***JVM***

JVM (Java Virtual Machine) is an abstract machine because it doesn't physically exist. **It is a specification that provides a runtime environment in which Java bytecode can be executed.**

**JRE**

JRE is an acronym for Java Runtime Environment which physically exists. It is a set of software tools which are used for developing Java applications. It is the implementation of JVM It contains a set of libraries + other files that JVM uses at runtime.

The ***JDK*** contains JVM, JRE and other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

**JIT and JVM:**

The Just-In-Time (JIT) compiler is a component of the runtime environment that improves the performance of Java™ applications by compiling bytecodes to native machine code at run time.

While both JVM and JIT are part of Java platform but key difference between them is that JVM is an interpreter while JIT is a compiler. JVM is mandatory to run a Java program as it converts Java byte code to machine code. JIT comes next to improver performance by replacing hot Java byte code with machine code.

**Object & Class:**

An entity that has state and behavior is known as an **object**.

State: represents the value of an object.

Behavior: represents the behavior of an object.

**Class** is a template or blueprint from which objects are created.

OOPS:

Encapsulation

Abstraction

Polymorphism

Inhertance

**Encapsulation**:

It is a process by which data (variables) and the code that acts upon them (methods) are integrated as a single unit. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class.

Encapsulation in Java can be achieved by:

1. Declaring the variables of a class as private.
2. Providing public setter and getter methods to modify and view the variables values.

It refers to the process of securing methods and data into a single unit may refer to this process as "data hiding."

Types of Encapsulations in OOPs

* Member Variable Encapsulation
* Function Encapsulation
* Class Encapsulation

**ABSTRACTION**:

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details

Abstraction in Java is achieved using either an abstract class or an interface. An interface in Java represents the blueprint of a class and contains abstract methods and static constants.

|  |  |
| --- | --- |
| ABSTRACT CLASS | INTERFACE |
| 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. |
| Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
|  |  |

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.

When a class inherit from an existing class, it can reuse methods and fields of the parent class. Moreover, it can add new methods and fields in current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

Single🡪 When a single class inherits another single class, it is known as a single inheritance

Multilevel 🡪 When there is a chain of single inheritance, , it is known as a multi level inheritance

Hierarchical 🡪When two or more classes inherits a single class, it is known as hierarchical inheritance.

Hybrid 🡪Hybrid inheritance is a combination of more than two types of inheritances single and multiple. It can be achieved through interfaces only as multiple inheritance is not supported by Java.

**Polymorphism**: In Java, polymorphism refers to the ability of a class to provide different implementations of a method, depending on the type of object that is passed to the method. To put it simply, polymorphism in Java allows us to perform the same action in different ways.It is achieved by overloading and overriding.

**Method overloading is a compile-time polymorphism** in which many methods share the same name but have distinct arguments, signatures, and return types

**Method overriding is a runtime polymorphism** in which the same method with the same arguments or signature is associated with several classes.

**Aggregation**: If a class have an entity reference, it is known as Aggregation. It represents HAS-A relationship.

**Upcasting:**

We are calling the method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Bike b = new Splendor(); //upcasting

**method hiding:**

"if a subclass defines a static method with the same signature as a static method in the super class, in such a case, the method in the subclass hides the one in the superclass

**Exception**:

An exception is an event that disrupts the normal flow of the program.

Exception is a sub class of throwable class. Error is another class of throwable.

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Throw-> postpone the handling of a checked exception and throws is used to invoke an exception explicitly.

TRY with MULTI CATCH:

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you must perform different tasks at the occurrence of different exceptions, use java multi-catch block.

CUSTOM EXCEPTION:

we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

**Serialization** :

serialization is the conversion of a Java object into a static stream (sequence) of bytes, which we can then save to a database or transfer over a network.

Implement java.io.serializable interface

Treading life Cycle:

1. NEW – a newly created thread that has not yet started the execution
2. RUNNABLE – either running or ready for execution but it's waiting for resource allocation
3. BLOCKED – waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
4. WAITING – waiting for some other thread to perform a particular action without any time limit
5. TIMED\_WAITING – waiting for some other thread to perform a specific action for a specified period
6. TERMINATED – has completed its execution

**Multithreading** --> how to implement? extending thread class , implementing runnable

Interface.

**marker interface** - An [interface](https://www.javatpoint.com/interface-in-java) that does not contain methods, fields, and constants is known as **marker interface**. In other words, an empty interface is known as **marker interface** or **tag interface.** It delivers the run-time type information about an object. In short, it indicates a signal or command to the JVM.

The **Serializable** and **Cloneable** interfaces are the example of marker interface.

Serialization (converting an object into byte stream) is a mechanism in which **object state is read from the memory and written into a file or database**.

**why Java is not 100% object oriented?**

because it supports primitive data type like it,byte,long etc.,which are not objects.

**How would you create an immutable class in Java?**

* The class must be declared as final so that child classes can’t be created.
* Data members in the class must be declared private so that direct access is not allowed.
* Data members in the class must be declared as final so that we can’t change the value of it after object creation.
* A parameterized constructor should initialize all the fields so that data members can’t be modified with an object reference.
* There should be no setter

**Methods of Object Class:**

tostring() method

hashCode() method

equals(Object obj) method

finalize() method

clone() method

wait()

**13. What is String Pool:**

It is a storage area in heap memory where string literals are stored. When we create a string literal, the JVM first check that literal in the String pool. If the literal is already present in the pool, it returns a reference to the pooled instance. If the literal is not present in the pool, a new String object takes place in the String pool.

String.intern() 🡪 When we create String with new keyword it occupies space in heap memory. In order to put them in SPC , intern() method is called.

Immutability 🡪once created cannot be modified.

How it is achieved? 🡪by defining class and variable as private & Final.

How to overcome immutability in strings 🡪 string builder and string buffer objects

Other immutable objects 🡪 Wrapper Classes – Byte,Interger,Boolean

Wrapper Class 🡪

Need for immutable🡪They are thread safe. Can be used by many threads without interference.

Can be used with external libraries.

|  |  |
| --- | --- |
| HashMap | HashTable |
| Not sync. Not thread safe. Hence faster | Sync. Thread safe |
| Null allowed in key or value | Null not allowed in key or value |
| Iterator | Enumarator + Iterator |

|  |  |
| --- | --- |
| Array | ArrayList |
| Static – cannot change size | Dynamic – you can resize |
| Work with objects and primitive types | Work with objects and not primitive type |

|  |  |
| --- | --- |
| ArrayList | Vector |
| Non - Synchronised | Synchronised |
| Not thread safe | Thread safe |

**equals()  and ==**

== is an *operator* and equals() is a *method*.

Operators are generally used for *primitive* type comparisons and thus == is used for memory address comparison and equals() method is used for comparing *objects*.

JAVA 8:

Functional **interface**:

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. **Runnable**, **ActionListener**, **Comparable** are some of the examples of functional interfaces.

Java 8 Functional Interfaces and Lambda Expressions help us in writing smaller and cleaner code by **removing a lot of boiler-plate code**