**Java Interview Questions**

1. **OOPS Concept**OOPs Concept circulates around classes, Objects and 4 pillars of OOPs

* Classes:- User Defined Blue print from which Object are created.
* Objects:- They are instances of classes, They represent real life entities.
* Abstraction:- It is a process, which displays only the information needed and hides the unnecessary information. We use Abstract classes and Interfaces for abstraction.
* Encapsulation:- It is defined as the wrapping up of data under a single unit. It bind together code and the data it manipulates.
* Inheritance:- It is a properties in which one object inherits properties another object.
* Polymorphism:- Polymorphism Means one name but many form. It is of 2 types:-
* *Overloading* :- Happens when there exists many functions with same name but different parameters
* *Overriding* :- Happens in case of inheritance.

1. **Interface**

Interfaces helps us to define the behaviour of the class. In other words, it tells a class must do and not how.

1. **Abstract Class**  
   These are the classes that can not be instantiated. they are similar to interfaces that can not instantiated.
2. **Abstract class vs Interface**

|  |  |  |
| --- | --- | --- |
|  | Abstract Class | Interface |
| Type Of Method | Abstract & Non Abstract methods. | Only abstract methods.  default and static methods(Java8) |
| Type Of Variable | Final, non final, static and non static variables. | Static and final variables |
| Accessibility of data member | Private, public and default | Public by default. |
| Multiple Inheritance | Does not supports | Supports |
| Multiple Implementation | Can extend one or more interfaces only. | Can extend only one java class and one or more interface. |

1. **What are the scenerios in which we should use interfaces instead of abstract classes and vice versa?**

Partial method implementation (here abstract classes should be preferred)  
Here deposit and viewDetails are partially implemented in abstract class and does need to implemented in child classes as there is common functionality. But withdraw is subject to change based on Account type hence implementation is required in every sub class.  
This behaviour is called partial implementation.  
Other Example given at https://www.youtube.com/shorts/WfjjBCBG9qU

abstract class Account {

protected String accountNumber;

protected double balance;

public Account() {}

public Account(String accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;}

abstract void withdraw(double amount);

void deposit(double amount) {balance+=amount;}

void viewDetails(){

System.out.println("Account "+accountNumber+ " has balance of "+ balance);

}

}

class SavingsAccount extends Account{

SavingsAccount(String accountNumber,double balance) {

super(accountNumber, balance);

}

@Override

void withdraw(double amount) {

if(balance>amount)balance -= amount;

else System.out.println("Insufficient balance");

}

}

class OverDraftAccount extends Account {

double overdraftValue;

public OverDraftAccount(double overdraftValue, double balance, String accountNumber) {

super(accountNumber, balance);

this.overdraftValue = overdraftValue;

}

@Override

void withdraw(double amount) {

if(balance+overdraftValue>amount)balance -= amount;

else System.out.println("Insufficient Balance");

}

}

**When Multiple inheritance is required then use Interface.**

Here Duck can fly and swim so it need both properties. Hence both interfaces need to be inherited i.e. Flyable and Swimmable (multiple inheritence)

Pigeon can only Fly hence Flyable

Penguin can only swim hence Swimmable

interface Flyable {void fly();}

interface Swimmable {void swim();}

class Duck implements Flyable, Swimmable {

public void fly() {

// Duck-specific flying behavior

}

public void swim() {

// Duck-specific swimming behavior

}

}

Class Pigeon implements Flyable {

public void fly() { }  
}

Class Penguin implements Swimable {  
 public void swim(){ }  
}

1. **Why do we have constructor for Abstract class even if we can't instantiate it**

**Initialization of Fields**: Abstract classes can define fields (instance variables), and constructors are essential for initializing these fields. When you create concrete subclasses that extend the abstract class, these constructors ensure that the fields in the abstract class are properly initialized when objects of the concrete subclasses are created.

**Code Reuse**: Constructors in abstract classes allow for code reuse. Common initialization logic can be placed in the abstract class's constructor, which is shared by all subclasses. Concrete subclasses can invoke this constructor using super(...) to reuse the initialization logic, reducing code duplication.

1. **Methods Possessed by Object class in java.**

toString() - returns the String representation of the object.

clone() - return exact replica of an object.

equals(), finalize(), etc.

1. **JDK, JRE,JVM**

JDK is S/W development environment used to develop applications.

JRE is implementation of JVM. It is set of S/W tools used to develop java applications.

JVM

1. **Access Specifiers.**

* Public : Variables, method and classes declared as public can be accessed by any class or method.
* Protected : Variables, method and classes declared as protected can be accessed with in that package and inherited class.
* Private : Variables, method and classes declared as private can be accessed with in that class.
* Default : Variables, method and classes declared as default can be accessed with in that package.

1. **Aggregation v/s Composition**

* *Aggregation* represents weak relationship. Also called Is-A relationship. We can Implement a aggregation using inheritance.
* *Composition* represents strong relationship. Also Called Has-A relationship. We can implement composition using "new" keyword. Hint : C S Has “new”keyword

For Example

Pulsar is a bike.

Bike has a engine.

1. Multi Threading. Why Multi Threading?

Execute multiple threads independently at the same time.

Why Multithreading? -> Efficient use of CPU.

1. Thread & Process.

* *Thread*:- Light weight and smallest unit of process.
* *Process*:- Program in execution is called Process.

1. Ways of implementing Multi Threading.

* Extending Thread Class
* Implementing Runnable Interface.

Implementing Runnable interface is better way.

class MyThread extends Thread {  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 System.out.println("Thread " + Thread.currentThread().getId() + ": Count " + i);  
 }  
 }  
}

public class ThreadExample {  
 public static void main(String[] args) {  
 MyThread thread1 = new MyThread();  
 MyThread thread2 = new MyThread();

// Start the threads  
 thread1.start();  
 thread2.start();  
 }  
}

class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 System.out.println("Thread " + Thread.currentThread().getId() + ": Count " + i);  
 }  
 }  
 }

public class RunnableExample {  
 public static void main(String[] args) {  
 // Create instances of the MyRunnable class  
 MyRunnable runnable1 = new MyRunnable();  
 MyRunnable runnable2 = new MyRunnable();  
 // Create threads and associate them with the runnables  
 Thread thread1 = new Thread(runnable1);  
 Thread thread2 = new Thread(runnable2);  
 // Start the threads  
 thread1.start();  
 thread2.start();  
 }  
}

1. User Thread & Daemon Thread

* *User Thread*:- User threads have a specific life cycle and its life is independent of any other thread. JVM waits for user threads to complete its tasks before terminating it. When user threads are finished, JVM terminates the whole program along with associated daemon threads.
* *Daemon Thread*:- daemon threads provides services and support to user threads. There are two methods available in thread class for daemon thread: setDaemon() and isDaemon().

We can create daemon threads in java using the thread class setDaemon(true). We should set a thread as Daemon Thread before thread exeution starts.

isDaemon() method is generally used to check whether the current thread is daemon or not.

1. Life Cycle Method of Thread.

a. New

b. Runnable

c. Blocked/Waiting

d. Timed Waiting.

e. Terminated State.

1. Different Methods used in Multi-threading.

* start() -> begin the execution of a newly created thread. Thread can call the start() method only once.
* run() -> begin the execution of the same thread. Thread can call the run() method multiple times.
* wait() -> tells the calling thread (a.k.a Current Thread) to wait until another thread invoke’s the notify() or notifyAll() method for this object. Here, Thread does loses its ownership of the resources and resume’s it’s execution only when resources are available after notify or notifyAll() is called. Belongs to Object Class.
* sleep() -> used to pause the execution of current thread for a specified time in Milliseconds. Here, Thread does not lose its ownership of the resources and resume’s it’s execution. Belongs to Thread class.
* Notify() -> Wakes up only one thread that is in waiting state.
* NotifyAll() -> Wakes up all threads that are in waiting state.

1. Synchronization

Two Types of Synchronization

a. Process Synchronization

b. Thread Synchronization

i. Mutual Exclusion: Can be achieved using Synchronized Method, Synchronized Block, Static Synchronization

ii. Cooperation/Inter-thread communication

1. Synchronized Method, Synchronized Block & Static Synchronization

* *Synchronized Method*:- In this method, the thread acquires a lock on the object when they enter the synchronized method and releases the lock either normally or by throwing an exception when they leave the method. No other thread can use the whole method unless and until the current thread finishes its execution and release the lock. It can be used when one wants to lock on the entire functionality of a particular method.
* *Synchronized Block*:- In this method, the thread acquires a lock on the object between parentheses after the synchronized keyword, and releases the lock when they leave the block. No other thread can acquire a lock on the locked object unless and until the synchronized block exists. It can be used when one wants to keep other parts of the programs accessible to other threads.
* *Static Synchronization*:- Synchronized blocks should be preferred more as it boosts the performance of a particular program. It only locks a certain part of the program (critical section) rather than the entire method and therefore leads to less contention.

1. Exceptions v/s Error.

* Exception: - Abnormal condition that disrupts the normal flow of the program.
* Error: - Error is caused by the environment in which the JVM is running, exceptions are caused by the program itself.

1. Checked & Unchecked Exception

* *Checked Exception*: - Also called as Compile time Exeption. e.g. ClassNotFoundException, SQLException, IOException, etc
* *Unchecked Exception*: - Also called as Run time exception. e.g. NumberFormatException, NullPointerException, ClassCastException, ArrayIndexOutOfBoundException, StackOverflowError etc.

1. How to create a custom exception and how to throw it?

**Creating a Custom Exception**:

To create a custom exception, you need to define a new class that extends either Exception or one of its subclasses (e.g., RuntimeException). You can add your own fields and methods to the custom exception class as needed. Here's an example:

public class CustomException extends Exception {  
 public CustomException(String message) { super(message); }  
}

In this example, we've created a custom exception named CustomException that extends the base Exception class. It accepts a message as a constructor parameter, which is passed to the superclass constructor using super(message).

**Throwing a Custom Exception**:

To throw your custom exception, you can use the throw statement within your code. For example:

public class CustomExceptionExample {

public static void main(String[] args) {

try {

// Simulate a condition where the custom exception is thrown

throw new CustomException("This is a custom exception.");

} catch (CustomException e) {

System.err.println("Custom Exception Caught: " + e.getMessage());

}

}

}

In this example, we use the throw statement to throw an instance of our custom exception (CustomException) within a try block. If the condition is met, the custom exception is thrown. We then catch and handle the custom exception in the catch block, where we can access the exception's message using e.getMessage().

1. Differentiate between throw, throws and throwable.

* *Throw* : ‘throw’ is used to throw an exception manually in Java. Using this keyword, it is possible to throw an exception from any method or block.
* *Throws* : ‘throws’ is used in the method signature in Java. If the method is capable of throwing exceptions, it is indicated by this method.
* *Throwable* : It is super class for all types of errors and exceptions in Java. In case customized exceptions are created, they should extend this class too.

1. Iterator vs listIterator.

|  |  |
| --- | --- |
| Iterator | List Iterator |
| Traverse elements only in the forward direction | Traverse elements both in forward and backward directions |
| Traverse Map, List and Set | Traverse List and not the other two. |
| Indexes cannot be obtained | It has methods like nextIndex() and previousIndex() to obtain indexes |
| Cannot modify or replace elements. | Can modify or replace elements with the help of set(E e) |
| Cannot add elements and it throws ConcurrentModificationException. | Can easily add elements to a collection at any time |
| Iterator itr = collection.iterator();  while(itr.hasNext()) { // add your code here.} | ListIterator ltr = list.listIterator();  while (ltr.hasNext()) {// add your code here.} |

String V/S String Buffer

|  |  |
| --- | --- |
| String | String Buffer |
| String is immutable | StringBuffer is mutable |
| String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when we concatenate strings. |
| String uses String constant pool. | StringBuffer uses Heap memory |
| To concat 2 Strings we use .concat method. | To concat 2 StringBuffers we need .append method. |

String Buffer v/s String Builder

|  |  |
| --- | --- |
| String Buffer | String Builder |
| StringBuffer is synchronized i.e. thread safe | StringBuilder is non-synchronized i.e. not thread safe. |
| Less Efficient | More Efficient |
| Introduced in java1.0 | Introduced in java 1.5 |

String Tokenizer-> String Tokenizer class allows you to break a String into tokens.It is a legacy class of Java.

1. Garbage Collection

Process of reclaiming runtime unused memory automatically is called Garbage Collection. It is automatically done by JVM.

1. Difference Between Comparable and Comparator Interface.

|  |  |  |
| --- | --- | --- |
|  | Comparable | Comparator |
| Method used | compareTo() | compare() |
| Package | Java.lang | Java.util |
| Sorting Sequence | Provides Single Sorting Sequence | Provides multiple Sorting Sequence |
| Sorting order | Sort data according to natural sorting order. | Sort data according to customized sorting order |
| Logic of sorting | Logic of sorting must be in same class | Logic of sorting should be in different class |
| Implemented By | Implemented By Calendar, Date & String | Implemented to sort instances of third party instances. |

1. What is difference between Method Overloading and Method Overriding?

|  |  |  |
| --- | --- | --- |
|  | Method Overloading | Method Overriding |
| When it happens? | Compile time. Hence called compile time polymorphism | Runtime. Hence called runtime polymorphism. |
| Where it happens? | In same class or within sub class. | Only in sub class. |

1. Can we Overload Main method?  
   Yes. But JVM will always call the main method with below signature.  
   public static void main(String [] args)
2. What is public static void main(String [] args)?

* public : This access specifier indicates that main method can be called from anywhere.
* static: No object is required to call the main method.
* void : Main method does not return any value.
* String[] args : arguments passed while running the program.

1. Can we change the order of public, static and void keywords?

Yes, but void should come just before method name. static public void and public static void are acceptable.

1. Can we start a thread twice?  
   No. We cannot do that. It may give illlegalThreadStateException
2. What is Serialization in java?  
   Serialization is the process of converting a java object into a format (ByteStream, Json or XML) that can be easily stored, transmitted while deserialization is the reverse process, where the serialized data is reconstructed into an object that can be used and manipulated in a program.  
   Serialization only happens if we have default constructor.
3. How can we Serialize a Java Object and how to deserialize it?
4. Suppose we have 5 variables in Object and out of which we want to convert only 3 variables into Byte Stream. How can we achieve this?  
   We can mark a variable as Transient. Via this we ensure that transient variables are not serialized to Byte Class. For example: Passwords can be marked as Transient.
5. How to create Immutable class in java? For more detailed answer refer GFG.

An immutable class is a class whose instances, once created, cannot be modified.  
a. Make variables private & final.  
b. Set the value of instance variable only via parameterized constructor.  
c. Provide only getters and no setters are required.

d. Make class final.

1. What are different types of Memory area allocated by JVM?  
   a. Class Area  
   b. Heap  
   c. Stack   
   d. Program Counter Register
2. What is fail-fast and fail-safe in java?  
   While iterating through the collection and if another thread tries to modify the list on which we are iterating, in that case there is concurrent modification exception and it is called **fail-fast**.  
   Fail-Fast happens on Arraylist, Linkedlist  
   While iterating over a collection we are given a copy of collection, so we don’t iterate on original collection, we iterate on copy of collection. By this we our iteration is not hampered. This is called **fail-safe**. Fail Safe happens on concurrent hashmap
3. What is Deserialization?  
   Converting of Byte Stream back to object is called deserialization. After deserialization object is stored back in heap.
4. findFirst() v/s findAny()

**findFirst()** method finds the **first element** in a Stream. So, we use this method when we specifically want the first element from a sequence. It returns an optional.

**findAny()** method allows us to **find any element** from a Stream. We use it when we're looking for an element without paying an attention to the encounter order. It also returns an Optional.

1. Difference between HashMap and concurrent HashMap?

HashMap is not thread safe where as Concurrent HashMap is thread safe.

1. What is ConcurrentModificationException?  
   It occurs when we try to modify a collection concurrently.   
   For example: While iterating any collection if we try to modify the structure or add or remove elements then it may result in ConcurrentModificationException.
2. In continuation to above question, In concurrentHashMap we will get concurrentModificationException . what if I use HashTable which is also thread safe. Will we get concurrentModificationException?  
   If multiple threads try to accept the hashtable, the thread will acquire a lock on entire table, but in case of concurrentHashMap, if there are 16 segments, thread will be acquired on only one segment and rest of segments are available for changes.
3. Why wait() and notify() method belongs to object and not to thread class?  
   Because locking mechanism belongs to Object class and not Thread class.
4. What does Arrays.stream do in java?

It converts Arrays into sequential Stream

1. What is Marker Interface?  
   An empty interface is called Marker Interface. SerialIzable And Clonable are example of Marker Interface. Concept off Marker Interface is Obsolute. Instead annotations are used

Ex: public interface MarkerInterface{}

1. What is Thread Safety?  
   When multiple threads are working on the same data structure, and the value of data structure is changing, that scenario is not thread-safe and we will get inconsistent results. When a thread is already working on an object and preventing another thread on working on the same object, this process is called Thread-Safety.
2. What are Interceptors in Spring Framework?  
   Interceptors are components that allow us to intercept *and manipulate the request/response* in an application. Interceptors add cross-cutting concerns to our application, such as logging, security checks, authorization, etc.
3. HashTable v/s Concurrent HashMap?

Similarities between HashTable and concurrent hash map

1. **Key-Value Storage**: Both HashTable and ConcurrentHashMap store data in the form of key-value pairs. They allow you to associate a value with a unique key for efficient retrieval.
2. **Hashing**: Both data structures use hashing to distribute keys across an underlying array or buckets. Hash functions are used to map keys to indices in the storage structure.
3. **Fast Lookup**: Both structures provide fast lookup times for retrieving values associated with keys. Hashing allows for constant-time or near-constant-time retrieval in average cases.
4. **Thread-Safe Operations**: While they differ in their level of thread safety, both HashTable and ConcurrentHashMap are designed to be used in multi-threaded environments. HashTable is synchronized, and ConcurrentHashMap provides concurrent operations for thread-safe access.
5. **Java Collections Framework**: Both HashTable and ConcurrentHashMap are part of the Java Collections Framework, making them easily integrable with other collections and compatible with various APIs.

differences between HashTable and ConcurrentHashMap:

1. **Concurrency Level**: HashTable is fully synchronized, meaning that only one thread can access it at a time, which can lead to performance bottlenecks in highly concurrent scenarios. In contrast, ConcurrentHashMap allows multiple threads to read and write concurrently, improving scalability.
2. **Granular Locking**: ConcurrentHashMap uses a technique called "segmentation," where the data is divided into segments, and each segment is guarded by a separate lock. This allows finer-grained locking compared to HashTable's global lock.
3. **Performance**: ConcurrentHashMap is designed to perform better in high-concurrency scenarios due to its fine-grained locking and reduced contention. HashTable's global synchronization can lead to contention and performance issues.
4. **Null Handling**: HashTable doesn't allow null keys or values, while ConcurrentHashMap allows null keys and values.
5. **Iterators**: Iterating over HashTable using Enumeration doesn't throw ConcurrentModificationException, while ConcurrentHashMap's iterators are designed to be fail-safe.
6. I have an interface in java.  
   In that, I have provided 10 methods.  
   Do I need to implement all 10 methods if any class implements that interface?

If we have an interface with 10 methods, any class that implements this interface must provide concrete implementations for all 10 of those methods. Failure to do so will result in a compilation error.

1. I have an abstract class in java.

In that class I have 5 abstract methods and 5 non abstract methods.

Do I need to provide implementation of all abstract and non abstract methods in class that extends abstract class?

In Java, when you have an abstract class that contains a combination of abstract methods and non-abstract methods, the rules for providing implementations in classes that extend the abstract class are as follows:

Abstract Methods: Any class that extends the abstract class must provide concrete (non-abstract) implementations for all the abstract methods declared in the abstract class. Failure to do so will result in a compilation error.

Non-Abstract (Concrete) Methods: Classes that extend the abstract class are not required to override or re-implement non-abstract (concrete) methods from the abstract class. However, they can choose to override them if they want to customize the behavior defined in the abstract class. If they don't override these methods, they will inherit the implementations provided by the abstract class.

1. Java New Features from Java 8 to Java 17.

https://reflectoring.io/java-release-notes/

1. What is SOAP & REST. How it is different from REST?

SOAP stand for Simple Object Access Protocol. It is a messaging protocol for exchanging structured information in the implementation of web services.

|  |  |  |
| --- | --- | --- |
|  | SOAP | REST |
|  | It is a Protocol | It is Architectural Style |
| Data Format | XML | XML, HTML, JSON |
| Security | SSL & WS-Security | SSL & HTTPS |
| BandWidth | Consumes High Bandwidth | Consumes Less Bandwidth |
| Caching | Can not be cached | Can be Cached |
| Payload handling | Has strict communication contract and need knowledge of everything before any interaction | Need no knowledge of API |
| ACID Compliance | Has built in ACID compliance to reduce anomalies | Lacks ACID Compliance |

1. What are atomic variables in java?

Atomic variables are typically used in multi-threaded applications to manage shared data without the need for explicit synchronization using locks.

1. How to create enum in java?

Example1

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

}

Example 2

enum Day { SUNDAY("Sun"), MONDAY("Mon"), TUESDAY("Tue"), WEDNESDAY("Wed"),

THURSDAY("Thu"), FRIDAY("Fri"), SATURDAY("Sat");

private final String abbreviation;

// Constructor for enum constants

private Day(String abbreviation) { this.abbreviation = abbreviation; }

public String getAbbreviation() { return abbreviation; }

}

public class EnumConstructorExample {

public static void main(String[] args) {

Day sunday = Day.SUNDAY;

System.out.println("Enum Constant: " + sunday);

System.out.println("Abbreviation: " + sunday.getAbbreviation());

}

}

**Spring Boot**

1. **Inversion Of Control**: Giving control of Object Creation to Spring so that Object can be created and injected into another class. For Example, Instead of creating Address object using new keyword Spring IOC Container creates object of address type and injects into student object. So here we can see that the control is inverted from you creating the object to spring creating the object.
2. **Dependency Injection**: It is a technique in which an object receives other object that it depends on.  
   For example: In a Student class we have a dependency on address, so whenever we try to create object of Student class, the address should be created and should be injected in student object.  
   There are 2 ways in which a dependency can be injected.  
   a. *Setter Injection (use setter method for a field)*  
   b. *Constructor Injection (use Constructor to set values for a field)*
3. **Spring IOC Container**:

Spring IOC container performs:-

1. It creates the object.
2. Hold the Object in memory.
3. And Inject them in another object (Dependency Injection)

It must know following things before performing its task.  
a. Beans or POJO classes it has to manage.  
b. Configuration Files. Generally XML Configurations. In configuration files we tell that which bean is dependent on other things.

1. **Application Context:**

* It is an interface which represents Spring IoC Container.
* In Application Context Spring beans are defined & managed.
* It is responsible for reading configurations and creating beans based on these configurations.
* It also implements BeanFactory.

Some of the important subclasses which implements this Interface are :-

1. **ClasspathXMLApplicationContext** : loads XML based bean definitions from one or more XML configuration files located on classpath.(loads bean definition from class path)
2. **FileSystemXMLApplicationContext** : loads XML based bean definitions from one or more XML configuration files located in file system. (loads bean definition from file system)
3. **AnnotationConfigApplicationContext** : loads bean definition from one or more configuration classes which are annotated with Spring’s @Configuration annotations.



1. **Bean Factory v/s application Context**

|  |  |  |
| --- | --- | --- |
|  | Bean Factory | Application Context |
| Type Of Applications | Suitable to build stand alone applications | Suitable to build web applications integrated with AOP & ORM. |
| Functionality | Fundamental container that provides basic functionality | Advanced container that extends Bean Factory that provides basic + advanced functionality. |
| Support for Annotation | Does not support annotation. In Bean Autowiring, we need to configure the properties in XML file only. | It supports Annotation based configuration in Bean Autowiring. |
| Memory | Requires less memory as it provides basic features | Requires more memory. |

Bean Factory is deprecated from Spring 3.0.

1. **LifeCycle Of Bean**

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the init() method and the destroy() method.

1. public void **init()** : used for initializing code, Loading config, connecting db etc.
2. public void **destroy() :** used for writing code clean up.

  
Configuration Techniques for init & destroy method  
a. XML ( “init-method” attribute & “destroy-method” attribute)  
b. Spring Interface(“Initialization Bean Interface” & “Disposable Bean Interface”)  
c. Annotation(@PostConstruct & @PreDestroy) -> @PostConstruct will provide init functionality & @PreDestroy will provide destroy functionality.

1. **What is class path and what it is used for?**   
   CLASSPATH defines the path, to find third-party and user-defined classes that are not part of Java platform.
2. **How to handle response timeout while calling an API. ReadTimeOut vs ServerTimeOut.***Server Time Out* :- Happens when we are not able to establish a connection with server with in a required time period.  
   *Read Time Out*:- Happens when we are able to establish a connection with server, but reading data takes a lot of time or we are not able to read the data.
3. **What is versioning in Rest? What are the ways that we can use to implement versioning?**Versions help us to identify the old version and new version in Rest API. We can implement versioning using multiple ways for Rest API.
4. Using request parameters
5. Using headers
6. Using URL/URI
7. Using produces.

@EnableAutoConfiguration

1. Enables the auto configurations of Application Context. Therefore it automatically configures specific class or bean from classpath and the beans which are defined by us. Use ‘exclude’ property to disable autoconfiguration of specific class

@SpringBootConfiguration

1. It is class level annotation
2. It indicates that class has bean definition methods.

@ComponentScan

1. Searches/Scans the provided package for classes annotated with such as @Controller, @Service & @Repository or @Component

**POJO v/s Java Bean v/s Spring Bean**

**POJO** = Plain Old Java Object. Object of any class is POJO. E.g. ABC abc = new ABC();  
Here ‘abc’ is POJO.

**Java Bean** = Also called Enterprise Java Bean. EJB is a POJO with few restrictions.

1. Restriction 1: Class should have default/ no-arg constructor.
2. Restriction 2: Every variable must have a getter & setter.
3. Restriction 3: The class must implement serializable interface.

**Spring Bean** = Any POJO maintained by Spring IOC container is called Spring Bean or Simply Bean. Application Context is implementation of Spring IOC Container. (All code of Application Context is written inside Application Context, its sub-classes and its sub-interfaces.)

***@Configuration*** *= Indicates that a class has some methods in which beans are defined a.k.a Bean Definition Methods. So Spring Container can process the class & generate Spring Beans to be used in the application.*

**@BEAN**

It tells that method will produce a Bean. It is used in conjunction with @Configuration. E.g.

public class MyCustomBean{

@override  
 public String toString() {return “MyCustomBean”;}

}

@Configuration   
public class MyApp {

@Bean  
 public MyCustomBean getCustomBean() {return new MyCustomBean();}

}  
Use case for @Bean : For writing custom business logic or for instantiating bean for 3rd party.

**@Primary v/s @Qualifier**

**@Qualifier** -> Autowire a specific bean among same type of Bean.

**@Primary ->** Used to give high preference to the specific beanamong multiple beans of same type to inject to a bean.

*@Qualifier has higher priority than @Primary*

@Component @Primary  
class QuickSort implements SortingAlogorithm {}

@Component  
class BubbleSort implements SortingAlgorithm {}

@Component @Qualifier(“RadixSortQualifier”)  
class RadixSort implements SortingAlgorithm {}

Just give me preferred algorithm (marked with @Primary)

@Component  
class ComplexAlgorithm   
@Autowired  
private SortingAlgorithm algorithm;

@Component  
class AnotherComplexAlgorithm  
@Autowired @Qualifier(“RadixSortQualifier”)  
private SortingAlgorithm iWantToUseRadixSortOnly;

**Internal Working of Hashmap**

A HashMap in Java is an implementation of a hash table data structure. It maps keys to values, allowing for efficient lookups based on the keys.

The internal working of a HashMap in Java can be understood by understanding the following components:

**Hash Function**: The hash function takes a key as input and returns an integer (the hash code) that is used to determine the index of the bucket in which the key-value pair will be stored. The hash code is calculated using the key's hashCode() method.

**Buckets**: The HashMap stores key-value pairs in "buckets". Each bucket is an array of Entry objects that store a key-value pair.

**Hash Code Collision**: If two keys have the same hash code, they are said to have collided. The HashMap handles collisions by chaining the entries in the same bucket using a linked list.

**Resizing**: The HashMap is implemented as an array of buckets, with the size of the array being a power of 2. The HashMap can be resized dynamically when the number of key-value pairs stored in the HashMap exceeds a certain load factor.

**get() and put() operations**: The get() operation takes a key as input and uses the hash function to calculate the hash code and find the index of the bucket. The linked list in the bucket is then searched for the key-value pair, and the value is returned if found. The put() operation is similar, but it also adds a new key-value pair to the HashMap if the key is not already present.

Overall, the HashMap in Java provides an efficient and flexible implementation of a hash table data structure, with the ability to handle hash code collisions and dynamic resizing.

Here's how the HashMap works in detail:

**Calculating the Hash Code**: When you put a key into the HashMap, the HashMap calculates the hash code of the key using the hashCode() method. The hash code is used to determine the index in the array where the value should be stored.

**Compression Function**: The hash code of the key is then passed through a compression function to determine the index in the array where the value should be stored. The compression function takes the hash code and computes a remainder when it's divided by the size of the array. This remainder is the index where the value should be stored.

**Handling Collisions**: In the unlikely event that two keys have the same hash code and therefore the same index in the array, a collision occurs. To handle collisions, the HashMap stores linked lists at each index in the array. When a collision occurs, the value is stored in the linked list at that index.

**Retrieving Values**: To retrieve a value from the HashMap, you provide the key. The HashMap calculates the hash code of the key and passes it through the compression function to determine the index in the array where the value is stored. It then searches the linked list at that index to find the value associated with the key.

In summary, the HashMap in Java uses a hash table to store key-value pairs, where the hash code of the key is used to determine the index in the array where the value should be stored. The compression function and linked lists are used to handle collisions and ensure efficient retrieval of values.

**@ExceptionHandler & @ControllerAdvice**

@ExceptionHandler is method level annotation which is used to define a method that should be called to handle exceptions that are thrown in our application. When an exception is thrown, Spring Boot will look for *@ExceptionHandler*-annotated method that is capable of handling the exception type that was thrown. If it finds such a method, it will invoke that method to handle the exception.

@ControllerAdvice is class level annotation that defines a single class to handle exceptions globally across all controllers in our application thus acting as central exception handler for an entire application. We can use this annotation to provide a consistent way of handling exceptions and to write code that is clean, concise and reusable.

For example, you can define a class that extends ResponseEntityExceptionHandler and annotate it with @ControllerAdvice to handle exceptions in your controllers. In this class, you can define methods that will handle specific exceptions and return a custom response.

@ControllerAdvice  
public class CustomExceptionHandler extends ResponseEntityExceptionHandler {

@ExceptionHandler(value = {Exception.class})  
 public ResponseEntity<Object> handleAnyException(Exception ex, WebRequest request) {  
 // return custom response  
 }

@ExceptionHandler(value = {CustomException.class})  
 public ResponseEntity<Object> handleCustomException(CustomException ex, WebRequest request) {  
 // return custom response  
 }  
 }

**Actuators in Spring Boot**

1. Actuator is a *set of tools for* ***monitoring******and managing*** *our application*.
2. These tools *provide insights into the internal operations* of your application, such as health status, metrics, and various other details.
3. Actuators allow you to *monitor the health* of your application and *diagnose any issues* that may arise in production.

Spring Boot Actuators expose a number of endpoints, such as: (HIT DM)

1. /health: Provides information about the health of your application
2. /metrics: Provides metrics about the performance of your application
3. /info: Provides information about the configuration of your application
4. /trace: Provides a trace of the last few HTTP requests handled by your application
5. /dump: Provides a dump of thread information for all threads in the JVM

We can also create custom endpoints to expose specific information about your application. To enable Actuators in Spring Boot, we need to add the *spring-boot-starter-actuator* dependency to our project. By default, the endpoints are secured and can only be accessed with a specific set of credentials. We can configure the security settings to suit our needs.

**Java 8 Features** (LSDFOM)

a. Lambda Expressions: - similar to methods, but they do not need a name.

b. Stream API

c. Default Methods: - these are methods which are defined in interface and tagged as default. The implementation can be changed in child classes.

d. Static Methods: - these are methods which are defined in interface and keyword static. The implementation can't be changed in child classes.

e. Functional Interface

f. Optional Class

g. Method Referencing

**Functional Interfaces in Java**

They are the interfaces which have only one abstract method.   
It can have any number of static and default method.   
Example of functional Interface is Runnable, Comparable, Predicate, Consumer, Function, Supplier, etc.  
Functional interfaces were introduced in Java 8 in order to implement lambda expressions.

**We can create our own functional interface.**

To create our own functional interface, we need to follow the following steps:-

a. Create an interface.  
b. Annotate it with @FunctionalInterface  
c. define only one abstract method. (Abstract method does not have a body)  
d. (Optional) Create any static or default methods.

Example of Functional Interface:-

@FunctionalInterface  
public interface FunctionalInterface {

void singleAbstractMethod(); // Abstract Method  
 default void defaultMethod(){  
 System.out.println("This is default Method"); // This is optional

}

}

**Method Referencing In Java 8**

It is a special type of lambda expression in which we reference existing methods. It uses ::(double colon)

Types Of method reference

1. Static Method Reference.  
    If a lambda expression just call a static method of a class

(args) -> Class.staticMethod(args)

Shorthand -> if a lambda expression just call a static method of a class

Class::staticMethod

1. Instance Method Reference of a particular object

If a lambda expression just call a default method of an object

(args) -> obj.instanceMethod(args)

Shorthand -> if a lambda expression just call a default method of an object

obj::instanceMethod.

1. Instance Method Reference of an arbitrary object of a particular type.
2. Constructor Reference.

**Predicates**

It is pre-defined functional interface.  
Abstract Method -> public Boolean **test**(T t); Return type is always Boolean;  
Predicate Joining:- It is used when we want to combine multiple predicates to find single results.  
There are 3 ways to join a predicate.

1. And:- Predicate1.and(Predicate2).test("the value which we want to test")
2. Or:- Predicate1.or(Predicate2).test("the value which we want to test")
3. Negate:- Predicate1.negate().test("the value which we want to test")

**Functions**It is pre-defined functional interface.  
Abstract Method -> **apply**(T t);  
Working of functions is such a way that "If we give some input & perform some operation, it will gives us the output. This output can be anything (not necessarily a Boolean value)".  
In Functions return type is not fixed. Hence we have to declare both input type and return type.  
Just like predicate joining, we have functional chaining. We can combine / chain multiple functions together with “andThen” and “compose”.  
 f1.andThen(f2).apply(Input); - first f1 then f2 // from left to right  
 f1.compose(f2).apply(Input) - first f2 then f1 // from right to left

Multiple functions can be chained together like :

f1.andThen(f2).andThen(f3).andThen(f4).apply(Inputs);

**Consumer**

It is predefined Functional Interface which never return anything (never supply), they just consume.

Abstract method: - **accept**.  
Example :- c1.andThen(c2).accept (Input); - first c1 then c2.

Multiple consumers can be chained together like: c1.andThen(c2).andThen(c3).andThen(c4).accept(Inputs);

**Supplier**

It is functional Interface which does not take any input but it will always supply an object.  
There is no chaining as no input is given. It is just opposite of Consumer.

Abstract method: - **get(T t);**

Example: current date.

**Stream API**

It is special Iterator class that allow processing of collection of Objects in a functional manner.

In simple words: Stream is sequence of Objects upon which we perform variety of operations to get desired output. Stream API is used to process these collections of Objects.

There are 2 types of streams:-

1. Parallel Stream(runs on multiple core & multiple thread; O/P is unpredictable; High Performance
2. Sequential Streams(Runs on single core & one thread; O/P is predictable; Poor performance)

**Optional in java 8**

Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.  
Some important methods in Optional Class are

a. *isPresent()* - Return true if value is present in the Optional, otherwise false.

b. *get()* - If a value is present in the Optional, returns the value, otherwise throws NoSuchElementException.

c. *ofNullable(T value)* - if non-null, Returns an Optional describing the specified value otherwise returns an empty Optional.

.collect(Collectors.toList())

.collect(Collectors.toSet())

.collect(Collectors.toCollection(List::new))

.collect(Collectors.toMap(City::getName, City::getTemperature,(key, identicalKey) -> key))

.collect(Collectors.groupingBy(City::getName, Collectors.collectingAndThen(Collectors.counting(),f -> f.intValue())))

.collect(Collectors.groupingBy(City::getName, Collectors.counting()))

.collect(Collectors.groupingBy(City::getName))

.collect(Collectors.joining(", "))

.collect(Collectors.groupingBy(City::getName,Collectors.mapping(City::getTemperature,Collectors.toList())))

.collect(Collectors.groupingBy(City::getName,Collectors.mapping(City::getTemperature,Collectors.toSet())))

.collect(Collectors.partitioningBy(city -> city.getTemperature() > 15))

**REST API**

@RestController

@RequestMapping("/college")

public class StudentController {  
@GetMapping("/student")

public Student getStudent() {

return new Student("Shivam", "Khandelwal");

}

}

**Rest API using Query Parameters**

Sample url = <http://localhost:8080/query?firstName=Shivam&lastName=Khandelwal>

@GetMapping("/student/query")

public Student studentQueryParam(

@RequestParam(name = "firstName") String firstName,

@RequestParam(name = "lastName") String lastName) {

return new Student(firstName, lastName);

}

**Rest API using path variables**

Sample url = <http://localhost:8080/student/shivam/khandelwal/>

@GetMapping("/student/{firstName}/{lastName}/")

public Student studentPathVariable(

@PathVariable("firstName") String firstName,

@PathVariable("lastName") String lastName) {

return new Student(firstName, lastName);}

}

**Rest API returning ResponseEntity**

@GetMapping("/getAllUsers")

public ResponseEntity<List<User>> getAllUsers() {

List <User> user;

user = userService.getAllUser();

return new ResponseEntity<>(user,HttpStatus.OK);

}

***Add Post mapping as well.***

**Adding Swagger**

1. Getting Swagger 2 Spring Dependency  
2. Enabling swagger in our code.  
3. Configuring Swagger.  
4. Adding details as annotations to API.

**Getting Swagger 2 Spring Dependency**Add springfox-swagger2 & springfox-swagger-ui maven dependency.

**Enabling swagger in our code.**  
We need to add @EnableSwagger2 annotation in Main class.