JAVA

who? | when? | which? |
JVM Arch | Literals |
Type conversion | casting | Generics |

=> Java Advantages
-> Platform Independence (Write Once, Run Anywhere - WORA)
-> Object-Oriented Programming (OOP)
-> Automatic memory management (Garbage Collection) prevents memory leaks.
-> Multi-threading Support.
-> Rich Standard Library (Java API)
-> open Source

=> JDK : JDK is a full package that includes, JRE, compilers and loader.
=> JRE: providing libraries and the JVM to run Java applications.
=> JVM: An engine that executes Java bytecode, enabling Java applications to run on any device/platform without modification, ensuring cross-platform compatibility.

====================
String
====================

=> String
-> is class in Java.String obj are by default immutable (unchanged).but can be mutable(changed) by using the String Buffer and StringBuilder.
-> String Buffer is Thread Safe bcz it uses multiple thread.
-> String builder is not thread Safe.

=> String Literals
-> String s1 = “ Hello”
-> String s2 = “Hello”
-> String Literals create only one object in SCP , no duplicate obj are not allowed here in SCP.

=> String Object
-> String new s1 = new String(“Hello”);
-> String new s2 = new String(“Hello”);
-> Now it will create 2 obj in heap memory

Example 1:
String name = "Praveen";

name = name + "Raj";
1). ”Praveen" is stored in the **String Pool**.
2). Raj" is a string literal, also stored in the **String Pool**.
3). When name + "Raj" is executed:

- A **new String object** is created in the **heap memory** because String is immutable.
- The result "PraveenRaj" is stored in the heap (not the String Pool unless explicitly interned).

4). Objects Created:

- "Praveen" (String Pool)
- "PraveenRaj" (Heap Memory

Example 2:
String s1 = "Raj";
String s2 = "Raj";
1)”Raj" is created in the **String Pool** when s1 is initialized.
For s2, the JVM finds "Raj" already in the String Pool, so s2 points to the **same object**.
Objects Created:

- Only **one object** ("Raj") in the **String Pool**.
- No new object in the heap.

=> Why String immutable ?
-> If a string is immutable, it can be safely reused across multiple places without creating multiple copies, saving memory and improving performance.

Advantages of String Immutability

1. **Memory Efficiency**:
   - With **string pooling**, Java reuses existing strings instead of creating new ones for identical values. For example:
     String s1 = "hello";
   - String s2 = "hello";
   - Both s1 and s2 refer to the same object in the string pool.
2. **Thread Safety**:
   - Since String objects cannot be modified, they can be safely shared across multiple threads without the need for synchronization.
3. **Security**:
   - Strings are commonly used for sensitive data like passwords, URLs, or file paths. If strings were mutable, their values could be altered, leading to potential security issues.
4. **Performance**:

- Immutable strings eliminate the need to recalculate hash codes, which is beneficial for collections like HashMap.

Array -> int num [] ={1,2,3,34} // int num[] = new int[3];
  cons : size is fixed to overcome this we are moving to collection ArrayList
Types of Array;
 => one dimensional, multi dimensional and jagged array

=================
  DATA-TYPES
=================

I) Primitive data Types
a) Integer TYPES
1) Byte- Size: 8 bits (1 byte)
2) Short- size: 16 bits (2 byte)
3) int - size : 32 bits (4 byte). (Default value = 0)
4) long- 64 bits (8 byte)
b) Floating-Points Types
5) Float- Size: 32 bits (4 bytes)
6) Double- Size: 64 bits (8 bytes)
c) Character Types
7) char - size: 16 bits(2 byte)
d) Boolean Types
8) boolean - size 1 bit (default value = false)
II) Non-Primitive Data Types
1) String (Default value = null)
2) Arrays
3) Classes
4) Interface

=================
  VARIABLES
=================

1. Local variable
   -> declared inside the method , constructor or blocks

-> access within the method only or block.
-> must be assigned a value before use.

2. Instance Variable
   -> Declare inside the class and outside the Method.they belongs to the object of the class.
   -> Accessible through out the class via object.
3. Static Variable
   Example: static int staticVar = 100;
   -> Declared with the Static keyword and shared across all the instance of a class(similar to instance variable).
   -> can access via class name
4. Final Variable.
   -> final variable declared with final keyword , cannot be changes after initialisation.

Static Variable -> in Java is a class-level variable that is shared among all instances(obj) of the class.
Static Method -> belongs to the class rather than any instance and can be called without creating an object of the class and cannot be overridden.
Static Block -> A static block in Java is used for static initialisation of a class and runs once when the class is loaded.

Transient
-> Only transient fields are skipped during serialization.
-> After deserialization, transient fields get default values (null for objects, 0 for integers, etc.).
-> Ex: transient int salary; // Will NOT be serialized

volatile
-> Changes made by one thread are immediately visible to others.
-> The variable is always read from main memory, not from thread-local cache.
-> Works only for **simple reads/writes** (not for compound actions like count++).
-> static volatile boolean flag = false; // Now always read from main memory

==========================
Control Flow Statements in Java
==========================

Control Flow Type	Examples	Purpose
Decision-Making	if, if-else, switch	To make decisions based on conditions.

Looping	for, while, do-while	To repeat a block of code.
Jump	break, continue, return	To alter the flow of loops or exit from methods.

==================================

Memory Management in Java

==================================

1. Method Area
   -> Store the Metadata ,Static variable and byte code of loaded class.
   -> Method area was the part of PermGen space.
   -> After java8 Method Area was replaced by MetaSpace which is dynamically resizable, eliminating many memory constraints.
2. Heap Area
   -> it stores objects and JRE classes/instance variables.
   -> when we create a keyword “new” for creating new object , the obj will store in heap. Reference will store in Stack memory.
3. Stack
   -> store the methods and local variable.
   -> LIFO , each thread has its own stack
4. Program Counter (PC) register
   -> holds the address of the next instruction to execute for a thread.
5. Native Method stack.
   -> It is also called as C Stack.
   -> It is stack for the native code written in a language other than java.

=> Garbage collection:
-> Garbage collection in Java is a powerful feature that ensures efficient memory management and prevents memory leaks
-> finalize() is method called by JVM to clean up memories in garbage collection.
-> finalize() method was depreciated after java 9. It replaced by try-with-resource.

=> String Constant Pool
-> comes inside the heap memory.
-> In SCP duplicates objects are not allowed
-> First JVM will check in SCP if not create new obj in SCP.

=> **Types of Access Modifiers**

Modifier	Class	Package	Subclass	World (Outside Package)
public	Yes	Yes	Yes	Yes

protected	Yes	Yes	Yes	No
(default)	Yes	Yes	No	No
private	Yes	No	No	No

====================
OOPS -Principles
====================

Class -> Blueprint for creating the object contains the methods and function.
Object -> is an instance of a class.
Inheritance -> class which inherit the method from the parent classes.
Types -> Single ,Multiple, Multilevel, Hierarchical,Hybrid
Single -> a class inherits from one super class.
Multilevel -> a class inherit from super class ,also a class inherit that sub class also.
Hierarchical -> Multiple classes inherit from single Super class.
Multiple ,Hybrid -> not supported in java.
Polymorphism -> single action different ways. The ability of different objects to respond to the same method call in different ways.
-> Achieved by Method overloading and over ridding.
-> addItem(Item) or addItem(List<Item>) over loading.
-> searchItem(Integer id) or searchItem(String Name) Over Loading.

Encapsulation -> Entity/ hiding or wrapping data (variables) and methods in single unit. Is a class hidden from the other classes.
=> Abstraction
-> Abstraction is achieved by interface and abstract class. shows only the function not the implementation
-> **Interface**:contains only the abstract methods without the body.
-> In interfaces, all methods are abstract by default.
-> **Abstract Method**: An abstract method does not have a body; it ends with a semicolon.
-> Abstract Class: must declare with abstract keyword, can have or not abstract method or normal method./we can also implement here but in interface it is not possible.
-> Abstract classes allow you to define shared behaviour across subclasses and force them to provide specific implementations for some methods.

Feature	Abstract Class	Interface
Contains Abstract Methods	Can contain abstract methods and non-abstract methods.	Contains only abstract methods by default (prior to Java 8).
Default Methods	Not supported (methods must be explicitly implemented).	Supports default and static methods (since Java 8).

Abstract Keyword	Mandatory for abstract methods.	Methods are implicitly abstract (no need to specify abstract).
Extend/Implement	Can extend another Java class (single inheritance).	Can implement multiple interfaces.
Multiple Inheritance	Not supported (can extend only one class).	Supported (can implement multiple interfaces).
Fields	Can have instance variables (with access modifiers).	Can only have public static final (constants).
Constructors	Can have constructors.	Cannot have constructors.
Type of Abstraction	Supports partial abstraction.	Ensures 100% abstraction (prior to Java 8).
Use Case	When you need to share code among related classes.	When you need to define a contract without implementation.
Performance	Slightly faster than interfaces due to direct inheritance.	Slower due to method resolution in multiple inheritance.

Constructor -> has same name as Class Name, Does not return type.it will call automatically when the obj are created.
Method Overloading -> same method but different args.
Method Overriding -> sub class has the same method same args from the parent class (parent class override by sub class) also called as dynamic binding

Dynamic Binding
-> it will point the object not the reference.
-> Ex. Animal animal = new Dogs();
-> It will call the methods only in dog class not from Animal class.

Serialisation -> Object into Byte-Stream/ serialised and stored in a file. (Archived by JPA ,hibernate)EX:Game App.
Wrapper class -> is used to convert the primitive data types into object. Also called as boxing and unboxing
-> auto boxing.
-> Some libraries and collections (like ArrayList, HashMap, etc.) work with objects, not primitive types
-> if u want to handle null values u need wrapper class , primitive class will not handle the null values.

===================
Exceptions
===================

-> Runtime time exception / unchecked.
-> Compile time exception/ checked.
-> Final : is an access specifier, or a keyword to declare constant to prevent inheritance and prevent overriding.
-> Final class : we can't extend the final || no child class || inheritance not possible
-> Final method : can't be over ride
-> Final Variable : always constant we can't change the variable
-> Finally: always executed after try block. Handle the cleanup code
-> Finalize: used to remove the memory from garbage/ before deleting object in garbage collector this method will call by JVM.
-> throw: inside the method , we can customise the exception as well by manually.only of unchecked exceptions.
-> throws: method level exception and compile/checked time error. Will throws the exception to the caller method or JVM
-> throwable: is part of super class for all types of error and Exception.

-> Exception : which throws by our programs.
-> Error : which throws by hardware or some memory issues.
-> Throwable is the Parent class of all errors and Exception.
=> To Print the Exception.
-> SOUT(e.printStackTrace)
-> SOUT(e.toString)
-> SOUT(e.getMessage())

=> Multiple Catch in single Try
-> the hierarchy need to follow from Child to parent Class(ArithmeticException -> Exception).
-> Finally: mainly to used to cleanUp the memory/cleanUp the code
-> Finally will handled always weather exception is handled or not
-> Finally will not execute only when JVM get showdown(system.exit(0)) ;
-> If there is any error in catch block the try won't work only finally work;

=> Nested-try-catch-finally
-> In Nested try it will check the inner try , then if not check the other catch block
-> try without catch block ? True
-> try without catch and finally ? False compile time error.
-> Inside the (catch || Finally) if we have the try-catch-finally ? True
=> Custom Exception
-> we need to create a class that should extend Exception or Run-time-Exception.
-> we need to create a constructor as well for that class.
-> then we can access our custom expedition.
-> Constructors enable you to pass specific details about the exception (e.g., error message, error code, root cause)

Types of Exception :

Exception	Type
NullPointerException	Unchecked/Runtime
ArrayIndexOutOfBoundsException	Unchecked
ArithmeticException	Unchecked
IOException	Checked/Compile
FileNotFoundException	Checked
SQLException	Checked
ClassNotFoundException	Checked

===================
Collection
===================

-> Iterable -> collection

=> Iterator: Iterator interface is the part of java collection.
-> Iterate is used to iterate the collection of object such as list, set or any collections.,
-> here we add, remove or replace the element in the collection while iterating.
-> boolean hasNext() -> return false if there is no element.
-> E next() -> iterate the next element
-> void remove() -> remove the last element

=> List Iterator
-> is an interface that extends Iterator.
-> it provide additional methods to iterate list.
-> iterate both backward and forward is available
-> hasPrevious(), previousIndex(), hasNext(), hasNextIndex(),set()

=> Array List
-> allow Duplicates, maintain insertion order.
-> internally uses a dynamic array to store the elements.
-> better for storing and accessing data.
-> not synchronized, not thread safe.

=> LinkedList
-> allow Duplicates, maintain insertion order
-> internally uses a doubly linked list to store the elements.
-> better for manipulating data

-> Contains Data and pointer(has the index of previous&next) by which we add or delete the data easily and fast.when compared to Array list
-> synchronized, thread safe.

=> Vector
-> Used to use for Thread safe
-> Synchronised
-> performance is slow when compared to array and linked list.

=> Stack
-> LIFO.
-> insertion and deletion take place from one side known as a top
-> push add the items ,pop delete the recent add items

=> Queue
-> FIFO
-> A *queue* is also ordered, but you'll only ever touch elements at one end,FIFO
-> Priority Queue -> sort and save in the array.

=> Set
-> Only accept Unique element
-> not maintain insertion order.
-> allows a single null value.

=> HashSet
-> unique element, not maintain order , use hash function.
-> HashSet is implemented using a Hash table.

=> LinkedHashSet
-> unique element, maintain order.

=> TreeSet
-> unique element, sort and store.

=> Map
-> Key value pair. A Map is useful if you have to search, update or delete elements on the basis of a key.

=> Hash map
-> Initially Empty hashMap: Here, the hashmap’s size is taken as 16.
-> Java HashMap maintains no order.
-> It allows us to store the null elements as well, but there should be only one null key.(index 0)

-> If you try to insert the duplicate key, it will replace the element of the corresponding key.
-> Ex: 16 buckets (hash:key:value:next)
-> find the hashCode() of the key.
-> find the bucket index using hashCode.
-> simply add to linked list as first node -> if key already present -> add to linked list by replacing the existing equal node
-> if not add the linked list in new node in the same bucket.

-> Hashing : used to cover the object into integer , for faster search.
-> Hash collision: when we get the same hash code for particular index collision will take place.it will fixed by equals method.
if k1.equals(k2) it will replace or else create new new node(new index) in hashMap.

hashCode(s)=s[0]×31n−1+s[1]×31n−2+⋯+s[n−1] -> “apple”. / s-> string n-> length of string
s[0] -> ASCII key value of that char
index=(hashCode(key))&(n−1)   -> key=“apple” n= 16
Default bucket size: 16
Default load factor: 0.75
Maximum capacity: 2^30 ;

=> HashTable
-> synchronised
-> HashTable is thread-safe.
-> HashTable does not allow null keys or values.

=> Concurrent Hash map // exception
-> provide thread safe implementation
-> allow multiple threads to access and modify the map concurrently without need of explicit synchronisation.

-> lock will applied for particular node where in hash map lock will for entire Linked list
-> scalable
-> thread safe but not for entire object , only in bucket level called fragments.
-> does not allow null key

=> Synchronised Hash map
-> lock the whole map
-> not scalable.
-> thread safe for whole object.
-> allow null key.

Collection	Default Initial Capacity	Explanation
Vector	10	Vector is synchronized, and its default initial capacity is 10.
ArrayList	10	ArrayList dynamically resizes, starting with a default capacity of 10.
LinkedList	N/A	LinkedList is a doubly-linked list and does not have an initial capacity. It grows as needed.
HashMap	16	HashMap has a default capacity of 16 and a load factor of 0.75, meaning resizing occurs after 12 elements.
LinkedHashMap	16	LinkedHashMap, like HashMap, has a default capacity of 16 and keeps insertion order.
ConcurrentHashMap	16	ConcurrentHashMap starts with a default capacity of 16 and is thread-safe for concurrent access.
HashSet	16	HashSet is backed by HashMap and shares the same default capacity of 16 and a load factor of 0.75.
LinkedHashSet	16	LinkedHashSet maintains insertion order and has a default capacity of 16.
TreeSet	N/A	TreeSet is based on a Red-Black tree and does not have an initial capacity.

Feature/Collection	List	Set	Map	Queue
Implementation Examples	ArrayList, LinkedList, Vector	HashSet, LinkedHashSet, TreeSet	HashMap, TreeMap, LinkedHashMap	LinkedList, PriorityQueue
Order	Maintains insertion order	No order	Order depends on implementation	FIFO (First-In-First-Out)
Duplicates	Allows duplicates	No duplicates	Keys are unique, values can be duplicate	Allows duplicates (depends on implementation)

Null Elements	Allows multiple null values	Allows one null value	Allows one null key and multiple null values	Allows null elements (depends on implementation)
Synchronization	ArrayList (not synchronized), Vector (synchronized)	Not synchronized	HashMap (not synchronized), Hashtable (synchronized)	Typically not synchronized (e.g., LinkedList)
Random Access	ArrayList (supports), LinkedList (does not)	Not applicable	Not applicable	Not applicable
Memory Overhead	ArrayList (lower), LinkedList (higher)	HashSet (lower), TreeSet (higher)	HashMap (higher), TreeMap (higher)	Depends on implementation
Performance	ArrayList (faster for random access), LinkedList (faster for insertions/removals)	HashSet (faster), TreeSet (slower)	HashMap (faster), TreeMap (slower)	LinkedList (good for frequent insertions/removals), PriorityQueue (sorted)
Thread Safety	Vector (thread-safe), ArrayList (not thread-safe)	Not applicable	HashMap (not thread-safe), Hashtable (thread-safe)	Typically not thread-safe (e.g., LinkedList)
Sorting	Not sorted	TreeSet (sorted)	TreeMap (sorted by keys)	PriorityQueue (sorted based on priority)

Difference between ArrayList and Vector

ArrayList	Vector
ArrayList is not synchronized.	Vector is synchronized.
ArrayList is not a legacy class.	Vector is a legacy class.
ArrayList increases size by 50% when resized.	Vector doubles its size when resized.
ArrayList is not thread-safe.	Vector is thread-safe as all methods are synchronized.

Difference between ArrayList and LinkedList

ArrayList	LinkedList
ArrayList uses a dynamic array.	LinkedList uses a doubly linked list.
ArrayList is not efficient for manipulation.	LinkedList is efficient for manipulation.
ArrayList is better for storing and fetching data.	LinkedList is better for manipulating data.
ArrayList provides random access.	LinkedList does not provide random access.
ArrayList takes less memory overhead.	LinkedList takes more memory overhead.

Difference between Iterator and ListIterator

Iterator	ListIterator
Iterator traverses elements only in forward direction.	ListIterator traverses in both forward and backward directions.
Iterator can be used in List, Set, and Queue.	ListIterator can only be used in List.
Iterator can perform remove operation only.	ListIterator can perform add, remove, and set operations.

Difference between Iterator and Enumeration

Iterator	Enumeration
Iterator can traverse both legacy and non-legacy elements.	Enumeration can traverse only legacy elements.
Iterator is fail-fast.	Enumeration is not fail-fast.
Iterator is slower than Enumeration.	Enumeration is faster than Iterator.
Iterator can perform remove operation.	Enumeration can only traverse the collection.

Difference between List and Set

List	Set
List can contain duplicate elements.	Set contains only unique elements.
List maintains insertion order.	Set does not maintain insertion order.
List contains a legacy class (Vector).	Set does not have a legacy class.
List allows multiple null values.	Set allows only one null value.

Difference between HashSet and TreeSet

HashSet	TreeSet
HashSet does not maintain any order.	TreeSet maintains ascending order.
HashSet is implemented using a Hash table.	TreeSet is implemented using a Tree structure.
HashSet is faster than TreeSet.	TreeSet is slower than HashSet.
HashSet is backed by HashMap.	TreeSet is backed by TreeMap.

Difference between Set and Map

Set	Map
Set contains values only.	Map contains key-value pairs.
Set contains unique values.	Map contains unique keys but can have duplicate values.
Set allows only one null value.	Map allows one null key and multiple null values.

Difference between HashSet and HashMap

HashSet	HashMap
HashSet contains only values.	HashMap contains key-value pairs.
HashSet implements the Set interface.	HashMap implements the Map interface.
HashSet does not allow duplicate values.	HashMap allows duplicate values but with unique keys.
HashSet allows one null value.	HashMap allows one null key and multiple null values.

Difference between HashMap and TreeMap

HashMap	TreeMap
HashMap does not maintain any order.	TreeMap maintains ascending key order.
HashMap is implemented using a Hash table.	TreeMap is implemented using a Tree structure.
HashMap can be sorted by key or value (using other means).	TreeMap can be sorted by key only.
HashMap allows one null key and multiple null values.	TreeMap does not allow null keys, but can have multiple null values.

Difference between HashMap and Hashtable

HashMap	Hashtable
HashMap is not synchronized.	Hashtable is synchronized.
HashMap allows one null key and multiple null values.	Hashtable does not allow null keys or values.
HashMap is not thread-safe.	Hashtable is thread-safe.
HashMap inherits the AbstractMap class.	Hashtable inherits the Dictionary class.

Difference between Collection and Collections

Collection	Collections
Collection is an interface.	Collections is a utility class.
Collection provides data structure functionality to List, Set, and Queue.	Collections provides static methods to operate on collections.
Collection defines methods for data structures.	Collections provides methods like sorting, synchronization, etc.

Difference between Comparable and Comparator

Comparable	Comparator
Comparable provides a single sort order.	Comparator provides multiple sort orders.
Comparable has a method compareTo().	Comparator has a method compare().
Comparable is in the java.lang package.	Comparator is in the java.util package.
Implementing Comparable modifies the class.	Using Comparator does not modify the class.

Feature	ConcurrentHashMap	Synchronised Hash Map
Package	java.util.concurrent	java.util.Collections
Locking Mechanism	Fine-grained locking	Whole map locking
Performance	Better in high concurrency	Slower in high concurrency
Null Keys/Values	Not allowed	Allowed
Thread-Safety	High	Moderate
Use Case	High-frequency multi-threaded operations	Low-concurrency or occasional use

Feature	Fast-Fail Iterator	Fail-Safe Iterator
Definition	Throws an exception if the collection is modified while iterating.	Allows modifications during iteration without throwing exceptions.
Implementation	Directly accesses the collection’s structure and detects modifications.	Uses a **copy of the collection** to iterate, so modifications do not affect iteration.
Concurrent Modification Exception (CME)	Yes, throws ConcurrentModificationException if modified.	No, because it works on a separate copy.
Performance	**Faster** (no extra memory usage).	**Slower** (creates a copy, uses more memory).
Used in	ArrayList, HashMap, HashSet, etc.	ConcurrentHashMap, CopyOnWriteArrayList, etc.

======================

Design Pattern in Java
========================

-> Creational: is all about creating object [singleton, factory, builder, prototype.,]
-> Structural : is one object get the features of another object [adapter, bridge, composite.]
-> Behavioural : just communication between two object not get the features of another object.[command, state, visitor,iterator]
=> Singleton Design Pattern :
-> A class has only one instance, a class that has only one instance and provides a global point of access to it.
-> create a static instance(obj) [static Abc obj = new Abc()] why static? Bcz we having the static method that will return the static obj. So the obj also should in static.
-> create a private constructor
-> create a static method [return the static obj]
(advantage) -> Saves memory because object is not created at each request. Only single instance is reused again and again.
(useCase) -> Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

EXAMPLE:
```
class Logger {
    private static Logger instance = new Logger(); // Early instance creation
    private Logger() {} // Private constructor

    public static Logger getInstance() {
        return instance;
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

=> Builder Pattern;
-> Allow us to create multiple optional to create an object, not mandatory to call all fields.
-> class having userName, id, rollNo, dob while creating obj we don’t need to call all the fields.

=>Factory Pattern

=> Iterator: The Iterator pattern is widely used in Java collections like List, Set, and Map, where you can call the iterator() method to get an Iterator object to traverse through the collection.

================
Threads
================

-> Threads allows a program to operate more efficiently by doing multiple things at the same time.
-> To perform complicated tasks in the background without interrupting the main program.
-> Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
There are two types of threads in Java as follows:

- User thread
- Daemon thread

To create a thread
i) By extending Thread class
ii) By implementing Runnable interface.
-> in java we can't achieve the multiple inheritance so we are implementing the Runnable interface.

Thread Class vs Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

Strat() -> It is a method belongs to thread class, it will create a new thread for u, Start will call the run() method
run () -> will run thread
waitl() -> it will stop the execution of the thread for particular time and it will start only when it call by notify() or notifyAll()
sleep() -> it will stop the thread and start automatically after time over.
notify() - >notify the waiting threads
notifyAll() -> will notify all waiting threads of a particular object.
join () -> will wait until thread complete

isAlive() ->it will check the thread return the boolean
setName -> set the same
priority -> when the processor get n no of thread at same time the priority will be assign.
    ->which thread want to send to the processor as the first priority. Default priority is 5, min= 1 ,max= 10.

Thread State
1. New
-> Thread t = new thread ();
-> Here thread is created but not started
2. Runnable
-> t.start()
-> now the thread is ready to run
3.Blocked
-> synchronized(t1) {
-> When the thread tries to enter a synchronized block, current thread is on hold
4.Waiting
-> Thread enter to the waiting state
-> t.wait()
5.Terminated
-> when in completes the run() method the thread is terminated.

Uses of thread
-> use all the core in machine
-> send a synchronised sequence
-> we can access web app using multi threading (handle by servlet)
-> for game developing multi threading Is there
-> Performance.
-> Better UI response.

Synchronised in Thread
-> synchronised should use before method of the thread
-> it will create lock mechanism for that thread method and will put remaining thread in hold.
-> when we need to access only one thread at a time , we can use the synchronised thread.
synchronization
-> is the capability to control the access of multiple threads to shared resources (like variables, objects, or methods) to prevent data inconsistency
-> Synchronization ensures that only one thread can access a synchronized method or block at a time,
-> thereby protecting shared resources from being corrupted when multiple threads attempt to update them simultaneously.

====================
JAVA 8
====================

=> Java 8(Features)->Functional Interface
  ->Lambda Expression
  ->Method Reference
  ->Stream API
  ->For Each Method
  -> java DATA and Time API
-> In java 8 interface can be implemented the method by two ways default method and static method,
interface A{ interface B{ class C {
default void show();   default void show (); void show();
Sop(a) sop(b); sop(c)
} } }

Class D extend C implement B{
D obj = new D();
obj.show()
}

The OP will “c” normal method(class) has more power than the Default(in interface)
  -> Lambda Expression only works in the Functional interface.

=> Functional Interface : only one abstract method, n no of default and static methods
-> main use of Functional interface is To implement Lambda Expression
-> if we have more than one abstract method lambda function get confusion to call which method.

=> Functional Interface Before Java 8 /or/ **s**ome standard Java pre-defined functional interface
# runnable
# callable
# Comparator**:** use to sort different objects in a user-defined order
# Comparable**:** use to sort objects in the natural sort order

-> Equals() : checks if two objects are the same or not and returns a boolean.
-> Compare(): It checks which of the two objects is "less than", "equal to" or "greater than" the other

=> can functional interface extends another functional Interface? Why ? False

=> Function ,Predicate, consumer , supplier, Operator five most important FI in java 8.

=> Default Method
-> It provides an implementation/body of methods within interfaces.
-> default method Call through an instance/obj of MyClass.
-> default method can be override in sub class by implementing the default method or the implementation in optional part.

=> Static Method
-> static method call via the interface name.
-> cannot be overridden by implementing the classes.
-> static method in an interface cannot access instance variables or instance methods.
-> It can only access other static methods or variables within the interface.

=> Stream Methods:
-> Intermediate Operations

Method	Description
filter(Predicate<T> predicate)	Filters elements based on a condition.
map(Function<T, R> mapper)	Transforms elements from one type to another.
flatMap(Function<T, Stream<R>> mapper)	Flattens nested structures (e.g., List<List<T>> → List<T>).
distinct()	Removes duplicate elements based on equals().
sorted()	Sorts elements in natural order.
sorted(Comparator<T> comparator)	Sorts elements using a custom comparator.
limit(long maxSize)	Truncates the stream to a given number of elements.
skip(long n)	Skips the first n elements.
peek(Consumer<T> action)	Performs an action on each element without modifying them (used for debugging).

->Terminal Operations

Method	Description
collect()	Collects elements into a collection (List, Set, Map)
count()	Returns the count of elements
min()	Finds the minimum element

max()	Finds the maximum element
findFirst()	Retrieves the first element
findAny()	Retrieves any element
allMatch()	Checks if all elements match a condition
anyMatch()	Checks if any element matches a condition
noneMatch()	Checks if no element matches a condition
reduce()	Combines elements into a single result
forEach()	Performs an action for each element
toArray()	Converts a stream to an array

=> Method reference.
-> It is used to refer to a method without invoking it. :: (double colon) is used for describing the method reference
-> Uses function as a parameter to invoke a method.
-> Syntax is class::methodName
-> Integer::parseInt(str) \\ method reference
-> str -> Integer.ParseInt(str); \\ equivalent lambda

=> Lambda Expression.
-> Syntax is (parameters) -> expression or (parameters) -> { statements } //syntax
-> A function that can be shared or referred to an object.
-> Lambdas help to reduce boilerplate code.
-> Lambda expressions can be passed as a parameter to another method.
-> Lambda expressions can be standalone without belonging to any class.

=> Stream vs Parallel Stream:

Feature	Stream	Parallel Stream
Execution	Processes data **sequentially**, one step at a time.	Processes data **in parallel**, dividing tasks across multiple threads.
Threading	Operates on a **single thread**.	Operates on **multiple threads** (using the ForkJoin framework internally).
Performance	Suitable for small datasets.	Can provide better performance for large datasets.
Order	**Preserves order** of the data (unless otherwise specified).	May not always preserve the order of processing due to parallel execution.
Use Case	When tasks are small or order must be preserved.	When tasks are large, independent, and performance can benefit from parallelism.

=> Optional Class
-> to handle null , can return an Optional.empty()
-> Optional provides methods like map(), filter(), and flatMap()

==================
Spring Boot
==================

=> Spring Boot Advantages and Features :
-> Stand alone.
-> Embedded.
-> Production Ready Features.
-> Reduce the Boiler plate codes.
-> Externalised Configurations
-> Spring Boot Staters.

=> IoC
-> Inversion of control (it is just principle)
-> is a kind of framework it provide the tyre instance
-> it was achieved by the DI and it was implemented by the DI(Dependency Injection).
-> shifting the creation, management and the lifecycle of objects from the programmer to the framework

=> DI
-> removes the dependency from the code.
-> it is loosely coupling we no need to create obj for that by Hard coding or manually created.
-> Easy manage the code and test the application.
-> By default Obj will created automatically in Spring after creating @component.
-> Spring Container contains the beans(Obj).how bean is created inside the Spring Container? At the time of run app.
-> @component : will create a Bean(ob)j for the class / create obj in the Spring container

-> Two type of scope : Prototype(create more than one obj) and Singleton(default create one obj)
-> Actually there are six Scope in beans: Prototype, Singleton, Request, Session, application and Web-socket.

=>Constructor Injection
-> Default @Autowire not need
-> when we need an object along with all its dependencies.
-> For example, a CAR can have various dependencies like engine, gear, steering, and other relevant components without which a CAR is useless.
-> So, we need all those components wired together in the object of a car.

=> Setter Injection
-> when we have some optional dependencies that are not mandatorily.
-> For example, let's say we have a new user in our application, and they have their phone number as an optional field in their profile details.
=> Field Injection
-> directly into class fields without using constructors or setter methods
-> not best way to use, bcz hard to use outside the spring container

=> Annotations
-> @Autowire : search the obj by type /automatic dependency injection
   for Field and Setter Injection @Autowired no need mentioned.
-> @Profile: used to set / map the bean for particular env (dev/prod/qa)
-> @SpringBootApplication -> @ComponentScan, @Configuration, and @EnableAutoConfiguration
-> @Configuration
-> @OneToMany(fetch = FetchType.EAGER) =>

=> @SpringBootApplication
-> is used to @EnableAuto-configuration, @ComponentScan, and @Configuration for a Spring Boot application.

=> @EnableAutoConfiguration
-> when spring find the starter web dependency it will tell Spring Boot to automatically configure beans for common use cases like data sources, JPA, web servers automatically.
-> scanning the classpath for certain libraries (e.g., Hibernate, Tomcat) and auto-configures beans accordingly.

=> @Confuguration

-> Class level configuration
-> allows us create a custom bean creation
-> is used to manually configure beans and dependencies.
=> @ComponentScan
-> It will scan the specified package (or sub-packages) for classes annotated with @Component, @Service, @Repository, or @Controller, and register them as Spring beans.

=> @Component
-> Class level component
-> is used to tell Spring that this class should be treated as a Spring-managed component (or bean).
-> will create a bean(obj) automatically, we no need create obj manually;
-> we can give the obj like this also [@Component (“objName”)]
-> Or Else it will take the class name as ObjName by changing the first letter as lowerCase.

=> @Bean
-> Method level component
-> Used within the @configuration class to explicitly declare a bean.
-> @Bean for more customised bean definitions or when you cannot modify the class (e.g., third-party libraries).
-> by default the bean name is same as the method name.
-> we can specify the bean name as @Bean(“beanName”).

=> @AutoWired
-> inject the bean Automatically.
-> it is used in constructor injection, setter injection, field injection.

=> @Controller
-> used to developing traditional web applications
-> @ResponseBody Must be added to methods when returning JSON/XML data
-> Return type will be ModelAndView for rendering views (JSP os Thymeleaf)

=>RestController
->used in RESTful web controller for APIs/RestAPI’s
-> Combination of @Controller and @ResponseBody
-> Return type will be Typically objects or ResponseEntity for returning JSON/XML

=> @Qualifier
->Search the obj by the Specific name when we have the same name of bean.
-> to avoid confusion we are using it.

=>@Primary
-> Class level annotation.

-> sets the default preference for a bean.

=> StereoType Annotation
-> are @Controller ,@Sevice, @Repository these all derived from the @Component
-> each annotation will create a bean/layer in the spring container.

=> @Lazy
-> class level annotation
-> Spring will not create bean at run time
-> we need to create or call the bean explicitly.

=> @Value
-> used to assign a default value for the var and methods
-> we can also the value from the properties files by calling property key.

=>@RequestMapping
-> used to map the web request
-> we will pass the value inside that which specify the path of the URL.

=>@RequestBody
-> is used to pass the java Obj in the request params
-> mainly used in POST,PUT or PATCH.

=> @ResponseBody
-> Map the java object into HTTP request
-> convert it java

=> @Transactional
-> If your service method interacts with multiple repositories
-> Whenever you need to ensure that multiple database operations are executed as a single unit of work (either all succeed or all fail).
-> rollback

=> Actuator in SpringBoot
-> Production ready endpoints to monitor your application.
-> We can monitor the request/response, liqubibase, threads, bean, health, env
-> we can enable it by adding it in properties files.
-> custom endpoints /@end-point we can write the logic
-> we can run it diff port

==================
Micro Service

==================

Micro service Advantages
-> Microservices are independently manageable services
-> upgrade each service individually
-> emanding service can be deployed on multiple servers to enhance performance
-> Dynamic scaling
-> Easy to test , easy to deploy
-> even in failure it will not affect the other service.

Service-registry (Eureka)
-> all the service will attached to the service registry
-> dependency : Eureka Server , Spring Web
-> @EnableEurekaServer
-> default port 8761
-> All the client server will register with Eureka Server
->

Service(dept)
-> dependency: Eureka Client
-> @EnableDiscoveryClient

API GateWay
-> All the API are exposed through API gateWay only not directly to the particular service.
-> All the request comes to the API gateway from there It will route to service.
-> accept the request from client and it knows to which service to go and get the response to the client.
->

Config Server
-> save all the default config in config server
->

ZipKin
-> log trace

Feing
-> service communicate with each other by Rest template or web client . IPC is not recommended.

-> connect one service to another service(Open Feing dependency)
-> Create an Interface
-> @FeignClient("QUESTION-SERVICE")

-> Then we need to @Autowired
-> @EnableFeignClients in the main application.

Actuator in spring ?

Config Server => @EnableConfigServer
Spring Reactive Web (for Web client) => service to service connection

================
DataBase
================

=> Index
-> to retrieve the data fast
-> An index is a database structure that improves the speed of data retrieval by maintaining a sorted order of key fields.
-> It reduces query execution time but can slightly slow down insert/update operations due to index maintenance.
=> Types of Indices
-> **Unique Index**, **Clustered Index**, **Non-Clustered Index**, **Full-Text Index**, and **Composite Index**.

Why Use Indexing?

->To enhance query performance by reducing the amount of data scanned.
-> It is essential for optimizing large-scale database operations.

=> Normalisation
-> Normalization organizes data in a database to reduce redundancy and improve integrity.
-> It ensures data consistency by dividing it into smaller, related tables.

Why is Normalization Important?

-> It eliminates data duplication, reducing storage space and anomalies during data updates.
-> It ensures data integrity and consistency across the database.

=> Foreign key / primary key

->A **primary key** uniquely identifies each row in a table and cannot contain NULL values.
-> A **foreign key** establishes a relationship between two tables by referring to the primary key of another table.

=> Inner Join
-> only give the perfect match not null values from the table
=> Left join
-> will give all the data from the left table but also from the matched data from the right table
=> Right join
-> will give all the date from the right table and also give the matched data from the left table
=> Cross Join
-> will match the every row form both the tables and give the all Datas.
=> Union
-> should match the no of column in the both the table
-> also have the same datatypes ini the column
=> OuterJoin
-> outer join will join all the data from the both the tables.
-> even if not having the match value.

1. DELETE

- Purpose: Removes rows from a table based on a condition.
- Transaction Support: Can be rolled back (uses COMMIT and ROLLBACK).
- Where Clause: Supports a WHERE clause to delete specific rows.
- Impact on Table: Does not remove the table structure or reset identity columns.

```
-- Delete rows where the age is greater than 30
DELETE FROM Employees
WHERE Age > 30;

-- Delete all rows (without removing the table)
DELETE FROM Employees;
```

2. DROP

- Purpose: Deletes the entire table or database, including its structure and data.
- Irreversible: Cannot be rolled back once executed.
- Impact on Table: Completely removes the table from the database.

```
-- Drop the Employees table
DROP TABLE Employees;

-- Drop the entire database
DROP DATABASE CompanyDB;

-- Drop the Employees table
DROP TABLE Employees;
```

```
-- Drop the entire database
DROP DATABASE CompanyDB;
```

3. TRUNCATE

- Purpose: Removes all rows from a table but retains the structure for future use.
- Faster than DELETE: Operates at the table level (non-logged operation).
- Irreversible: Cannot use ROLLBACK after execution.
- Resets Identity Columns: Resets any auto-increment columns back to their initial value.

```
-- Remove all rows from the Employees table
TRUNCATE TABLE Employees;
```

Feature	DELETE	DROP	TRUNCATE
Operation	Deletes specific rows	Deletes the entire table	Deletes all rows
WHERE Clause	Yes	No	No
Transaction Support	Yes (can rollback)	No	No
Resets Identity	No	Not applicable	Yes
Removes Structure	No	Yes	No

=> Second largest value in the table;

```
SELECT MAX (column_name)
FROM table_name
WHERE column_name NOT IN (SELECT Max (column_name)
                         FROM table_name);
```

Or

```
select rate from job
order by rate desc
offset 1
fetch first row only;
```

=> Lazy loading & Eager Loading
-> lazy loading will fetch the date from child class, only when it explicitly called. Improve the performance of application.
-> Eager loading it will call all the data from child class also all times

=> CaseCadeTypes
-> PERSIST(save),REMOVE,MERGE(updated),REFRESH,DETACH,ALL

=> Advantage of Hibernate
-> ORM (object-relational-Mapping)
-> Automatic table creation
-> Database independent
-> Lazy and Eager Loading

====================
Data -Structure
====================

=> DS
-> Make the code more efficient with less memory and less storage
-> Easy to data manipulation (insert, delete, sort, search)
-> optimised search and retrieval precess

=> Time complexity
-> measure how the running time of an algorithm increases with size of the input data.
-> algorithm for large dataset with less time and less memory and make the code more efficient.
=>Big O notation
-> O(1)
-> O(log n) Binary search
-> O(n) Linear search
-> O(n log n)
-> O(n^2)
-> O(2^n)
-> O(n!)

=> Linear Search
-> Linear search will search the elements by one by one.
-> Time Complexity:O(n).
=> Binary Search
-> Binary search will search the element by diving the array into two part and it will search now.
-> Time Complexity : O(log n).

=> Bubble Sort
-> swap occurs for every iteration until reach its correct position.
-> Time complexity : O(n^2).

=> selection sort

-> swap occurs only once for its position.
-> swap iteration is reduced here when compared to bubble sort , here we are taking the max/min value and swap it max position.
-> Time complexity : $O(n^2)$.

=> Insertion Sort
-> here we are going to shift the element instead of swapping the element.
-> array[j+1] = array[j];   which shift left or right ?
-> Time Complexity: $O(n^2)$

=> Quick Sort.
-> Dived and conquer
-> Recursion will take place here: when u call the same function call by itself.
  -> Pivot:will be the centre place/element but in the code it should in correct position of an array.EX: int[] array = {8,2,4,1,6,3} here 2 is the pivot because it was in correct position.
-> Partition will also done in this sorting.
-> Time complexity :Best & Average case O(n log n)
-> In worst case : $O(n^2)$.