two forms of singleton design pattern

* **Early Instantiation**: creation of instance at load time.
* **Lazy Instantiation**: creation of instance when required.

Usage of Singleton design pattern

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

How to create Singleton design pattern?

* To create the singleton class, we need to have static member of class, private constructor and static factory method.
* Static member: It gets memory only once because of static, it contains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

Abstraction

What is abstraction?

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

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

What is difference between Interface and Abstract class?

|  |  |  |
| --- | --- | --- |
| Sr. No | Abstract | Interface |
| 1 | Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2 | Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3 | Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4 | Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5 | An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 6 | An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 7 | A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |

When to use interface and abstract class?

* In the java application, there are some related classes that need to share some lines of code then you can put these lines of code within the abstract class and this abstract class should be extended by all these related classes.
* You can define the non-static or non-final field(s) in the abstract class so that via a method you can access and modify the state of the object to which they belong.
* You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Consider using interfaces if any of these statements apply to your situation:

* It is total abstraction, All methods declared within an interface must be implemented by the class(es) that implements this interface.
* A class can implement more than one interface. It is called multiple inheritances.
* You want to specify the behavior of a particular data type but are not concerned about who implements its behavior.

Relationship (Association)

Association in Java defines the connection between two classes

Types of Association

1. IS-A Association
   1. Simple
   2. multilevel
   3. hierarchy
   4. multiple
   5. hybrid

Diagram

Description automatically generated

1. HAS-A Association
   1. Aggregation
   2. Composition

Aggregation vs Composition

|  |  |  |
| --- | --- | --- |
| Sr No | Aggregation | Composition |
| 1 | In Aggregation, objects can remain in the scope of a system without each other. | In a composition relationship, objects cannot remain in the scope of a system without each other. |
| 2 | In Aggregation, linked objects are independent of each other. | In Composition, objects are tightly coupled or dependent on each other. |
| 3 | In Aggregation, child objects don't have a lifetime. | In Composition, child objects have a lifetime. |
| 4 | In Aggregation, if we delete an assembly, it will never affect its parts. | In the case of owning a class, it affects the containing class object. |
|  |  |  |



Polymorphism

What is polymorphism?

**Polymorphism in Java** is a concept by which we can perform a single action in different ways.

Runtime Polymorphism in Java

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

Diagram

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Q)Can we return different return type in method overriding?

Ans-> From java 5.0 it is possible to have different return type for a overriding method in child class it is allow only covariant return type

Q) Overriding and Access modifier -> the access modifier for an overriding method can allow greater access modifier but not weaker.

A picture containing diagram

Description automatically generated

Q) Overriding and exception handling.

Ans-> Rule 1 – If the super class does not throw exception then child class overiden method can throw runtime exception

Rule 2 – if the super class throw exception then child method should throw same or children of parent exception

Table

Description automatically generated

Exception Handling

Exception hierarchy

Exception try catch

Finally block vs finalize method

Throw vs throws

Custom exception

Exception chaining

Collections