**Java Interview Preparation**

**1.What is bean in Spring?**

In Spring, the objects that form the backbone of your application and that are managed by Spring IoC Container are called beans. Beans are instantiated, assembled, and managed by a Spring IoC container.

In Spring Core , beans are defined through **@Configuration** annotated classes. Beans are defined in these classes using **@Bean** annotated methods.

@Configuration

public class AppConfig {

@Bean

public MyBean myBean() {

return new MyBean(dependencyBean());

}

@Bean

public DependencyBean dependencyBean() {

return new DependencyBean();

}

}

**2.What is Inversion of Control in Spring?**

Inversion of Control (IoC) is a design principle that refers to reversal of control flow in a software application. Instead of object itself controlling the instantiation (control flow), the process is delegated to external container like IoC container in spring.

In context of Spring, IoC is implemented using Dependency Injection, whereby objects define their

Dependencies. The IoC container then injects those dependencies when it creates the bean.

**3. What is difference between BeanFactory and ApplicationContext ?**

Both BeanFactory and ApplicationContext are key components of Spring Framework.

**BeanFactory**

* BeanFactory is the core interface for accessing Spring IoC Container.
* It provides fundamental mechanism or managing java objects called beans, which are often referred to as beans.
* It supports basic features of dependency injection and bean lifecycle management.
* BeanFactory loads bean definitions, instantiates beans, and manages their lifecycle based on configuration metadata provided in XML files , java annotations, or Java Code.
* BeanFactory provides lazy loading of beans , meaning beans are instantiated only when requested.

**ApplicationContext**

* ApplicationContext is a sub-interface of BeanFactory.
* ApplicationContext adds more enterprise-specific functionality to BeanFactory such as event propagation, internationalization support , application layer configuration, AOP integration, and more.
* ApplicationContext eagerly loads and pre-instantiates singleton beans upon initialization , providing better performance than lazy loading in Bean Factory.
* It supports different resource loading mechanisms, environment awareness, and message resolution.

**4. Explain LifeCycle of a Bean**

The lifecycle of beans in Spring refers to the various stages a bean goes through from its instantiation to its destruction. Understanding the bean lifecycle is crucial for managing resources, initialization, and cleanup tasks efficiently. The bean lifecycle in Spring typically consists of the following stages:

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 custom **init()** method and the **destroy()** method.

To provide the facility to the created bean to invoke custom **init()** method on the startup of a spring container and to invoke the custom **destroy()** method on closing the container, we need to implement our bean with two interfaces namely **InitializingBean**, **DisposableBean** and will have to override **afterPropertiesSet()** and **destroy()** method. **afterPropertiesSet()** method is invoked as the container starts and the bean is instantiated whereas, the **destroy()** method is invoked just after the container is closed.

**Note:** To invoke destroy method we have to call a **close()** method of ConfigurableApplicationContext.

// Java program to create a bean

// in the spring framework

**package** beans;

**import** org.springframework

    .beans.factory.DisposableBean;

**import** org.springframework

    .beans.factory.InitializingBean;

// HelloWorld class which implements the

// interfaces

**public** **class** HelloWorld

**implements** InitializingBean,

 DisposableBean {

    @Override

    // It is the init() method

    // of our bean and it gets

    // invoked on bean instantiation

**public** **void** afterPropertiesSet()

**throws** Exception

    {

        System.out.println(

            "Bean HelloWorld has been "

            + "instantiated and I'm the "

            + "init() method");

    }

    @Override

    // This method is invoked

    // just after the container

    // is closed

**public** **void** destroy() **throws** Exception

    {

        System.out.println(

            "Container has been closed "

            + "and I'm the destroy() method");

    }

}

* Now, we need to configure the spring XML file **spring.xml** and define the bean.
* XML

|  |
| --- |
| <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"              "<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">    <**beans**>      <**bean** id="hw" class="beans.HelloWorld"/>    </**beans**> |

Finally, we need to create a driver class to run this bean.

// Java program to call the

// bean initialized above

**package** test;

**import** org.springframework

    .context

    .ConfigurableApplicationContext;

**import** org.springframework

    .context.support

    .ClassPathXmlApplicationContext;

**import** beans.HelloWorld;

// Driver class

**public** **class** Client {

**public** **static** **void** main(String[] args)

**throws** Exception

    {

        // Loading the Spring XML configuration

        // file into the spring container and

        // it will create the instance of the bean

        // as it loads into container

        ConfigurableApplicationContext cap

            = **new** ClassPathXmlApplicationContext(

                "resources/spring.xml");

        // It will close the spring container

        // and as a result invokes the

        // destroy() method

        cap.close();

    }

}

**5. Explain different scopes of bean in Spring**

In Java, when you work with the Spring Framework, beans are objects managed by the Spring IoC (Inversion of Control) container. The scope of a bean defines the lifecycle and visibility of that bean within the container. Different scopes allow you to control how and when the container creates and manages instances of beans. Here are some common scopes of beans in Spring:

1. **Singleton Scope:**
   * The singleton scope means that the Spring container creates only one instance of the bean for each container (or application context).
   * All requests for the bean result in the same object instance being returned.
   * This is the default scope in Spring.
2. **Prototype Scope:**
   * The prototype scope means that the Spring container creates a new instance of the bean every time a request for that bean is made (i.e., every time **getBean()** is called).
   * This leads to a new object instance being returned for each request.
3. **Request Scope:**
   * Beans with request scope are created once per HTTP request.
   * This scope is applicable only in the context of a web-aware Spring ApplicationContext.
   * The same bean instance is provided to all request processing threads, but different instances are used for processing different requests.
4. **Session Scope:**
   * Beans with session scope are created once per HTTP session.
   * Like request scope, this is applicable only in the context of a web-aware Spring ApplicationContext.
   * The same bean instance is provided for the entire duration of the user's session.
5. **Application Scope:**
   * Beans with application scope are created once per ServletContext.
   * They are available across all HTTP requests and sessions within the web application.
6. **WebSocket Scope:**
   * Introduced in Spring 4.2, beans with WebSocket scope are created once per WebSocket session.
   * They are available across WebSocket lifecycle.
7. **Custom Scope:**
   * Spring allows developers to define custom scopes to fulfill specific requirements.
   * Developers can implement their own **Scope** interface and define the bean lifecycle as per their needs.

Choosing the appropriate scope for a bean depends on factors such as the lifecycle of the bean, the concurrency requirements, and the context in which the bean will be used.

**6.Explain @SpringBootAnnotation in SpringBoot**

The @SpringBootApplication annotation is a convenience annotation provided by the Spring Boot framework.It is often used to mark the main class of a Spring Boot application.This annotation combines several other annotations in to one :

* **@Configuration :** Indicates that the class contains bean definitions that should be processed by the spring container.
* **@EnableAutoConfiguration :** Enables Spring Boot’s auto-configuration mechanism, which automatically configures the spring application based on dependencies and properties present in the classpath. The feature simplifies the setup of Spring applications by reducing the need for explicit configuration.
* **@ComponentScan :** Instructs Spring to scan the specified packages for components such as controllers , services , repositories etc and automatically register them as Spring beans in the application context.

By using **@SpringBootApplication**, developers can bootstrap a Spring Boot application with minimal configuration. It's a convenient way to set up a Spring application with sensible defaults and auto-configuration while also enabling component scanning and defining additional beans if needed.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

In this example, **MyApplication** is marked as the main class for the Spring Boot application. When executed, **SpringApplication.run()** method bootstraps the application, triggering the auto-configuration, component scanning, and other necessary setup defined by Spring Boot.

**7.Explain AutoConfiguration in SpringBoot**

In Spring Boot, Auto-Configuration is a feature that automatically configures the Spring application based on the classpath dependencies, reducing the need for manual configuration. It leverages convention over configuration principle to set up your application with sensible defaults.

Here's how Auto-Configuration works:

1. **Classpath Scanning**: Spring Boot scans the classpath for certain libraries and frameworks. For example, if it finds Hibernate on the classpath, it assumes that you're using JPA for database access.
2. **Conditional Configuration**: Spring Boot applies conditions to determine whether to apply certain configurations. For instance, it might check if a particular bean is already defined or if a specific property is set.
3. **@Configuration**: When Spring Boot detects that certain conditions are met, it generates **@Configuration** classes that define beans necessary for the detected libraries and frameworks. These configurations are typically exposed as Spring beans and can be injected into your application.
4. **Customization**: Spring Boot allows you to customize or override auto-configured beans by providing your own configurations. You can define your beans in your own configuration classes and Spring Boot will use those instead of the auto-configured ones.

With **spring-boot-starter-web** and **spring-boot-starter-data-jpa** dependencies included:

* Spring Boot detects **spring-boot-starter-web** and sets up a web application context with sensible defaults like embedded Tomcat server, dispatcher servlet, etc.
* It also detects **spring-boot-starter-data-jpa** and assumes you're using JPA for database access. It then auto-configures the **EntityManager**, **TransactionManager**, and other necessary components for JPA.
* You can further customize this auto-configuration by providing your own **@Configuration** classes. For example, if you want to use a different datasource or configure JPA entities differently, you can define your own **DataSource** bean and entity classes, and Spring Boot will use those instead.

1. **What is Actuator in Spring Boot**

Spring Boot Actuator is a set of production-ready features that provides monitoring, management, and operational support for Spring Boot applications. It offers various endpoints and metrics to monitor and manage your application at runtime. Actuator enables you to gather insights into the application's health, metrics, environment, configuration, and more.

1. **Explain Functional Interfaces in java**

In java , functional interface is an interface that contains only one abstract method. Functional interfaces are key concept in Java’s support for functional programming, introduced in Java 8. They enable the use of lambda expressions and method references to represent instances of single method interfaces more concisely and effectively.

Here are key points about functional interfaces in Java:

1.**Single Abstract Method(SAM) Interface** : A functional Interface must contain exactly one abstract method.It can contain multiple default or static methods, but only one abstract method.This single method represents the behaviour encapsulated by the functional interface.

[2.**@FunctionalInterface**](mailto:2.@FunctionalInterface) **Annotation :**

While it's not strictly required, it's a good practice to annotate functional interfaces with **@FunctionalInterface** annotation. This annotation ensures that the interface is intended to be used as a functional interface, and the compiler will produce an error if more than one abstract method is added to it.

**3.Lambda Expressions and Method References**: Functional interfaces are primarily used to represent behaviors as lambda expressions or method references. Instead of creating separate implementations of the interface for each behavior, you can pass lambda expressions or method references that define the behavior directly, making the code more concise and expressive.

**4**. **Built-in Functional Interfaces**: Java provides several built-in functional interfaces in the **java.util.function** package, such as **Predicate**, **Function**, **Consumer**, **Supplier**, etc. These interfaces cover common functional use cases and can be used directly or extended to create custom functional interfaces.

5.**Usage in Streams and APIs** :

Functional Interfaces are used extensively in Java Stream Api and other APIs introduced in Java 8 and later. They allow for a more functional style of programming, where operations like mapping, filtering, and reducing can be performed concisely using lambda expressions.

Example :

@FunctionalInterface

interface Printer{

void printMessage(String message);

}

class Main{

public static void main(String[] args){

Printer printer = (String msg) -> System.out.println(msg);

printer.printMessage(“Hello”);

}

}

1. **Explain Inbuilt Functional Interfaces in Java .**

In java, the **java.util.function** package provides a set of built-in functional interfaces that represent common function types. These interfaces are widely used in stream API and many other cases of functional programming.

Here are few built-in functional interfaces in java :

* **Predicate<T>** : Represents a predicate(Boolean valued function) of one argument. It is commonly used to filter elements in collections or streams

**Example** :

Predicate<Integer> evenChecker = n -> n % 2 == 0;

System.out.println(evenChecker.test(10));

* **Function<T, R>**: Represents a function that accepts one argument and produces a result. It's commonly used for mapping or transforming elements.

**Example** :

**Function<String,Integer> findStringLength = str -> str.length();**

**System.out.println(findStringLength.apply(“Shiva”));**

* **Consumer<T> :** Represents an operation that accepts a single input argument and returns no result. It's often used for consuming or processing elements.

**Example:**

Consumer<String> welcomeGreeter = str -> System.out.println(“Welcome “+str);

System.out.println(welcomeGreeter.accept(“Shiva Teja”));

* **Supplier<T>**: Represents a supplier of results. It has no input arguments and produces a result. It's commonly used for lazy initialization or generating values.

**Example**:

**Supplier<Double> randomDouble = () -> Math.random();**

**System.out.println(randomDouble.get());**

* **UnaryOperator<T>**: Represents an operation on a single operand that produces a result of the same type as its operand. It's often used for unary operations.

**Example**:

UnaryOperator<Integer> square = n -> n \*n;

System.out.prinltn(square.apply(2));

* **BinaryOperator<T> :** Represents an operation on two operands of same type and produces a result of same type.

**Example**

BinaryOperator<Integer,Integer> multipler = (n,m) -> n \*m ;

System.out.println(multipler.apply(2,4));

**11.Explain Stream API in java 8**

In java, the stream API was introduced in java 8 as new abstraction that allows developers to process collections of data in functional and declarative manner. It provides a set of methods to perform operations on sequences of elements, such as filtering, mapping, sorting, and reducing. Streams allow for concise and expressive code, promoting readability and maintainability.

Here's an overview of some key concepts and functionalities of the Stream API:

**1.Stream Creation :** Stream can be created from various data sources such as collections,arrays, or even I/O operations.They can also be generated using static factory methods provided by stream interface.

List<Integer> numbers = Arrays.asList(1,2,3,4,5);

Stream<Integer> stream = numbers.stream();

**2.Intermediate Operations :**

These operations are applied to stream and return another stream. Intermediate operations include filtering, mapping, sorting etc. These operations are lazy, meaning they do not perform any processing until the terminal operation is invoked.

Stream<Integer> filteredStream = numbers.stream().filter(n -> n % 2==0);

Stream<Integer> mappedStream = numbers.stream().map(n-> n+1);

**3.Terminal Operations :**

These operations consume the stream and produce a result. Terminal operations include reduction operations like **reduce**, **collect**, and **forEach**, as well as operations to find elements like **findFirst**, **findAny**, and **count.**

int sum = numbers.stream().reduce((a,b)-> a+b);

List<Integer> evenNumbers = numbers.stream().filter(n -> n%2==0).collect(Collectors.toList());

**4.Parallel Streams :**

Streams can be processed sequentially or parallel.Parallel streams automatically divide the workload across multiple threads, which can lead to improved performance for large datasets.

**Example:**

List<Integer> evenNumbers = numbers.parallelStream().filter(n -> n % 2 ==0).collect(Collectors.toList());

**5.Lazy Evaluation :**

Intermediate operations are lazily evaluated, meaning they are only executed only when a terminal operation is invoked.This allows for efficient processing of large datasets because unnecessary computations are avoided.

List<String> strings = Arrays.asList(“abc”,”efg”,”hij”);

Stream<String> filteredStrings = strings.stream().filter(s -> {

System.out.println(“Filtering” +s);

return s.contains(“a”);

});

filteredStrings.forEach(System.out::println); // without this statement nothing is printed

**12.Explain difference between map and flatMap.**

map and flatMap are higher order functions used in java.

**map** :

The map function takes each element of a collection and applies a function to it, returning a new collection of same size where each element is the result of applying the function to the corresponding element in the original collection.

**flatMap** :

* The flatMap function is similar to map, but it’s used when mapping function returns a collection itself.It then flattens these nested collections in to a single collection.
* It first applies the transformation function to each element of the collection, resulting in a collection of collections.Then it flattens this collection of collections in to a single collection.
* flatMap is often used when you want to perform a one-to-many transformation and then flatten the result.

public class Main {

public static void main(String[] args) {

List<List<Integer>> nestedList = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5, 6),

Arrays.asList(7, 8, 9)

);

// Using flatMap to flatten the nested lists

List<Integer> flattenedList = nestedList.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println(flattenedList); // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

}

}

In this example :

* We have a nested list of integers (nestedList) , where each element is a list of integers.
* We use the flatMap operation to transform each nested list in to a stream of its elements.
* The resulting stream is a single stream containing all the elements from the nested lists.
* Finally, we collect the elements of the flattened stream in to a list.

**13.Explain method reference and constructor reference in java.**

**14.Explain Comparator and Comparable in java**

In Java, ‘Comparator’ and ‘Comparable’ are interfaces used for sorting and ordering objects in collections.They both serve the similar purpose but differ in implementation and usage scenarios.

**1.Comparable**

* **Comparable** is an interface that allows objects to be compared to each other for the purpose of natural ordering.
* It is used to define default sorting behaviour of objects of class.
* The **compareTo** method is implemented in the class whose objects need to be sorted.
* This interface is typically implemented by the class of objects that need to be sorted.

import java.util.\*;

class Student implements Comparable<Student> {

private int id;

private String name;

// Constructor, getters, setters...

@Override

public int compareTo(Student otherStudent) {

// Compare based on student names

return this.name.compareTo(otherStudent.getName());

}

}

public class ComparableExample {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student(101, "Alice"));

students.add(new Student(102, "Bob"));

students.add(new Student(103, "Charlie"));

Collections.sort(students);

for (Student student : students) {

System.out.println(student.getName());

}

}

}

**2.Comparator**

* **Comparator** is an interface that provides a way to sort objects based on different criteria.
* It allows the definition of custom comparison logic separate from the class whose objects are being sorted.
* It is useful when you want to define multiple sorting criteria or when you cannot modify the class of objects being sorted.
* The  **compare** method is used to compare two objects.

Usage:

import java.util.\*;

class Student {

private int id;

private String name;

// Constructor, getters, setters...

// Static inner class implementing Comparator

static class NameComparator implements Comparator<Student> {

@Override

public int compare(Student student1, Student student2) {

// Compare based on student names

return student1.getName().compareTo(student2.getName());

}

}

}

public class ComparatorExample {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student(101, "Alice"));

students.add(new Student(102, "Bob"));

students.add(new Student(103, "Charlie"));

// Create a Comparator object

Student.NameComparator nameComparator = new Student.NameComparator();

// Sort the list using the Comparator

students.sort(nameComparator);

for (Student student : students) {

System.out.println(student.getName());

}

}

}

* For **Comparable**, sorting is done using **Arrays.sort()** or **Collections.sort()**, where the objects themselves define the sorting logic.
* For **Comparator**, sorting can be done using overloaded versions of **sort()** method or using classes like **TreeSet** and **TreeMap**, where the sorting logic is provided externally through a **Comparator** object.

**15.Explain volatile and transient keyword in java**

**Volatile:**

* The **volatile** keyword is used to indicate that a variable’s value may be modified by multiple threads that are executing concurrently.When a variable is declared as **volatile,** it guarantees visibility of changes made to the variable across threads.
* Essentially , when a variable is declared as **volatile,** any read or write operation on that variable will be performed directly on the main memory rather than in thread’s cache.This ensures that changes made by one thread to a **volatile** variable are immediately visible to other threads.
* However **volatile** does not provide atomicity or synchronization . It only ensures visibility.

**Code Snippet:**

Suppose we have a boolean flag variable shared among multiple threads, and one thread modifies the flag while other threads read it. We want to ensure that changes to the flag made by one thread are immediately visible to other threads.

class SharedResource {

private volatile boolean flag = false;

public void setFlag(boolean value) {

flag = value;

}

public boolean getFlag() {

return flag;

}

}

In this example, the **volatile** keyword is used for the **flag** variable. Now, any changes made to **flag** by one thread will be immediately visible to other threads.

public class Main {

public static void main(String[] args) {

SharedResource sharedResource = new SharedResource();

// Thread 1 - sets the flag

new Thread(() -> {

sharedResource.setFlag(true);

}).start();

// Thread 2 - reads the flag

new Thread(() -> {

System.out.println("Flag value: " + sharedResource.getFlag());

}).start();

}

}

**Transient:**

* The **transient** keyword is used to indicate that a variable should not be serialized when the object is serialized. In Java, serialization is the process of converting an object into a byte stream, which can then be saved to a file or sent over a network.
* When an object is serialized , all of its non-transient fields are written to output stream.However,if field is marked as **transient** , its value will not be persisted during serialization. When the object is deserialized,the transient variable will be initialized to its default value(eg: null for object references and , 0 for primitive values).
* The **transient** keyword is often used for fields that hold temporary or derived data, or for fields that should not be serialized due to security or performance reasons.
* **Code snippet**

import java.io.Serializable;

import java.io.\*;

public class Person implements Serializable {

private String name;

private transient String ssn;

public Person(String name, String ssn) {

this.name = name;

this.ssn = ssn;

}

public String getName() {

return name;

}

public String getSsn() {

return ssn;

}

}

public class SerializationExample {

public static void main(String[] args) {

Person person = new Person("John Doe", "123-45-6789");

// Serialize the object

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {

oos.writeObject(person);

System.out.println("Object has been serialized");

} catch (IOException e) {

e.printStackTrace();

}

// Deserialize the object

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {

Person deserializedPerson = (Person) ois.readObject();

System.out.println("Deserialized Person: " + deserializedPerson.getName());

System.out.println("Deserialized SSN: " + deserializedPerson.getSsn());

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}