

GROKKING The JAVA DEVELOPER INTERVIEW

More Than 200 Questions To Crack The Java, Spring, SpringBoot & Hibernate Interview

JATIN ARORA

GROKKI

The

JAVA DEVELOPER INTERVIEW

More Than 200 Questions To Crack
SpringBoot & Hibernate Interviews



Grokking The Java Developer Interview

**More Than 200 Questions To Crack The Java, Spring,
SpringBoot**

& Hibernate Interview

JATIN ARORA

GROKKING THE JAVA DEVELOPER INTERVIEW

COPYRIGHT © 2020 BY JATIN ARORA.

All rights reserved. No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

For permission or any other information, contact me at:

jatinarora208@gmail.com

Preface

Grokking The Java Developer Interview helps you to crack a Java, Spring & Hibernate interview.

This book covers enough details of a Java/Spring topic that will surely help the absolute beginners and if you are already familiar with Java & Spring, then this book will help you to brush-up on the concepts.

The book has more than 200 questions that are frequently asked during an interview for Java, Spring, SpringBoot & Hibernate profile. Some of the important topics like Multi-threading, Collection framework, Singleton Pattern, SpringBoot Annotations and many more are covered in detail. Most of these topics are explained with code examples that will help you to quickly grasp the concept.

Who is this book for?

This book is for you if you are either preparing for an interview, planning to move into this field in future, brushing up your Java & Spring skills or just want to get an in-depth overview of the field. This book provides the most important and frequently asked questions along with their solutions.

Who is this book NOT for?

This book is not for you if you are looking for an in-depth study of Java or Spring. The objective of this book is not to discuss the ongoing research or challenges in this industry or serve as a substitute for a course book.

If you follow this book diligently, you would be better equipped to face any Java interview. Whether you are a beginner or an intermediate level expert in Java, this book has enough juice for you.

This book contains a lot of code examples, most of the code snippets are displayed in image format that you can zoom-in and out of. However, if some image is not clearly visible on your device, you can also refer to the GitHub repository for this book. The GitHub repo contains all of the snippets shown in this book and it can be found at:

<https://github.com/reachjatin/Grokking-The-Java-Developer-Interview>

Contact Information

I would love to hear from you.

For feedback or queries, you can contact me at

jatinarora208@gmail.com

Your valuable suggestions to improve the book are always welcomed.

I wish you all the best and I am confident that this book will help you in making that job switch that you are looking for.

Happy Learning, Cheers :)

-Jatin Arora

Acknowledgements

I thank my brother, Sumit Kumar, for encouraging me to take up this project. From reading early drafts to advising me on the structure of the book to making sure that it was written in a manner that was coherent, organized and engaging at the same time, his role has been pivotal in the making of this book.

To elaborate a little on his background, Sumit is an UChicago Alum, and presently working for Amazon, Seattle as a Research Scientist at Alexa AI. He has also worked with Samsung, India as a Lead Engineer. He is an active blogger and regularly post articles on Data structures and Algorithms, System Design, NLP and many more interesting topics. You can find those at:

<https://medium.com/@sumit.arora>

<https://blog.reachsumit.com>

To Sumit, for his patience in reading through various preliminary versions of this book, and his vital suggestions. Thank you brother for never saying no to my numerous feedback requests and providing all the constructive criticism. I dedicate this book to him.

I would like to express my sincere gratitude to my family, friends, and everyone who motivated me throughout. Thanks for the support, the honest feedback, and for everything else that has helped to make this book possible in its current form.

-Jatin Arora

Table of Content s

[Preface](#)

[Acknowledgements](#)

[Question 1: What are the 4 pillars of OOPS?](#)

[Question 2: What is an abstract class?](#)

[Question 3: Does Abstract class have constructor?](#)

[Question 4: What is an Interface?](#)

[Question 5: Difference between abstract class and interface](#)

[Question 6: What to choose – interface or abstract class](#)

[Question 7: Why Java 8 has introduced default methods?](#)

[Question 8: Why Java 8 has introduced static methods?](#)

[Question 9: Why Java does not allow multiple inheritance?](#)

[Question 10: What are the rules for Method Overloading and Method Overriding?](#)

[Question 11: Can we override final methods?](#)

[Question 12: Can constructors and private methods be overridden?](#)

[Question 13: What is final keyword and where it can be used?](#)

[Question 14: What is exception and exception handling?](#)

[Question 15: Difference between error and exception](#)

[Question 16: What are the different types of exceptions?](#)

[Question 17: How exception handling is done in java?](#)

[Question 18: Can we write a try block without catch block?](#)

[Question 19: How to handle multiple exceptions together?](#)

[Question 20: When finally block will not get executed](#)

[Question 21: Difference between throw and throws keyword. And discuss Exception Propagation](#)

[Question 22: Exception handling w.r.t. method overriding](#)

[Question 23: Programs related to Exception handling and return keyword](#)

[Question 24: How to make your own custom exception class?](#)

[Question 25: How to make custom checked / unchecked exception?](#)

[Question 26: What happens when you throw an exception from finally block?](#)

[Question 27: What will be Output of below program related to try-catch-finally?](#)

[Question 28: Explain try-with-resources](#)

[Question 29: Why String is Immutable?](#)

[Question 30: What does the equals\(\) method of String class do?](#)

[Question 31: Explain StringBuffer and StringBuilder](#)

[Question 32: Explain the output of below program related to equals\(\) method of StringBuilder](#)

[Question 33: When to use String/StringBuffer/StringBuilder](#)

[Question 34: Explain equals and hashCode contract](#)

[Question 35: What is Marker Interface?](#)

[Question 36: Can you write your own custom Marker interface?](#)

[Question 37: What is Comparable and Comparator?](#)

[Question 38: How to compare a list of Employees based on name and age such that if name of the employee is same then sorting should be based on age](#)

[Question 39: Difference between Comparable and Comparator](#)

[Question 40: Different methods of Object class](#)

[Question 41: What type of arguments are allowed in System.out.println\(\) method?](#)

[Question 42: Explain System.out.println\(\) statement](#)

[Question 43: Explain Auto-boxing and Un-boxing](#)

Question 44: Find the output of below program

Question 45: Can you pass primitive long value in switch statement?

Question 46: Explain static keyword in Java

Question 47: What is an Inner Class in Java, how it can be instantiated and what are the types of Inner Classes?

Question 48: What is Constructor Chaining in java?

Question 49: What is init block?

Question 50: What is called first, constructor or init block?

Question 51: What is Variable shadowing and Variable hiding in Java?

Question 52: What is a constant and how we create constants in Java?

Question 53: Explain enum

Question 54: What is Cloneable?

Question 55: What is Shallow Copy and Deep Copy?

Question 56: What is Serialization and De-serialization?

Question 57: What is serialVersionUID?

Question 58: Serialization scenarios with Inheritance

Question 59: Stopping Serialization and De-serialization

Question 60: What is Externalizable Interface?

Question 61: Externalizable with Inheritance

Question 62: Difference between Serializable and Externalizable

Question 63: How to make a class Immutable?

Question 64: Explain Class loaders in Java

Question 65: What is Singleton Design Pattern and how it can be implemented?

Question 66: What are the different ways in which a Singleton Design pattern can break and how to prevent that from happening?

Question 67: What are the different design patterns you have used in your projects?

[Question 68: Explain Volatile keyword in java](#)

[Question 69: What is Garbage Collection in Java, how it works and what are the different types of Garbage Collectors?](#)

[Question 70: Explain Generics in Java](#)

[Question 71: What is Multi-threading?](#)

[Question 72: How to create a thread in Java?](#)

[Question 73: Which way of creating threads is better: Thread class or Runnable interface](#)

[Question 74: What will happen if I directly call the run\(\) method and not the start\(\) method to execute a thread](#)

[Question 75: Once a thread has been started can it be started again](#)

[Question 76: Why wait, notify and notifyAll methods are defined in the Object class instead of Thread class](#)

[Question 77: Why wait\(\), notify\(\), notifyAll\(\) methods must be called from synchronized block](#)

[Question 78: difference between wait\(\) and sleep\(\) method](#)

[Question 79: join\(\) method](#)

[Question 80: yield\(\) method](#)

[Question 81: Tell something about synchronized keyword](#)

[Question 82: What is static synchronization?](#)

[Question 83: What will be output of below program where one synchronized method is calling another synchronized method?](#)

[Question 84: Programs related to synchronized and static synchronized methods](#)

[Question 85: What is Callable Interface?](#)

[Question 86: How to convert a Runnable to Callable](#)

[Question 87: Difference between Runnable and Callable](#)

[Question 88: What is Executor Framework in Java, its different types and how to create these executors?](#)

[Question 89: Tell something about awaitTermination\(\) method in executor](#)

[Question 90: Difference between shutdown\(\) and shutdownNow\(\) methods of executor](#)

[Question 91: What is Count down latch in Java?](#)

[Question 92: What is Cyclic Barrier?](#)

[Question 93: Atomic classes](#)

[Question 94: What is Collection Framework?](#)

[Question 95: What is Collections?](#)

[Question 96: What is ArrayList?](#)

[Question 97: What is default size of ArrayList?](#)

[Question 98: Which data structure is used internally in an ArrayList?](#)

[Question 99: How add\(\) method works internally or How the ArrayList grows at runtime](#)

[Question 100: How to make an ArrayList as Immutable](#)

[Question 101: What is LinkedList?](#)

[Question 102: When to use ArrayList / LinkedList](#)

[Question 103: What is HashMap?](#)

[Question 104: Explain the internal working of put\(\) and get\(\) methods of HashMap class and discuss HashMap collisions](#)

[Question 105: equals and hashCode method scenarios in HashMap when the key is a custom class](#)

[Question 106: How to make a HashMap synchronized?](#)

[Question 107: What is Concurrent HashMap?](#)

[Question 108: What is HashSet class and how it works internally?](#)

[Question 109: Explain Java's TreeMap](#)

[Question 110: Explain Java's TreeSet](#)

[Question 111: Difference between fail-safe and fail-fast iterators](#)

[Question 112: Difference between Iterator and ListIterator](#)

[Question 113: Difference between Iterator.remove and Collection.remove\(\)](#)

[Question 114: What is the difference between a Monolith and Micro-service architecture?](#)

[Question 115: What is Dependency Injection in Spring?](#)

[Question 116: What are the different types of Dependency Injection?](#)

[Question 117: Difference between Constructor and Setter injection](#)

[Question 118: What is @Autowired annotation?](#)

[Question 119: What is the difference between BeanFactory and ApplicationContext?](#)

[Question 120: Explain the life-cycle of a Spring Bean](#)

[Question 121: What are the different scopes of a Bean?](#)

[Question 122: What is the Default scope of a bean?](#)

[Question 123: What happens when we inject a prototype scope bean in a singleton scope bean?](#)

[Question 124: How to inject a prototype scope bean in a singleton scope bean?](#)

[Question 125: Explain Spring MVC flow](#)

[Question 126: What is the difference between <context:annotation-config> and <context:component-scan>?](#)

[Question 127: What is the difference between Spring and SpringBoot?](#)

[Question 128: What is auto-configuration in SpringBoot?](#)

[Question 129: What are SpringBoot starters?](#)

[Question 130: What is @SpringBootApplication Annotation?](#)

[Question 131: Where does a Spring Boot application start from?](#)

[Question 132: How to remove certain classes from getting auto-configured in SpringBoot?](#)

[Question 133: How to autowire a class which is in a package other than SpringBoot application class's package or any of its sub-packages](#)

[Question 134: What is application.properties file in a SpringBoot application?](#)

[Question 135: How to configure the port number of a SpringBoot application?](#)

[Question 136: Which jar builds our springboot application automatically whenever we change some code just like a node.js application?](#)

[Question 137: What default embedded server is given in spring boot web starter and how we can change the default embedded server to the one of our choice](#)

[Question 138: Where should we put our html and javascript files in a spring boot application?](#)

[Question 139: What are the different stereotype annotations?](#)

[Question 140: Difference between @Component, @Controller, @Service, @Repository annotations?](#)

[Question 141: Difference between @Controller and @RestController annotation](#)

[Question 142: What is @RequestMapping and @RequestParam annotation?](#)

[Question 143: How to define a Get or Post endpoint?](#)

[Question 144: Which annotation is used for binding the incoming json request to the defined pojo class?](#)

[Question 145: What is @Qualifier annotation?](#)

[Question 146: What is @Transactional annotation?](#)

[Question 147: What is @ControllerAdvice annotation?](#)

[Question 148: What is @Bean annotation?](#)

[Question 149: Difference between @Component and @Bean](#)

[Question 150: How to do profiling in a SpringBoot application](#)

[Question 151: What is RestTemplate?](#)

[Question 152: What is Spring Data JPA?](#)

[Question 153: What is the difference between JpaRepository, CrudRepository, PagingAndSortingRepository and which one you have used in your applications?](#)

[Question 154: What is Spring AOP?](#)

[Question 155: Have you used Spring Security in your application](#)

[Question 156: What do you know about Spring Batch framework?](#)

[Question 157: Difference between SOAP and REST](#)

[Question 158: What is Restful api?](#)

[Question 159: What is a stateless object?](#)

[Question 160: What is the difference between Web Server and Application Server?](#)

[Question 161: What do you know about CommandLineRunner and ApplicationRunner?](#)

[Question 162: What do you know about Eureka Naming Server?](#)

[Question 163: What do you know about Zuul?](#)

[Question 164: What do you know about Zipkin?](#)

[Question 165: What do you know about Hysterix?](#)

[Question 166: What is JPA?](#)

[Question 167: What is Hibernate?](#)

[Question 168: Difference between JPA and Hibernate](#)

[Question 169: What is @Entity?](#)

[Question 170: How to give a name to a table in JPA?](#)

[Question 171: What is @Id, @GeneratedValue?](#)

[Question 172: How to use a custom database sequence in Hibernate to generate primary key values?](#)

[Question 173: How to give names to the columns of a JPA Entity](#)

[Question 174: How to define a @OneToMany relationship between entities](#)

[Question 175: Why annotations should be imported from JPA and not from Hibernate?](#)

[Question 176: What is the difference between get\(\) and load\(\) method of Hibernate Session?](#)

[Question 177: What is the difference between save\(\), saveOrUpdate\(\) and persist\(\) method of Hibernate Session?](#)

[Question 178: What is Session and SessionFactory in Hibernate?](#)

[Question 179: What is First Level and Second Level Cache in Hibernate?](#)

[Question 180: What is session.flush\(\) method in Hibernate?](#)

[Question 181: How can we see the SQL query that gets generated by Hibernate?](#)

[Question 182: What is Hibernate Dialect and why we need to configure it?](#)

[Question 183: What do you know about hibernate.hbm2ddl.auto property in Hibernate?](#)

[Question 184: What is Maven?](#)

[Question 185: What is pom.xml?](#)

[Question 186: What is local repo and central repo?](#)

[Question 187: Where we define our local repo path?](#)

[Question 188: Where do we define proxies so that maven can download jars from the internet in a corporate environment?](#)

[Question 189: Explain Maven build life-cycle](#)

[Question 190: What do you know about SQL Joins?](#)

[Question 191: Difference between TRUNCATE & DELETE statements](#)

[Question 192: Difference between Function and Stored Procedure](#)

[Question 193: What is DDL, DML statements?](#)

[Question 194: How to find the nth highest salary from Employee table](#)

[Question 195: Difference between UNION and UNION ALL commands in SQL](#)

[Question 196: Difference between Unique Key and Primary Key in SQL](#)

[Question 197: What is the difference between Primary and Foreign key in SQL?](#)

[Question 198: What is the difference between clustered and non-clustered index?](#)

[Question 199: What is the difference between WHERE and HAVING clause in SQL](#)

[Question 200: How to change the gender column value from Male to Female and Female to Male using single Update statement](#)

[Question 201: Find first 3 largest numbers in an array.](#)

[Question 202: Move all negative numbers at the beginning of an array and all positive numbers at the end](#)

[About the Author](#)

Question 1: What are the 4 pillars of OOPS?

Answer: 4 pillars of OOPS are:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Let's take a look at them:

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

Real world examples:

- TV remote: To start the TV, you have to press the power button, you don't have to know about the internal circuit operations like how infrared waves are passing.
- Car gears: We know what happens when we change the gear. But we don't know how changing gear works under the hood, that information is irrelevant to us, so it is abstracted.

In java, Abstraction can be achieved in two ways:

- Abstract classes
- Interfaces

2. **Encapsulation** : Encapsulation is a process of *Binding data and methods within a class* . Think of it like showing the essential details of a class by using the access control modifiers (*public, private, protected*). So, we can say that Encapsulation leads to the desired level of Abstraction.

Example:

Java Bean, where all data members are made private and you define certain public methods to the outside world to access them.

3. Inheritance : Using inheritance means defining a parent-child relationship between classes, by doing so, you can reuse the code that is already defined in the parent class. Code reusability is the biggest advantage of Inheritance.

Java does not allow multiple inheritance through classes but it allows it through interfaces.

4. Polymorphism : Poly means many and Morph means forms. Polymorphism is the process in which an object or function takes different forms. There are 2 types of Polymorphism :

- Compile Time Polymorphism (Method Overloading)
- Run Time Polymorphism (Method Overriding)

In Method overloading, two or more methods in one class have the same method name but different arguments. It is called as *Compile time polymorphism* because it is decided at compile time which overloaded method will be called.

Overriding means when we have two methods with same name and same parameters in parent and child class. Through overriding, child class can provide specific implementation for the method which is already defined in the parent class.

Question 2: What is an abstract class?

Answer: A class that is declared using “abstract” keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (methods with body).

Some points to remember:

- An abstract class cannot be instantiated, which means you are not allowed to create an object of the abstract class. This also means, an abstract class has no use unless it is extended by some other class
- If there is any abstract method in a class then that class must be declared abstract
- The first non-abstract class which is extending from an abstract class will have to give implementation of the abstract methods defined in abstract class

Example:

```
package com.tech;

abstract class MyAbstractClass {

    //abstract method
    abstract void print();

    //concrete method
    public void display() {
        System.out.println("In display method");
    }

}
```

```
public class AbstractDemo extends MyAbstractClass {  
  
    @Override  
    void print() {  
        System.out.println("In print method");  
    }  
  
    public static void main(String[] args) {  
        AbstractDemo obj = new AbstractDemo();  
        obj.print();  
        obj.display();  
    }  
}
```

Output:

```
In print method  
In display method
```

Question 3: Does Abstract class have constructor?

Answer: This is a famous interview question and the answer is: Yes, abstract classes have constructor. Either you can provide it or the default one will be provided by Java. Now, you must be wondering if you cannot create an object of abstract class then what is the need of a constructor.

One thing you must know is that the constructors are used when you are creating an object of a class, to initialize the data members of that class and your abstract class can have data members.

Now, when your class extends abstract class then the same abstract class will become super class for your extending class and remember when you have constructor of your class then first line of your constructor is always a

call to super class constructor and this is the time when your abstract class constructor will get called.

Example 1:

```
package com.tech;

abstract class MyAbstractClass {

    public MyAbstractClass() {
        System.out.println("inside MyAbstractClass constructor");
    }

}

public class AbstractDemo extends MyAbstractClass {

    public AbstractDemo() {
        System.out.println("inside AbstractDemo constructor");
    }

    public static void main(String[] args) {
        AbstractDemo obj = new AbstractDemo();
    }

}
```

Output:

```
inside MyAbstractClass constructor
inside AbstractDemo constructor
```

Example 2:

```
package com.tech;

abstract class MyAbstractClass {

    public int a;
    public int b;

    public MyAbstractClass(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void print() {
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}

public class AbstractDemo extends MyAbstractClass {

    public AbstractDemo(int x, int y) {
        super(x, y);
    }

    public static void main(String[] args) {
        AbstractDemo obj = new AbstractDemo(5, 10);
        obj.print();
    }
}
```

Output:

a = 5

b = 10

Question 4: What is an Interface?

Answer:

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- Interface specify what a class must do but not how to do
- An interface is like defining a contract that is fulfilled by implementing classes
- An interface is used to achieve full abstraction.
- All methods in an interface are public and abstract by default and all variables declared in an interface are constants i.e. public, static and final
- A class which implements an interface will have to provide implementation of all the methods that are defined in the interface
- A class can implement more than one interface, this is how Java allows multiple inheritance.
- Since Java 8, we can have default and static methods in an interface

Question 5: Difference between abstract class and interface

Answer: The differences are:

- Abstract class can have both abstract and concrete methods but interface can only have abstract methods (Java 8 onwards, it can have default and static methods as well)
- Abstract class methods can have access modifiers other than public but interface methods are implicitly public and abstract
- Abstract class can have final, non-final, static and non-static variables but interface variables are only static and final
- A subclass can extend only one abstract class but it can implement multiple interfaces
- An Abstract class can extend one other class and can implement multiple interfaces but an interface can only extend other interfaces

In this question, the interviewer may try to confuse you by saying that from Java 8 onwards, you can have static and default methods in an Interface so now what is the difference between abstract class and interface and the answer you should tell is – We can still extend only one class but can implement multiple interfaces.

Question 6: What to choose – interface or abstract class

Answer: Consider these points while choosing between the two:

- When you want to provide default implementation to some of the common methods that can be used directly by the subclasses then you can use abstract class because it can have concrete methods also, this is not the case with Interface because the child classes that are implementing this interface will have to provide implementation for all the methods that are declared in the interface
- If your contract keeps on changing then Interface will create problems because then you will have to provide implementation of those new methods in all the implementing classes, whereas with abstract class you can provide one default implementation to the new methods and only change those implementing classes that are actually going to use these new methods

Most of the times, interfaces are a good choice. It is also one of the best practices, when you code in terms of interfaces.

Question 7: Why Java 8 has introduced default methods?

Answer: To extend the capability of an already existing interface, default methods are introduced in Java 8. Let's understand this by one example:

Consider there are 100 classes that are implementing one interface. Now you want to define one new method inside your interface. In this case you will have to change all the implementation classes to fulfill the interface contract. So, Java introduced default methods, here you can provide default implementation of that new method inside your interface and as it is not mandatory to provide implementation of default methods by the implementing classes, all the 100 classes can use the default implementation or if they want they can provide their own implementation by overriding the default method.

Now consider one interesting scenario: You have two interfaces, *Interface1* and *Interface2* both having default method *hello()* and one class is implementing these 2 interfaces without giving implementation to this default method. You see the problem here? Yes, it is the famous ***Diamond***

Problem (Refer to *Question 9*, if you're not already familiar with this problem).

```
interface Interface1 {
    default void hello() {
        System.out.println("Hello from Interface1");
    }
}

interface Interface2 {
    default void hello() {
        System.out.println("Hello from Interface2");
    }
}

public class Child implements Interface1, Interface2 {

}

▼ ① Errors (1 item)
  ↗ Duplicate default methods named hello with the parameters () and () are inherited from the types Interface2 and Interface1
```

So, to avoid this error, it is mandatory to provide implementation for common default methods of interfaces

```
public class Child implements Interface1, Interface2 {
    @Override
    public void hello() {
        System.out.println("inside Child class hello method");
        Interface1.super.hello();
    }

    public static void main(String[] args) {
        Child obj = new Child();
        obj.hello();
    }
}
```

Output:

```
inside Child class hello method
Hello from Interface1
```

Question 8: Why Java 8 has introduced static methods?

Answer: Consider an example where you want to define a utility class, what you usually do is you define a class which contains static methods and then you call these methods using class name. Now, Java 8 onwards you can do the same thing using an Interface by giving only static methods inside your interface. This way of using Interface for defining utility classes is better as it helps in performance also, because using a class is more expensive operation than using an interface.

Question 9: Why Java does not allow multiple inheritance?

Answer: Multiple inheritance occurs when a class has more than one parent classes.

Why Java does not allow this : let us consider there are 2 parent classes having a method named *hello()* with same signature and one child class is extending these 2 classes, if you call this *hello()* method which is same in both parents, which parent class method will get executed – it results into an ambiguous situation, this is also called **Diamond Problem** .

You will get a compile time error if you try to extend more than one class.

```

class Parent1 {
    public void hello() {
        System.out.println("Hello from Parent1 class");
    }
}

class Parent2 {
    public void hello() {
        System.out.println("Hello from Parent2 class");
    }
}

public class Child extends Parent1, Parent2 {
}

```

▼  Errors (1 item)

 Syntax error on token ',', . expected

Question 10: What are the rules for Method Overloading and Method Overriding?

Answer: **Method Overloading Rules:** Two methods can be called overloaded if they follow below rules:

- Both have same method name
- Both have different arguments

If both methods follow above two rules, then they may or may not:

- Have different access modifiers
- Have different return types
- Throw different checked or unchecked exceptions

Method Overriding Rules: The overriding method of child class must follow below rules:

- It must have same method name as that of parent class method
- It must have same arguments as that of parent class method
- It must have either the same return type or covariant return type (child classes are covariant types to their parents)
- It must not throw broader checked exceptions
- It must not have a more restrictive access modifier (if parent method is public, then child method cannot be private/protected)

Question 11: Can we override final methods?

Answer: No, final methods cannot be overridden.

Question 12: Can constructors and private methods be overridden?

Answer: No

Question 13: What is final keyword and where it can be used?

Answer: If you use final with a primitive type variable, then its value cannot be changed once assigned.

If you use final with a method, then you cannot override it in the subclass.

If you use final with class, then that class cannot be extended.

If you use final with an object type, then that object cannot be referenced again.

Question 14: What is exception and exception handling?

Answer: An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime, so exception handling is a mechanism by which normal flow of the program is maintained.

Program showing the exception is thrown:

```
public class TestException {  
    public static void main(String[] args) {  
        System.out.println("Program Started");  
        int a = 15/0;  
        System.out.println("Program End");  
    }  
}
```

Output:

```
Program Started  
Exception in thread "main" java.lang.ArithmetiException: / by zero  
at com.exception.TestException.main(TestException.java:6)
```

Program showing that the exception is handled:

```
public class TestException {  
    public static void main(String[] args) {  
        System.out.println("Program Started");  
        try{  
            int a = 15/0;  
        } catch (ArithmetiException e) {  
            System.out.println("Exception handled");  
        }  
        System.out.println("Program End");  
    }  
}
```

Output:

```
Program Started  
Exception handled  
Program End
```

Question 15: Difference between error and exception

Answer: **Error** : Errors in a program are irrecoverable, they indicate that something severe has gone wrong in the application and the program gets terminated in case of error occurrence e.g. running out of memory: *OutOfMemoryError* , making too many recursive calls: *StackOverflowError* etc.

Exception : Exceptions on the other hand are something that we can recover from by handling them properly e.g.: trying to access a property/method from a null object: *NullPointerException* , dividing an integer by zero: *ArithmaticException* etc.

Question 16: What are the different types of exceptions?

Answer: There are 2 types of exceptions:

- **Checked Exceptions:** All exceptions other than *RuntimeException* and *Error* are known as Checked exception. These exceptions are checked by the compiler at the compile time itself. E.g. when you are trying to read from a file, then compiler enforces us to handle the *FileNotFoundException* because it is possible that the file may not be present. Some other checked exceptions are *SQLException* , *IOException* etc.

```
public class DemoException {
    public static void main(String[] args) {
        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } finally {
            System.out.println("Inside finally block");
        }
    }
}
```

Errors (1 item)

Unhandled exception type FileNotFoundException

- **Unchecked Exceptions:** Runtime Exceptions are known as Unchecked exceptions. Compiler does not force us to handle these exceptions but as a programmer, it is our responsibility to handle runtime exceptions e.g. *NullPointerException* , *ArithmaticException* , *ArrayIndexOutOfBoundsException* etc.

Question 17: How exception handling is done in java?

Answer: try-catch block is used for exception handling. If you think that certain statements may throw an exception, surround them with try block.

A try block is always followed by a catch block or finally or both.

You cannot use the try block alone:

```
public class DemoException {  
    public static void main(String[] args) {  
        try {  
            FileReader f = new FileReader("C:\\temp\\dummy.txt");  
        }  
    }  
  
    ▾ ✘ Errors (1 item)  
        ✘ Syntax error, insert "Finally" to complete BlockStatements
```

Question 18: Can we write a try block without catch block?

Answer: Yes, we can write a try block with finally, but we cannot write a try block alone.

Question 19: How to handle multiple exceptions together?

Answer: You can write multiple catch blocks one after another for each exception or you can write a single catch block using a pipe symbol (|) to separate the exceptions.

While writing multiple catch block, you have to follow the below rule:

- Handle the most specific exception and then move down to the most generic ones, means you cannot handle Exception (base class of exception) before FileNotFoundException

```
public class DemoException {  
    public static void main(String[] args) {  
  
        try {  
            FileReader f = new FileReader("C:\\temp\\dummy.txt");  
        } catch (Exception e) {  
            e.printStackTrace();  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } finally {  
            System.out.println("Inside finally block");  
        }  
    }  
}
```

Errors (1 item)
Unreachable catch block for FileNotFoundException. It is already handled by the catch block for Exception

Suppose, your method is throwing more than one exception and you want to perform some specific action based on the exception thrown, you should use multiple catch blocks in this case.

Example using multiple catch blocks:

```

public class DemoException {
    public static void main(String[] args) {

        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } catch (FileNotFoundException e) {
            System.out.println("Action when File is not found");
        } catch (NullPointerException e) {
            System.out.println("Action for NullPointerException");
        } catch (Exception e) {
            System.out.println("Action for exceptions other than "
                + "FileNotFoundException/NullPointerException");
        } finally {
            System.out.println("Inside finally block");
        }

    }
}

```

When using pipe (|) symbol:

```

public class DemoException {

    public static void main(String[] args) {

        try {
            FileReader f = new FileReader("C:\\temp\\dummy.txt");
        } catch (FileNotFoundException | NullPointerException e ) {
            System.out.println("Same action for both");
        } finally {
            System.out.println("Inside finally block");
        }

    }
}

```

Question 20: When finally block will not get executed

Answer:

- when System.exit() is called
- when JVM crashes

Question 21: Difference between throw and throws keyword. And discuss Exception Propagation

Answer:

- **throw** is a keyword which is used to explicitly throw an exception in the program, inside a function or inside a block of code, whereas **throws** is a keyword which is used with the method signature to declare an exception which might get thrown by the method while executing the code
- **throw** keyword is followed by an instance of an Exception class whereas **throws** is followed by Exception class names
- You can throw one exception at a time but you can declare multiple exceptions using **throws** keyword
- Using **throw** keyword, only unchecked exceptions are propagated, whereas using **throws** keyword both checked and unchecked exceptions can be propagated.

Exception propagation :

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

- When method **m1()** calls method **m2()** which calls method **m3()**, a stack is formed which gets unfold from the top, so if method **m3()** throws an exception and it is not handled there, it will drop down the call stack to method **m2()**, if it is not handled there, it will drop down the call stack to method **m1()**, this happens until we reach the bottom of the stack or until the exception is caught. This is called Exception Propagation in java.

Example: unchecked exception is thrown and it can be seen from the call stack that it is propagated

```
public class DemoException {  
  
    public static void method1() {  
        method2();  
    }  
  
    public static void method2() {  
        throw new ArithmeticException("Arithmetic Exception from method2");  
    }  
  
    public static void main(String[] args) {  
        method1();  
    }  
  
}
```

Output:

```
Exception in thread "main" java.lang.ArithmetiException: Arithmetic Exception from method2  
at com.tech.DemoException.method2(DemoException.java:10)  
at com.tech.DemoException.method1(DemoException.java:6)  
at com.tech.DemoException.main(DemoException.java:14)
```

Example: Here the unchecked exception is handled

```
public class DemoException {  
  
    public static void method1() {  
        method2();  
    }  
  
    public static void method2() {  
        throw new ArithmeticException("Arithmetic Exception from method2");  
    }  
  
    public static void main(String[] args) {  
        try {  
            method1();  
        } catch (ArithmetiException e) {  
            System.out.println("Exception handled");  
        }  
    }  
}
```

Output:

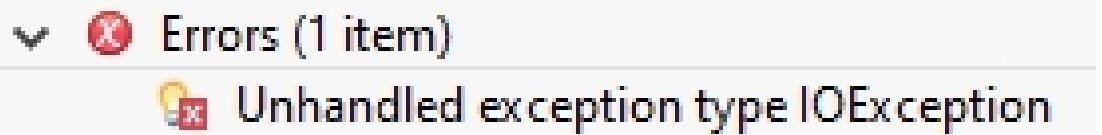
Exception handled

Example: *checked exceptions are not propagated down the call chain by default*,

```
public class DemoException {  
  
    public static void method1() {  
        throw new IOException("IO Exception occurred");  
    }  
  
    public static void main(String[] args) {  
        method1();  
    }  
  
}  
  
▼  Errors (1 item)  
     Unhandled exception type IOException
```

You have to use **throws keyword** if you want to propagate the checked exception, like

```
public class DemoException {  
  
    public static void method1() throws IOException {  
        throw new IOException("IO Exception occurred");  
    }  
  
    public static void main(String[] args) {  
        method1();  
    }  
  
}
```



Notice in the above example that the checked exception is propagated and now it is the responsibility of the caller method to either handle the exception or throw it further. Below example is showing that the **checked exception is handled**,

```
public class DemoException {  
    public static void method1() throws IOException {  
        throw new IOException("IO Exception occurred");  
    }  
  
    public static void main(String[] args) {  
        try {  
            method1();  
        } catch (IOException e) {  
            System.out.println("Exception handled");  
        }  
    }  
}
```

Output:

Exception handled

throws can be used with unchecked exceptions also, though it is of no use because unchecked exceptions are by default propagated. See this in the below program:

```
public class DemoException {  
    public static void method1() throws ArithmeticException{  
        int a = 10/0;  
    }  
  
    public static void main(String[] args) {  
        try {  
            method1();  
        } catch (ArithmetricException e) {  
            System.out.println("Exception handled");  
        }  
    }  
}
```

Output:

Exception handled

Unchecked exceptions are by default propagated:

```
public class DemoException {  
    public static void method1() {  
        int a = 10/0;  
    }  
  
    public static void main(String[] args) {  
        try {  
            method1();  
        } catch (ArithmetricException e) {  
            System.out.println("Exception handled");  
        }  
    }  
}
```

Output:

Exception handled

Question 22: Exception handling w.r.t. method overriding

Answer:

- If the parent class method does not declare an exception then child class overridden method cannot declare checked exceptions but it can declare unchecked exceptions
- If the parent class method declares an exception, then child class overridden method
 - can declare no exception
 - can declare same exception
 - can declare a narrower exception (more broader exception declaration than parent one is not allowed)

Example when parent class method does not declare an exception and child class declares checked exception :

```
package com.exception;

import java.io.IOException;

class Parent {
    public void hello() {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws IOException{
        System.out.println("Child class hello method");
    }
}

▼ ✘ Errors (1 item)
  ✘ Exception IOException is not compatible with throws clause in Parent.hello()
```

Example when parent class method does not declare an exception and child class declares unchecked exception :

```
package com.exception;

class Parent {
    public void hello() {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws ArithmeticException{
        System.out.println("Child class hello method");
    }
}

public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        p.hello();
    }
}
```

Output:

Child class hello method

Example when Child class overridden method throws a broader exception than the parent one :

```
package com.exception;

class Parent {
    public void hello() throws ArithmeticException {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws Exception {
        System.out.println("Child class hello method");
    }
}

▼ ✘ Errors (1 item)
  ✘ Exception Exception is not compatible with throws clause in Parent.hello()
```

Example when child class overridden method throws same exception as the parent one :

```
package com.exception;

class Parent {
    public void hello() throws Exception {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws Exception {
        System.out.println("Child class hello method");
    }
}
```

```
public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        try {
            p.hello();
        } catch (Exception e) {
            System.out.println("Handled");
        }
    }
}
```

Output:

Child class hello method

Example when child class overridden method declares a narrower exception than the parent one :

```
import java.io.IOException;

class Parent {
    public void hello() throws Exception {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() throws IOException {
        System.out.println("Child class hello method");
    }
}
```

```
public class TestException {
    public static void main(String[] args) {
        Parent p = new Child();
        try {
            p.hello();
        } catch (Exception e) {
            System.out.println("Handled");
        }
    }
}
```

Output:

Child class hello method

Example when parent class method declares an exception and child class overridden method does not declare any exception :

```
package com.exception;

class Parent {
    public void hello() throws Exception {
        System.out.println("Parent class hello method");
    }
}

class Child extends Parent {
    public void hello() {
        System.out.println("Child class hello method");
    }
}
```

```
public class TestException {  
    public static void main(String[] args) {  
        Parent p = new Child();  
        try {  
            p.hello();  
        } catch (Exception e) {  
            System.out.println("Handled");  
        }  
    }  
}
```

Output:

Child class hello method

Question 23: Programs related to Exception handling and return keyword

Answer: These questions are asked a lot of times to the new programmers.

One thing you should remember is, if you write anything after return statement / throw exception statement, then that will give Compile time error as 'Unreachable code'.

Program 1:

```
package com.tech;

public class DemoException {
    public static int method1() {
        try {
            throw new ArithmeticException();
        return 1;
    } catch(Exception e) {
        return 2;
    } finally {
        return 3;
    }
}

    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

Errors (1 item)

Unreachable code

Program 2 : finally block is always executed

```
package com.tech;

public class DemoException {
    public static int method1() {
        try {
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

Output:

3

Program 3 : program execution returns from the finally block

```
public class DemoException {
    public static int method1() {
        try {
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
        return 4;
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

▼  Errors (1 item)

  Unreachable code

Program 4 :

```
public class DemoException {
    public static int method1() {
        try {
            int a = 15/0;
            return 1;
        } catch(Exception e) {
            return 2;
        } finally {
            return 3;
        }
    }
    public static void main(String[] args) {
        int result = method1();
        System.out.println(result);
    }
}
```

Output:

3

Program 5 :

```
public class DemoException {  
    public static int method1() {  
        try {  
            int a = 15/0;  
            return 1;  
        } catch(Exception e) {  
            return 2;  
        }  
    }  
    public static void main(String[] args) {  
        int result = method1();  
        System.out.println(result);  
    }  
}
```

Output:

2

Question 24: How to make your own custom exception class?

Answer: In java, you can create your own custom exception class which are basically derived classes from the Exception class.

For Example:

```
package com.tech;  
  
class MyException extends Exception {  
    public MyException(String s) {  
        super(s);  
    }  
}
```

```
public class DemoException {  
  
    public static void main(String[] args) {  
        try {  
            method1();  
        } catch (MyException e) {  
            System.out.println(e.getMessage());  
        }  
        System.out.println("Program end");  
    }  
  
    public static void method1() throws MyException {  
        throw new MyException("My own custom exception");  
    }  
}
```

Output:

My own custom exception
Program end

Question 25: How to make custom checked / unchecked exception?

Answer: If you want to make a custom unchecked exception class then extend *RuntimeException* and for creating a custom checked exception class, extend *Exception* class.

Question 26: What happens when you throw an exception from finally block?

Answer: When exception is thrown from finally block, then it takes precedence over the exceptions that are thrown from try/catch block

Question 27: What will be Output of below program related to try-catch-finally?

```

class MyException1 extends Exception { }
class MyException2 extends Exception { }

public class DemoException {
    public static void method1() throws Exception {
        try{
            System.out.println("5");
            throw new MyException1();
        } catch (Exception e) {
            System.out.println("6");
            throw new MyException2();
        } finally {
            System.out.println("7");
            throw new Exception();
        }
    }

    public static void main(String[] args) throws Exception {
        try {
            System.out.println("1");
            method1();
            System.out.println("2");
        } catch (Exception e) {
            System.out.println("3");
            throw new MyException2();
        } finally {
            System.out.println("4");
            throw new MyException1();
        }
    }
}

```

With everything you have read so far, you should be able to answer this easily. I will leave this one for you.

Question 28: Explain try-with-resources

Answer: try-with-resources concept was introduced in Java 7. It allows us to declare resources which will be used inside the try block and it assures us that the resources will be closed after execution of this block. A resource

is an object that must be closed after finishing the program. The resources declared must implement *AutoCloseable* interface.

Syntax:

```
public class Demo {  
    public static void main(String[] args) {  
        try (FileReader reader = new FileReader("C:\\temp\\dummy.txt")){  
  
            //program statements  
  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Question 29: Why String is Immutable?

Answer: String is immutable for below reasons:

- 1. String Pool :** String Pool is possible only because String is Immutable in Java. String pool is a special storage area in Java heap. If the string is already present in the pool, then instead of creating a new object, old object's reference is returned. This way different String variables can refer to the same reference in the pool, thus saving a lot of heap space also. If String is not immutable then changing the string with one reference will lead to the wrong values to other string variables having the same reference.
- 2. Security :** String parameters are used in network connections, database URL's, username and passwords etc. Because String is immutable, these values can't be changed. Otherwise any hacker could change the referenced values which will cause severe security issues in the application.
- 3. Multi-threading :** Since String is immutable, it is safe for multithreading. A single String instance can be shared across different threads. This avoids the use of synchronization for thread safety. Strings are implicitly thread-safe.

4. **Caching** : The hashCode of string is frequently used in Java.
Since string is immutable, the hashCode will remain the same, so it can be cached without worrying about the changes. This makes it a great candidate for using it as a Key in Map.
5. **Class Loaders** : Strings are used in Java ClassLoaders and since String is made immutable, it provides security that correct class is being loaded.

Question 30: What does the equals() method of String class do?

Answer: As we know, Object class is the parent of all classes, and Object class has a equals() method that compares the reference of two objects, but String class has overridden this method, and String class's equals() method compares the contents of two strings.

Program:

```
public class Test {  
    public static void main(String[] args) {  
        String s1 = "Mark";  
        String s2 = "John";  
        String s3 = "Mark";  
  
        System.out.println("s1.equals(s2): " + s1.equals(s2));  
        System.out.println("s1.equals(s3): " + s1.equals(s3));  
    }  
}
```

Output:

```
s1.equals(s2): false  
s1.equals(s3): true
```

Question 31: Explain StringBuffer and StringBuilder

Answer: Both StringBuffer and StringBuilder classes are used for String manipulation. These are mutable objects. But StringBuffer provides thread-safety as all its methods are synchronized, this makes performance of StringBuffer slower as compared to StringBuilder.

StringBuffer class is present from Java 1.0, but due to its slower performance, StringBuilder class was introduced in Java 1.5

If you are in a single-threaded environment or don't care about thread safety, you should use StringBuilder. Otherwise, use StringBuffer for thread-safe operations.

Question 32: Explain the output of below program related to equals() method of StringBuilder

```
public class Demo {  
    public static void main(String[] args) {  
  
        StringBuilder sb1 = new StringBuilder("hello");  
        StringBuilder sb2 = new StringBuilder("hello");  
  
        if(sb1.equals(sb2)) {  
            System.out.println("Equal");  
        } else {  
            System.out.println("Not Equal");  
        }  
    }  
}
```

Output:

Not Equal

Answer: This is another very famous interview question. If you were expecting 'Equal' as output, then you were wrong. The output is not 'Equal' because StringBuffer and StringBuilder does not override equals and hashCode methods. In the above program, Object's class equals() method

is getting used and as it compares the reference of two objects, the output of above program is ‘Not Equal’.

Since hashCode is used in data structures that use hashing algorithm to store the objects. Examples are HashMap, HashSet, HashTable, ConcurrentHashMap etc. and all these data structures require their keys not to be changed so that stored values can be found by using hashCode method but StringBuffer/StringBuilder are mutable objects. This makes them a very poor choice for this role.

You must have heard about the equals and hashCode contract in Java, which states that if two objects are equals according to equals() method then their hashCode must be same, vice-versa is not true. Now, had the equals method for StringBuilder/StringBuffer been overridden, their corresponding hashCode method would also need to be overridden to follow that rule. But as explained earlier, these classes don’t need to have their own hashCode implementation and hence same is with their equals method.

***Question 33: When to use
String/StringBuffer/StringBuilder***

Answer: You should use String class if you require immutability, use StringBuffer if you require mutability + Thread safety and use StringBuilder if you require mutability and no thread safety.

Question 34: Explain equals and hashCode contract

Answer: The equals and hashCode contract says:

- If two objects are equals according to equals() method, then their hashCode must be same but reverse is not true i.e. if two objects have same hashCode then they may/may not be equals.

Question 35: What is Marker Interface?

Answer: Marker interface is an interface which is empty. Some of the Marker interfaces are Cloneable, Serializable, Remote etc. If you have read that Marker interfaces indicate something to the compiler or JVM, then you have read it wrong, it has nothing to do with JVM.

Consider the example of Cloneable:

It is said that you cannot call `clone()` method on a class object unless the class implements Cloneable interface. Well this statement is true, because when you call `clone()` method then the first statement in `clone()` is, *obj instanceof Cloneable* .

The class object on which `clone()` method is getting called is checked, whether the class implements Cloneable interface or not, by using `instanceOf` operator. If class does not implement Cloneable interface then `CloneNotSupportedException` is thrown.

Same is true with `writeObject(Object)` method of `ObjectOutputStream` class. Here, *obj instanceof Serializable* is used to check whether the class implements Serializable interface or not. If class does not implement Serializable interface then `NotSerializableException` is thrown.

Question 36: Can you write your own custom Marker interface?

Answer: Yes. As you already know that Marker interfaces have got nothing to do with indicating some signal to JVM or compiler, instead it is just a mere check of using `instanceOf` operator to know whether the class implements Marker interface or not.

In your method, you can put a statement like: `object instanceof MyMarkerInterface`.

You can use Marker interface for classification of your code.

Question 37: What is Comparable and Comparator?

Answer: Both Comparable and Comparator interfaces are used to sort the collection of objects. These interfaces should be implemented by your

custom classes, if you want to use Arrays/Collections class sorting methods.

Comparable interface has compareTo(Obj) method, you can override this method in your class, and you can write your own logic to sort the collection.

General rule to sort a collection of objects is:

If ‘this’ object is less than passed object, return negative integer.

If ‘this’ object is greater than passed object, return positive integer.

If ‘this’ object is equal to the passed object, return zero.

Comparable Example :

Employee.java

```
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int age;
    private long salary;

    public Employee(int id, String name, int age, long salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
}
```

```
public int getId() { return id; }
public String getName() { return name; }
public int getAge() { return age; }
public long getSalary() { return salary; }

public void setId(int id) { this.id = id; }
public void setName(String name) { this.name = name; }
public void setAge(int age) { this.age = age; }
public void setSalary(long salary) { this.salary = salary; }

@Override
public int compareTo(Employee obj) {
    return this.id - obj.id;
}

@Override
public String toString() {
    return "Employee [id=" + id + ", name=" + name + ", age=" + age + ", "
           + "salary=" + salary + "]"
           + "\n";
}
}
```

ComparableDemo.java :

```
public class ComparableDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));

        Collections.sort(empList);
        System.out.println(empList);
    }
}
```

Output:

```
[Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=20, name=Mike, age=20, salary=10000]
]
```

Here, we have sorted the Employee list based on 'id' attribute.

Now, if we want to sort the employee list based on any other attribute, say name, we will have to change our compareTo() method implementation for this. So, Comparable allows only single sorting mechanism.

But Comparator allows sorting based on multiple parameters. We can define another class which will implement Comparator interface and then we can override it's compare(Obj, Obj) method.

Suppose we want to sort the Employee list based on name and salary.

NameComparator.java :

```
public class NameComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getName().compareTo(emp2.getName());
    }

}
```

String class already implements Comparable interface and provides a lexicographic implementation for compareTo() method which compares 2 strings based on contents of characters or you can say in lexical order. Here, Java will determine whether passed String object is less than, equal to or greater than the current object.

ComparatorDemo.java :

```
public class ComparatorDemo {  
  
    public static void main(String[] args) {  
        List<Employee> empList = new ArrayList<>();  
  
        empList.add(new Employee(4, "Dave", 25, 28000));  
        empList.add(new Employee(20, "Mike", 20, 10000));  
        empList.add(new Employee(9, "Abhi", 32, 5000));  
        empList.add(new Employee(1, "Lisa", 40, 19000));  
  
        Collections.sort(empList, new NameComparator());  
        System.out.println(empList);  
    }  
  
}
```

Output:

```
[Employee [id=9, name=Abhi, age=32, salary=5000]  
, Employee [id=4, name=Dave, age=25, salary=28000]  
, Employee [id=1, name=Lisa, age=40, salary=19000]  
, Employee [id=20, name=Mike, age=20, salary=10000]  
]
```

The output list is sorted based on employee's names.

SalaryComparator.java :

```
import java.util.Comparator;  
  
public class SalaryComparator implements Comparator<Employee> {  
  
    @Override  
    public int compare(Employee emp1, Employee emp2) {  
        return (int) (emp1.getSalary() - emp2.getSalary());  
    }  
}
```

ComparatorDemo.java :

```

public class ComparatorDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));

        Collections.sort(empList, new SalaryComparator());
        System.out.println(empList);
    }

}

```

Output:

```

[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=20, name=Mike, age=20, salary=10000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=4, name=Dave, age=25, salary=28000]
]

```

Question 38: How to compare a list of Employees based on name and age such that if name of the employee is same then sorting should be based on age

Answer: When comparing by name, if both names are same, then comparison will give 0. If the compare result is 0, we will compare based on age.

NameAgeComparator.java :

```
public class NameAgeComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee emp1, Employee emp2) {
        int flag = emp1.getName().compareTo(emp2.getName());
        if(flag == 0) {
            flag = emp1.getAge() - emp2.getAge();
        }
        return flag;
    }

}
```

ComparatorDemo.java :

```
public class ComparatorDemo {

    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));
        empList.add(new Employee(15, "Mike", 25, 15000));
        empList.add(new Employee(8, "Mike", 30, 20000));

        Collections.sort(empList, new NameAgeComparator());
        System.out.println(empList);
    }

}
```

Output:

```
[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=20, name=Mike, age=20, salary=10000]
, Employee [id=15, name=Mike, age=25, salary=15000]
, Employee [id=8, name=Mike, age=30, salary=20000]
]
```

Question 39: Difference between Comparable and Comparator

Answer:

- Comparable interface can be used to provide single way of sorting whereas Comparator interface is used to provide multiple ways of sorting
- Comparable interface is present in 'java.lang' package whereas Comparator interface is present in 'java.util' package
- For using Comparable, the class needs to implement Comparable interface whereas for using Comparator, there is no need to make changes in the class
- Comparable provides compareTo() method to sort elements, whereas Comparator provides compare() method to sort elements
- We can sort the list elements of Comparable type by using Collections.sort(listObj) method, whereas to sort the list elements of Comparator type, we have to provide a Comparator object like, Collections.sort(listObj, Comparator)

At times, when you are using any third-party classes or the classes where you are not the author of the class, then in that case Comparator is the only choice to sort those objects

Question 40: Different methods of Object class

Answer: Object class sits at the top of class hierarchy tree. Every class is a child of Object class. Below methods are present inside Object class:

```
//Creates and returns a copy of this object
protected native Object clone() throws
CloneNotSupportedException

//Indicates whether some other object is "equal to"
//this one
public boolean equals(Object obj )
```

```
//Returns a hash code value for the object
public native int hashCode()

//Returns the runtime class of an Object
public final native Class<?> getClass()

//Returns a string representation of the object
public String toString()

//Called by the garbage collector on an object when
garbage collection
//determines that there are no more references to the
object
protected void finalize() throws Throwable

//Wakes up a single thread that is waiting on this
object's monitor
public final native void notify()

//Wakes up all threads that are waiting on this
object's monitor
public final native void notifyAll()

public final native void wait( long timeout ). throws
InterruptedException

public final void wait( long timeout , int nanos ). throws
InterruptedException

public final void wait(). throws InterruptedException
```

**Question 41: What type of arguments are allowed in
System.out.println() method?**

Answer: println() method of PrintStream class is overloaded, and it accepts below arguments :

```
public class DemoPrintln {  
    public static void main(String[] args) {  
        System.out.println;  
    }  
}
```

- `println() : void - PrintStream`
- `println(boolean x) : void - PrintStream`
- `println(char x) : void - PrintStream`
- `println(char[] x) : void - PrintStream`
- `println(double x) : void - PrintStream`
- `println(float x) : void - PrintStream`
- `println(int x) : void - PrintStream`
- `println(long x) : void - PrintStream`
- `println(Object x) : void - PrintStream`
- `println(String x) : void - PrintStream`

Question 42: Explain System.out.println() statement

Answer:

- System is a class in java.lang package
- out is a static member of System class and is an instance of java.io.PrintStream
- println() is a method of PrintStream class

Question 43: Explain Auto-boxing and Un-boxing

Answer: In Java 1.5, the concepts of Auto-boxing and Un-boxing were introduced to automatically convert primitive type to object and vice-versa.

When Java automatically converts a primitive type, like int into its corresponding wrapper class object i.e. Integer, then this is called Auto-boxing. While the opposite of this is called Un-boxing, where an Integer object is converted into primitive type int.

Example:

```
public class Test {  
    public static void main(String[] args) {  
        ArrayList<Integer> list = new ArrayList<>();  
  
        //Auto-boxing : int to Integer  
        list.add(5);  
        list.add(10);  
  
        //Unboxing : Integer to int  
        int a = list.get(0);  
  
        System.out.println(a);  
    }  
}
```

Output:

5

Question 44: Find the output of below program

```

public class Test {
    public void print(int a, long b) {
        System.out.println("Method 1");
    }

    public void print(long a, int b) {
        System.out.println("Method 2");
    }

    public static void main(String[] args) {
        Test obj = new Test();
        obj.print(5, 10);
    }
}

```

Answer: Here, the compiler is confused as to which method to be called, so it throws Compile Time error.

▼ Errors (1 item)

The method print(int, long) is ambiguous for the type Test

Question 45: Can you pass primitive long value in switch statement?

Answer: No, switch works only with 4 primitives and their wrappers, as well as with the enum type and String class:

- byte and Byte
- short and Short
- char and Character
- int and Integer
- enum
- String

```
public class Test {  
    public static void main(String[] args) {  
  
        long num = 5;  
  
        switch(num) {  
            }  
    }  
}
```

Errors (1 item)
Cannot switch on a value of type long. Only convertible int values, strings or enum variables are permitted

Question 46: Explain static keyword in Java

Answer: In Java, a static member is a member of a class that isn't associated with an instance of a class. Instead, the member belongs to the class itself.

In Java, Static is applicable for the following:

- Variable
- Method
- Block
- Nested class

Static Variable : if any variable is declared as static, then it is known as 'static variable'. Only single copy of the variable gets created and all instances of the class share same static variable. The static variable gets memory only once in the class area at the time of class loading.

When to use static variable : static variables should be used to declare common property of all objects as only single copy is created and shared among all class objects, for example, the company name of employees etc.

Static Method : When a method is declared with static keyword then it is known as static method. These methods belong to the class rather than the object of the class. As a result, a static method can be directly accessed using class name without the need of creating an object.

One of the basic rules of working with static methods is that you can't access a non-static method or field from a static method because the static method doesn't have an instance of the class to use to reference instance methods or fields. Another restriction is, 'this' and 'super' cannot be used in static context.

For example: main() method is static, Java Runtime uses this method to start an application without creating an object.

Static Block : Static block gets executed exactly once when the class is first loaded, use static block to initialize the static variables.

Static nested classes :

Static nested classes are a type of inner class in java where the inner class is static. Static nested classes can access only the static members of the outer class. The advantage of using static nested classes is that it makes the code more readable and maintainable.

In the case of normal inner class, you cannot create inner class object without first creating the outer class object, but in the case of static inner class, there can be a static inner class object without the outer class object.

How to create object of static inner class:

```
OuterClass.StaticNestedClass nestedClassObject = new  
OuterClass.StaticNestedClass();
```

Compile Time Error comes when we try to access non-static member inside static nested class:

```
class OuterClass {  
    int a = 10;  
    static int b = 20;  
    private static int c = 30;  
  
    static class InnerClass {  
        void print() {  
            System.out.println("Outer class variable a : " + a);  
            System.out.println("Outer class variable b : " + b);  
            System.out.println("Outer class variable c : " + c);  
        }  
    }  
}
```

▼  Errors (1 item)

 Cannot make a static reference to the non-static field a

Using inner class object:

```
class OuterClass {  
    int a = 10;  
    static int b = 20;  
    private static int c = 30;  
  
    static class InnerClass {  
        void print() {  
            //System.out.println("Outer class variable a : " + a);  
            System.out.println("Outer class variable b : " + b);  
            System.out.println("Outer class variable c : " + c);  
        }  
    }  
}  
  
public class StaticNestedTestClass {  
    public static void main(String[] args) {  
        OuterClass.InnerClass innerClassObject=new OuterClass.InnerClass();  
        innerClassObject.print();  
    }  
}
```

Output:

```
Outer class variable b : 20
Outer class variable c : 30
```

If you have static members in your Static Inner class then there is no need to create the inner class object:

```
class OuterClass {
    static int x = 20;

    static class InnerClass {
        static int y = 30;

        static void display() {
            System.out.println("Outer x : " + x);
        }
    }
}

public class StaticNestedTestClass {
    public static void main(String[] args) {
        OuterClass.InnerClass.display();
        System.out.println(OuterClass.InnerClass.y);
    }
}
```

Output:

```
Outer x : 20
30
```

Question 47: What is an Inner Class in Java, how it can be instantiated and what are the types of Inner Classes?

Answer: In Java, when you define one non-static class within another class, it is called Inner Class/Nested Class. Inner class enables you to logically group classes that are only used in one place, thus, this increases the use of encapsulation and creates more readable and maintainable code.

Java inner class is associated with the object of the class and they can access all the variables and methods of the outer class. Since inner classes are associated with the instance, we can't have any static variables in them.

The object of java inner class is part of the outer class object and to create an instance of the inner class, we first need to create an instance of outer class.

Java Inner classes can be instantiated like below:

```
OuterClass outerObject = new OuterClass();
```

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Example:

```
class OuterClass {
    static int outer_x = 10;
    int outer_y = 20;
    private int outer_z = 30;

    class InnerClass {
        void print() {
            System.out.println("outer_x : " + outer_x);
            System.out.println("outer_y : " + outer_y);
            System.out.println("outer_z : " + outer_z);
        }
    }
}
```

```
public class InnerClassDemo {  
    public static void main(String[] args) {  
        OuterClass outerClass = new OuterClass();  
        OuterClass.InnerClass innerClass = outerClass.new InnerClass();  
  
        innerClass.print();  
    }  
}
```

Output:

```
outer_x : 10  
outer_y : 20  
outer_z : 30
```

Compile time error when static variable and static method is present in Inner class:

```
class OuterClass {  
  
    class InnerClass {  
        static int inner_x = 20;  
  
        static void print() {  
        }  
    }  
}
```

Errors (2 items)

- The field inner_x cannot be declared static in a non-static inner type, unless initialized with a constant expression
- The method print cannot be declared static; static methods can only be declared in a static or top level type

There are 2 special types of Inner Classes:

- local inner class
- anonymous inner class

local inner class : Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a *for loop*, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block in which they are defined in, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it. Local inner class must be instantiated in the block they are defined in.

Points to remember:

- local inner class cannot be instantiated from outside of the block where they are defined
- local inner class has access to the members of the enclosing class
- till Java 1.7, local inner class can access only final local variable of the enclosing block where they are defined. But from Java 1.8 onwards, it is possible to access the non-final local variable of the enclosing block
- the scope of local inner class is restricted to the block where they are defined
- A local inner class can extend an abstract class or can implement an interface

Example:

```

class OuterClass {
    int outer_x = 10;
    static int outer_y = 20;
    private String outer_name = "Mike";

    public void print() {
        int non_final_block_level_j = 30;
        final int final_block_level_k = 40;

        class Inner {
            public void display() {
                System.out.println("outer_x : " + outer_x);
                System.out.println("outer_y : " + outer_y);
                System.out.println("outer_name : " + outer_name);

                System.out.println("non_final_block_level_j : " + non_final_block_level_j);
                System.out.println("non_final_block_level_j : " + final_block_level_k);
            }
        }

        Inner inner = new Inner();
        inner.display();
    }
}

public class InnerClassDemo {
    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass();
        outerClass.print();
    }
}

```

Output:

```

outer_x : 10
outer_y : 20
outer_name : Mike
non_final_block_level_j : 30
non_final_block_level_j : 40

```

Remember, you can only access the block level variables, and cannot change them. You will get compile time error if you try to change them:

```
class OuterClass {  
    public void print() {  
        int print_j = 10;  
        final int print_k = 20;  
  
        class Inner {  
            public void display() {  
                print_j = 50;  
            }  
        }  
  
        Inner inner = new Inner();  
        inner.display();  
    }  
}
```

▼ ✖ Errors (1 item)
✖ Local variable print_j defined in an enclosing scope must be final or effectively final

A variable whose value is not changed once initialized is called as ***effectively final variable***.

Anonymous inner class :

An inner class that does not have any name is called Anonymous Inner class. You should use them when you want to use local class only once. It does not have a constructor since there is no class name and it cannot be declared as static.

Generally, they are used when you need to override the method of a class or an interface.

When to use :

Example 1 : Let's understand this by an example, Suppose you are want to return a list of employee class objects and they should be sorted based on employee name, now for this you can write a comparator in a separate class and pass its object inside the Collections.sort(list, comparatorObject)

Instead, you can use the anonymous inner class and you don't have to create a new class just for writing a comparator that you are not going to use later on.

Program:

Employee.java :

```
public class Employee {  
    private int id;  
    private String name;  
    private int age;  
    private long salary;  
  
    public Employee(int id, String name, int age, long salary) {  
        this.id = id;  
        this.name = name;  
        this.age = age;  
        this.salary = salary;  
    }  
  
    public int getId() { return id; }  
    public String getName() { return name; }  
    public int getAge() { return age; }  
    public long getSalary() { return salary; }  
  
    public void setId(int id) { this.id = id; }  
    public void setName(String name) { this.name = name; }  
    public void setAge(int age) { this.age = age; }  
    public void setSalary(long salary) { this.salary = salary; }  
  
    @Override  
    public String toString() {  
        return "Employee [id=" + id + ", name=" + name + ", age=" + age + ", "  
               + "salary=" + salary + "]" + "\n";  
    }  
}
```

AnonymousInnerDemo.java :

```

public class AnonymousInnerDemo {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        empList.add(new Employee(4, "Dave", 25, 28000));
        empList.add(new Employee(20, "Mike", 20, 10000));
        empList.add(new Employee(9, "Abhi", 32, 5000));
        empList.add(new Employee(1, "Lisa", 40, 19000));
        empList.add(new Employee(15, "Mike", 25, 15000));
        empList.add(new Employee(8, "Mike", 30, 20000));

        Collections.sort(empList, new Comparator<Employee>() {

            @Override
            public int compare(Employee emp1, Employee emp2) {
                return emp1.getName().compareTo(emp2.getName());
            }
        });
        System.out.println(empList);
    }
}

```

Output:

```

[Employee [id=9, name=Abhi, age=32, salary=5000]
, Employee [id=4, name=Dave, age=25, salary=28000]
, Employee [id=1, name=Lisa, age=40, salary=19000]
, Employee [id=20, name=Mike, age=20, salary=10000]
, Employee [id=15, name=Mike, age=25, salary=15000]
, Employee [id=8, name=Mike, age=30, salary=20000]
]

```

Example 2 : Using anonymous inner class, you can implement a Runnable also

```
public class RunnableInnerDemo {  
  
    public static void main(String[] args) {  
  
        Thread t = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                System.out.println("Thread 1");  
            }  
        });  
  
        t.start();  
  
        System.out.println("Main Thread");  
    }  
}
```

Output:

Main Thread
Thread 1

Example 3 : You can use anonymous inner class in situations where you want to override the parent class method without creating a separate child class:

```
package com.demo.inner;  
  
class Parent {  
    public void display() {  
        System.out.println("display method in parent class");  
    }  
}
```

```
public class TestAnonymousInner {
    public static void main(String[] args) {

        Parent p = new Parent() {
            public void display() {
                System.out.println("display method in anonymous inner class");
            }
        };

        p.display();
    }
}
```

Output:

display method in anonymous inner class

Some points to remember:

- anonymous class does not have a constructor as it does not have any class name
- in anonymous class, we cannot have static members except static constants
- an anonymous class has access to the members of its enclosing class
- an anonymous class cannot access local variables in its enclosing scope(block) that are not final or effectively final

Anonymous Inner Class Example :

```
package com.demo.inner;

class Parent {
    public void display() {
        System.out.println("display method in parent class");
    }
}
```

```
public class TestAnonymousInner {

    private int x = 10;
    private static String HELLO = "Hello";
    public static void main() {
        System.out.println("Overloaded main method");
    }
    public void dummy() {
        System.out.println("dummy");
    }

    private void print() {
        int y = 20;
        final int z = 30;
        int w = 40;
        Parent p = new Parent() {
            static final int w = 50;
            public void display() {
                System.out.println("display method in anonymous inner class");
                System.out.println("Enclosing class x : " + x);
                System.out.println("Enclosing class constant : " + HELLO);
                main();
                dummy();
                System.out.println("Enclosing block y : " + y);
                System.out.println("Enclosing block z : " + z);
                System.out.println("w variable shadowing : " + w);
            }
        };
        p.display();
    }

    public static void main(String[] args) {
        TestAnonymousInner obj = new TestAnonymousInner();
        obj.print();
    }
}
```

Output:

```
display method in anonymous inner class
Enclosing class x : 10
Enclosing class constant : Hello
Overloaded main method
dummy
Enclosing block y : 20
Enclosing block z : 30
w variable shadowing : 50
```

Compile time error in case of using static variable which is not final :

The screenshot shows a Java code editor with two files:

- Parent.java**:

```
package com.demo.inner;

class Parent {
    public void display() {
        System.out.println("display method in parent class");
    }
}
```
- TestAnonymousInner.java**:

```
public class TestAnonymousInner {

    public static void main(String[] args) {
        Parent p = new Parent() {
            static final int a = 10;
            static int b = 50;
            public void display() {
                System.out.println("display method in anonymous inner class");
            }
        };
        p.display();
    }
}
```

A tooltip at the bottom right indicates an error: "The field b cannot be declared static in a non-static inner type, unless initialized with a constant expression".

Question 48: What is Constructor Chaining in java?

Answer: when one constructor calls another constructor, it is known as constructor chaining. This can be done in two ways:

- `this()` : it is used to call the same class constructor
- `super()` : it is used to call the parent class constructor

this() Example:

```
public class Employee {

    private String name;
    private int age;
    private int salary;

    public Employee() {
        this("Mike");
    }
    public Employee(String name) {
        this(name, 20);
    }
    public Employee(String name, int age) {
        this(name, age, 20000);
    }
    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    void print() {
        System.out.println("Employee name : " + name);
        System.out.println("Employee age : " + age);
        System.out.println("Employee salary : " + salary);
    }

    public static void main(String[] args) {
        Employee obj = new Employee();
        obj.print();
    }
}
```

Output:

```
Employee name : Mike  
Employee age : 20  
Employee salary : 20000
```

super() example:

```
package com.test;  
  
class Company {  
    private String name;  
    private int age;  
  
    public Company(String name) {  
        System.out.println("ABC Company");  
        System.out.println("Details : ");  
    }  
    public Company(String name, int age) {  
        this(name);  
        this.name = name;  
        this.age = age;  
    }  
    void display() {  
        System.out.println("Name : " + name);  
        System.out.println("Age : " + age);  
    }  
}
```

```
class Employee extends Company {  
    public Employee() {  
        super("Mike", 20);  
    }  
    public Employee(String name) {  
        this();  
    }  
}  
  
public class ConstructorChaining {  
  
    public static void main(String[] args) {  
        Employee obj = new Employee();  
        obj.display();  
    }  
}
```

Output:

```
ABC Company  
Details :  
Name : Mike  
Age : 20
```

Some points to remember:

- `this()` can call same class constructor only
- `super()` can call immediate super class constructor only
- `this()` and `super()` call must be the first statement, because of this reason both cannot be used at the same time
- you must use `super()` to call the parent class constructor but if you do not provide a `super()` call, JVM will put it automatically

Question 49: What is init block?

Answer: init block is called instance initializer block

- init block is used to initialize the instance data members
- init block runs each time when object of the class is created
- init block runs in the order they appear in the program
- compiler replaces the init block in each constructor after the `super()` statement
- init block is different from static block which runs at the time of class loading

Example 1:

```
class TestInit {
    {
        System.out.println("init block called");
    }
    TestInit() {
        System.out.println("default constructor called");
    }
}

public class InitBlockTest {

    public static void main(String[] args) {
        new TestInit();
    }
}
```

Output:

**init block called
default constructor called**

Example 2:

```
package com.test;

class Parent {
    {
        System.out.println("Parent class init block 1");
    }

    public Parent() {
        System.out.println("Parent constructor called");
    }

    {
        System.out.println("Parent class init block 2");
    }
}

class Child extends Parent {
    {
        System.out.println("Child class init block 1");
    }

    public Child() {
        System.out.println("Child constructor called");
    }

    {
        System.out.println("Child class init block 2");
    }
}
```

```
public class InitBlockTest {  
    public static void main(String[] args) {  
        new Child();  
    }  
}
```

Output:

```
Parent class init block 1  
Parent class init block 2  
Parent constructor called  
Child class init block 1  
Child class init block 2  
Child constructor called
```

Example 3:

```
package com.test;

class Parent {
{
    System.out.println("Parent class init block 1");
}

static {
    System.out.println("Parent class static block 1");
}

public Parent() {
    System.out.println("Parent constructor called");
}

{
    System.out.println("Parent class init block 2");
}

static {
    System.out.println("Parent class static block 2");
}
}
```

```
class Child extends Parent {
{
    System.out.println("Child class init block 1");
}

static {
    System.out.println("Child class static block 1");
}

public Child() {
    System.out.println("Child constructor called");
}

{
    System.out.println("Child class init block 2");
}

static {
    System.out.println("Child class static block 2");
}
}

public class InitBlockTest {

    public static void main(String[] args) {
        new Child();
    }
}
```

Output:

Parent class static block 1
Parent class static block 2
Child class static block 1
Child class static block 2
Parent class init block 1
Parent class init block 2
Parent constructor called
Child class init block 1
Child class init block 2
Child constructor called

Order of execution :

- static blocks of super classes
- static blocks of the class
- init blocks of super classes
- constructors of super classes
- init blocks of the class
- constructors of the class

1. The code in static initialization block will be executed at class load time (and yes, that means only once per class load), before any instances of the class are constructed and before any static methods are called.
2. The instance initialization block is actually copied by the Java compiler into every constructor the class has. So,

the code in instance initialization block is executed **exactly** before the code in constructor.

Question 50: What is called first, constructor or init block?

Answer: Constructor is invoked first. Compiler copies all the code of instance initializer block into the constructor after first statement super().

Question 51: What is Variable shadowing and Variable hiding in Java?

Answer: **Variable shadowing** : When a local variable inside a method has the same name as one of the instance variables, the local variable shadows the instance variable inside the method block.

Example 1:

```
public class Test {  
    //instance variables  
    String name = "Mike";  
    int age = 15;  
  
    public void display() {  
        //local variables  
        String name = "John";  
        int age = 20;  
  
        System.out.println("Name : " + name);  
        System.out.println("Age : " + age);  
    }  
  
    public static void main(String[] args) {  
        Test obj = new Test();  
        obj.display();  
    }  
}
```

Output:

Name : John

Age : 20

If you want to access instance variables then you can do so using ‘this’ keyword like below:

Example 2:

```
public class Test {  
    //instance variables  
    String name = "Mike";  
    int age = 15;  
  
    public void display() {  
        //local variables  
        String name = "John";  
        int age = 20;  
  
        System.out.println("Name : " + name);  
        System.out.println("Age : " + age);  
        System.out.println("Name : " + this.name);  
        System.out.println("Age : " + this.age);  
    }  
    public static void main(String[] args) {  
        Test obj = new Test();  
        obj.display();  
    }  
}
```

Output:

```
Name : John  
Age : 20  
Name : Mike  
Age : 15
```

Variable Hiding : When the child and parent classes both have a variable with the same name, the child class variable hides the parent class variable.

Example 1:

```
package com.test;

class ParentClass {
    String x = "Parent's x";

    public void print() {
        System.out.println(x);
    }
}

class ChildClass extends ParentClass {
    String x = "Child's x";

    public void print() {
        System.out.println(x);
    }
}

public class Test {
    public static void main(String[] args) {
        ParentClass obj = new ChildClass();
        obj.print();
    }
}
```

Output:

Child's x

If you want to access parent's class variable then you can do this using super keyword:

Example 2:

```
package com.test;

class ParentClass {
    String x = "Parent's x";

    public void print() {
        System.out.println(x);
    }
}

class ChildClass extends ParentClass {
    String x = "Child's x";

    public void print() {
        System.out.println(x);
        System.out.println(super.x);
    }
}

public class Test {
    public static void main(String[] args) {
        ParentClass obj = new ChildClass();
        obj.print();
    }
}
```

Output:

Child's x
Parent's x

Variable hiding is not same as Method Overriding :

While variable hiding looks like overriding a variable (similar to method overriding), it is not. Overriding is applicable only to methods while hiding is applicable to variables.

In the case of method overriding, overridden methods completely replace the inherited methods, so when we try to access the method from a parent's reference by holding a child's object, the method from the child class gets called.

But in variable hiding, the child class hides the inherited variables instead of replacing them, so when we try to access the variable from the parent's reference by holding the child's object, it will be accessed from the parent class.

When an instance variable in a subclass has the same name as an instance variable in a super class, then the instance variable is chosen from the reference type.

```
package com.test;

class ParentClass {
    String x = "Parent's x";

    public void print() {
        System.out.println(x);
    }
}

class ChildClass extends ParentClass {
    String x = "Child's x";

    public void print() {
        System.out.println(x);
    }
}
```

```
public class Test {  
    public static void main(String[] args) {  
        ParentClass obj = new ChildClass();  
        System.out.println(obj.x);  
        obj.print();  
    }  
}
```

Output:

Parent's x
Child's x

Question 52: What is a constant and how we create constants in Java?

Answer: A constant is a variable whose value cannot change once it has been assigned. To make any variable a constant, we can use ‘static’ and ‘final’ modifier like below:

```
public static final TYPE NAME_OF_CONSTANT_VARIABLE = VALUE;
```

We can also use “enum” to define constants.

Question 53: Explain enum

Answer: enum in java is a data type which contains a fixed set of constants. In enum, we can also add variables, methods and constructors. Some common examples of enums are: days of week, colors, excel report columns etc.

Some points to remember:

- enum constants are static and final implicitly
- enum improves type safety
- enum can be declared inside or outside of a class
- enum can have fields, constructors (private) and methods
- enum cannot extend any class because it already extends Enum class implicitly but it can implement many interfaces
- We can use enum in switch statement
- We can have main() method inside an enum
- enum has values(), ordinal() and valueOf() methods. values() return an array containing all values present inside enum, ordinal() method returns the index of given enum value and valueOf() method returns the value of given constant enum
- enum can be traversed
- enum can have abstract methods
- enum cannot be instantiated because it contains private constructor only
- The constructor is executed for each enum constant at the time of enum class loading
- While defining enum, constants should be declared first, prior to any fields or methods, or else compile time error will come

Example 1:

main() method in enum, iterating over enum:

```
public enum Color {  
    RED, GREEN, BLUE, BLACK;  
  
    private Color() {  
        System.out.println("Constructor called for : " + this.toString());  
    }  
  
    public static void main(String[] args) {  
        for(Color c : Color.values()) {  
            System.out.println(c + " at index : " + c.ordinal());  
        }  
    }  
}
```

Output:

```
Constructor called for : RED  
Constructor called for : GREEN  
Constructor called for : BLUE  
Constructor called for : BLACK  
RED at index : 0  
GREEN at index : 1  
BLUE at index : 2  
BLACK at index : 3
```

Example 2:

```
package com.enumeration.demo;

enum Employee {
    Mike(15), John(20), Lisa(12), Dave(25);
    private int age;
    int getAge() {
        return age;
    }
    private Employee(int age) {
        this.age = age;
        System.out.println("Constructor called for : " + this.toString());
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.println("Age of Lisa is : " + Employee.Lisa.getAge());
    }
}
```

Output:

```
Constructor called for : Mike
Constructor called for : John
Constructor called for : Lisa
Constructor called for : Dave
Age of Lisa is : 12
```

Question 54: What is Cloneable?

Answer: Cloneable is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.

A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

If you try to Clone an object which doesn't implement the Cloneable interface, it will throw CloneNotSupportedException.

Example without implementing Cloneable interface:

Program 1:

```
public class Employee {  
  
    private String name;  
    private int age;  
  
    public Employee(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
    public int getAge() { return age; }  
    public void setName(String name) { this.name = name; }  
    public void setAge(int age) { this.age = age; }  
  
    public static void main(String[] args) {  
        Employee e1 = new Employee("Mike", 10);  
        System.out.println("Employee 1, name : " + e1.getName());  
  
        Employee e2 = e1;  
        e2.setName("John");  
  
        System.out.println("Employee 1, name : " + e1.getName());  
        System.out.println("Employee 2, name : " + e2.getName());  
    }  
}
```

Output:

```
Employee 1, name : Mike  
Employee 1, name : John  
Employee 2, name : John
```

Here, we have created Employee object e1 with new keyword but then we created another object Employee e2 which has the same reference as of

e1. So, any change in e2 object will reflect in e1 object and vice-versa.

Now, let's implement Cloneable interface in our Employee class and invoke the clone() method on e1 object to make its clone:

Program 2:

```
public class Employee implements Cloneable{
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }

    public static void main(String[] args)
        throws CloneNotSupportedException {

        Employee e1 = new Employee("Mike", 10);
        System.out.println("Employee 1, name : " + e1.getName());

        Employee e2 = (Employee) e1.clone();
        e2.setName("John");

        System.out.println("Employee 1, name : " + e1.getName());
        System.out.println("Employee 2, name : " + e2.getName());
    }
}
```

Output:

```
Employee 1, name : Mike
Employee 1, name : Mike
Employee 2, name : John
```

In the above example, we have only primitive types in our Employee class, what if we have an object type i.e. another class object reference, see the example below:

Program 3:

```
package com.clonealbe.demo;

class Company {
    private String name;
    public Company(String name) {
        this.name = name;
    }
    public String getName() {    return name;    }
    public void setName(String name) {    this.name = name;    }
}

public class Employee implements Cloneable{
    private String name;
    private int age;
    private Company company;

    public Employee(String name, int age, Company company) {
        this.name = name;
        this.age = age;
        this.company = company;
    }
    public String getName() {    return name;    }
    public int getAge() {    return age;    }
    public void setName(String name) {    this.name = name;    }
    public void setAge(int age) {    this.age = age;    }
    public Company getCompany() {    return company;    }
    public void setCompany(Company company) {    this.company = company;    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Company c1 = new Company("Company_ABC");
        Employee e1 = new Employee("Mike", 10, c1);
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());

        Employee e2 = (Employee) e1.clone();
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
        e2.getCompany().setName("XYZ");
        System.out.println("-----");
        System.out.println("Employee 1, company name : " + e1.getCompany().getName());
        System.out.println("Employee 2, company name : " + e2.getCompany().getName());
    }
}
```

Can you guess the output of the last 2 sysout? Here is the output:

Employee 1, company name : Company_ABC

Employee 2, company name : Company_ABC

Employee 1, company name : XYZ

Employee 2, company name : XYZ

If you are surprised with the above output, then let me make it clear by saying that, by default Object's clone() method provide Shallow copy. This brings us to the next interview question: What is shallow copy and deep copy

Question 55: What is Shallow Copy and Deep Copy?

Answer: **Shallow Copy** : When we use the default implementation of clone() method, a shallow copy of object is returned, meaning if the object that we are trying to clone contains both primitive variables and non-primitive or reference type variable, then only the object's reference is copied not the entire object itself.

Consider this with the example:

Employee object is having Company object as a reference, now when we perform cloning on Employee object, then for primitive type variables, cloning will be done i.e. new instances will be created and copied to the cloned object but for non-primitive i.e. Company object, only the object's reference will be copied to the cloned object. It simply means Company object will be same in both original and cloned object, changing the value in one will change the value in other and vice-versa.

Now, if you want to clone the Company object also, so that your original and cloned Employee object will be independent of each other, then you have to perform Deep Copy.

Deep Copy : in Deep copy, the non-primitive types are also cloned to make the original and cloned object fully independent of each other.

Program 1:

```

package com.clonealbe.demo;

class Company implements Cloneable {
    private String name;
    public Company(String name) {
        this.name = name;
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Employee implements Cloneable {
    private String name;
    private int age;
    private Company company;

    public Employee(String name, int age, Company company) {
        this.name = name;
        this.age = age;
        this.company = company;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
    public Company getCompany() { return company; }
    public void setCompany(Company company) { this.company = company; }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Employee employee = (Employee) super.clone();
        employee.company = (Company) company.clone();
        return employee;
    }
}

public static void main(String[] args) throws CloneNotSupportedException {
    Company c1 = new Company("Company_ABC");
    Employee e1 = new Employee("Mike", 10, c1);
    System.out.println("Employee 1, company name : " + e1.getCompany().getName());

    Employee e2 = (Employee) e1.clone();
    System.out.println("Employee 2, company name : " + e2.getCompany().getName());
    e2.getCompany().setName("XYZ");
    System.out.println("-----");
    System.out.println("Employee 1, company name : " + e1.getCompany().getName());
    System.out.println("Employee 2, company name : " + e2.getCompany().getName());
}
}

```

Output:

```
Employee 1, company name : Company_ABC  
Employee 2, company name : Company_ABC  
-----  
Employee 1, company name : Company_ABC  
Employee 2, company name : XYZ
```

In above example, we have overridden the clone method in our employee class and we called the clone method on mutable company object.

We can also use ***Copy constructor*** to perform deep copy:

Program 2:

```
package com.clonealbe.demo;  
  
class Company {  
    private String name;  
    public Company(String name) {  
        this.name = name;  
    }  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
}  
  
public class Employee {  
    private String name;  
    private int age;  
    private Company company;  
    public Employee(String name, int age, Company company) {  
        this.name = name;  
        this.age = age;  
        this.company = company;  
    }  
  
    //Copy constructor  
    public Employee(Employee emp) {  
        this.name = emp.getName();  
        this.age = emp.getAge();  
        Company company = new Company(emp.getCompany().getName());  
        this.company = company;  
    }  
}
```

```

public String getName() { return name; }
public int getAge() { return age; }
public void setName(String name) { this.name = name; }
public void setAge(int age) { this.age = age; }
public Company getCompany() { return company; }
public void setCompany(Company company) { this.company = company; }

public static void main(String[] args) {
    Company c1 = new Company("Company_ABC");
    Employee e1 = new Employee("Mike", 10, c1);
    System.out.println("Employee 1, company name : " + e1.getCompany().getName());

    //Invoking copy constructor
    Employee e2 = new Employee(e1);
    System.out.println("Employee 2, company name : " + e2.getCompany().getName());
    e2.getCompany().setName("XYZ");
    System.out.println("-----");
    System.out.println("Employee 1, company name : " + e1.getCompany().getName());
    System.out.println("Employee 2, company name : " + e2.getCompany().getName());
}
}

```

Output:

```

Employee 1, company name : Company_ABC
Employee 2, company name : Company_ABC
-----
Employee 1, company name : Company_ABC
Employee 2, company name : XYZ

```

There are 2 other methods by which you can perform deep copy:

- By using **Serialization**, where you serialize the original object and returns the deserialized object as a clone
- By using external library of Apache Commons Lang. Apache Common Lang comes with `SerializationUtils.clone()` method for performing deep copy on an object. It expects all classes in the hierarchy to implement `Serializable` interfaces else `SerializableException` is thrown by the system

Question 56: What is *Serialization* and *De-serialization*?

Answer: Serialization is a mechanism to convert the state of an object into a byte stream while De-serialization is the reverse process where the byte stream is used to recreate the actual object in memory. The byte stream created is platform independent that means objects serialized on one platform can be deserialized on another platform.

To make a Java Object serializable, the class must implement Serializable interface. Serializable is a Marker interface. ObjectOutputStream and ObjectInputStream classes are used for Serialization and Deserialization in java.

We will serialize the below Employee class:

```
import java.io.Serializable;

public class Employee implements Serializable{
    private String name;
    private int age;
    private transient int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
               + ", salary=" + salary + "]";
    }
}
```

SerializationDemo.java:

```

public class SerializationDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("Mike", 15, 20000);
        String file = "C:\\temp\\byteStream.txt";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);

            fos.close();
            oos.close();
            System.out.println("Employee object is serialized : " + emp);
        } catch (IOException e1) {
            System.out.println("IOException is caught");
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            System.out.println("IOException is caught");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException is caught");
        }
    }
}

```

Output:

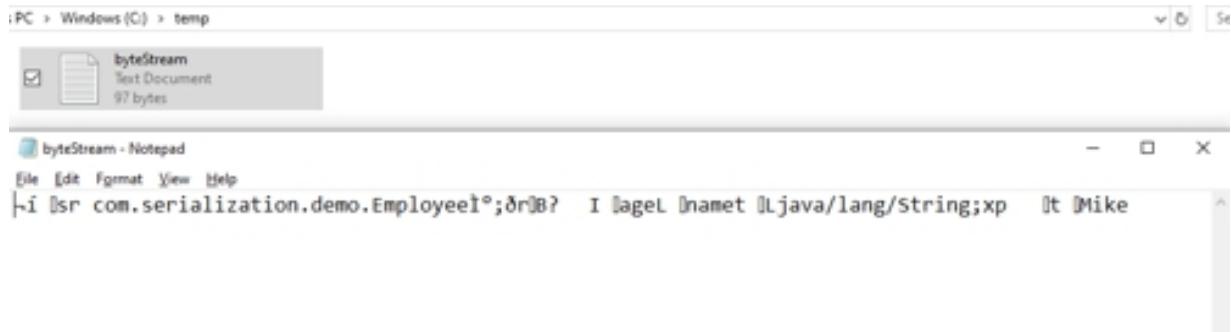
```

Employee object is serialized : Employee [name=Mike, age=15, salary=20000]
Employee object is de-serialized : Employee [name=Mike, age=15, salary=0]

```

Here, while de-serializing the employee object, salary is 0, that is because we have made salary variable to be ‘transient’. ‘static’ and ‘transient’ variables do not take part in Serialization process. During de-serialization, transient variables will be initialized with their default values i.e. if objects, it will be null and if “int”, it will be 0 and static variables will be having the current value.

And if you look at the file present in C:/temp/bytestream.txt, you can see how the object is serialized into this file,



Question 57: What is SerialVersionUID?

Answer: **SerialVersionUID** : The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must have the same SerialVersionUID, otherwise, *InvalidClassException* will be thrown when you deserialize the object. A Serializable class can declare its own UID explicitly by declaring a field. It must be static, final and of type long. *Remember, there is an exception for SerialVersionUID that although it is static, it gets serialized too, so that at the object deserialization the sender and receiver can be verified.*

If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class. This default serialVersionUID gets changed, when you add a new field or remove the transient keyword from a variable or convert the static variable to non-static variable. And if you are modifying the class structure after Serialization has been done, you will not be able to deserialize the object, see the example below:

Let's change the Employee class in our previous example and remove the transient keyword from salary variable:

Program 1:

```

public class Employee implements Serializable {
    private String name;
    private int age;
    private int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
               + ", salary=" + salary + "]";
    }
}

```

We have already serialized the Employee object in our previous example, now let's try to de-serialize it back:

```

public class DeserializationTest {
    public static void main(String[] args) {
        String file = "C:\\temp\\byteStream.txt";
        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

```

java.io.InvalidClassException : com.serialization.demo.Employee; local
class incompatible: stream classdesc serialVersionUID =
-3697389390179909057, local class serialVersionUID = -3759917827722067163
at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)

```

```
at java.io.ObjectInputStream.readClassDesc(Unknown Source)
at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
at java.io.ObjectInputStream.readObject0(Unknown Source)
at java.io.ObjectInputStream.readObject(Unknown Source)
at com.serialization.demo.DeserializationTest.main()
DeserializationTest.java:13 )
```

Hence, it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, in case you are using an IDE and not giving any serialVersionUID, compiler will give you a warning like below:



Example with SerialVersionUID:

Program 2:

```
public class Employee implements Serializable {

    private static final long serialVersionUID = 21L;

    private String name;
    private int age;
    private transient int salary;
    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
               + ", salary=" + salary + "]";
    }
}
```

SerializationDemo.java:

```

public class SerializationDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("John", 20, 31000);
        String file = "C:\\temp\\emp.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);

            fos.close();
            oos.close();
            System.out.println("Employee object is serialized : " + emp);
        } catch (IOException e1) {
            System.out.println("IOException is caught");
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            System.out.println("IOException is caught");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException is caught");
        }
    }
}

```

Output:

```

Employee object is serialized : Employee [name=John, age=20, salary=31000]
Employee object is de-serialized : Employee [name=John, age=20, salary=0]

```

Now, let's add one more field 'company' and remove the transient keyword from our Employee class. Here, as we are changing the class structure, let's see if we get the error of *InvalidClassException* again:

Program 3:

```
public class Employee implements Serializable {  
    private static final long serialVersionUID = 21L;  
  
    private String name;  
    private int age;  
    private int salary;  
    private String companyName;  
    public Employee(String name, int age, int salary) {  
        this.name = name;  
        this.age = age;  
        this.salary = salary;  
    }  
    @Override  
    public String toString() {  
        return "Employee [name=" + name + ", age=" + age  
               + ", salary=" + salary + ", companyName=" + companyName + "]";  
    }  
}
```

DeserializationTest.java:

```
public class DeserializationTest {  
    public static void main(String[] args) {  
        String file = "C:\\temp\\emp.ser";  
        try {  
            FileInputStream fis = new FileInputStream(file);  
            ObjectInputStream ois = new ObjectInputStream(fis);  
            Employee emp1 = (Employee) ois.readObject();  
  
            fis.close();  
            ois.close();  
            System.out.println("Employee object is de-serialized : " + emp1);  
        } catch (IOException e) {  
            e.printStackTrace();  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Employee object is de-serialized : Employee [name=John, age=20, salary=0, companyName=null]
```

Some Points to remember:

- If a parent class has implemented Serializable interface then child class doesn't need to implement it but the reverse is not true
- Static data members and transient data members are not saved via Serialization process (serialVersionUID is an exception). So, if you don't want to save value of a non-static data member then make it transient
- Constructor of serialized class is never called when the serialized object is deserialized (in case of inheritance, no-arg constructor of parent gets called during de-serialization)

Question 58: Serialization scenarios with Inheritance

Case 1: If super class is Serializable then by default, its sub-classes are also Serializable

```

class Parent implements Serializable {
    int x;
    public Parent(int x) {
        this.x = x;
    }
}

class Child extends Parent {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
    }
}

public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(10,50);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
}

```

```
private static void serializeObject(String file, Child child) {  
    try {  
        FileOutputStream fos = new FileOutputStream(file);  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        oos.writeObject(child);  
        fos.close();  
        oos.close();  
        System.out.println("The object has been serialized");  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
  
private static void deserializeObject(String file) {  
    try {  
        FileInputStream fis = new FileInputStream(file);  
        ObjectInputStream ois = new ObjectInputStream(fis);  
        Child child1 = (Child) ois.readObject();  
        fis.close();  
        ois.close();  
        System.out.println("The object has been deserialized");  
        System.out.println("x : " + child1.x);  
        System.out.println("y : " + child1.y);  
    } catch (IOException e) {  
        e.printStackTrace();  
    } catch (ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
}  
}
```

Output:

x : 10

y : 50

The object has been serialized

The object has been deserialized

x : 10

y : 50

Case 2: When super class does not implement the Serializable Interface, then also we can serialize the subclass provided that it implements Serializable interface.

In this case, when we de-serialize the subclass object, then no-arg constructor of its parent class gets called. So, the serializable sub-class must have access to the default no-arg constructor of its parent class (general rule is that the Serializable sub-class must have access to the no-arg constructor of first non-Serializable super class).

```
class Parent {  
    int x;  
    public Parent(int x) {  
        this.x = x;  
    }  
}  
  
class Child extends Parent implements Serializable {  
    int y;  
    public Child(int x, int y) {  
        super(x);  
        this.y = y;  
    }  
}  
  
public class TestSerialization {  
    public static void main(String[] args) {  
        Child child = new Child(20,40);  
        System.out.println("x : " + child.x);  
        System.out.println("y : " + child.y);  
        String file = "C:\\temp\\child1.ser";  
  
        serializeObject(file, child);  
        deserializeObject(file);  
    }  
}
```

(Note: serializeObject() and deserializeObject() remains same as the Case 1 program)

Output:

```
x : 20
y : 40
The object has been serialized
java.io.InvalidClassException: com.serialization.demo.Child; no valid constructor
    at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(Unknown Source)
    at java.io.ObjectStreamClass.checkDeserialize(Unknown Source)
    at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
    at java.io.ObjectInputStream.readObject0(Unknown Source)
    at java.io.ObjectInputStream.readObject(Unknown Source)
    at com.serialization.demo.TestSerialization.deserializeObject(TestSerialization.java:54)
    at com.serialization.demo.TestSerialization.main(TestSerialization.java:33)
```

When no-arg constructor is present in Super class:

```
class Parent {
    int x;
    public Parent(int x) {
        this.x = x;
        System.out.println("Parent class one-arg constructor");
    }
    public Parent() {
        x = 100;
        System.out.println("Parent class no-arg constructor");
    }
}

class Child extends Parent implements Serializable {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
        System.out.println("Child class two-arg constructor");
    }
    public Child() {
        System.out.println("Child class no-arg constructor");
    }
}
```

```
public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(20,40);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child2.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
}
```

(Note: serializeObject() and deserializeObject() remains same as the Case 1 program)

Output:

```
Parent class one-arg constructor
Child class two-arg constructor
x : 20
y : 40
The object has been serialized
Parent class no-arg constructor
The object has been deserialized
x : 100
y : 40
```

Question 59: Stopping Serialization and De-serialization

Suppose, parent class implements Serializable interface but we don't need the child class to be serialized

Here, we can implement writeObject() and readObject() methods in sub-class and throw NotSerializableException from these methods :

```

class Parent implements Serializable {
    int x;
    public Parent(int x) {
        this.x = x;
    }
}

class Child extends Parent {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
    }
    private void writeObject(ObjectOutputStream oos) throws IOException {
        throw new NotSerializableException("Serialization is not supported on this object");
    }
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        throw new NotSerializableException("Serialization is not supported on this object");
    }
}

public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(5,25);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child3.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
}

```

(Note: serializeObject() and deserializeObject() remains same as the Question 58 - Case 1 program)

Output:

```

x : 5
y : 25
java.io.NotSerializableException: Serialization is not supported on this object
    at com.serialization.demo.Child.writeObject(TestSerialization.java:25)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at java.io.ObjectStreamClass.invokeWriteObject(Unknown Source)
    at java.io.ObjectOutputStream.writeSerialData(Unknown Source)
    at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)
    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
    at java.io.ObjectOutputStream.writeObject(Unknown Source)
    at com.serialization.demo.TestSerialization.serializeObject(TestSerialization.java:47)
    at com.serialization.demo.TestSerialization.main(TestSerialization.java:39)

```

```
java.io.NotSerializableException: Serialization is not supported on this object
    at com.serialization.demo.Child.readObject(TestSerialization.java:28)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at java.io.ObjectStreamClass.invokeReadObject(Unknown Source)
    at java.io.ObjectInputStream.readSerialData(Unknown Source)
    at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
    at java.io.ObjectInputStream.readObject0(Unknown Source)
    at java.io.ObjectInputStream.readObject(Unknown Source)
    at com.serialization.demo.TestSerialization.deserializeObject(TestSerialization.java:61)
    at com.serialization.demo.TestSerialization.main(TestSerialization.java:40)
```

Serialization and De-serialization process can be customized also by providing writeObject() and readObject() methods in the class that we want to serialize.

```
private void writeObject(ObjectOutputStream oos) throws IOException {
    // Any Custom logic
    oos.defaultWriteObject();
    // Any Custom logic
}
private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
    // Any Custom logic
    ois.defaultReadObject();
    // Any Custom logic
}
```

Declaring both methods as private is necessary (public methods will not work), so other than the JVM nothing else can see them. This also proves that neither method is not inherited nor overridden or overloaded. The JVM automatically checks these methods and calls them during the serialization-deserialization process. The JVM can call these private methods, but other objects cannot. Thus, the integrity of the class is maintained and the serialization protocol can continue to work as normal.

For example, you can have encryption and decryption logic in these methods.

Question 60: What is Externalizable Interface?

Answer: The default serialization process is very slow as it is fully recursive, so whenever we try to serialize one object, the serialization process tries to serialize all the fields of our class (except static and transient variables). So, if we have a class with lots of variables present and we do not want to serialize all of them, we have to make all of those variables as transient, all

these fields will always be assigned with default values. This makes the entire process very slow.

Externalizable interface is used when we want to implement custom logic to serialize/deserialize an object. Externalizable interface extends the Serializable interface, and it has two methods, writeExternal() and readExternal() which are used for serialization and de-serialization. This way, we can change the JVM's default serialization behavior because while using Externalizable, we decide what to store in stream.

Program 1:

Employee.java:

```
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Employee implements Externalizable {
    private String name;
    private int age;
    private transient int salary;
    private String companyName;
    public Employee() {
        System.out.println("No-arg constructor of Employee class");
    }

    public Employee(String name, int age, int salary, String companyName) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        this.companyName = companyName;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + ", salary=" +
               salary + ", companyName=" + companyName + "]";
    }
}
```

```

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(name);
    out.writeInt(age);
    out.writeInt(salary);
}
@Override
public void readExternal(ObjectInput in) throws IOException,
                                ClassNotFoundException {
    name = (String) in.readObject();
    age = in.readInt();
    salary = in.readInt();
}
}

```

TestExternalizable.java:

```

public class TestExternalizable {

    public static void main(String[] args) {
        Employee emp = new Employee("Mike", 15, 25000, "ABC");
        String file = "C:\\\\temp\\\\object.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);

            fos.close();
            oos.close();
            System.out.println("Employee object is serialized : \n" + emp);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : \n" + emp1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

```
Employee object is serialized :  
Employee [name=Mike, age=15, salary=25000, companyName=ABC]  
No-arg constructor of Employee class  
Employee object is de-serialized :  
Employee [name=Mike, age=15, salary=25000, companyName=null]
```

Now, one thing to remember here is that the public no-arg constructor gets called before readExternal() method, so we have to provide this no-arg constructor or else we will get an exception during run-time.

Comment the public no-arg constructor from Employee.java:

Program 2:

```
public class Employee implements Externalizable {  
    private String name;  
    private int age;  
    private transient int salary;  
    private String companyName;  
    // public Employee() {  
    //     System.out.println("No-arg constructor of Employee class");  
    // }
```

Now, run TestExternalizable.java, you will get below output:

```
Employee object is serialized :  
Employee [name=Mike, age=15, salary=25000, companyName=ABC]  
java.io.InvalidClassException: com.externalizable.demo.Employee; no valid constructor  
    at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(Unknown Source)  
    at java.io.ObjectStreamClass.checkDeserialize(Unknown Source)  
    at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)  
    at java.io.ObjectInputStream.readObject0(Unknown Source)  
    at java.io.ObjectInputStream.readObject(Unknown Source)  
    at com.externalizable.demo.TestExternalizable.main(TestExternalizable.java:28)
```

So, if interviewer asks a question that you have a class which has 1000 variables and you want to serialize only 10 specific variables, your answer should be using an Externalizable interface.

Some points to remember:

- Using Externalizable interface, we can implement custom logic for serialization and deserialization of object
- When using Externalizable, we have to explicitly mention what fields or variables we want to serialize
- When using Externalizable, a public no-arg constructor is required
- Using Externalizable, we can also serialize transient and static variables
- `readExternal()` method must read the values in the same sequence and with the same types as were written by `writeExternal()` method

Question 61: Externalizable with Inheritance

If a class is implementing Externalizable and also has a parent class, how we can save the data of parent class and how we can recover it back. In case of Externalizable, we have to implement the `readExternal()` and `writeExternal()` methods in each and every class we want to serialize or deserialize.

Case 1 : When both parent and child classes are implementing Externalizable interfaces:

Department.java:

```
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Department implements Externalizable {
    private int deptId;
    private String deptName;
    private int capacity;

    public Department() {
        System.out.println("No-arg constructor of Department class");
    }

    public void writeExternal(ObjectOutput output) throws IOException {
        output.writeInt(deptId);
        output.writeObject(deptName);
        output.writeInt(capacity);
    }

    public void readExternal(ObjectInput input) throws IOException {
        deptId = input.readInt();
        deptName = (String) input.readObject();
        capacity = input.readInt();
    }
}
```

```
public void setDeptId(int deptId) { this.deptId = deptId; }
public void setDeptName(String deptName) { this.deptName = deptName; }
public void setCapacity(int capacity) { this.capacity = capacity; }

public int getDeptId() { return deptId; }
public String getDeptName() { return deptName; }
public int getCapacity() { return capacity; }

@Override
public String toString() {
    return "Department [deptId=" + deptId + ", deptName=" + deptName
           + ", capacity=" + capacity + "]";
}

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(deptId);
    out.writeUTF(deptName);
}
@Override
public void readExternal(ObjectInput in) throws IOException,
                               ClassNotFoundException {
    deptId = in.readInt();
    deptName = in.readUTF();
}
}
```

Student.java:

```
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Student extends Department implements Externalizable {

    private int studentId;
    private String studentName;
    private int age;

    public Student() {
        System.out.println("No-arg constructor of Student class");
    }
}
```

```

public void setStudentId(int studentId) { this.studentId = studentId; }
public void setStudentName(String studentName) { this.studentName = studentName; }
public void setAge(int age) { this.age = age; }

public int getStudentId() { return studentId; }
public String getStudentName() { return studentName; }
public int getAge() { return age; }
@Override
public String toString() {
    return super.toString() + "\n" +
        "Student [studentId=" + studentId + ", studentName=" + studentName
        + ", age=" + age + "]";
}

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    super.writeExternal(out);
    out.writeInt(studentId);
    out.writeUTF(studentName);
}
@Override
public void readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException {
    super.readExternal(in);
    studentId = in.readInt();
    studentName = in.readUTF();
}
}

```

Here, in the overridden writeExternal() and readExternal() methods in child class Student, we are also calling super.writeExternal() and super.readExternal() methods to serialize and de-serialize fields of parent class Department.

ExternalizableDemo.java:

```

public class ExternalizableDemo {
    public static void main(String[] args) {
        Student st = new Student();
        st.setStudentId(1);
        st.setStudentName("Lisa");
        st.setAge(20);
        st.setDeptId(10);
        st.setDeptName("IT");
        st.setCapacity(60);
        String file = "C:\\temp\\object1.ser";
    }
}

```

```

    try {
        FileOutputStream fos = new FileOutputStream(file);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(st);
        fos.close();
        oos.close();
        System.out.println("Student object is serialized : \n" + st);
    } catch (IOException e) {
        e.printStackTrace();
    }

    try {
        FileInputStream fis = new FileInputStream(file);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Student st1 = (Student) ois.readObject();
        fis.close();
        ois.close();
        System.out.println("Student object is de-serialized : \n" + st1);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

Output:

```

No-arg constructor of Department class
No-arg constructor of Student class
Student object is serialized :
Department [deptId=10, deptName=IT, capacity=60]
Student [studentId=1, studentName=Lisa, age=20]
No-arg constructor of Department class
No-arg constructor of Student class
Student object is de-serialized :
Department [deptId=10, deptName=IT, capacity=0]
Student [studentId=1, studentName=Lisa, age=0]

```

The capacity and age variable values are 0 because we did not serialize these two variables.

Case 2 : When only child class is implementing the Externalizable interface

Department.java:

```
package com.externalizable.demo;

public class Department {
    private int deptId;
    private String deptName;
    private int capacity;

    public Department() {
        System.out.println("No-arg constructor of Department class");
    }

    public void setDeptId(int deptId) { this.deptId = deptId; }
    public void setDeptName(String deptName) { this.deptName = deptName; }
    public void setCapacity(int capacity) { this.capacity = capacity; }

    public int getDeptId() { return deptId; }
    public String getDeptName() { return deptName; }
    public int getCapacity() { return capacity; }

    @Override
    public String toString() {
        return "Department [deptId=" + deptId + ", deptName=" + deptName
               + ", capacity=" + capacity + "]";
    }
}
```

Student.java:

```
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Student extends Department implements Externalizable {
    private int studentId;
    private String studentName;
    private int age;
    public Student() {
        System.out.println("No-arg constructor of Student class");
    }
}
```

```

public void setStudentId(int studentId) { this.studentId = studentId; }
public void setStudentName(String studentName) { this.studentName = studentName; }
public void setAge(int age) { this.age = age; }

public int getStudentId() { return studentId; }
public String getStudentName() { return studentName; }
public int getAge() { return age; }

@Override
public String toString() {
    return super.toString() + "\n" +
        "Student [studentId=" + studentId + ", studentName=" + studentName
        + ", age=" + age + "]";
}

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(getDeptId());
    out.writeUTF(getDeptName());
    out.writeInt(studentId);
    out.writeUTF(studentName);
}
@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    setDeptId(in.readInt());
    setDeptName(in.readUTF());
    studentId = in.readInt();
    studentName = in.readUTF();
}
}

```

Here, we are using getters and setters of parent class, Department, to serialize and de-serialize its fields.

ExternalizableDemo.java:

```

public class ExternalizableDemo {
    public static void main(String[] args) {
        Student st = new Student();
        st.setStudentId(1);
        st.setStudentName("Lisa");
        st.setAge(20);
        st.setDeptId(10);
        st.setDeptName("IT");
        st.setCapacity(60);
        String file = "C:\\temp\\object2.ser";
    }
}

```

```

try {
    FileOutputStream fos = new FileOutputStream(file);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(st);
    fos.close();
    oos.close();
    System.out.println("Student object is serialized : \n" + st);
} catch (IOException e) {
    e.printStackTrace();
}

try {
    FileInputStream fis = new FileInputStream(file);
    ObjectInputStream ois = new ObjectInputStream(fis);
    Student st1 = (Student) ois.readObject();
    fis.close();
    ois.close();
    System.out.println("Student object is de-serialized : \n" + st1);
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
}

```

Output:

```

No-arg constructor of Department class
No-arg constructor of Student class
Student object is serialized :
Department [deptId=10, deptName=IT, capacity=60]
Student [studentId=1, studentName=Lisa, age=20]
No-arg constructor of Department class
No-arg constructor of Student class
Student object is de-serialized :
Department [deptId=10, deptName=IT, capacity=0]
Student [studentId=1, studentName=Lisa, age=0]

```

Question 62: Difference between Serializable and Externalizable

Answer:

- Serializable is a marker interface which means it does not contain any method whereas Externalizable is a child interface of Serializable and it contains two methods writeExternal() and readExternal()
- When using Serializable, JVM takes full responsibility for serializing the class object but in case of Externalizable, the programmer has full control over serialization logic
- Serializable interface is a better fit when we want to serialize the entire object whereas Externalizable is better suited for custom serialization
- Default serialization is easy to implement but it comes with some issues and performance cost whereas in case of Externalizable, the programmer has to provide the complete serialization logic which is a little hard but results in better performance
- Default serialization does not call any constructor whereas a public no-arg constructor is needed when using Externalizable interface
- When a class implements Serializable interface, it gets tied with default serialization which can easily break if structure of the class changes like adding/removing any variable whereas using Externalizable, you can create your own binary format for your object

Question 63: How to make a class Immutable?

Answer: As we know, String is an Immutable class in Java, i.e. once initialized its value never change. We can also make our own custom Immutable class, where the class object's state will not change once it is initialized.

Benefits of Immutable class:

- Thread-safe: With immutable classes, we don't have to worry about the thread-safety in case of multi-threaded environment as these classes are inherently thread-safe
- Cacheable: An immutable class is good for Caching because while we don't have to worry about the value changes

How to create an Immutable class in java:

- Declare the class as final so that it cannot be extended
- Make all fields as private so that direct access to them is not allowed
- Make all fields as final so that its value can be assigned only once
- Don't provide 'setter' methods for variables
- When the class contains a mutable object reference,
 1. While initializing the field in constructor, perform a deep copy
 2. While returning the object from its getter method, make sure to return a copy rather than the actual object reference

Example:

We will make Employee class as immutable, but Employee class contains a reference of Address class

Address.java:

```
package com.immutable.demo;

public class Address {
    private String city;
    private String state;

    public Address(String city, String state) {
        this.city = city;
        this.state = state;
    }
}
```

```

public String getCity() { return city; }
public String getState() { return state; }
public void setCity(String city) { this.city = city; }
public void setState(String state) { this.state = state; }

@Override
public String toString() {
    return "Address [city=" + city + ", state=" + state + "]";
}
}

```

Employee.java:

```

package com.immutable.demo;

public final class Employee {
    private final String name;
    private final int age;
    private final Address address;

    public Employee(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        Address cloneAddress = new Address(address.getCity(), address.getState());
        this.address = cloneAddress;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public Address getAddress() {
        return new Address(address.getCity(), address.getState());
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
               + ", address=" + address + "]";
    }
}

```

TestImmutable.java:

```

package com.immutable.demo;

public class TestImmutable {
    public static void main(String[] args) {
        Address address = new Address("Chennai", "Tamil Nadu");
        Employee employee = new Employee("Mike", 15, address);

        System.out.println("Original Employee object : \n" + employee);
    }
}

```

```

        address.setCity("Mumbai");
        address.setState("Maharashtra");
        System.out.println("Employee object after local variable address change :\n" + employee);

        Address empAddress = employee.getAddress();
        empAddress.setCity("Jaipur");
        empAddress.setState("Rajasthan");
        System.out.println("Employee object after employee address change:\n" + employee);
    }
}

```

Here, after creating Employee object, the first change is done in local address object and then we used the employee's getter method to access the address object and tried to change the value in it.

Output:

```

Original Employee object :
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
Employee object after local variable address change :
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
Employee object after employee address change:
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]

```

As, you can see that the value remained the same.

If we don't follow the rule about mutable object reference present in the class, let's see what will happen in that case.

Let's change the Employee class constructor and getter method:

Employee.java:

```

package com.immutable.demo;

public final class Employee {
    private final String name;
    private final int age;
    private final Address address;

    public Employee(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
}

```

```

public String getName() { return name; }
public int getAge() { return age; }
public Address getAddress() {
    return address;
}
@Override
public String toString() {
    return "Employee [name=" + name + ", age=" + age
           + ", address=" + address + "]";
}
}

```

Now, if we run our *TestImmutable.java* class, below is the output:

```

Original Employee object :
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
Employee object after local variable address change :
Employee [name=Mike, age=15, address=Address [city=Mumbai, state=Maharashtra]]
Employee object after employee address change:
Employee [name=Mike, age=15, address=Address [city=Jaipur, state=Rajasthan]]

```

Why we perform deep copy in constructor:

- When you assign the actual address object in the constructor, then remember it is storing the reference of address object, so if you change the value in this address object, it will reflect in the employee object

Why we don't return original reference from the getter:

- When you return the original address object from the getter method then you can use the returned object reference to change the values in employee object

Question 64: Explain Class loaders in Java

Answer: **ClassLoader** is a java class which is used to load .class files in memory. When we compile a java class, JVM creates a bytecode which is platform independent. The bytecode is present in .class file. When we try to use a class, then classloader loads it into memory.

There are 3 types of built-in class loaders in java:

1. *Bootstrap class loader* : it loads JDK class files from jre/lib/rt.jar and other core classes. It is the parent of all class loaders, it is also called Primordial classloader.
2. *Extensions class loader* : it loads classes from JDK extensions directory, it delegates class loading request to its parent, Bootstrap and if the loading of class is unsuccessful, it loads classes from jre/lib/ext directory or any other directory pointed by `java.ext.dirs` system property.
3. *System class loader* : It loads application specific classes from the CLASSPATH. We can set classpath while invoking the program using -cp or classpath command line options. It is a child of Extension ClassLoader.

Java class loader is based on three principles:

1. Delegation principle: It forwards the request for class loading to its parent class loader. It only loads the class if the parent does not find or load the class.
2. Visibility principle: According to Visibility principle, the child ClassLoader can see all the classes loaded by parent ClassLoader. But the parent class loader cannot see classes loaded by the child class loader.
3. Uniqueness principle: According to this principle, a class loaded by Parent should not be loaded by Child ClassLoader again. It is achieved by delegation principle.

Suppose, you have created a class Employee.java and compiled this class and Employee.class file is created. Now, you want to use this class, the first request to load this class will come to System/Application ClassLoader, which will delegate the request to its parent, Extension ClassLoader which further delegates to Primordial or Bootstrap class loader

Now, Bootstrap ClassLoader will look for this class in rt.jar, since this class is not there, the request will come to Extension ClassLoader which looks in jre/lib/ext directory and tries to locate this class there, if this class is found

there then Extension ClassLoader will load this class and Application ClassLoader will not load this class, this has been done to maintain the Uniqueness principle. But if the class is not loaded by Extension ClassLoader, then this Employee.class will be loaded by Application ClassLoader from the CLASSPATH.

Employee.java:

```
package com.classloader.demo;

public class Employee {

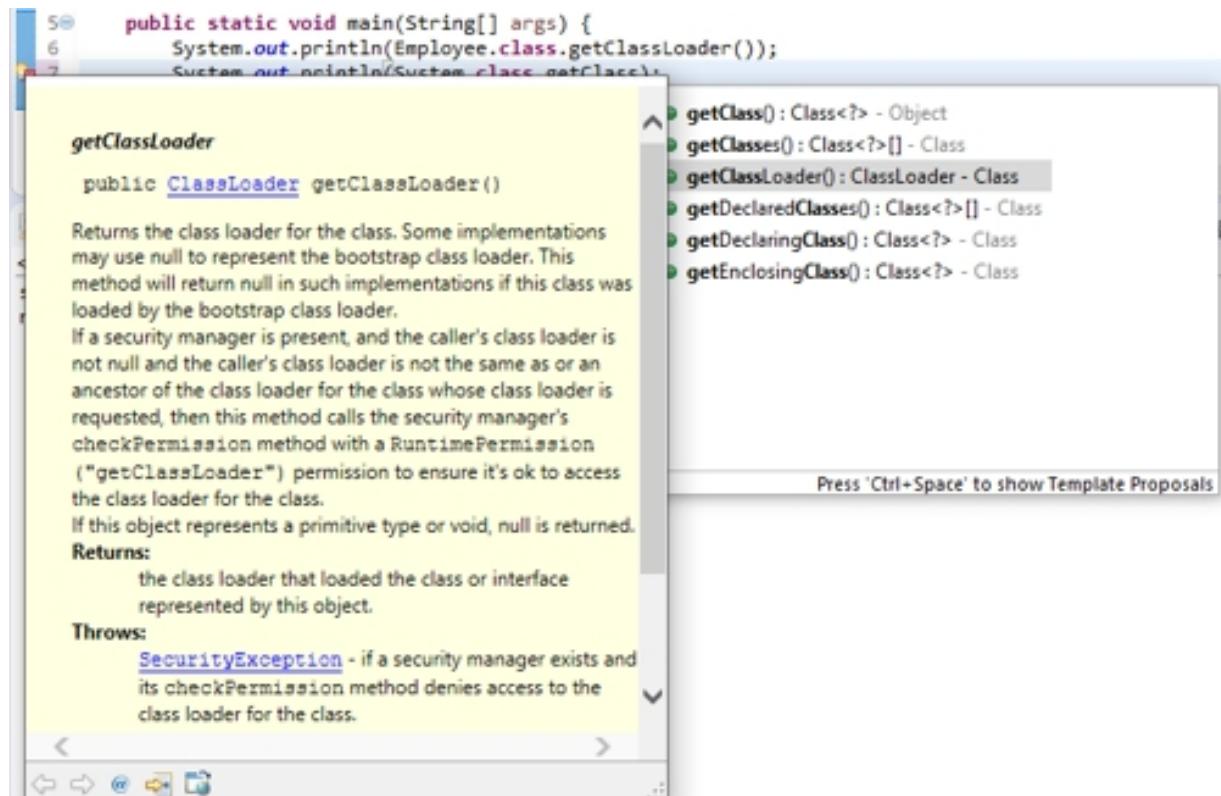
    public static void main(String[] args) {
        System.out.println(Employee.class.getClassLoader());
        System.out.println(System.class.getClassLoader());
    }

}
```

Output:

```
sun.misc.Launcher$AppClassLoader@73d16e93
null
```

If you are thinking why null is printed when we tried to know which classloader is loading the java.lang.System class then take a look at the Javadoc :



We can also create our own custom class loader by extending the `ClassLoader` class.

Question 65: What is Singleton Design Pattern and how it can be implemented?

Answer: This is a famous interview question, as it involves many follow-up questions as well. I will cover all of them here:

What is Singleton Design Pattern?

Singleton design pattern comes under Creational Design Patterns category and this pattern ensures that only one instance of class exists in the JVM.

Singleton pattern is used in:

- logging, caching, thread pool etc.
- other design patterns like Builder, Prototype, Abstract Factory etc.
- core java classes like `java.lang.Runtime` etc.

How to implement Singleton Pattern

To implement a Singleton pattern, we have different approaches but all of them have the below common concepts:

- private constructor to restrict instantiation of the class from other classes.
- private static variable of the same class that is the only instance of the class.
- public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

The approaches to implement singleton pattern are:

Eager Initialization : In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not even use it.

Example:

A.java:

```
public class A {  
    //private constructor  
    private A() { }  
  
    //only instance of this class  
    private static final A instance = new A();  
  
    //public static method to return the only instance  
    public static A getInstance() {  
        return instance;  
    }  
  
    public void print() {  
        System.out.println("Singleton class print method");  
    }  
}
```

Test.java:

```
public class Test {  
    public static void main(String[] args) {  
        A obj = A.getInstance();  
        obj.print();  
    }  
}
```

Output:

Singleton class print method

We can also access the instance by using the classname like A.instance because of static keyword but we will have to change its access modifier to ‘public’, so that it is visible outside the singleton class.

Let’s suppose you are creating a large object by using a lot of resources, there may be a chance that object creation may throw an exception but the above way of creating a singleton class does not provide any option for

exception handling as you can write try-catch only inside a block of code. There is a solution to tackle this particular problem where you can create the class instance inside a static block.

Static Block Eager Initialization:

```
public class A {  
    private A() {}  
  
    private static A instance;  
  
    static {  
        try{  
            instance = new A();  
        } catch (Exception e) {  
            throw new RuntimeException("Exception while creating singleton object");  
        }  
    }  
  
    public static A getInstance() {  
        return instance;  
    }  
}
```

Both the above approaches create the object even before it is used (initialized at the time of class loading because of static variable and static block), but there are other approaches where we can create a Singleton class instance in a lazy initialization way i.e. only when someone asks for it. These approaches are discussed below.

Lazy Initialization: using this way of creating Singleton class, the object will not get created unless someone asks for it. Here we will create the class instance inside the global access method.

```
public class A {  
  
    private A() { }  
  
    private static A instance;  
  
    public static A getInstance() {  
        if(instance == null) {  
            instance = new A();  
        }  
        return instance;  
    }  
}
```

This approach is suitable for only single-threaded application, suppose there are 2 threads and both have checked that the instance is null and now they are inside the “if(...)” condition, it will destroy our singleton pattern and both threads will have different instances. So, we must overcome this problem so that our singleton pattern doesn’t break in case of multi-threaded environment.

Thread Safe Singleton implementation: here the easiest way to prevent multiple threads from creating more than one instance is to make the global access method ‘synchronized’, this way threads will acquire a lock first before entering the getInstance() method.

```
public class A {  
  
    private A() { }  
  
    private static A instance;  
  
    public synchronized static A getInstance() {  
        if(instance == null) {  
            instance = new A();  
        }  
        return instance;  
    }  
}
```

Synchronizing the entire method comes with performance degradation, also acquiring the lock and releasing the lock on every call to getInstance() method seems un-necessary, because only first few calls to getInstance() method needs to be synchronized, what I mean to say by this statement is: let's suppose there are 10 threads that are trying to call getInstance() method, now you need to apply synchronization to only these 10 threads at this time and the thread which first acquires the lock will create the object. After that, every thread will get the same object because of null check in if condition, so we can optimize the above code by using **double-checked locking principle**, where we will use a synchronized block inside the if condition, like shown below:

```
public class A {  
  
    private A() {}  
  
    private static A instance;  
  
    public static A getInstance() {  
        if(instance == null) {  
            synchronized (A.class) {  
                if(instance == null) {  
                    instance = new A();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

The reason of second if condition inside the synchronized block : suppose there are 2 threads and both called the getInstance() method at the same time, now they will both be inside the first if condition as instance is null at this time, and the first thread which acquires the lock will create the object and as soon as it exits the synchronized block, other thread which was waiting, will acquire the lock and it will also create another object thus breaking the singleton pattern. This is why it is called “double-checked locking”.

Now, some people have faced issues with the above approach in java 1.4 and earlier versions, which was solved in later versions **by using ‘volatile’ keyword** with the above approach like below:

```
public class A {  
  
    private A() { }  
  
    private static volatile A instance;  
  
    public static A getInstance() {  
        A localRef = instance;  
        if(localRef == null) {  
            synchronized (A.class) {  
                localRef = instance;  
                if(localRef == null) {  
                    instance = localRef = new A();  
                }  
            }  
        }  
        return localRef;  
    }  
}
```

localRef variable is there for the cases where instance is already initialized (discussed above), the volatile field is only accessed once because we have written *return localRef* not *return instance* .

There is another approach where an inner static class is used to create the Singleton class instance and it is returned from the global access method. This approach is called ***Bill Pugh Singleton Implementation*** .

Bill Pugh Singleton Implementation Program:

```
public class A {  
  
    private A() { }  
  
    private static class InnerA {  
        private static final A instance = new A();  
    }  
  
    public static A getInstance() {  
        return InnerA.instance;  
    }  
}
```

The inner class does not get loaded at the time of class A loading, only when someone calls getInstance() method, it gets loaded and creates the Singleton instance.

Question 66: What are the different ways in which a Singleton Design pattern can break and how to prevent that from happening?

Answer: There are 3 ways which can break Singleton property of a class, they are discussed below with examples:

Reflection : Reflection API in java is used to change the runtime behavior of a class. Hibernate, Spring's Dependency injection also uses Reflection. So, even though in the above singleton implementations, we have defined the constructor as private, but using Reflection, even private constructor can be accessed, so Reflection can be used to break the singleton property of a class.

A.java:

```
package com.singleton.demo;

public class A {

    private A() { }

    public static final A instance = new A();

}
```

Notice, I am using public access modifier with the only instance of singleton class so that it can be accessed outside this class using just the class name.

Now, let's see how Reflection can break Singleton:

Test.java:

```
package com.singleton.demo;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Test {
    public static void main(String[] args) {

        A instance1 = A.instance;
        A instance2 = null;
        Constructor[] constructors = A.class.getDeclaredConstructors();
```

```
        for(Constructor constructor : constructors) {
            constructor.setAccessible(true);
            try {
                instance2 = (A) constructor.newInstance();
                break;
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Instance1 hashCode : " + instance1.hashCode());
        System.out.println("Instance2 hashCode : " + instance2.hashCode());
    }
}
```

Output:

```
Instance1 hashCode : 366712642
Instance2 hashCode : 1829164700
```

As we can see from the output, the 2 instances have different hashCode, thus destroying Singleton.

Now to prevent Singleton from Reflection, one simple solution is to throw an exception from the private constructor, so when Reflection tries to invoke private constructor, there will be an error.

```
package com.singleton.demo;

public class A {

    private A() {
        if(A.instance != null) {
            throw new InstantiationException("This object creation is not allowed");
        }
    }
    public static final A instance = new A();
}
```

Now, if you run *Test.java* , you will get below output:

```
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at com.singleton.demo.Test.main(Test.java:15)
Caused by: java.lang.InstantiationException: This object creation is not allowed
    at com.singleton.demo.A.<init>(A.java:7)
    ... 5 more
Instance1 hashCode : 2018699554Exception in thread "main" java.lang.NullPointerException
    at com.singleton.demo.Test.main(Test.java:29)
```

The other solution to prevent Singleton from Reflection is **using Enums**, as its constructor cannot be accessed via Reflection, JVM internally handles the creation and invocation of enum constructor

SingletonEnum.java:

```
package com.singleton.demo;

public enum SingletonEnum {

    INSTANCE;

    public void print() {
        System.out.println("Inside print method");
    }
}
```

TestSingletonEnum.java:

```
package com.singleton.demo;

public class TestSingletonEnum {

    public static void main(String[] args) {
        SingletonEnum.INSTANCE.print();
    }

}
```

Output:

Inside print method

Another way which can break Singleton property of a class is:

Serialization and Deserialization : Our Singleton class may implement Serializable interface so that the object's state can be saved and at a later point in time, it can be accessed back using Deserialization, now the problem here is, when we deserialize the object, a new instance of the class will be created, thus breaking the singleton pattern.

See, the example below:

A.java:

```
package com.singleton.demo;

import java.io.Serializable;

public class A implements Serializable{

    private static final long serialVersionUID = -2020L;

    private A() { }
    public static final A instance = new A();

}
```

Test.java:

```
public class Test {
    public static void main(String[] args) {

        A instance1 = A.instance;
        A instance2 = null;
        String file = "C:\\\\temp\\\\object.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(instance1);

            fos.close();
            oos.close();

            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            instance2 = (A) ois.readObject();

            fis.close();
            ois.close();
        } catch (Exception e) {
            System.out.println("Exception occurred");
        }

        System.out.println("Instance1 hashCode : " + instance1.hashCode());
        System.out.println("Instance2 hashCode : " + instance2.hashCode());
    }
}
```

Output:

```
Instance1 hashCode : 865113938
Instance2 hashCode : 2003749087
```

To prevent our Singleton class from Serialization, there is a method called ***readResolve()*** which is called when ObjectInputStream has read an object from the stream and is preparing to return it to the caller, so we can return the only instance of this class from this method, and this way the only instance of singleton will be assigned to instance2, see below:

A.java:

```
public class A implements Serializable {

    private static final long serialVersionUID = -2020L;

    private A() { }
    public static final A instance = new A();

    protected Object readResolve() {
        System.out.println("readResolve() method is called");
        return instance;
    }
}
```

Test.java:

```
public class Test {
    public static void main(String[] args) {

        A instance1 = A.instance;
        A instance2 = null;
        String file = "C:\\temp\\object1.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(instance1);

            fos.close();
            oos.close();
        }
```

```

        System.out.println("Starting De-serialization");
        FileInputStream fis = new FileInputStream(file);
        ObjectInputStream ois = new ObjectInputStream(fis);
        instance2 = (A) ois.readObject();
        System.out.println("De-serialization completes");
        fis.close();
        ois.close();
    } catch (Exception e) {
        System.out.println("Exception occurred");
    }

    System.out.println("Instance1 hashCode : " + instance1.hashCode());
    System.out.println("Instance2 hashCode : " + instance2.hashCode());
}
}

```

Output:

```

Starting De-serialization
readResolve() method is called
De-serialization completes
Instance1 hashCode : 865113938
Instance2 hashCode : 865113938

```

You can also throw an exception from the readResolve() method, but returning the only instance approach is better as your program execution will not stop.

One last way which can break Singleton property of a class is:

Cloning : As we know, Cloning is used to create duplicate objects (copy of the original object). If we create a clone of the instance of our Singleton class then a new instance will be created thus breaking our Singleton pattern.

See the program below:

TestSingleton.java:

```
package com.singleton.demo;

class Parent implements Cloneable {
    int x = 20;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Singleton extends Parent {
    public static final Singleton instance = new Singleton();

    private Singleton() { }
}

public class TestSingleton {
    public static void main(String[] args) throws CloneNotSupportedException {

        Singleton instance1 = Singleton.instance;
        Singleton instance2 = (Singleton) instance1.clone();

        System.out.println("Instance1 hashCode : " + instance1.hashCode());
        System.out.println("Instance2 hashCode : " + instance2.hashCode());
    }
}
```

Output:

Instance1 hashCode : 366712642
Instance2 hashCode : 1829164700

As you can see, both instances have different hashcodes indicating our Singleton pattern is broken, so to prevent this we can override clone method in our Singleton class and either return the same instance or throw CloneNotSupportedException from it.

See the program changes below:

```
class Singleton extends Parent {  
    public static final Singleton instance = new Singleton();  
  
    private Singleton() { }  
  
    @Override  
    protected Object clone() throws CloneNotSupportedException {  
        return new CloneNotSupportedException();  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.CloneNotSupportedException cannot be cast to com.singleton.demo.Singleton  
at com.singleton.demo.TestSingleton.main(TestSingleton.java:27)
```

clone() returning the same instance, see the program changes below:

```
class Singleton extends Parent {  
    public static final Singleton instance = new Singleton();  
  
    private Singleton() { }  
  
    @Override  
    protected Object clone() throws CloneNotSupportedException {  
        return instance;  
    }  
}
```

Output:

```
Instance1 hashCode : 366712642  
Instance2 hashCode : 366712642
```

Question 67: What are the different design patterns you have used in your projects?

Answer: You will be asked this question almost in all interviews nowadays. So, be prepared with some design patterns that are used in mostly all projects, like:

- Factory Design Pattern
- Abstract Factory Design Pattern
- Strategy Design Pattern
- Builder Design Pattern
- Singleton Design Pattern
- Observer Design Pattern

And any other if you have used in your projects.

Question 68: Explain Volatile keyword in java

Answer: Volatile is a keyword in java, that can be applied only to variables. You cannot apply volatile keyword to classes and methods. Applying volatile to a shared variable that is accessed in a multi-threaded environment ensures that threads will read this variable from the main memory instead of their own local cache.

Consider below code:

```
public class Test {  
  
    static int x = 10;  
  
}
```

Now, let's suppose 2 threads are working on this class and both threads are running on different processors having their own local copy of variable x. If any thread modifies its value, the change will not be reflected back in the original variable x in the main memory leading to data inconsistency because the other thread is not aware of the modification.

So, to prevent data inconsistency, we can make variable x as volatile:

```
public class Test {  
  
    static volatile int x = 10;  
  
}
```

Now, all the threads will read and write the variable x from/to the main memory. Using volatile, also prevents compiler from doing any reordering or any optimization to the code.

Question 69: What is Garbage Collection in Java, how it works and what are the different types of Garbage Collectors?

Answer: Garbage collection in java is the process of looking at heap memory, identifying which objects are in use and which are not and deleting the unused objects. An unused object or unreferenced object, is no longer referenced by any part of your program.

Garbage collector is a daemon thread that keeps running in the background, freeing up heap memory by destroying the unreachable objects.

There was an analysis done on several applications which showed that most objects are short lived, so this behavior was used to improve the performance of JVM. In this method, the heap space is divided into smaller parts or generations. These are, ***Young Generation , Old or Tenured Generation*** and ***Permanent Generation*** .

The **Young Generation** is where all new objects are allocated and aged. The young generation is further divided into 3 parts: Eden Space, Survivor space S0 and Survivor space S1. When the young generation fills up, this causes a **minor garbage collection** . Some surviving objects are aged and eventually move to the old generation. All minor garbage collections are

"Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**. Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So, for Responsive applications, major garbage collections should be minimized. Also note that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection. And Perm Gen was available till Java 7, it is removed from Java 8 onwards and JVM uses native memory for the representation of class metadata which is called MetaSpace.

There is a flag `MaxMetaspaceSize`, to limit the amount of memory used for class metadata. If we do not specify the value for this, the Metaspace re-sizes at runtime as per the demand of the running application.

How Garbage collection works:

When new objects are first created, they are stored in the eden space of Young Generation and at that time both Survivor spaces are empty. When the eden space is filled, then a minor garbage collection is triggered. All the

unused or un-referenced objects are cleared from the eden space and the used objects are moved to first Survivor space S0.

At the next minor garbage collection, same process happens, un-referenced objects are cleared from the eden space but this time, the surviving objects are moved to Survivor space S1. In addition, the objects that were in S0 will also be matured and they also get moved to S1. Once all surviving objects are moved to S1, both eden and S0 are cleared.

At the next minor GC, the same process repeats. When surviving objects reached a certain threshold, they get promoted from Young generation to Old generation. These minor GC will continue to occur and objects will continue to be promoted to the Old generation.

Eventually, a major GC will be performed on the Old generation which cleans up and compacts the space.

Types of Garbage collector in Java:

Serial GC:

Serial GC is designed for smaller applications that have small heap sizes of up to a few hundred MBs. It only uses single virtual CPU for its garbage collection and the collection is done serially. It takes around couple of second for Full garbage collections.

It can be turned on by using **-XX:+UseSerialGC**

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m  
-XX:+UseSerialGC -jar C:\temp\test.jar
```

Parallel/Throughput GC:

Parallel garbage collector uses multiple threads to perform the garbage collection. By default, on a host with N CPUs, this collector uses N garbage collector threads for collection. The number of collector threads can be controlled with the command line option: **-XX:ParallelGCThreads=<N>**

It is called Throughput collector as it uses multiple CPUs to speed up the application throughput. A drawback of this collector is that it pauses the

application threads while performing minor or full GC, so it is best suited for applications where long pauses are acceptable. It is the default collector in JDK 8.

It can be turned on by using below 2 options:

-XX:+UseParallelGC

With this command line option, you get a multi-thread young generation collector with a single-threaded old generation collector. The option also does single-threaded compaction of old generation.

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m  
-XX:+UseParallelGC -jar C:\temp\test.jar
```

-XX:+UseParallelOldGC

With this option, the GC is both a multithreaded young generation collector and multithreaded old generation collector. It is also a multithreaded compacting collector.

Compacting describes the act of moving objects in a way that there are no holes between objects. After a garbage collection sweep, there may be holes left between live objects. Compacting moves objects so that there are no remaining holes. This compacting of memory makes it faster to allocate new chunks of memory to the heap.

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m  
-XX:+UseParallelOldGC -jar C:\temp\test.jar
```

Concurrent Mark Sweep (CMS) Collector:

The CMS collector, also called as the concurrent low pause collector, collects the tenured generation. It attempts to minimize the pauses due to garbage collection, by doing most of the garbage collection work concurrently with the application threads.

It can be turned on by passing `-XX:+UseConcMarkSweepGC` in the command line option.

If you want to set number of threads with this collector, pass -
XX:ParallelCMSThreads=<N>

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m  
-XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2 -jar  
C:\temp\test.jar
```

G1 Garbage Collector:

The Garbage First or G1 collector is a parallel, concurrent and incrementally compacting low-pause garbage collector

G1 collector partitions the heap into a set of equal-sized heap regions. When G1 performs garbage collection then a concurrent global marking phase is performed to determine the liveness of objects throughout the heap. After this mark phase is complete, G1 knows which regions are mostly empty. It collects unreachable objects from these regions first, which usually yields a large amount of free space, also called Sweeping. So G1 collects these regions (containing garbage) first, and hence the name Garbage-First.

It can be turned on by passing -XX:+UseG1GC in the command line options

```
java -Xmx25g -Xms5g -XX:+UseG1GC -jar C:\temp\test.jar
```

Java 8 has introduced one JVM parameter for reducing the unnecessary use of memory by creating too many instances of the same String. This optimizes the heap memory by removing duplicate String values to a global single char[] array. We can use the -XX:+UseStringDeduplication JVM argument to enable this optimization.

G1 is the default garbage collector in JDK 9.

Question 70: Explain Generics in Java

Answer: Java Generics provides a way to reuse the same code with different inputs.

Advantages:

-
Generics provide compile-time type safety that allows programmers to catch invalid types at compile time.

Before Generics:

```
List list = new ArrayList();
list.add("Mike");
list.add(10);
```

After Generics:

```
List<String> list = new ArrayList<>();
list.add("Mike");
list.add(10);
```

▼ ✖ Errors (1 item)
✖ The method add(int, String) in the type List<String> is not applicable for the arguments (int)

-
When using Generics, there is no need of type-casting.

Before Generics:

```
List list = new ArrayList();
list.add("Mike");
```

```
String name = list.get(0);
```

▼ ✖ Errors (1 item)

✖ Type mismatch: cannot convert from Object to String

After Generics:

```
List<String> list = new ArrayList<>();
list.add("Mike");

String name = list.get(0);
```

-

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized and are type safe and easier to read.

```
package com.generic.demo;

class Test<T> {
    private T obj;

    public Test(T obj) {
        this.obj = obj;
    }

    public T getObject() {
        return this.obj;
    }
}

public class GenericDemo {
    public static void main(String[] args) {
        Test<String> t1 = new Test<>("Mark");
        System.out.println(t1.getObject());

        Test<Integer> t2 = new Test<>(10);
        System.out.println(t2.getObject());
    }
}
```

Output:

Mark

10

Multi-threading: you will be asked many questions on multi-threading, so, read as much as you can and whatever you can. Here, I am including some of the important questions that are mostly asked in every interview.

Question 71: What is Multi-threading?

Answer: Multi-threading is a process of executing two or more threads concurrently, utilizing available CPU resources. A single thread is a lightweight sub-process and the smallest unit of processing. Threads are independent, if any exception occurs in one thread, it does not affect other threads.

When we execute a Java program without making any separate thread, then also our program runs on a thread called ‘main thread’.

There are 2 types of threads in an application, **user** thread and **daemon** thread. When the application is first started, main thread is the first user thread created. We can create multiple user threads and daemon threads.

One thing to remember here is that, JVM does not have any control on a thread’s execution. The thread execution is controlled by Thread scheduler which is part of Operating System. A thread can be assigned a priority using `setPriority(int)` method, where 1 is the minimum and 10 is the maximum priority, however thread priority is not guaranteed as it is platform dependent.

Multi-threading is used in a time-consuming task, one common example is File Upload.

Question 72: How to create a thread in Java?

Answer: There are 2 ways to create a thread:

- By extending Thread class
- By implementing Runnable interface

By extending Thread class:

ThreadTest.java:

```
package com.multithreading.demo;

class Task extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
            + " is running");
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        for(int i=1; i<=5; i++) {
            Task task = new Task();
            task.setName("Thread " + i);
            task.start();
        }
    }
}
```

Output:

```
Thread 1 is running
Thread 3 is running
Thread 2 is running
Thread 4 is running
Thread 5 is running
```

Here a Task class extends Thread class and overrides the run() method which contains the business logic, then we make an object of this Task and call the start() method, which starts the thread execution. *start() method internally calls run() method .*

By implementing Runnable interface:

ThreadTest.java:

```
package com.multithreading.demo;

class Task implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
            + " is running");
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        for(int i=1; i<=5; i++) {
            Thread task = new Thread(new Task());
            task.setName("Thread " + i);
            task.start();
        }
    }
}
```

Output:

```
Thread 2 is running
Thread 1 is running
Thread 4 is running
Thread 3 is running
Thread 5 is running
```

If you see the outputs of this and previous program, you will see that they are different, because any thread can get a chance to execute its run() method, when the CPU resources are available.

***Question 73: Which way of creating threads is better:
Thread class or Runnable interface***

Answer: Implementing Runnable is always the preferred choice, see the reasons below:

- As you know, Java does not allow multiple inheritance through classes (because of Diamond problem discussed in Question 9), so if you are creating threads by extending Thread class then you will not be able to extend any other class.
- When we are working with multi-threading, we are not looking to overwrite any existing functionality of Thread class, we just want to execute the code with multiple threads, so in that way also, Runnable is a good choice.
- One more reason to choose Runnable is that, most people don't work with just Raw Threads, they use the Executor framework that is provided from Java 5, that separates the task from its execution and we can execute Runnables using ***execute(Runnable Task)*** method of ***Executor interface***.

Question 74: What will happen if I directly call the run() method and not the start() method to execute a thread

Answer: if run() method is called directly, then a new thread will not be created instead the code will run on the current thread which is main thread. Calling run() method directly will make it behave as any other normal method call. Only a call to start() method creates separate thread.

Question 75: Once a thread has been started can it be started again

Answer: No. A thread can be started only once in its lifetime. If you try to start a thread which has already been started, an ***IllegalThreadStateException*** is thrown, which is a runtime exception. A thread in runnable state or a dead thread cannot be restarted.

Question 76: Why wait, notify and notifyAll methods are defined in the Object class instead of Thread class

Answer: This is another very famous multi-threading interview question. The methods wait, notify and notifyAll are present in the Object class, that means they are available to all class objects, as Object class is the parent of all classes.

wait() method – it tells the current thread to release the lock and go to sleep until some other thread enters the same monitor and calls *notify()*

notify() method – wakes up the single thread that is waiting on the same object's monitor

notifyAll() method – wakes up all the threads that called *wait()* on the same object

if these methods were in Thread class, then thread T1 must know that another thread T2 is waiting for this particular resource, so T2 can be notified by something like *T2.notify()*

But in java, the object itself is shared among all the threads, so one thread acquires the lock on this object's monitor, runs the code and while releasing the lock, it calls the *notify* or *notifyAll* method on the object itself, so that any other thread which was waiting on the same object's monitor will be notified that now the shared resource is available. That is why these methods are defined in the Object class.

Threads have no specific knowledge of each other. They can run asynchronously and are independent. They do not need to know about the status of other threads. They just need to call *notify* method on an object, so whichever thread is waiting on that resource will be notified.

Let's consider this with a real-life example:

Suppose there is a petrol pump and it has a single washroom, the key of which is kept at the service desk. This washroom is a shared resource for all. To use this shared resource, the user must acquire the key to the

washroom lock. So, the user goes to service desk, acquires the key, opens the door, locks it from the inside and use the facility.

Meanwhile if another user arrives at the petrol pump and wants to use the washroom, he goes to the washroom and found that it is locked. He goes to the service desk and the key is not there because it is with the current user. When the current user finishes, he unlocks the door and returns the key to the service desk. He does not bother about the waiting users. The service desk gives the key to waiting user. If there are more than one user waiting to use the facility, then they must form a queue.

Now, apply this analogy to Java, one user is one thread and the washroom is the shared resource which the threads wish to execute. The key will be **synchronized keyword** provided by Java, through which thread acquires a lock for the code it wants to execute and making other threads wait until the current thread finishes. Java will not be as fair as the service station, because any of the waiting threads may get a chance to acquire the lock, regardless of the order in which the threads came. The only guarantee is that all the waiting threads will get to use the shared resource sooner or later.

In this example, the lock can be acquired on the key object or the service desk and none of them is a thread. These are the objects that decide whether the washroom is locked or not.

Question 77: Why wait(), notify(), notifyAll() methods must be called from synchronized block

Answer: these methods are used for inter-thread communication. So, a wait() method only makes sense when there is a notify() method also.

If these methods are not called from a synchronized block then

- IllegalMonitorStateException will be thrown
- Race condition can occur

Let's first look at the IllegalMonitorStateException:

WaitDemo.java:

```
package com.multithreading.demo;

public class WaitDemo {
    public static void main(String[] args) {
        WaitDemo wd = new WaitDemo();

        try {
            wd.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Exception in thread "main" java.lang.IllegalMonitorStateException
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Unknown Source)
    at com.multithreading.demo.WaitDemo.main(WaitDemo.java:8)
```

Now, let's understand how a race condition can occur:

Producer-Consumer problem : The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer, if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Pseudo code:

```
class Test{
    List<String> buffer;

    public void produce(String data) {
        buffer.add(data);
        notify();
    }

    public String consume() {
        while(buffer.isEmpty())
            wait();
        return buffer.remove();
    }
}
```

How the race condition problem can occur:

Suppose, a thread has called the consume() method and it finds that the buffer is Empty

Now, just before the wait() method is called, another thread calls produce() and adds an item to the buffer and calls notify()

And The first thread calls the wait() method. In this case, notify() call by the second thread will be missed

if by any chance, produce() method is not called then the consumer thread will be stuck in waiting state indefinitely, even though there is data available in the buffer.

The solution to above problem is using synchronized method/block to make sure that notify() is never called between the condition isEmpty() and wait()

Question 78: difference between wait() and sleep() method

Answer: The differences are:

- wait() method can only be called from a synchronized context while sleep() method can be called without synchronized context
- wait() method releases the lock on the object while waiting but sleep() method does not release the lock it holds while waiting, it means if the thread is currently in a synchronized block/method then no other thread can enter this block/method
- wait() method is used for inter-thread communication while sleep() method is used to introduce a pause on execution
- waiting thread can be waked by calling notify() or notifyAll(), while sleeping thread will wake up when the specified sleep time is over or the sleeping thread gets interrupted
- wait() method is non-static, it gets called on an object on which synchronization block is locked while sleep() is a static method, we call this method like Thread.sleep(), that means it always affects the currently executing thread
- wait() is normally called when a condition is fulfilled like if the buffer size of queue is full then producer thread will wait, whereas sleep() method can be called without a condition

Question 79: join() method

Answer: join() method causes the current thread to pause execution until the thread which has called join() method is dead.

join() method can be used to execute the threads sequentially or in some specific order.

Let's see an example below:

There are 3 threads and I want to execute them in the order 1, 3, 2:

JoinMethodDemo.java:

```
package com.multithreading.demo;

class JoinTask implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName());
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class JoinMethodDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new JoinTask());
        t1.setName("First Thread");
        Thread t2 = new Thread(new JoinTask());
        t2.setName("Second Thread");
        Thread t3 = new Thread(new JoinTask());
        t3.setName("Third Thread");

        t1.start();
        System.out.println("Current Thread : "
                           + Thread.currentThread().getName());
        t1.join();

        t3.start();
        System.out.println("Current Thread : "
                           + Thread.currentThread().getName());
        t3.join();

        t2.start();
        System.out.println("Current Thread : "
                           + Thread.currentThread().getName());
        t2.join();

        System.out.println("Exiting from Current Thread : "
                           + Thread.currentThread().getName());
    }
}
```

Output:

```
Current Thread : main
First Thread
Current Thread : main
Third Thread
Current Thread : main
Second Thread
Exiting from Current Thread : main
```

In the code, if you don't write `t2.join()`, then current thread will not wait from the `t2` thread to die, see the output below when `t2.join()` statement is commented from the code :

```
Current Thread : main
First Thread
Current Thread : main
Third Thread
Current Thread : main
Exiting from Current Thread : main
Second Thread
```

There are overloaded versions of `join()` method also,

- `join(long milliseconds)` : when this method is called, then the current thread will wait at most for the specified milliseconds
- `join(long milliseconds, long nanoseconds)` : when this method is called, then the current thread will wait at most for the specified milliseconds plus nanoseconds.

These `join` methods are dependent on the underlying Operating system for timing. So, you should not assume that `join()` will wait exactly as long as

you specify.

You can execute threads in a sequence using CountDownLatch also.

Question 80: yield() method

Answer: yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution.

When the yielded thread will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent. Yield method doesn't guarantee that the current thread will pause or stop but it guarantees that CPU will be relinquished by current Thread as a result of a call to Thread.yield() method in java.

Question 81: Tell something about synchronized keyword

Answer: synchronized keyword in java is used to control the access of multiple threads to any shared resource, so that any consistency problem can be avoided.

We can make the entire method as synchronized or just the part where the shared resource is getting used, to do this synchronized blocks are used.

Synchronized method/block can only have one thread executing inside it, all the other threads trying to enter into the synchronized method/block will get blocked until the thread inside finishes its execution. When the thread exits the synchronized method/block then Java guarantees that changes to the state of the object is visible to all the threads. This eliminates the memory inconsistency errors.

Question 82: What is static synchronization?

Answer: When synchronized keyword is used with a static method, then that is called static synchronization. In this, lock will be on the class not the object. This means only one thread can access the class at a time.

The purpose of static synchronization is to make the static data thread-safe.

Let's look at some programs:

Here, we have a Hello class which has a synchronized method:

```
package com.multithreading.demo;

class Hello {

    synchronized void sayHello() {
        System.out.println("in sayHello() method " +
                           Thread.currentThread().getName());
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName() + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

A Task class which implements Runnable and its run() method simply calls the synchronized method of Hello class:

```
class Task implements Runnable {  
  
    Hello h;  
  
    Task(Hello h) {  
        this.h = h;  
    }  
  
    @Override  
    public void run() {  
        h.sayHello();  
    }  
}
```

Our main class:

```
public class SynchronizationDemo {  
    public static void main(String[] args) {  
        Hello obj1 = new Hello();  
        Hello obj2 = new Hello();  
  
        Thread task1 = new Thread(new Task(obj1));  
        task1.setName("First Thread");  
        Thread task2 = new Thread(new Task(obj1));  
        task2.setName("Second Thread");  
        Thread task3 = new Thread(new Task(obj2));  
        task3.setName("Third Thread");  
        Thread task4 = new Thread(new Task(obj2));  
        task4.setName("Fourth Thread");  
  
        task1.start();  
        task2.start();  
        task3.start();  
        task4.start();  
    }  
}
```

We have 2 objects of our Hello class, one object is shared among First and Second thread, and one object is shared among Third and Fourth thread, and we are starting these threads.

Output:

```
in sayHello() method First Thread
in sayHello() method Third Thread
First Thread , i = 1
Third Thread , i = 1
First Thread , i = 2
Third Thread , i = 2
First Thread , i = 3
Third Thread , i = 3
Third Thread , i = 4
First Thread , i = 4
First Thread , i = 5
Third Thread , i = 5
in sayHello() method Second Thread
```

```
in sayHello() method Second Thread  
Second Thread , i = 1  
in sayHello() method Fourth Thread  
Fourth Thread , i = 1  
Second Thread , i = 2  
Fourth Thread , i = 2  
Second Thread , i = 3  
Fourth Thread , i = 3  
Fourth Thread , i = 4  
Second Thread , i = 4  
Second Thread , i = 5  
Fourth Thread , i = 5
```

As you can see from the output, the First and Second thread are not having any thread interference. Same way, Third and Fourth thread does not have any thread interference but First and Third thread are entering the synchronized method at the same time with their own object locks (Hello obj1 and obj2).

Lock which is hold by First thread will only stop the Second thread from entering the synchronized block, because they are working on the same instance i.e. obj1, but it cannot stop Third or Fourth thread as they are working on another instance i.e. obj2.

If we want our synchronized method to be accessed by only one thread at a time then we have to use a **static synchronized method/block** to have the synchronization on the class level rather than on the instance level.

```
class Hello {  
    static synchronized void sayHello() {  
        System.out.println("in sayHello() method " +  
                           Thread.currentThread().getName());  
        for(int i=1; i<=5; i++) {  
            System.out.println(Thread.currentThread().getName() + " , i = " + i);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Let's see the output now:

```
in sayHello() method First Thread  
First Thread , i = 1  
First Thread , i = 2  
First Thread , i = 3  
First Thread , i = 4  
First Thread , i = 5  
in sayHello() method Fourth Thread  
Fourth Thread , i = 1  
Fourth Thread , i = 2  
Fourth Thread , i = 3  
Fourth Thread , i = 4  
Fourth Thread , i = 5
```

```
in sayHello() method Third Thread  
Third Thread , i = 1  
Third Thread , i = 2  
Third Thread , i = 3  
Third Thread , i = 4  
Third Thread , i = 5  
in sayHello() method Second Thread  
Second Thread , i = 1  
Second Thread , i = 2  
Second Thread , i = 3  
Second Thread , i = 4  
Second Thread , i = 5
```

Here, only one thread is accessing the static synchronized method.

Same can be done by synchronized block also:

```
class Hello {  
  
    static void sayHello() {  
        synchronized(Hello.class) {  
            System.out.println("in sayHello() method " +  
                Thread.currentThread().getName());  
            for(int i=1; i<=5; i++) {  
                System.out.println(Thread.currentThread().getName() + " , i = " + i);  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

Question 83: What will be output of below program where one synchronized method is calling another synchronized

method?

```
package com.multithreading.demo;

class Demo {
    public synchronized void m1() {
        m2();
        System.out.println("inside m1()");
    }

    public synchronized void m2() {
        m3();
        System.out.println("inside m2()");
    }

    public synchronized void m3() {
        System.out.println("inside m3()");
    }
}

public class Test {
    public static void main(String[] args) {
        Demo d1 = new Demo();

        d1.m1();
    }
}
```

Output:

inside m3()
inside m2()
inside m1()

One thing to remember here is that Java synchronized keyword is re-entrant in nature, it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding the lock can enter into that method without acquiring lock.

Question 84: Programs related to synchronized and static synchronized methods

There are some confusing programs that interviewer can ask where some methods are static synchronized and some methods are non-static synchronized, and sometimes they are calling each other, so let's discuss those.

Scenario 1 : There are 2 threads that are calling 2 different static synchronized methods.

Here, these 2 threads will block each other, as only one lock per class exists. So, these 2 static synchronized methods will not be executed at the same time.

Program:

```
package com.multithreading.demo;

class Demo {
    public static synchronized void m1() {
        System.out.println("inside m1()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static synchronized void m2() {
        System.out.println("inside m2()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Demo d1 = new Demo();
        Demo d2 = new Demo();

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                d1.m1();
            }
        });
        t1.setName("First thread");
    }
}
```

```
Thread t2 = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        d2.m2();  
    }  
});  
t2.setName("Second thread");  
  
t1.start();  
t2.start();  
}  
}
```

Output:

```
inside m1()
First thread , i = 1
First thread , i = 2
First thread , i = 3
First thread , i = 4
First thread , i = 5
inside m2()
Second thread , i = 1
Second thread , i = 2
Second thread , i = 3
Second thread , i = 4
Second thread , i = 5
```

Scenario 2 : There are 2 threads, one is calling static synchronized method on one object, and the other thread is calling non-static synchronized method on another object.

Here, these 2 threads will not block each other and will be executed concurrently as both locks are different, the thread executing the static synchronized method holds a lock on the class and the thread executing the non-static synchronized method holds the lock on the object on which the method has been called.

In short, static synchronized method do not block a non-static synchronized method.

Program:

```
package com.multithreading.demo;

class Demo {
    public static synchronized void m1() {
        System.out.println("inside m1()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public synchronized void m2() {
        System.out.println("inside m2()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

(Main class is same as Scenario 1)

Output:

```
inside m2()
inside m1()
Second thread , i = 1
First thread , i = 1
Second thread , i = 2
First thread , i = 2
Second thread , i = 3
First thread , i = 3
First thread , i = 4
Second thread , i = 4
First thread , i = 5
Second thread , i = 5
```

Question 85: What is Callable Interface?

Answer: Callable interface represents an asynchronous task which can be executed by a separate thread, it has one call() method, which returns a Future object.

A Runnable can be executed by passing it into the Thread class constructor but Thread class does not have a constructor that accepts a Callable, so Callable can be executed only by submit() method of ExecutorService Interface.

Callable's call() method can throw checked exceptions, the exceptions are collected in Future object which can be checked by making a call to Future.get() method, an ExecutionException is thrown which wraps the original exception. We can get the original checked exception by making a call to getCause() method on the exception object thrown. However, if we don't call Future.get() method then the exception will not be reported back and the task will be marked as completed.

One thing you should remember is that the Future.get() method blocks the execution, so timeouts should be used when using this method to avoid unexpected waits.

Program showing how Callable is used and how the result is returned in Future object:

```
class Task implements Callable<Integer> {
    private int num;

    public Task(int num) {
        this.num = num;
    }

    @Override
    public Integer call() throws Exception {
        if(num < 0) {
            throw new InvalidParameterException("Negative number not allowed");
        }
        return num*num;
    }
}

public class CallableDemo {
    public static void main(String[] args) {

        Task task = new Task(5);

        ExecutorService es = Executors.newFixedThreadPool(2);
        Future<Integer> f = es.submit(task);

        try {
            System.out.println("Result: " + f.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        es.shutdownNow();
    }
}
```

Output:

Result: 25

Let's pass a negative number, so that exception will be thrown:

```
public class CallableDemo {  
    public static void main(String[] args) {  
  
        Task task = new Task(-10);  
  
        ExecutorService es = Executors.newFixedThreadPool(2);  
        Future<Integer> f = es.submit(task);  
  
        try {  
            System.out.println("Result: " + f.get());  
        } catch (InterruptedException | ExecutionException e) {  
            e.printStackTrace();  
        }  
        es.shutdownNow();  
    }  
}
```

Output:

```
java.util.concurrent.ExecutionException: java.security.InvalidParameterException: Negative number not allowed  
at java.util.concurrent.FutureTask.report(Unknown Source)  
at java.util.concurrent.FutureTask.get(Unknown Source)  
at com.callable.demo.CallableDemo.main(CallableDemo.java:35)  
Caused by: java.security.InvalidParameterException: Negative number not allowed  
at com.callable.demo.Task.call(CallableDemo.java:20)  
at com.callable.demo.Task.call(CallableDemo.java:1)  
at java.util.concurrent.FutureTask.run(Unknown Source)  
at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)  
at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)  
at java.lang.Thread.run(Unknown Source)
```

You can call e.getCause() to get the original exception.

Question 86: How to convert a Runnable to Callable

Answer: We have a utility method in “Executors” class:

- callable(Runnable task) : Returns a Callable object that, when called, runs the given task and returns null.
- callable(Runnable task, T result) : Returns a Callable object that, when called, runs the given task and returns the given result

Question 87: Difference between Runnable and Callable

Answer: The difference is:

- Runnable tasks can be executed by using Thread class or ExecutorService interface whereas Callable tasks can be executed by using ExecutorService interface only
- Return type of Runnable's run() method is void whereas Callable's call() method returns Future object
- Runnable's run() method does not throw checked exceptions whereas Callable's call() method can throw checked exceptions

Question 88: What is Executor Framework in Java, its different types and how to create these executors?

Answer: Executor Framework is an abstraction to managing multiple threads by yourself. So, it decouples the execution of a task and the actual task itself. Now, we just have to focus on the task that means, only implement the Runnables and submit them to executor. Then these runnables will be managed by the executor framework. It is available from Java 1.5 onwards.

Also, we don't have to create new threads every time. With executor framework, we use Thread pools. Think of Thread Pool as a user-defined number of threads which are called worker threads, these are kept alive and reused. The tasks that are submitted to the executor will be executed by these worker threads. If there are more tasks than the threads in the pool, they can be added in a Queue and as soon as one of thread is

finished with a task, it can pick the next one from this Queue or else, it will be added back in the pool waiting for a task to be assigned.

So, it saves the overhead of creating a new thread for each task. If you are thinking about what is the problem with creating a new thread every time we want to execute a task, then you should know that creating a thread is an expensive operation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead and new threads without any throttling will lead to the creation of large number of threads. These threads will cause wastage of resources.

There are 2 main interfaces that you must know, one is *Executor* and the other is *ExecutorService* .

Executor interface contains `execute(Runnable task)` method through which you can execute only Runnables. Also, the return type of `execute()` method is void, since you are passing a Runnable to it and it does not return any result back.

ExecutorService interface contains the `submit()` method which can take both Runnable and Callable, and its return type is Future object. ExecutorService extends the Executor Interface, so it also has the `execute()` method.

Now, we have an idea of what is an Executor Framework, let's look at different types of Executors:

SingleThreadExecutor :

This executor has only one thread and is used to execute tasks in a sequential manner. If the thread dies due to an exception while executing the task, a new thread is created to replace the old thread and the subsequent tasks are executed in the new thread.

How to create a SingleThreadExecutor:

```
ExecutorService executor =  
Executors.newSingleThreadExecutor();
```

Executors is a utility class which contains many factory methods to create different types of ExecutorService, like the one called SingleThreadExecutor, we just created.

FixedThreadPoolExecutor :

As its name suggests, this is an executor with a fixed number of threads. The tasks submitted to this executor are executed by the specified number of threads and if there are more tasks than the number of threads, then those tasks will be added in a queue (e.g. LinkedBlockingQueue).

How to create a FixedThreadPoolExecutor:

```
ExecutorService executor = Executors.newFixedThreadPool  
(5);
```

Here, we have created a thread pool executor of 5 threads, that means at any given time, 5 tasks can be managed by this executor. If there are more active tasks, they will be added to a queue until one of the 5 threads becomes free.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

CachedThreadPoolExecutor :

This executor is mainly used when there are many short-lived tasks to be executed. If you compare this with the fixed thread pool, here the number of threads of this executor pool is not bounded. If all the threads are busy executing the assigned tasks and if there is a new task, then a new thread

will be created and added to the pool. If a thread remains idle for close to sixty seconds, it is terminated and removed from the cache.

Use this one, if you are sure that the tasks will be short-lived, otherwise there will be a lot of threads in the pool which will lead to performance issues.

How to create a CachedThreadPoolExecutor:

```
ExecutorService executor =  
Executors.newCachedThreadPool();
```

ScheduledExecutor :

Use this executor, when you want to schedule your tasks, like run them at regular intervals or run them after a given delay. There are 2 methods which are used for scheduling tasks: *scheduleAtFixedRate* and *scheduleWithFixedDelay* .

How to create ScheduledExecutor:

```
ExecutorService executor =  
Executors.newScheduledThreadPool(4);
```

ScheduledExecutorService interface extends the ExecutorService interface.

Now, apart from using Executors class to create executors, you can use ThreadPoolExecutor and ScheduledThreadPoolExecutor class also. Using these classes, you can manually configure and fine-tune various parameters of the executor according to your need. Let's see at some of those parameters:

```
public ThreadPoolExecutor(int corePoolSize,  
                           int maximumPoolSize,  
                           long keepAliveTime,  
                           TimeUnit unit,  
                           BlockingQueue<Runnable> workQueue,  
                           ThreadFactory threadFactory,  
                           RejectedExecutionHandler handler)
```

Core and Max Pool sizes:

A ThreadPoolExecutor will automatically adjust the pool size according to the bounds set by corePoolSize and maximumPoolSize

When a new task is submitted to the executor then:

- If the number of threads running are less than the corePoolSize, a new thread is created to handle the request
- If the number of threads running are more than corePoolSize but less than maximumPoolSize then a new thread will be created only if the queue is full

Let's understand this with an example:

You have defined the core pool size as 5, maximum pool size as 10 and the queue capacity as 100. Now as tasks are coming in, new threads will be created up to 5, then other new tasks will be added to queue until it reaches 100. Now when the queue is full and if new tasks are coming in, threads will be created up to the maximumPoolSize i.e. 10. Once all the threads are in use and the queue is also full, the new tasks will be rejected. As the queue reduces, so does the number of active threads.

Keep Alive Time and TimeUnit:

When the number of threads are greater than the core size, this is the maximum time that excess idle threads will wait for new tasks before terminating. It is used to avoid the overhead of creating a new thread.

Let's understand this with an example:

You have defined the core pool size as 5 and maximum pool size as 15 and all the 15 threads are getting used at the moment. Now when the threads are getting finished with their work, the excess 10 threads (15-5) become idle and eventually die. To avoid these 10 threads being killed too quickly, we can specify the keep alive time for these by using the keepAliveTime parameter in the ThreadPoolExecutor constructor. If you have given its value as 1 and time unit as TimeUnit.MINUTE, each thread will wait for 1 min after it had finished executing a task. Basically, it is waiting for a new task to be assigned. If it is not given any task, it would let itself complete. And in the end, the executor will be left with the core threads (5).

BlockingQueue:

The queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method, you can use a ArrayBlockingQueue or LinkedBlockingQueue like:

```
BlockingQueue<Runnable> queue = new  
ArrayBlockingQueue<>(100);
```

ThreadFactory:

The factory to use when the executor creates a new thread. Using thread factories removes hardwiring of calls to *new Thread* , enabling applications to use special thread subclasses, priorities, etc.

RejectedExecutionHandler:

This handler is used when a task is rejected by the executor because all the threads are busy and the queue is full.

When this handler is not provided and the task submitted to execute() method is rejected, then an unchecked *RejectedExecutionException* is thrown.

But adding a handler is a good practice to follow, there is a method:

```
void rejectedExecution(Runnable r, ThreadPoolExecutor  
executor);
```

This method will be invoked by ThreadPoolExecutor when execute() cannot accept a task.

Putting it all together:

```
BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100);  
  
ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 15, 50, TimeUnit.SECONDS, queue);  
  
executor.setRejectedExecutionHandler(new RejectedExecutionHandler() {  
  
    @Override  
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  
        //run this code when task is rejected  
    }  
});
```

Question 89: Tell something about awaitTermination() method in executor

Answer: This method blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

```
public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
```

It returns true if this executor is terminated and false if the timeout is elapsed before termination.

Question 90: Difference between shutdown() and shutdownNow() methods of executor

Answer: An executor will not shut down automatically even when there is no task to process. It will stay alive and wait for new work. It will keep the JVM running.

When shutdown() method is called on an executor, then the executor will not accept new tasks and it will wait for the currently executing tasks to finish.

When shutdownNow() is called, it tries to interrupt the running threads and shutdown the executor immediately. However, there is no guarantee that all the running threads will be stopped at the same time.

One good way to shutdown an executor is to use both of these methods along with awaitTermination(). With this approach, the executor will stop accepting new tasks and waits up to the specified duration for all running tasks to be completed. If the time expires, it will shutdown immediately.

```
executor.shutdown();

try {
    if(executor.awaitTermination(5, TimeUnit.MINUTES)) {
        executor.shutdownNow();
    }
} catch (InterruptedException e) {
    executor.shutdownNow();
}
```

Question 91: What is Count down latch in Java?

Answer: CountDownLatch is used in requirements where you want one or more tasks to wait for some other tasks to finish before it starts its own execution. One example can be a server which is dependent on some services to be up and running before it can start processing requests.

How to use: When we create an object of CountDownLatch, then we specify the number of threads that it should wait for, then waiting thread calls the countDownLatch.await() method and until all the specified thread calls countDownLatch.countDown() method, the waiting thread will not start its execution.

For example, there are 3 services which a server is dependent on, before the server accepts any request, these services should be up and running. We will create a CountDownLatch by specifying 3 threads that the main thread should wait for, then main thread will call await() method means it will wait for all 3 threads. Once the threads are complete they will call countdown() method, decreasing the count by 1. The main thread will start its execution only when count reaches to zero.

```
package com.countdownlatch.demo;

import java.util.concurrent.CountDownLatch;

class Task implements Runnable {
    String service;
    CountDownLatch latch;

    public Task(String service, CountDownLatch latch) {
        this.service = service;
        this.latch = latch;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(service + " is up");
        latch.countDown();
    }
}
```

```
public class CountDownLatchDemo {  
    public static void main(String[] args) {  
  
        CountDownLatch latch = new CountDownLatch(3);  
  
        Thread t1 = new Thread(new Task("Service1", latch));  
        Thread t2 = new Thread(new Task("Service2", latch));  
        Thread t3 = new Thread(new Task("Service3", latch));  
  
        t1.start();  
        t2.start();  
        t3.start();  
  
        try {  
            latch.await();  
            System.out.println("All services are up, "  
                + "Starting Main Application now");  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Service1 is up  
Service2 is up  
Service3 is up  
All services are up, Starting Main Application now
```

CountDownLatch can also be used to start multiple threads at the same time, you can create a CountDownLatch of size 1, make all the other threads wait by calling countDownLatch.await(), then a single call to countDownLatch.countDown() method will resume execution for all the waiting threads at the same time.

CountDownLatch cannot be reused once the count reaches to zero, therefore in those scenarios, CyclicBarrier is used.

Question 92: What is Cyclic Barrier?

Answer: CyclicBarrier is used to make multiple threads wait for each other to reach a common barrier point. It is used mainly when multiple threads perform some calculation and the result of these threads needs to be combined to form the final output.

All the threads that wait for each other to reach the barrier are called parties, CyclicBarrier is created with a number of parties to wait for, and the threads wait for each other at the barrier by calling cyclicBarrier.await() method which is a blocking method and it blocks until all threads have called await().

The barrier is called Cyclic because it can be re-used after the waiting threads are released, by calling cyclicBarrier.reset() method which resets barrier to the initial state.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

```
public CyclicBarrier(int parties, Runnable barrierAction)
```

The CyclicBarrier uses an all-or-none breakage model for failed synchronization attempts: If a thread leaves a barrier point prematurely because of interruption, failure, or timeout, all other threads waiting at that barrier point will also leave abnormally via BrokenBarrierException (or InterruptedException if they too were interrupted at about the same time).

Example: One thread is adding first 5 natural numbers to the list, the other thread is adding next 5 numbers to the list and we will perform an addition of all these numbers to compute the sum of first 10 natural numbers

```
package com.cyclicbarrier.demo;

import java.util.List;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CyclicBarrier;

class Task1 implements Runnable {
    String name;
    List<Integer> numbers;
    CyclicBarrier barrier;

    public Task1(String name, List<Integer> numbers, CyclicBarrier barrier) {
        this.name = name;
        this.numbers = numbers;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        System.out.println(name + " is running");

        for(int i=1; i<=5; i++) {
            numbers.add(i);
        }

        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
        System.out.println(name + " has crossed the barrier");
    }
}

class Task2 implements Runnable {
    String name;
    List<Integer> numbers;
    CyclicBarrier barrier;

    public Task2(String name, List<Integer> numbers, CyclicBarrier barrier) {
        this.name = name;
        this.numbers = numbers;
        this.barrier = barrier;
    }
}
```

```

@Override
public void run() {
    System.out.println(name + " is running");

    for(int i=6; i<=10; i++) {
        numbers.add(i);
    }

    try {
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        e.printStackTrace();
    }
    System.out.println(name + " has crossed the barrier");
}

class FinalTask implements Runnable {
    String name;
    List<Integer> numbers;

    public FinalTask(String name, List<Integer> numbers) {
        this.name = name;
        this.numbers = numbers;
    }
    @Override
    public void run() {
        int sum = 0;
        for(Integer i : numbers) {
            sum = sum + i;
        }
        System.out.println("Sum of first 10 natural numbers : " + sum);
    }
}

public class CyclicBarrierDemo {
    public static void main(String[] args) {
        List<Integer> numbers = new CopyOnWriteArrayList<Integer>();

        CyclicBarrier barrier
            = new CyclicBarrier(2, new FinalTask("Final Thread", numbers));

        Thread t1 = new Thread(new Task1("First Thread", numbers, barrier));
        Thread t2 = new Thread(new Task2("Second Thread", numbers, barrier));

        t1.start();
        t2.start();
    }
}

```

Output:

```
First Thread is running
Second Thread is running
Sum of first 10 natural numbers : 55
Second Thread has crossed the barrier
First Thread has crossed the barrier
```

Question 93: Atomic classes

Answer: The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set call has a happens-before relationship with any subsequent get call on the same variable.

For example, consider below code:

```
class Task implements Runnable {  
    private int count;  
  
    public int getCount() {  
        return this.count;  
    }  
  
    @Override  
    public void run() {  
        for(int i=1; i<=50; i++) {  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            count++;  
        }  
    }  
}
```

In the above runnable task, we are just incrementing an integer 50 times,

```
public class AtomicDemo {  
    public static void main(String[] args) {  
        Task task = new Task();  
  
        Thread t1 = new Thread(task);  
        //Thread t2 = new Thread(task);  
  
        t1.start();  
        //t2.start();  
  
        try {  
            t1.join();  
            //t2.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("Count is : " + task.getCount());  
    }  
}
```

We are using join() method so that the main thread will wait for the thread t1 to die, otherwise the sysout will be executed before t1 has finished execution. Let's see what will be the output when only one thread is running:

Output:

Count is : 50

As, you can see, the output is as expected because only one thread is accessing the count, let's see what will happen in case the count variable is accessed by more than one thread, un-comment the code regarding second thread t2 and run the main class:

Output:

Count is : 83

The expected output was 100 but we got a different output, if you run the above program you will see a different output and it will be anywhere

between 50 and 100. The reason for this is that 2 threads are accessing a mutable variable without any synchronization. One solution that will be coming to your mind will be using synchronization block, and yes this problem can be solved using that but it will have a performance impact, as threads will acquire the lock, update the value and release the lock, and then giving other threads access to the shared mutable variable.

But java has provided Atomic wrapper classes for this purpose that can be used to achieve this atomic operation without using Synchronization.

Let's see the change in our Runnable:

```
class Task implements Runnable {  
    private AtomicInteger count = new AtomicInteger();  
  
    public int getCount() {  
        return this.count.get();  
    }  
  
    @Override  
    public void run() {  
        for(int i=1; i<=50; i++) {  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            count.incrementAndGet();  
        }  
    }  
}
```

Output:

Count is : 100

Question 94: What is Collection Framework?

Answer: Collection framework represents an architecture to store and manipulate a group of objects. All the classes and interfaces of this framework are present in java.util package.

Some points:

- Iterable interface is the root interface for all collection classes, it has one abstract method iterator()
- Collection interface extends the Iterable interface

Question 95: What is Collections?

Answer: Collections is a utility class present in java.util package that we can use while using collections like List, Set, and Map etc.

Some commonly used methods are Collections.sort(), Collections.unmodifiableList() etc.

Question 96: What is ArrayList?

Answer: ArrayList is a resizable-array implementation of List Interface. When we create an object of ArrayList then a contiguous block of memory is allocated to hold its elements. As it is a contiguous block, all the elements address is already known, that is how ArrayList allows random access.

Some points about ArrayList class:

- ArrayList class maintains insertion order
- ArrayList class can contain duplicate elements
- ArrayList class allows random access to its elements as it works on index basis
- ArrayList class is not synchronized
- You can add any number of null elements in the ArrayList class

Time complexity of ArrayList's get(), add(), remove() operations:

get(): constant time

add(): here the new element can be added at the first, middle or last positions, if you are using add(element), then it will append element at the last position and will take O(1), provided that the arrayList is not full otherwise it will create a new arrayList of one and a half times the size of previous arrayList and copy all arrayList elements to this new arrayList, making it O(n).

If the element is added in the middle or at any specific index, let's say at index 2, then a space needs to be made to insert this new element by shifting all the elements one position to its right, making it O(n).

add() operation runs in amortized constant time.

remove(): it is also same as add(), if you want to remove element from a specific index, then all elements to its right needs to be shifted one position to their left, making it O(n) but if element needs to be removed from the last, then it will take O(1).

Question 97: What is default size of ArrayList?

Answer: 10

```
/**  
 * Default initial capacity.  
 */  
private static final int DEFAULT_CAPACITY = 10;
```

Question 98: Which data structure is used internally in an ArrayList?

Answer: Internally, ArrayList uses Object[]

```
/**  
 * The array buffer into which the elements of the  
 * ArrayList are stored.  
 * The capacity of the ArrayList is the length of this  
 * array buffer. Any
```

```
* empty ArrayList with elementData ==  
DEFAULTCAPACITY_EMPTY_ELEMENTDATA  
* will be expanded to DEFAULT_CAPACITY when the first  
element is added.  
*/  
transient Object[] elementData ;
```

Question 99: How add() method works internally or How the ArrayList grows at runtime

Answer: this is what happens when we create an ArrayList object using its default constructor,

```
/**  
* Constructs an empty list with an initial capacity of  
ten.  
*/  
public ArrayList() {  
    this .elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA  
};  
}
```

Here, elementData is a transient variable and DEFAULTCAPACITY_EMPTY_ELEMENTDATA is an empty Object[] array:

```
transient Object[] elementData ;  
  
private static final Object[]  
DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

Now, let's see Javadoc of add() method

```
/**  
* Appends the specified element to the end of this  
list.  
*  
* @param e element to be appended to this list  
* @return <tt> true </tt> (as specified by {@link  
Collection#add} )  
*/
```

```

public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData [size ++] = e ;
    return true ;
}

```

Here size is a private variable

```

/*
 * The size of the ArrayList (the number of elements it
contains).
*
* @serial
*/
private int size ;
```

The default value of size will be 0, so call to ensureCapacityInternal() will have value 1, now let's see ensureCapacityInternal() Javadoc:

```

private void ensureCapacityInternal(int minCapacity ) {
    ensureExplicitCapacity(calculateCapacity (elementData
,minCapacity ));
}

```

Here minCapacity is holding value 1, calculateCapacity() method is:

```

private static int calculateCapacity(Object[]
elementData , int minCapacity ) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
) {
        return Math.max (DEFAULT_CAPACITY , minCapacity );
    }
    return minCapacity ;
}

```

Now, as both elementData and DEFAULTCAPACITY_EMPTY_ELEMENTDATA are same (see the default ArrayList constructor above), if condition will be true and then Math.max(10,1) will return 10 from calculateCapacity() method

Now, 10 will be passed to ensureExplicitCapacity()

```
private void ensureCapacityInternal(int minCapacity) {  
    ensureExplicitCapacity(calculateCapacity(elementData,  
        minCapacity));  
}  
  
private void ensureExplicitCapacity(int minCapacity) {  
    modCount++;  
    // overflow-conscious code  
    if (minCapacity - elementData.length > 0)  
        grow(minCapacity);  
}
```

modCount is used when we are iterating over ArrayList using Iterator or ListIterator, here minCapacity is 10 and elementData.length will be 0, so if condition will be satisfied and grow() method will be called with value 10:

```
private void grow(int minCapacity) {  
    // overflow-conscious code  
    int oldCapacity = elementData.length;  
    int newCapacity = oldCapacity + (oldCapacity >> 1);  
    if (newCapacity - minCapacity < 0)  
        newCapacity = minCapacity;  
    if (newCapacity - MAX_ARRAY_SIZE > 0)  
        newCapacity = hugeCapacity(minCapacity);  
    // minCapacity is usually close to size, so this is a  
    // win:  
    elementData = Arrays.copyOf(elementData,  
        newCapacity);  
}
```

Here, oldCapacity will be 0, and newCapacity will also be 0, so the first if condition will be satisfied because ($0-10 < 0$), so newCapacity will be minCapacity i.e. 10, the second if condition will not be satisfied as MAX_ARRAY_SIZE is a very huge number,

```
private static final int MAX_ARRAY_SIZE =  
Integer.MAX_VALUE - 8;
```

So, the ensureCapacityInternal() of add() will be executed :

```
public boolean add(E e) {  
    ensureCapacityInternal(size + 1); // Increments  
    modCount!!  
    elementData [size ++] = e ;  
    return true ;  
}
```

Element e will be added to object array elementData at index 0 and size will be incremented and finally add() method will return true, now this same process will repeat.

So, we can say before adding the element in arrayList, first it is ensured that the array can hold the element, if not the capacity will be increased and for this, grow() method is called. Suppose you are trying to add 11th element in the list, then grow() method will be called and the statement int newCapacity = oldCapacity + (oldCapacity >> 1); will make the new capacity to be one and a half times(50% increase) the size of list.

Let's make a simple code to understand this line:

```
public class TestArrayList {  
    public static void main(String[] args) {  
  
        int oldCapacity = 10;  
  
        int newCapacity = oldCapacity + (oldCapacity >> 1);  
  
        System.out.println(newCapacity);  
    }  
}
```

Output:

15

Question 100: How to make an ArrayList as Immutable

Answer: This is also a common interview question nowadays. If your answer is making the list as “final” then see the code below:

Program 1:

```
public class TestArrayList {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();

        list.add("John");
        list.add("Mike");
        list.add("Lisa");

        System.out.println(list);
    }
}
```

Output:

[John, Mike, Lisa]

Although, we have made the list as final but still we are able to add elements into it, remember applying final keyword to a reference variable ensures that it will not be referenced again meaning you cannot give a new reference to list variable:

```
public class TestArrayList {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();

        list.add("John");
        list.add("Mike");
        list.add("Lisa");

        list = new ArrayList<>();
        System.out.println(list);
    }
}
```



So, to make the list as unmodifiable, there is a method `unmodifiableList()` in Collections utility class,

Program 2:

```
public class TestArrayList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Mike");
        list.add("Lisa");
        System.out.println(list);

        list = Collections.unmodifiableList(list);
        list.add("Jack");
        System.out.println(list);
    }
}
```

Output:

```
[John, Mike, Lisa]
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at com.collections.demo.TestArrayList.main(TestArrayList.java:16)
```

Here, if you assign `Collections.unmodifiableList (list);` to a new reference then you will be able to change the original list which will in turn change the new list also, see below:

Program 3:

```

public class TestArrayList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Mike");
        list.add("Lisa");

        List<String> newList = Collections.unmodifiableList(list);
        list.add("Jack");
        System.out.println(newList);
    }
}

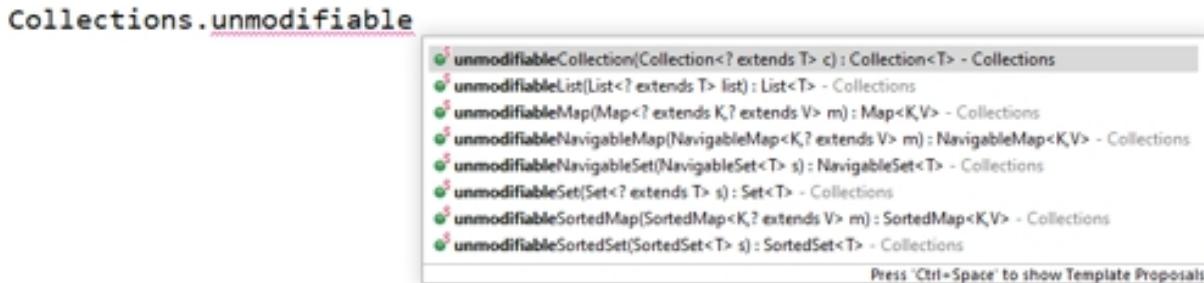
```

Output:

[John, Mike, Lisa, Jack]

Guava library also provides certain ways to make immutable list and Java 9 has List.of() method.

There are other utility methods also, to make unmodifiable collections:



Question 101: What is LinkedList?

Answer: Java LinkedList class is an implementation of linked list data structure and it uses a doubly linked list to store the elements. In Java LinkedList, elements are not stored in contiguous locations, they are stored at any available space in memory and they are linked with each other using pointers and addresses.

As Java LinkedList internally uses doubly linked list, so LinkedList class represents its elements as Nodes. A Node is divided into 3 parts:

Previous, Data, Next

Where Previous points to the previous Node in the list, Next points to the next Node in the list and Data is the actual data.

Some points about LinkedList class:

- LinkedList class maintains insertion order
- LinkedList class can contain duplicate elements
- LinkedList class is not synchronized
- LinkedList class can be used as list, stack or queue
- You can add any number of null elements in LinkedList

Time complexity of LinkedList's get(), add() and remove():

get(): As LinkedList does not store its elements in contiguous block of memory, random access is not supported here, elements can be accessed in sequential order only, so get() operation in LinkedList is O(n).

add() and **remove()**: Both add and remove operations in LinkedList is O(1), because no elements shifting is needed, just pointer modification is done (although remember getting to the index where you want to add/remove will still be O(n)).

Here, I am showing some portions of LinkedList Javadoc's:

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable,
    java.io.Serializable
{
    transient int size = 0;

    /**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     *             (first.prev == null && first.item !=
     * null)
    */
}
```

```

transient Node<E> first ;

/*
 * Pointer to last node.
 * Invariant: (first == null && last == null) ||
 *             (last.next == null && last.item != null)
 */

transient Node<E> last ;

/*
 * Constructs an empty list.
 */

public LinkedList() {
}

```

We can see that the `LinkedList` class implements `List` and `Deque` interfaces. There are `first` and `last` `Node` references also.

Let's see the `add()` method:

```

public boolean add(E e ) {
    linkLast(e );
    return true ;
}

void linkLast(E e ) {
    final Node<E> l = last ;
    final Node<E> newNode = new Node<>(l , e , null );
    last = newNode ;
    if (l == null )
        first = newNode ;
    else
        l .next = newNode ;
    size++;
    modCount++;
}

```

Here, in `linkLast()` method, `Node` class is used, let's see that:

```
private static class Node<E> {  
    E item ;  
    Node<E> next ;  
    Node<E> prev ;  
  
    Node(Node<E> prev , E element , Node<E> next ) {  
        this .item = element ;  
        this .next = next ;  
        this .prev = prev ;  
    }  
}
```

Here, we can see that Node class has 3 fields: item, prev and next.

Question 102: When to use ArrayList / LinkedList

Answer: When you have a requirement in which you will be doing a lot of add or remove operations near the middle of list, then prefer LinkedList, and when you have a requirement, where the frequently used operation is searching an element from the list, then prefer ArrayList as it allows random access.

Question 103: What is HashMap?

Answer: HashMap class implements the Map interface and it stores data in key, value pairs. HashMap provides constant time performance for its get() and put() operations, assuming the equals and hashCode method has been implemented properly, so that elements can be distributed correctly among the buckets.

Some points to remember:

- Keys should be unique in HashMap, if you try to insert the duplicate key, then it will override the corresponding key's value
- HashMap may have one null key and multiple null values
- HashMap does not guarantee the insertion order (if you want to maintain the insertion order, use LinkedHashMap class)
- HashMap is not synchronized
- HashMap uses an inner class Node<K, V> for storing map entries
- Hashmap has a default initial capacity of 16, which means it has 16 buckets or bins to store map entries, each bucket is a singly linked list. The default load factor in HashMap is 0.75
- Load factor is that threshold value which when crossed will double the hashmap's capacity i.e. when you add 13th element in hashmap, the capacity will increase from 16 to 32

Question 104: Explain the internal working of put() and get() methods of HashMap class and discuss HashMap collisions

Answer: If you are giving a Core java interview, then you must prepare for this question, as you will most likely be asked about this. So, let's get right into it:

put() method internal working:

When you call map.put(key,value), the below things happens:

- Key's hashCode() method is called
- Hashmap has an internal hash function which takes the key's hashCode and it calculates the bucket index
- If there is no element present at that bucket index, our <key, value> pair along with hash is stored at that bucket
- But if there is an element present at the bucket index, then key's hashCode is used to check whether this key is already present with the same hashCode or not.

If there is key with same hashCode, then equals method is used on the key. If equals method returns true, then the key's previous value is replaced with the new value otherwise a new entry is appended to the linked list.

get() method internal working:

When you call map.get(key), the below things happen:

- Key's hashCode() method is called
- Hash function uses this hashCode to calculate the index, just like in put method
- Now the key of element stored in bucket is compared with the passed key using equals() method, if both are equals, value is returned otherwise the next element is checked if it exists.

See HashMap's Javadoc:

Default capacity:

```
/**  
 * The default initial capacity - MUST be a power of  
 * two.  
 */  
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; //  
aka 16
```

Load factor:

```
/**  
 * The load factor used when none specified in  
 * constructor.  
 */  
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

Node class:

```
static class Node<K,V> implements Map.Entry<K,V> {  
    final int hash ;
```

```

final K key ;
V value ;
Node<K,V> next ;

Node(int hash , K key , V value , Node<K,V> next ) {
    this .hash = hash ;
    this .key = key ;
    this .value = value ;
    this .next = next ;
}

```

Internal Hash function:

```

static final int hash(Object key ) {
    int h ;
    return (key == null ) ? 0 : (h = key .hashCode()) ^
(h >>> 16);
}

```

Internal Data structure used by HashMap to hold buckets:

```
transient Node<K,V>[] table ;
```

HashMap's default constructor:

```

/*
 * Constructs an empty <tt> HashMap </tt> with the
 * default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this .loadFactor = DEFAULT_LOAD_FACTOR ; // all other
    fields defaulted
}


```

So, to conclude, Hashmap internally uses an array of Nodes named as table where each Node contains the calculated hash value, the key-value pair and the address to the next node.

HashMap collisions: it is possible that multiple keys will make the hash function generate the same index, this is called a collision. It happens because of poor hashCode method implementation.

One collision handling technique is called Chaining. Since every element in the array is a linked list, the keys which have the same hash function will be appended to the linked list.

Performance improvement in Java 8 : It is possible that due to multiple collisions, the linked list size has become very large, and as we know, searching in a linked list is $O(n)$, it will impact the constant time performance of hashmap's get() method. So, in Java 8, if the linked list size becomes more than 8, the linked list is converted to a binary search tree which will give a better time complexity of $O(\log n)$.

```
/**  
 * The bin count threshold for using a tree rather than  
list for a  
* bin. Bins are converted to trees when adding an  
element to a  
* bin with at least this many nodes. The value must be  
greater  
* than 2 and should be at least 8 to mesh with  
assumptions in  
* tree removal about conversion back to plain bins upon  
* shrinkage.  
*/  
static final int TREEIFY_THRESHOLD = 8;
```

Program showing the default capacity:

```
public class TestHashMap {  
    public static void main(String[] args) throws Exception {  
        HashMap<String, String> map = new HashMap<>();  
        map.put("name", "Mike");  
  
        Field tableField = HashMap.class.getDeclaredField("table");  
        tableField.setAccessible(true);  
        Object[] table = (Object[]) tableField.get(map);  
        System.out.print("hashmap capacity: ");  
        System.out.print(table == null ? 0 : table.length);  
        System.out.println("\nhashmap size:" + map.size());  
    }  
}
```

Output:

```
hashmap capacity: 16  
hashmap size:1
```

Program showing that hashmap's capacity gets doubled after load factor's threshold value breaches :

```
package com.hashmap.demo;  
  
import java.lang.reflect.Field;  
import java.util.HashMap;  
  
class Employee {  
    private int age;  
  
    public Employee(int age) {  
        this.age = age;  
    }  
}
```

```
public class TestHashMap {
    public static void main(String[] args) throws Exception {
        HashMap<Employee, String> map = new HashMap<>();
        for(int i=1;i<13;i++) {
            map.put(new Employee(i), "Hello " + i);
        }
        Field tableField = HashMap.class.getDeclaredField("table");
        tableField.setAccessible(true);
        Object[] table = (Object[]) tableField.get(map);
        System.out.print("hashmap capacity: ");
        System.out.print(table == null ? 0 : table.length);
        System.out.println("\nhashmap size:" + map.size());
    }
}
```

Output:

```
hashmap capacity: 16
hashmap size:12
```

Change the for loop condition from $i < 13$ to $i \leq 13$, see below:

```
package com.hashmap.demo;

import java.lang.reflect.Field;
import java.util.HashMap;

class Employee {
    private int age;

    public Employee(int age) {
        this.age = age;
    }
}
```

```

public class TestHashMap {
    public static void main(String[] args) throws Exception {
        HashMap<Employee, String> map = new HashMap<>();
        for(int i=1;i<=13;i++) {
            map.put(new Employee(i), "Hello " + i);
        }

        Field tableField = HashMap.class.getDeclaredField("table");
        tableField.setAccessible(true);
        Object[] table = (Object[]) tableField.get(map);
        System.out.print("hashmap capacity: ");
        System.out.print(table == null ? 0 : table.length);
        System.out.println("\nhashmap size:" + map.size());
    }
}

```

Output:

```

hashmap capacity: 32
hashmap size:13

```

Question 105: equals and hashCode method scenarios in HashMap when the key is a custom class

Answer: equals and hashCode methods are called when we store and retrieve values from hashmap. So, when the interviewer asks this question, it is mostly asked with an example, where the hashmap's key is a custom class and you are given some situations where either equals() is implemented or hashCode() is implemented, sometimes properly, sometimes not. We will discuss all combinations with programs below so you can give the correct answer in any situation.

Before we continue, just remember if in your custom class, you are not implementing equals() and hashCode(), then the Object class equals() and hashCode() will be called, and also remember about the contract between these 2 methods. It says when 2 objects are equal according to equals() method, then their hashCode must be same, reverse may not be true.

Scenario 1: when custom class does not implement both equals and hashCode methods

```
package com.hashmap.demo;

public class Employee {
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Here, Employee class has not given equals() and hashCode() method implementation, so Object's class equals() and hashCode() methods will be used when we use this Employee class as hashmap's key, and remember, equals() method of Object class compares the reference.

TestHashMap.java:

```
package com.hashmap.demo;

import java.util.HashMap;
import java.util.Map;

public class TestHashMap {
    public static void main(String[] args) {

        Map<Employee, Integer> map = new HashMap<>();

        Employee e1 = new Employee("Mike", 15);
        Employee e2 = new Employee("Mike", 15);
        Employee e3 = new Employee("John", 20);
        Employee e4 = e3;
```

```
System.out.println("e1 hashCode: " + e1.hashCode());
System.out.println("e2 hashCode: " + e2.hashCode());
System.out.println("e3 hashCode: " + e3.hashCode());
System.out.println("e4 hashCode: " + e4.hashCode());

System.out.println("e1 equals e2: " + e1.equals(e2));
System.out.println("e3 equals e4: " + e3.equals(e4));

map.put(e1, 100);
map.put(e2, 200);
map.put(e3, 300);
map.put(e4, 400);

System.out.println(map.get(e1));
System.out.println(map.get(e2));
System.out.println(map.get(e3));
System.out.println(map.get(e4));
System.out.println("hashmap size: " + map.size());
    }
}
```

Can you predict the output of this one?

Output:

```
e1 hashCode: 366712642
e2 hashCode: 1829164700
e3 hashCode: 2018699554
e4 hashCode: 2018699554
e1 equals e2: false
e3 equals e4: true
100
200
400
400
hashmap size: 3
```

Here, Employee objects e1 and e2 are same but they are both inserted in the HashMap because both are created using new keyword and holding a different reference, and as the Object's equals() method checks reference, they both are unique.

And as for objects e3 and e4, they both are pointing to same reference (e4 = e3), so they are equal according to Object's equals() method hence the value of e3 which was 300 gets replaced with the value 400 of the same key e4, and finally size of HashMap is 3.

Scenario 2: when only equals() method is implemented by Employee class

```
package com.hashmap.demo;

public class Employee {

    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (age != other.age)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
```

Now, can you predict the output of our TestHashMap class?

Let's see the output:

```
e1 hashCode: 366712642
e2 hashCode: 1829164700
e3 hashCode: 2018699554
e4 hashCode: 2018699554
e1 equals e2: true
e3 equals e4: true
100
200
400
400
hashmap size: 3
```

Well, nothing's changed here. Because even though e1 and e2 are equal according to our newly implemented equals() method, they still have different hashCode as the Object's class hashCode() is used. So the equals and hashCode contract is not followed and both e1, e2 got inserted in HashMap.

Scenario 3: when only hashCode() method is implemented:

```
package com.hashmap.demo;

public class Employee {
    private String name;
    private int age;
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
}
```

Let's run our TestHashMap class again and see the output:

```
e1 hashCode: 2399656
e2 hashCode: 2399656
e3 hashCode: 2316120
e4 hashCode: 2316120
e1 equals e2: false
e3 equals e4: true
100
200
400
400
hashmap size: 3
```

Well, now we have same hashCode for e1 and e2, but Object's equals method still checks the references and as references are different, both are not equal and are inserted in the hashmap.

Scenario 4: When both equals and hashCode are implemented properly:

```
package com.hashmap.demo;

public class Employee {
    private String name;
    private int age;
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + age;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Employee other = (Employee) obj;
    if (age != other.age)
        return false;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    return true;
}
}
```

Output:

```
e1 hashCode: 2399656
e2 hashCode: 2399656
e3 hashCode: 2316120
e4 hashCode: 2316120
e1 equals e2: true
e3 equals e4: true
200
200
400
400
hashmap size: 2
```

Here, both e1 and e2 are equals as we are comparing the contents of them in our equals() method, so their hashCodes must be same, which they are. So value of e1 which was 100 got replaced by 200, and size of hashmap is 2.

You can be asked to write the equals() and hashCode() methods implementation by hand also, so you should pay attention to how these are implemented.

Question 106: How to make a HashMap synchronized?

Answer: Collections.synchronizedMap(map);

Question 107: What is ConcurrentHashMap?

Answer: ConcurrentHashMap class provides concurrent access to the map, this class is very similar to HashTable, except that ConcurrentHashMap provides better concurrency than HashTable or even synchronizedMap.

Some points to remember:

- ConcurrentHashMap is internally divided into segments, by default size is 16 that means, at max 16 threads can work at a time
- Unlike HashTable, the entire map is not locked while reading/writing from the map
- In ConcurrentHashMap, concurrent threads can read the value without locking
- For adding or updating the map, the lock is obtained on segment level, that means each thread can work on each segment during high concurrency
- Concurrency level defines a number, which is an estimated number of threads concurrently accessing the map
- ConcurrentHashMap does not allow null keys or null values
- put() method acquires lock on the segment
- get() method returns the most recently updated value
- iterators returned by ConcurrentHashMap are fail-safe and never throw ConcurrentModificationException

Question 108: What is HashSet class and how it works internally?

Answer: HashSet is a class in Java that implements the Set Interface and it allows us to have the unique elements only. HashSet class does not maintain the insertion order of elements, if you want to maintain the insertion order, then you can use LinkedHashSet.

Internal implementation of HashSet:

HashSet internally uses HashMap and as we know the keys are unique in hashmap, the value passed in the add() method of HashSet is stored as the key of hashmap, that is how Set maintains the unique elements.

```
/*
 * Constructs a new, empty set; the backing <tt> HashMap
 </tt> instance has
 * default initial capacity (16) and load factor (0.75).
 */
public HashSet() {
    map = new HashMap<>();
}

private transient HashMap<E, Object> map ;
```

Let's see add() method's Javadoc:

```
public boolean add(E e) {
    return map.put(e, PRESENT) == null;
}

// Dummy value to associate with an Object in the
// backing Map
private static final Object PRESENT = new Object();
```

So, when we call hashSet.add(element) method then

- map.put() is called where key is the element and value is the dummy value (the map.put() method internal working has already been discussed above)
- if value is added in the map then put method will return null which will be compared with null, hence returning true from hashSet.add() method indicating the element is added
- however if the element is already present in the map, then the value associated with the element will be returned which in turn will be compared with null, returning false from hashSet.add() method

set.contains() method:

```
public boolean contains(Object o) {  
    return map.containsKey(o);  
}
```

The passed object is given to map.containsKey() method, as the HashSet's values are stored as the keys of internal map.

NOTE: If you are adding a custom class object inside the HashSet, do follow equals and hashCode contract. You can be asked the equals and hashCode scenarios questions, just like we discussed in HashMap (Question 105).

Question 109: Explain Java's TreeMap

Answer: TreeMap class is one of the implementation of Map interface.

Some points to remember:

- TreeMap entries are sorted based on the natural ordering of its keys. This means if we are using a custom class as the key, we have to make sure that the custom class is implementing Comparable interface
- TreeMap class also provides a constructor which takes a Comparator object, this should be used when we want to do a custom sorting
- TreeMap provides guaranteed $\log(n)$ time complexity for the methods such as containsKey(), get(), put() and remove()
- TreeMap iterator is fail-fast in nature, so any concurrent modification will result in ConcurrentModificationException
- TreeMap does not allow null keys but it allows multiple null values
- TreeMap is not synchronized, so it is not thread-safe. We can make it thread-safe by using utility method, Collections.synchronizedSortedMap(treeMap)
- TreeMap internally uses Red-Black tree based NavigableMap implementation.

Red-Black tree algorithm has the following properties:

- Color of every node in the tree is either red or black.
- Root node must be Black in color.
- Red node cannot have a red color neighbor node.
- All paths from root node to the null should consist the same number of black nodes

Program 1: Using Wrapper class as key

```
public class TestTreeMap {  
    public static void main(String[] args) {  
  
        TreeMap<Integer, String> map = new TreeMap<>();  
  
        map.put(4, "Mike");  
        map.put(1, "John");  
        map.put(3, "Jack");  
        map.put(2, "Lisa");  
  
        map.forEach((k,v) -> System.out.println(k + ":" + v));  
    }  
}
```

Output:

```
1:John  
2:Lisa  
3:Jack  
4:Mike
```

Here, Integer class already implements Comparable interface, so the keys are sorted based on the Integer's natural sorting order (ascending order).

Let's see, when key is a custom class:

Program 2:

```
class Employee {  
    String name;  
    int age;  
    Employee(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```

public class TestTreeMap {
    public static void main(String[] args) {

        TreeMap<Employee, Integer> map = new TreeMap<>();

        map.put(new Employee("Mike", 20), 100);
        map.put(new Employee("John", 10), 500);
        map.put(new Employee("Ryan", 15), 200);
        map.put(new Employee("Lisa", 20), 400);

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}

```

Output:

```

Exception in thread "main" java.lang.ClassCastException: com.treemap.demo.Employee cannot be cast to java.lang.Comparable
at java.util.TreeMap.compare(Unknown Source)
at java.util.TreeMap.put(Unknown Source)
at com.treemap.demo.TestTreeMap.main(TestTreeMap.java:19)

```

We get ClassCastException at runtime. Now, let's implement Comparable interface in Employee class and provide implementation of its compareTo() method:

Program 3:

```

class Employee implements Comparable<Employee> {
    String name;
    int age;
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int compareTo(Employee emp) {
        return this.name.compareTo(emp.name);
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + "]";
    }
}

```

```
public class TestTreeMap {
    public static void main(String[] args) {

        TreeMap<Employee, Integer> map = new TreeMap<>();
        map.put(new Employee("Mike", 20), 100);
        map.put(new Employee("John", 10), 500);
        map.put(new Employee("Ryan", 15), 200);
        map.put(new Employee("Lisa", 20), 400);

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Here, we are sorting based on Employee name,

Output:

```
Employee [name=John, age=10]:500
Employee [name=Lisa, age=20]:400
Employee [name=Mike, age=20]:100
Employee [name=Ryan, age=15]:200
```

Let's look at a program where we pass a Comparator in the TreeMap constructor, and sort the Employee object's based on age in descending order:

Program 4:

```
class Employee {
    String name;
    int age;
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + "]";
    }
}

public class TestTreeMap {
    public static void main(String[] args) {
        TreeMap<Employee, Integer> map = new TreeMap<>(
            new Comparator<Employee>() {
                @Override
                public int compare(Employee e1, Employee e2) {
                    if(e1.age < e2.age){
                        return 1;
                    } else if(e1.age > e2.age) {
                        return -1;
                    }
                    return 0;
                }
            }
        );
        map.put(new Employee("Mike", 20), 100);
        map.put(new Employee("John", 10), 500);
        map.put(new Employee("Ryan", 15), 200);
        map.put(new Employee("Lisa", 40), 400);

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Output:

```
Employee [name=Lisa, age=40]:400
Employee [name=Mike, age=20]:100
Employee [name=Ryan, age=15]:200
Employee [name=John, age=10]:500
```

Here, in Employee class, I have not implemented equals() and hashCode()

TreeMap's Javadoc:

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable,
java.io.Serializable
{
    /**
     * The comparator used to maintain order in this tree
     * map, or
     * null if it uses the natural ordering of its keys.
     *
     * @serial
     */
    private final Comparator<? super K> comparator ;
    private transient Entry<K,V> root ;
```

No-arg TreeMap constructor:

```
public TreeMap() {
    comparator = null ;
```

TreeMap constructor which takes comparator object:

```
public TreeMap(Comparator<? super K> comparator ) {
    this .comparator = comparator ;
}
```

TreeMap.put() method excerpt:

```
public V put(K key , V value ) {  
    Entry<K,V> t = root ;  
    if (t == null ) {  
        compare(key , key ); // type (and possibly null)  
        check  
  
        root = new Entry<>(key , value , null );  
        size = 1;  
        modCount ++;  
        return null ;  
    }  
    int cmp ;  
    Entry<K,V> parent ;  
    // split comparator and comparable paths  
    Comparator<? super K> cpr = comparator ;  
    if (cpr != null ) {  
        do {  
            parent = t ;  
            cmp = cpr .compare(key , t .key );  
            if (cmp < 0)  
                t = t .left ;  
            else if (cmp > 0)  
                t = t .right ;  
            else  
                return t .setValue(value );  
        } while (t != null );  
    }  
}
```

Question 110: Explain Java's TreeSet

Answer: TreeSet class is one of the implementation of Set interface

Some points to remember:

- TreeSet class contains unique elements just like HashSet
- TreeSet class does not allow null elements
- TreeSet class is not synchronized
- TreeSet class internally uses TreeMap, i.e. the value added in TreeSet is internally stored in the key of TreeMap
- TreeSet elements are ordered using their natural ordering or by a Comparator which can be provided at the set creation time
- TreeSet provides guaranteed $\log(n)$ time cost for the basic operations (add, remove and contains)
- TreeSet iterator is fail-fast in nature

TreeSet Javadoc:

```
public class TreeSet<E> extends AbstractSet<E>
implements NavigableSet<E>, Cloneable,
java.io.Serializable

public TreeSet() {
    this (new TreeMap<E, Object>());
}

public TreeSet(Comparator<? super E> comparator ) {
    this (new TreeMap<>(comparator ));
}
```

Question 111: Difference between fail-safe and fail-fast iterators

Answer: Iterators in Java are used to iterate over the Collection objects.

Fail-fast iterators : immediately throw *ConcurrentModificationException*, if the collection is modified while iterating over it. Iterator of ArrayList and HashMap are fail-fast iterators.

All the collections internally maintain some sort of array to store the elements, Fail-fast iterators fetch the elements from this array. Whenever, we modify the collection, an internal field called modCount is updated.

This modCount is used by Fail-safe iterators to know whether the collection is structurally modified or not. Every time when the Iterator's next() method is called, it checks the modCount. If it finds that modCount has been updated after the Iterator has been created, it throws ConcurrentModificationException.

Program 1:

```
public class FailFastIteratorTest {  
    public static void main(String[] args) {  
  
        ArrayList<Integer> list = new ArrayList<>();  
        list.add(1);  
        list.add(2);  
        list.add(3);  
  
        Iterator<Integer> itr = list.iterator();  
        while(itr.hasNext()) {  
            System.out.println(itr.next());  
            list.add(4);  
        }  
    }  
}
```

Output:

```
Exception in thread "main"  
java.util.ConcurrentModificationException  
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)  
at java.util.ArrayList$Itr.next(ArrayList.java:859)  
at com.iterator.demo.FailFastIteratorTest.main(FailFastIteratorTest.java:16)
```

But they don't throw the exception, if the collection is modified by Iterator's remove() method.

Program 2:

```
public class FailFastIteratorTest {  
    public static void main(String[] args) {  
  
        ArrayList<Integer> list = new ArrayList<>();  
        list.add(1);  
        list.add(2);  
        list.add(3);  
        System.out.println("List: " + list);  
        Iterator<Integer> itr = list.iterator();  
        while(itr.hasNext()) {  
            System.out.println(itr.next());  
            itr.remove();  
        }  
        System.out.println("List: " + list);  
    }  
}
```

Output:

List: [1, 2, 3]

1

2

3

List: []

Javadoc:

arrayList.iterator() method:

```
public Iterator<E> iterator() {  
    return new Itr();  
}
```

Itr is a private nested class in ArrayList:

```
private class Itr implements Iterator<E> {
    int cursor ;           // index of next element to return
    int lastRet = -1;      // index of last element returned;
    -1 if no such
    int expectedModCount = modCount ;
    Itr() {}
    public boolean hasNext() {
        return cursor != size ;
    }
}
```

Itr.next() method:

```
@SuppressWarnings ("unchecked" )
public E next() {
    checkForComodification();
    int i = cursor ;
    if (i >= size )
        throw new NoSuchElementException();
    Object[] elementData = ArrayList.this .elementData ;
    if (i >= elementData .length )
        throw new ConcurrentModificationException();
    cursor = i + 1;
    return (E) elementData [lastRet = i ];
}
```

See the first statement is a call to checkForComodification():

```
final void checkForComodification() {
    if (modCount != expectedModCount )
        throw new ConcurrentModificationException();
}
```

On the other hand, ***Fail-safe iterators*** does not throw *ConcurrentModificationException*, because they operate on the clone of the collection, not the actual collection. This also means that any

modification done on the actual collection goes unnoticed by these iterators. The last statement is not always true though, sometimes it can happen that the iterator may reflect modifications to the collection after the iterator is created. But there is no guarantee of it.

CopyOnWriteArrayList, ConcurrentHashMap are the examples of fail-safe iterators.

Program 1: ConcurrentHashMap example

```
public class FailSafeIteratorTest {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
        map.put(1, "Mike");
        map.put(2, "John");
        map.put(3, "Lisa");

        Iterator<Integer> itr = map.keySet().iterator();
        while(itr.hasNext()) {
            Integer key = itr.next();
            System.out.println(key + " : " + map.get(key));
            map.put(4, "Ryan");
        }
        System.out.println("Map: " + map);
    }
}
```

Output:

```
1 : Mike
2 : John
3 : Lisa
4 : Ryan
Map: {1=Mike, 2=John, 3=Lisa, 4=Ryan}
```

Here, iterator is reflecting the element which was added during the iteration operation.

Program 2: CopyOnWriteArrayList example

```
public class FailSafeIteratorTest {  
    public static void main(String[] args) {  
        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();  
        list.add(1);  
        list.add(2);  
        list.add(3);  
  
        Iterator<Integer> itr = list.iterator();  
        while(itr.hasNext()) {  
            System.out.println(itr.next());  
            list.add(4);  
        }  
        System.out.println("List: " + list);  
    }  
}
```

Output:

1

2

3

List: [1, 2, 3, 4, 4, 4]

Question 112: Difference between Iterator and ListIterator

Answer:

- Iterator can traverse the collection only in one direction i.e. forward direction but ListIterator can traverse the list in both directions, forward as well as backward, using previous() and next() method
- Iterator cannot add element to a collection while iterating over it, but ListIterator can add elements while iterating over the list
- Iterator cannot modify an element while iterating over a collection, but ListIterator has set(E e) method which can be used to modify the element
- Iterator can be used with List, Set or Map, but ListIterator only works with List
- Iterator has no method to obtain an index of the collection elements but ListIterator has methods like previousIndex() and nextIndex() which can be used to obtain the index

Question 113: Difference between Iterator.remove and Collection.remove()

Answer: Iterator.remove() does not throw ConcurrentModificationException while iterating over a collection but Collection.remove() method will throw ConcurrentModificationException.

Java Collection framework is very important topic when preparing for the interviews, apart from the above questions, you can read about Stack, Queue topics also, but if you are short on time, what we have discussed so-far should be enough for the interview.

Question 114: What is the difference between a Monolith and Micro-service architecture?

Answer: In monolithic architecture, applications are built as one large system, whereas in micro-service architecture we divide the application into modular components which are independent of each other.

Monolithic architecture has some advantages:

- Development is quite simple
- Testing a monolith is also simple, just start the application and do the end-to-end testing, Selenium can be used to do the automation testing
- These applications are easier to deploy, as only one single jar/war needs to be deployed
- Scaling is simple when the application size is small, we just have to deploy one more instance of our monolith and distribute the traffic using a load balancer
- Network latency is very low/none because of one single codebase

However, there are various disadvantages of monolith architecture as well:

- Rolling out a new version means redeploying the entire application
- Scaling a monolith application becomes difficult once the application size increases. It also becomes difficult to manage
- The size of the monolith can slow down the application start-up and deployment time
- Continuous deployment becomes difficult
- A bug in any module can bring down the entire application
- It is very difficult to adopt any new technology in a monolith application, as it affects the whole application, both in terms of time and cost

Micro-service architecture gives following advantages:

- Micro-services are easier to manage as they are relatively smaller in size
- Scalability is a major advantage of using a micro-service architecture, each micro-service can be scaled independently
- Rolling out a new version of micro-service means redeploying only that micro-service
- A bug in micro-service will affect only that micro-service and its consumers, not the entire application
- Micro-service architecture is quite flexible. We can choose different technologies for different micro-services according to business requirements
- It is not that difficult to upgrade to newer technology versions or adopt a newer technology as the codebase is quite smaller (this point is debatable in case the micro-service becomes very large)
- Continuous deployment becomes easier as we only have to re-deploy that micro-service

Despite all these advantages, Micro-services also comes with various disadvantages:

- As micro-services are distributed, it becomes complex compared to monolith. And this complexity increases with the increase in micro-services
- As micro-services will need to communicate with each other, it increases the network latency. Also, extra efforts are needed for a secure communication between micro-services
- Debugging becomes very difficult as one micro-service will be calling other micro-services and to figure out which micro-service is causing the error, becomes a difficult task in itself
- Deploying a micro-service application is also a complex task, as each service will have multiple instances and each instance will need to be configured, deployed, scaled and monitored
- Breaking down a large application into individual components as micro-services is also a difficult task

We have discussed both advantages and disadvantages of monolith and micro-services, you can easily figure out the differences between them, however, I am also stating them below.

The differences are:

- In a monolithic architecture, if any fault occurs, it might bring down the entire application as everything is tightly coupled, however, in case of micro-service architecture, a fault affects only that micro-service and its consumers
- Each micro-service can be scaled independently according to requirement. For example, if you see that one of your micro-service is taking more traffic, then you can deploy another instance of that micro-service and then distribute the traffic between them. Now, with the help of cloud computing services such as AWS, the applications can be scaled up and down automatically. However, in case of monolith, even if we want to scale one service within the monolith, we will have to scale the entire monolith
- In case of monolith, the entire technology stack is fixed at the start. It will be very difficult to change the technology at a later stage in time. However, as micro-services are independent of each other, they can be coded in any language, taking the advantage of different technologies according to the use-case. So micro-services gives you the freedom to choose different technologies, frameworks etc.
- Deploying a new version of a service in monolith requires more time and it increases the application downtime, however, micro-services entails comparatively lesser downtime

Question 115: What is Dependency Injection in Spring?

Answer: Dependency Injection is the most important feature of Spring framework. Dependency Injection is a design pattern where the dependencies of a class are injected from outside, like from an xml file. It ensures loose-coupling between classes.

In a Spring MVC application, the controller class has dependency of service layer classes and the service layer classes have dependencies of DAO layer classes.

Suppose class A is dependent on class B. In normal coding, you will create an object of class B using ‘new’ keyword and call the required method of class B. However, what if you can tell someone to pass the object of class B in class A? Dependency injection does this. You can tell Spring, that class A needs class B object and Spring will create the instance of class B and provide it in class A.

In the above example, we can see that we are passing the control of objects to Spring framework, this is called Inversion of Control (IOC) and Dependency injection is one of the principles that enforce IOC.

Question 116: What are the different types of Dependency Injection?

Answer: Spring framework provides 2 ways to inject dependencies:

- By Constructor
- By Setter method

Constructor-based DI : when the required dependencies are provided as arguments to the constructor, then it is known as constructor-based dependency injection, see the examples below:

Using XML based configuration:

Injecting a dependency is done through the bean-configuration file, for this <constructor-arg> xml tag is used:

```
<bean id="classB" class="com.dependency.demo.B" />

<bean id="classA" class="com.dependency.demo.A">
    <constructor-arg ref="classB" />
</bean>
```

In case of more than 1 dependency, the order sequence of constructor arguments should be followed to inject the dependencies.

Java Class A:

```
package com.dependency.demo;

public class A {
    B b;
    public A (B b) {
        this.b = b;
    }
}
```

Java Class B:

```
package com.dependency.demo;

public class B {
}
```

Using Java Based Configuration:

When using Java based configuration, the constructor needs to be annotated with @Autowired annotation to inject the dependencies,

Our classes A and B will be annotated with @Component (or any other stereotype annotation), so that they will be managed by Spring.

```
package com.dependency.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class A {

    B b;

    @Autowired
    public A (B b) {
        this.b = b;
    }
}

package com.dependency.demo;
import org.springframework.stereotype.Component;

@Component
public class B {
```

Before Spring version 4.3, @Autowired annotation was needed for constructor dependency injection, however, in newer Spring versions, @Autowired is optional, if the class has only one constructor.

But, if the class has multiple constructors, we need to explicitly add @Autowired to one of the constructors so that Spring knows which constructor to use for injecting the dependencies.

Setter-method injection: in this, the required dependencies are provided as the field parameters to the class and the values are set using setter methods of those properties. See the examples below.

Using XML based configuration:

Injecting a dependency is done through the bean configuration file and <property> xml tag is used where ‘name’ attribute defines the name of the field of java class.

```
<bean id="classB" class="com.dependency.demo.B" />

<bean id="classA" class="com.dependency.demo.A">
    <property name="b">
        <ref bean="classB" />
    </property>
</bean>
```

Java class A:

```
package com.dependency.demo;
public class A {

    B b;

    public void setB (B b) {
        this.b = b;
    }

}
```

Java class B:

```
package com.dependency.demo;

public class B {

}
```

Using Java based configuration:

The setter method needs to be annotated with @Autowired annotation.

```
package com.dependency.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class A {

    B b;

    @Autowired
    public void setB (B b) {
        this.b = b;
    }
}

package com.dependency.demo;
import org.springframework.stereotype.Component;

@Component
public class B {
```

There is also a Field injection, where Spring injects the required dependencies directly into the fields when those fields are annotated with @Autowired annotation.

Question 117: Difference between Constructor and Setter injection

Answer: The differences are:

- Partial dependency is not possible with Constructor based injection, but it is possible with Setter based injection. Suppose there are 4 properties in a class and the class has setter methods and a constructor with 4 parameters. In this case, if you want to inject only one/two property, then it is only possible with setter methods (unless you can define a new parametrized constructor with the needed properties)
- Cyclic dependency is also not possible with Constructor based injection. Suppose class A has dependency on class B and class B has dependency on class A and we are using constructor based injection, then when Spring tries to create object of class A, it sees that it needs class B object, then it tries to resolve that dependency first. But when it tries to create object of class B, it finds that it needs class A object, which is still under construction. Here Spring recognizes that a circular reference may have occurred and you will get an error in this case. This problem can easily be solved by using Setter based injection because dependencies are not injected at the object creation time
- While using Constructor injection, you will have to remember the order of parameters in a constructor when the number of constructor parameters increases. This is not the case with Setter injection
- Constructor injection helps in creating immutable objects, because a bean object is created using constructor and once the object is created, its dependencies cannot be altered anymore. Whereas with Setter injection, it's possible to inject dependency after object creation which leads to mutable objects.

Use constructor-based injection, when you want your class to not even be instantiated if the class dependencies are not resolved because Spring container will ensure that all the required dependencies are passed to the constructor.

Question 118: What is @Autowired annotation?

Answer: @Autowired is a way of telling Spring that auto-wiring is required. It can be applied to field, constructor and methods.

Question 119: What is the difference between BeanFactory and ApplicationContext?

Answer: The differences are:

- BeanFactory is the most basic version of IOC containers which should be preferred when memory consumption is critical whereas ApplicationContext extends BeanFactory, so you get all the benefits of BeanFactory plus some advanced features for enterprise applications
- BeanFactory instantiates beans on-demand i.e. when the method `getBean(beanName)` is called, it is also called Lazy initializer whereas ApplicationContext instantiates beans at the time of creating the container where bean scope is Singleton, so it is an Eager initializer
- BeanFactory only supports 2 bean scopes, *singleton and prototype* whereas ApplicationContext supports all bean scopes
- ApplicationContext automatically registers BeanFactoryPostProcessor and BeanPostProcessor at startup, whereas BeanFactory does not register these interfaces automatically
- Annotation based dependency injection is not supported by BeanFactory whereas ApplicationContext supports it
- If you are using plain BeanFactory, features like transactions and AOP will not take effect (not without some extra steps), even if nothing is wrong with the configuration whereas in ApplicationContext, it will work
- ApplicationContext provides additional features like MessageSource access (i18n or Internationalization) and Event Publication

Use an *ApplicationContext* unless you have a really good reason for not doing so.

Question 120: Explain the life-cycle of a Spring Bean

Answer: Spring beans are java classes that are managed by Spring container and the bean life-cycle is also managed by Spring container.

The bean life-cycle has below steps:

- Bean instantiated by container
- Required dependencies of this bean are injected by container
- Custom Post initialization code to be executed (if required)
- Bean methods are used
- Custom Pre destruction code to be executed (if required)

When you want to execute some custom code that should be executed before the bean is in usable state, you can specify an *init()* method and if some custom code needs to be executed before the bean is destroyed, then a *destroy()* method can be specified.

There are various ways to define these init() and destroy() method for a bean:

By using xml file,

<bean> tag has 2 attributes that can be used to specify its init and destroy methods,

```
<bean id="testClass" class="com.example.demo.Test"  
      init-method="init" destroy-method="destroy" />
```

You can give any name to your initialization and destroy methods, and here is our Test class

```
package com.example.demo;

public class Test {
    public void init() throws Exception{
        //do some initialization task
        System.out.println("init method");
    }

    public void destroy() throws Exception{
        //do some cleanup task
        System.out.println("destroy method");
    }
}
```

By implementing InitializingBean and DisposableBean interfaces

InitializingBean interface has afterPropertiesSet() method which can be used to execute some initialization task for a bean and DisposableBean interface has a destroy() method which can be used to execute some cleanup task.

Here is our Test class,

```
package com.example.demo;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Test implements InitializingBean, DisposableBean {
    @Override
    public void afterPropertiesSet() throws Exception{
        //do some initialization task
        System.out.println("init method");
    }

    @Override
    public void destroy() throws Exception{
        //do some cleanup task
        System.out.println("destroy method");
    }
}
```

And, in the xml file:

```
<bean id="testClass" class="com.example.demo.Test" />
```

By using @PostConstruct and @PreDestroy annotations

```
package com.example.demo;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class Test {
    @PostConstruct
    public void init() throws Exception{
        //do some initialization task
        System.out.println("init method");
    }

    @PreDestroy
    public void destroy() throws Exception{
        //do some cleanup task
        System.out.println("destroy method");
    }
}
```

And, in the xml file:

```
<bean id="testClass" class="com.example.demo.Test" />
```

Question 121: What are the different scopes of a Bean?

Answer: Spring framework supports 5 scopes,

singleton – only one bean instance per Spring IOC container

prototype – it produces a new instance each and every time a bean is requested

request – a single instance will be created and made available during complete life-cycle of an HTTP request

session – a single instance will be created and made available during complete life-cycle of an HTTP session

global session – a single instance will be created during the life-cycle of a *ServletContext*

@Scope annotation or *scope attribute* of *bean tag* can be used to define bean scopes in Spring.

Question 122: What is the Default scope of a bean?

Answer: Default scope of a bean is ***Singleton*** that means only one instance per context.

Question 123: What happens when we inject a prototype scope bean in a singleton scope bean?

Answer: When you define a bean scope to be singleton, that means only one instance will be created and whenever we request for that bean, that same instance will be returned by the Spring container, however, a prototype scoped bean returns a new instance every time it is requested.

Spring framework gets only one chance to inject the dependencies, so if you try to inject a prototyped scoped bean inside a singleton scoped bean, Spring will instantiate the singleton bean and will inject one instance of prototyped scoped bean. This one instance of prototyped scoped bean is the only instance that is ever supplied to the singleton scoped bean.

So here, whenever the singleton bean is requested, you will get the same instance of prototyped scoped bean.

Question 124: How to inject a prototype scope bean in a singleton scope bean?

Answer: We have discussed in the previous question that when a prototyped scoped bean is injected in a singleton scoped bean, then on each request of singleton bean, we will get the same instance of prototype

scoped bean, but there are certain ways where we can get a new instance of prototyped scoped bean also.

The solutions are:

- Injecting an ApplicationContext in Singleton bean and then getting the new instance of prototyped scoped bean from this ApplicationContext
- Lookup method injection using @Lookup
- Using scoped proxy

Injecting ApplicationContext:

To inject the ApplicationContext in Singleton bean, we can either use @Autowired annotation or we can implement ApplicationContextAware interface,

```
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
            throws BeansException {
        this.applicationContext = applicationContext;
    }

    public PrototypeBean getPrototypeBean() {
        return applicationContext.getBean(PrototypeBean.class);
    }
}
```

Here, whenever the getPrototypeBean() method is called, it will return a new instance of PrototypeBean.

But this approach contradicts with Spring IOC (Inversion of Control), as we are requesting the dependencies directly from the container.

Lookup Method Injection using @Lookup:

```

import org.springframework.beans.factory.annotation.Lookup;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean {

    @Lookup
    public PrototypeBean getPrototypeBean() {
        return null;
    }

}

```

Here, Spring will dynamically overrides `getPrototypeBean()` method annotated with `@Lookup` and it will look up the bean which is the return type of this method. Spring uses CGLIB library to do this.

Using Scoped Proxy

```

import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean {

    @Bean
    @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
           proxyMode = ScopedProxyMode.TARGET_CLASS)
    public PrototypeBean getPrototypeBean() {
        return new PrototypeBean();
    }

}

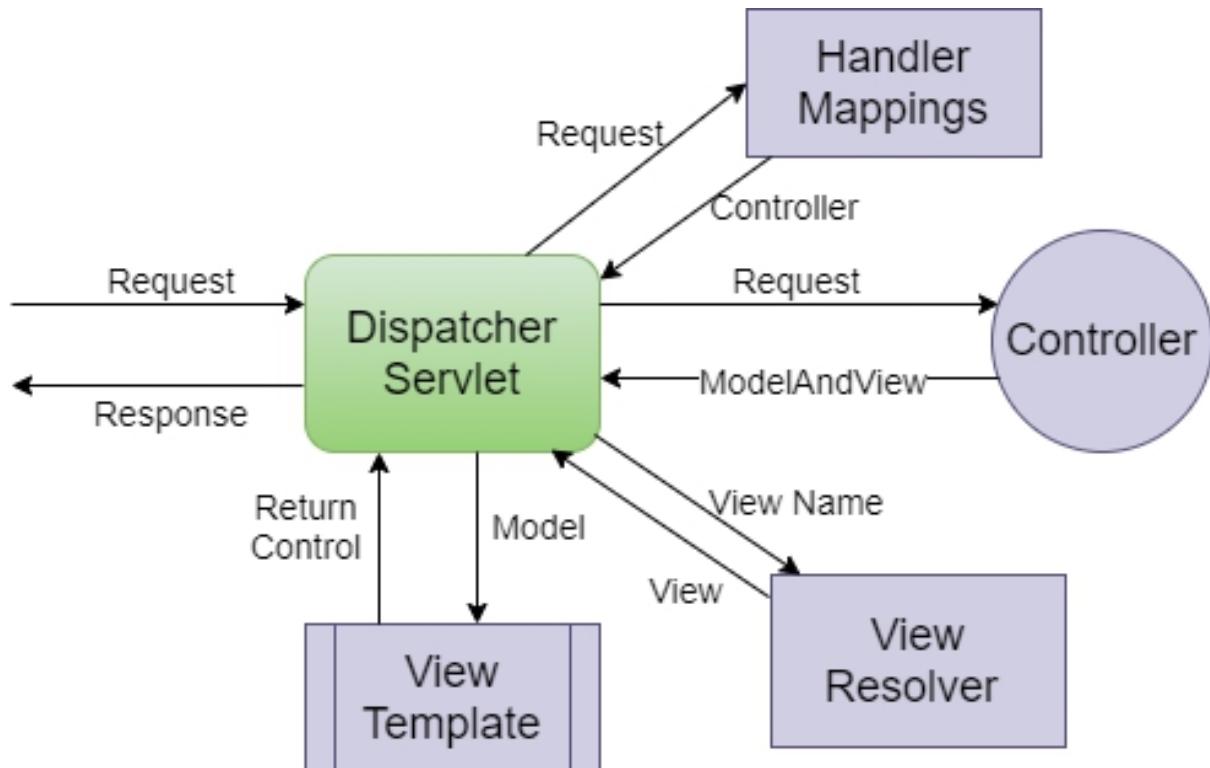
```

Spring uses CGLIB to create the proxy object and the proxy object delegates method calls to the real object. In the above example, we are using `ScopedProxyMode.TARGET_CLASS` which causes an AOP proxy to be injected at the target injection point. The default Proxy mode is `ScopedProxyMode.NO`.

To avoid CGLIB usage, configure the proxy mode with `ScopedProxyMode.INTERFACES` and it will use JDK dynamic proxy.

Question 125: Explain Spring MVC flow

Answer:



In Spring, DispatcherServlet acts as the front Controller. When a request comes in Spring MVC application, below steps get executed,

- the request is first received by the DispatcherServlet
- DispatcherServlet will take the help of HandlerMapping and it will get to know the specific Controller that is associated with this request using @RequestMapping's
- Now, the request gets transferred to its associated Controller, the Controller will process this request by executing appropriate methods and returns the ModelAndView object back to the DispatcherServlet
- The DispatcherServlet will transfer this object to ViewResolver to get the actual view page
- Finally, DispatcherServlet passes the Model object to the View page which displays the result

Remember, in developing REST services, the Controller's request mapping methods are annotated with @ResponseBody annotations, so they don't return a logical view name to DispatcherServlet, instead it writes the output directly into the HTTP response body.

Question 126: What is the difference between <context:annotation-config> and <context:component-scan>?

Answer: The differences are:

<context:annotation-config> is used to activate annotations in beans that are already registered in the application context. So for example, if you have a class A that is already registered in the context and you have @Autowired, @Qualifier annotations in the class A, then <context:annotation-config> resolves these annotations.

<context:component-scan> can also do what <context:annotation-config> does, but *component-scan* also scans the packages for registering the beans to application context. If you are using, *component-scan*, then there is no need to use *annotation-config*.

Question 127: What is the difference between Spring and SpringBoot?

Answer: SpringBoot makes it easy to work with Spring framework. When using Spring framework, we have to take care of all the configuration ourselves like, for making a web application, DispatcherServlet, ViewResolver etc configurations are needed. SpringBoot solves this problem through a combination of Auto Configuration and Starter projects.

Question 128: What is auto-configuration in SpringBoot?

Answer: Spring applications have a lot of XML or Java Bean Configurations. Spring Boot Auto configuration looks at the jars available on the CLASSPATH and configuration provided by us in the application.properties file and it tries to auto-configure those classes.

For example, if Spring MVC jar is on the classpath, then DispatcherServlet will be automatically configured and if Hibernate jar is on the classpath, then a DataSource will be configured (Of course, we will have to provide datasource url, username and password).

Question 129: What are SpringBoot starters?

Answer: Think of SpringBoot starters as a set of related jars that we can use for our application, we don't have to go out and add the dependencies one by one and worry about which version will be compatible with the spring boot version that you are using, starters take care of all that.

For example, when you want to make a web application, you simply will add 'spring-boot-starter-web' as a dependency and you will have all the jars that are needed for a web application, like, DispatcherServlet, ViewResolver, Embedded Tomcat etc. Similarly, 'spring-boot-starter-data-jpa' dependency can be used when you want to work with Spring and JPA.

Question 130: What is @SpringBootApplication Annotation?

Answer: @SpringBootApplication is a combination of 3 different annotations:

@Configuration: This annotation marks a class as a Configuration class in Java-based configuration, it allows to register extra beans in the context or import additional configuration classes

@ComponentScan: to enable component scanning

@EnableAutoConfiguration: to enable Spring Boot's auto-configuration feature

These 3 annotations are frequently used together, so SpringBoot designers bundled them into one single @SpringBootApplication, now instead of 3 annotations you just need to specify only one annotation on the Main class. However, if you don't need one of these annotation depending on your requirement, then you will have to use them separately.

Question 131: Where does a Spring Boot application start from?

Answer: The *SpringApplication* class provides a convenient way to bootstrap a Spring application that is started from a main() method. In many situations, you can delegate to the static *SpringApplication.run* method, as shown in the following example:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

Question 132: How to remove certain classes from getting auto-configured in SpringBoot?

Answer: @SpringBootApplication annotation accepts below parameters that can be used to remove certain classes from taking part in auto-configuration.

exclude: Exclude the list of classes from the auto-configuration

excludeNames: Exclude the list of class names from the auto-configuration

Question 133: How to autowire a class which is in a package other than SpringBoot application class's package or any of its sub-packages

Answer: When you specify @ComponentScan annotation on a class, then it starts scanning for the components from that package and its sub-packages, so if your class is in a different package altogether, then you will have to explicitly tell Spring to look into your package,

@ComponentScan annotation has below parameters:

scanBasePackages : Base packages to scan for annotated components

scanBasePackageClasses: Type-safe alternative to scanBasePackages() for specifying the packages to scan for annotated components. The package of each class specified will be scanned.

Question 134: What is application.properties file in a SpringBoot application?

Answer: SpringBoot allows us to configure our application configuration using application.properties file. In this file, we can define the configurations that will be needed to configure a datasource, like

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb  
spring.datasource.username=adminUser  
spring.datasource.password=adminPass
```

We can define the logging level of certain packages, using

```
logging.level.somePackagePath=INFO  
logging.level.someOtherPackagePath=DEBUG
```

We can also define the port number that our embedded server will run on, using

```
server.port=9000
```

We can have different application.properties file for different environments like dev, stage and prod.

Question 135: How to configure the port number of a SpringBoot application?

Answer: By default, the embedded server starts on port 8080, however you can change it by defining a property in application.properties file, like:

```
server.port=8082
```

You can also pass it as a vm argument:

```
java -jar C:\temp\app.jar -server.port=8082
```

or

```
java -jar -Dserver.port=8082 C:\temp\app.jar
```

Question 136: Which jar builds our springboot application automatically whenever we change some code just like a node.js application?

Answer: Devtools dependency, just add the below maven dependency in your pom.xml,

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-devtools</artifactId>  
</dependency>
```

Question 137: What default embedded server is given in spring boot web starter and how we can change the default embedded server to the one of our choice

Answer: The default embedded server is Tomcat, that comes with Spring boot web starter, if you want to change this, then use <exclusion> tag in web starter and add a separate dependency of the server that you want.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

When you add the exclusion tag and save the pom.xml, you will see that the tomcat dependency will be removed from Maven Dependencies, then you can add another server's dependency like the one below:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Question 138: Where should we put our html and javascript files in a spring boot application?

Answer: If you are adding html and javascript files, then you should put them in *static folder* in **src/main/resources/**. You should also make separate folders for html, css, javascript files.

Question 139: What are the different stereotype annotations?

Answer: @Component, @Controller, @Service and @Repository annotations are called stereotype annotations and they are present in `org.springframework.stereotype` package.

Question 140: Difference between @Component, @Controller, @Service, @Repository annotations?

Answer:

@Component: it is a general purpose stereotype annotation which indicates that the class annotated with it, is a spring managed component.

@Controller, @Service and @Repository are special types of @Component, these 3 themselves are annotated with @Component, see below

```
@Target (value=TYPE)
@Retention (value=RUNTIME)
@Documented
@Component
public @interface Controller

@Target (value=TYPE)
@Retention (value=RUNTIME)
@Documented
@Component
public @interface Service
```

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
@Component
public @interface Repository
```

So, the classes annotated with these annotations gets picked up in Component scanning and they are managed by Spring.

@Controller: the classes annotated with @Controller will act as Spring MVC controllers. DispatcherServlet looks for @RequestMapping in classes that are annotated with @Controller. That means you cannot replace @Controller with @Component, if you just replace it with @Component then yes it will be managed by Spring but it will not be able to handle the requests.

(Note: if a class is registered with Spring using @Component, then @RequestMapping annotations within class can be picked up, if the class itself is annotated with @RequestMapping)

@Service: the service layer classes that contain the business logic should be annotated with @Service. Apart from the fact that it is used to indicate that the class contains business logic, there is no special meaning to this annotation, however it is possible that Spring may add some additional feature to @Service in future, so it is always good idea to follow the convention.

@Repository: the classes annotated with this annotation defines data repositories. It is used in DAO layer classes. @Repository has one special feature that it catches platform specific exceptions and re-throw them as one of the Spring's unified unchecked exception i.e. *DataAccessException* .

Question 141: Difference between @Controller and @RestController annotation

Answer: The differences are:

- `@Controller` annotation is used to mark a class as Spring MVC controller where the response is a view name which will display the Model object prepared by controller, whereas `@RestController` annotation is a specialization of `@Controller` and it is used in RESTful web services where the response is usually JSON/XML.
- `@RestController` is made up of 2 annotations, `@Controller` and `@ResponseBody`. `@ResponseBody` annotation is used to attach the generated output directly into the body of http response.
- `@Controller` can be used with `@ResponseBody` which will have same effect as `@RestController`. `@ResponseBody` annotation can be used at the class level or at the individual methods also. When it is used at the method level, Spring will use HTTP Message Converters to convert the return value to HTTP response body (serialize the object to response body).

Question 142: What is `@RequestMapping` and `@RequestParam` annotation?

Answer: `@RequestMapping` annotation maps the incoming HTTP request to the handler method that will serve this request, we can apply `@RequestMapping` on a class as well as a method. When used on a class, it maps a specific request path or pattern onto a controller, then the method level annotation will make the mappings more specific to handler methods.

`@RequestParam` annotation is used to bind a web request parameter to the parameter of handler method

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/home")
public class DemoController {

    @RequestMapping(value="/getById")
    public String getById(@RequestParam(value = "id") String requestId) {
        return "dummy response";
    }
}
```

In the above `@RequestParam` example, we have defined a value parameter that just denotes that the incoming request parameter is by the name ‘id’ which will be mapped to the variable name ‘requestId’, you can define `@RequestParam` without this value attribute also.

By default, all requests are assumed to be of type HTTP GET, however we can specify the request type by using ‘method’ attribute of `@RequestMapping` annotation, like

```
@RequestMapping(method = RequestMethod.POST)
public String post() {
    return "Post request";
}
```

```
@RequestMapping(method = RequestMethod.PUT)
public String put() {
    return "Put request";
}
```

```
@RequestMapping(method = RequestMethod.PATCH)
public String patch() {
    return "Patch request";
}

@RequestMapping(method = RequestMethod.DELETE)
public String delete() {
    return "Delete request";
}
```

Question 143: How to define a Get or Post endpoint?

Answer: You can either use the @RequestMapping and defining its 'method' attribute as RequestMethod.GET, RequestMethod.POST or you can also use the shortcuts like @GetMapping and @PostMapping annotations.

```
@GetMapping("/getByName/{name}")
public String getMapping(@PathVariable String name) {
    return "Get";
}

@PostMapping("/saveUser")
public String postMapping() {
    return "Post";
}
```

There are similar shortcut annotations present for PUT, PATCH, DELETE requests too.

Question 144: Which annotation is used for binding the incoming json request to the defined pojo class?

Answer: @RequestBody annotation is used to bind the incoming json request to the pojo class,

```
@RequestMapping(value = "/createUser", method = RequestMethod.POST)
public String createUser(@RequestBody User user) {
    return "Created user " + user.getName();
}
```

Behind the scenes, Spring uses Jackson library (that comes with spring-boot-starter-web) to map the json request to the pojo class.

MappingJackson2HttpMessageConverter is used when the incoming request is of content-type application/json.

Question 145: What is @Qualifier annotation?

Answer: Let's consider an example to understand @Qualifier annotation better. Suppose we have an interface called Shape and there are 2 classes Rectangle and Circle that are implementing this interface. We are autowiring our Shape interface in our controller class using @Autowired, now here a conflict will happen, because there are 2 beans of the same type.

```
public interface Shape {  
  
}  
  
@Service
public class Rectangle implements Shape {  
  
}  
  
@Service
public class Circle implements Shape {  
  
}
```

```
@RestController
public class ShapeController {

    @Autowired
    Shape shape;
```

When you try to start your application, you will get:

```
*****
APPLICATION FAILED TO START
*****
Description:

Field shape in com.example.demo.controller.ShapeController required a single bean, but 2 were found:
 - circle: defined in file [C:\demo\target\classes\com\example\service\Circle.class]
 - rectangle: defined in file [C:\demo\target\classes\com\example\service\Rectangle.class]
```

Now, to resolve this you can give names to your Rectangle and Circle class, like:

```
@Service("rectangle")
public class Rectangle implements Shape {

}

@Service("circle")
public class Circle implements Shape {

}
```

And you will use @Qualifier annotation to specify which bean should be autowired, like:

```
@RestController
public class ShapeController {

    @Autowired
    @Qualifier("circle")
    Shape shape;
```

Now, Spring will not get confused as to what bean it has to autowire.

NOTE, you can also use @Qualifier annotation to give names to your Rectangle and Circle classes, like

```
@Service
@Qualifier("rectangle")
public class Rectangle implements Shape {

}
```

Question 146: What is @Transactional annotation?

Answer: Spring provides Declarative Transaction Management via @Transactional annotation. When a method is applied with @Transactional, then it will execute inside a database transaction. @Transactional annotation can be applied at the class level also, in that case, all methods of that class will be executed inside a database transaction.

How @Transactional works:

When @Transactional annotation is detected by Spring, then it creates a proxy object around the actual bean object. So, whenever the method annotated with @Transactional is called, the request first comes to the proxy object and this proxy object invokes the same method on the target

bean. These proxy objects can be supplied with interceptors. Spring creates a TransactionInterceptor and passes it to the generated proxy object. So, when the @Transactional annotated method is called, it gets called on the proxy object first, which in turn invokes the TransactionInterceptor that begins a transaction. Then the proxy object calls the actual method of the target bean. When the method finishes, the TransactionInterceptor commits/rollbacks the transaction.

One thing to remember here is that the Spring wraps the bean in the proxy, the bean has no knowledge of it. So, only the external calls go through the proxy. As for the internal calls (@Transactional method calling the same bean method), they are called using ‘this’.

Using @Transactional annotation, the transaction’s propagation and isolation can be set directly, like:

```
@Transactional(propagation = Propagation.REQUIRES_NEW,  
           isolation = Isolation.READ_UNCOMMITTED,  
           rollbackFor = Exception.class)  
public String process() {  
    return "Success";  
}
```

Also, you can specify a ‘rollbackFor’ attribute and specify which exception types must cause a transaction rollback (a transaction with Runtime exceptions and errors are by default rolled back).

If your process() method is calling another bean method, then you can also annotate that method with @Transactional and set the propagation level to decide whether this method should execute in the same transaction or it requires a new transaction.

Question 147: What is @ControllerAdvice annotation?

Answer: @ControllerAdvice annotation is used to intercept and handle the exceptions thrown by the controllers across the application, so it is a global exception handler. You can also specify @ControllerAdvice for a specific package,

```
@ControllerAdvice(basePackages="com.example.demo.controller")
public class Test {
```

Or a specific controller,

```
@ControllerAdvice(assignableTypes=DemoController.class)
public class Test {
```

Or even a specific annotation,

```
@ControllerAdvice(annotations=RestController.class)
public class Test {
```

@ExceptionHandler annotation is used to handle specific exceptions thrown by controllers, like,

```
@ControllerAdvice
public class Test {

    @ExceptionHandler(SQLException.class)
    public String handleSQLException() {
        return null;
    }

    @ExceptionHandler(UserNotFoundException.class)
    public String handleUserNotFoundException() {
        return null;
    }

}
```

Here, we have defined a global exception handler using @ControllerAdvice. If a SQLException gets thrown from a controller, then handleSQLException() method will be called. In this method, you can customize the exception and send a particular error page/error code. Also, custom exceptions can be handled.

If you don't want to create a global exception handler, then you can also define some `@ExceptionHandler` methods in a particular controller itself.

Question 148: What is `@Bean` annotation?

Answer: `@Bean` annotation is used when you want to explicitly declare and register a bean into application context, so that it will be managed by Spring.

Some points to remember:

- When using `@Bean`, you have the control over the bean creation logic.
- `@Bean` is a method level annotation, the body of the method contains the logic for creating the bean instance and this method returns the instance which will be registered in the spring application context.
- Using `@Bean`, you can register the classes from 3rd party libraries into the application context
- `@Bean` annotation is usually declared in configuration classes.

Question 149: Difference between `@Component` and `@Bean`

Answer: The differences are:

- `@Component` auto-detects and configures the beans using classpath scanning, whereas `@Bean` explicitly declares a single bean rather than letting Spring do it automatically
- `@Component` is a class level annotation, whereas `@Bean` is a method level annotation
- `@Component` has different specializations called stereotype annotations like `@Controller`, `@Service` and `@Repository`, whereas `@Bean` has no specializations
- `@Bean` lets you create and configure beans exactly how you choose it to be, whereas in `@Component`, Spring has the control
- `@Bean` lets you configure classes from 3rd party libraries where you are not the owner of the source code, but you can't use `@Component` in this case

Question 150: How to do profiling in a SpringBoot application

Answer: Every application has many environments like DEV, STAGE, UAT, Pre-PROD, PROD. And all these environments have different configurations like the database properties, port number etc. In Spring boot, we define all properties in `application.properties` file. But if we want to maintain different profiles, then we can have multiple `application.properties` file for each environment and at the runtime, we can specify which profile should be active.

The profiles can be created using the syntax, ***application-{profile_name}.properties***

For example,

`application-dev.properties` for dev environment specific configs

`application-uat.properties` for uat environment specific configs

`application-prod.properties` for prod environment specific configs

How to activate a particular profile:

There are many ways to activate a particular profile, one of them is by defining a property in our master application.properties file, like

`spring.profiles.active=uat`

We can also set a particular profile to be active by passing it as a VM argument, Maven settings, JVM system parameters, environment variables. In xml, we use `<beans profile="uat">` to activate a profile.

There is one `@Profile` annotation which when applied to a component tells that this component will be created only when a particular profile is set.

Question 151: What is RestTemplate?

Answer: RestTemplate is used to consume other REST services programmatically. It can also bind the api response to the custom domain classes. You must have used/seen the HttpURLConnection to consume other services, it contains a lot of boilerplate code, like configuring the HTTP request, executing it and then converting the HTTP response to a Java object. But when using RestTemplate, all these things happen in the background.

```
public String getUserId() {
    User user = new RestTemplate().getForObject("http://localhost:8080/getUser/1", User.class);
    return user.getName();
}
```

In this example, we are consuming a GET web service and converting the response to the object of User class.

Similarly, there are other methods to consume POST, DELETE web services like `exchange()` and `delete()` respectively.

Question 152: What is Spring Data JPA?

Answer: Spring Data is part of Spring framework. It provides an abstraction to significantly reduce the amount of boilerplate code required to implement data access layers. It does this by providing Repositories. Repositories are just interfaces that do not have any implementation class and provide various database operations like save, delete etc. We have

several repositories to choose from, `CRUDRepository`, `JPARepository`, `PagingAndSortingRepository`.

If you are thinking how Spring Data JPA saves us from writing boilerplate code, then consider an example, you have an `Employee` class and you are writing its Dao interface and implementation class for performing some CRUD operations. You will also have other classes and similarly, you will write CRUD operations logic for these classes as well. So, there is a lot of boilerplate code here. Spring Data JPA takes care of this, and provides you with a `Repository` interface which have all the common DAO operations. We just have to extend these Repositories and Spring Data JPA will provide the DAO implementation at runtime.

Spring Data JPA can also generate JPA queries on our behalf, we just have to follow certain method naming conventions and the database query will be automatically generated for us.

For example, let's say we have a table named `Employee` with `id`, `name` and `age` columns. If you want to write a query that will give the `Employee` object by name, then you can use Spring Data JPA, like:

```
public EmployeeEntity findByName(String name);
```

Spring Data JPA will transform this method into:

```
select * from Employee where name='passed_name_value';
```

Question 153: What is the difference between `JPARepository`, `CRUDRepository`, `PagingAndSortingRepository` and which one you have used in your applications?

Answer: `JPARepository` extends the `PagingAndSortingRepository` which in turn extends the `CRUDRepository`. So if you are using `JPARepository`, you can use all the methods of other two also.

`CRUDRepository` – this repo mainly provides methods related to CRUD operations

PagingAndSortingRepository – this repo extends the CRUDRepository and provides methods to do pagination and sorting of records

JPARepository – this repo extends the PagingAndSortingRepository and provides JPA related methods like flushing the persistence context and deleting records in batches

You can extend any of these repository and make a customized repo to have methods according to your need.

Question 154: What is Spring AOP?

Answer: Spring AOP (Aspect Oriented Programming) is one of the key components of Spring framework.

Consider an example to understand the AOP better, let's say we have 100 methods and you have to maintain logs in these methods, like what is the method name and what is the result of each method. One way is to go in each method and write logger statements. What if this requirement changes and you don't need this logger anymore then you will again go in each method and remove this logger. One thing is clear from this example that Logging is a cross-cutting concern.

A cross-cutting concern is a concern that can affect the entire application and should be centralized in one location as much as possible. Few examples of cross-cutting concerns are, transaction management, logging, authentication etc.

AOP helps you to refactor the different necessary repeating codes into different modules. By doing this, the cluttering of code can be removed and the focus can be applied on the actual business logic. AOP provides modularity but here the key unit of modularity is Aspect. Using AOP, you can add extra functionality before or after a method execution.

AOP terminology :

Aspect : Aspect is a class that implements the application concerns that cuts across multiple classes, such as transaction management and logging.

Aspects can be a normal class configured through Spring XML configuration or we can use @Aspect annotation.

Join Point : a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

Advice : advices are actions that are taken for a particular join point. There are different types of advices, like, before, after and around advice.

Pointcut : pointcut is an expression that is matched with join points to determine whether advice needs to be applied or not.

Target Object : objects on which advices are applied by one or more aspects. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.

AOP Proxy : an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

Weaving : It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

AOP Advice types:

Before advice : these advices run before the execution of join point methods. @Before annotation is used to mark a method as before advice.

After returning advice : Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception. @AfterReturning annotation is used to mark a method as after returning advice.

After throwing advice : Advice to be executed if a method exits by throwing an exception. @AfterThrowing annotation marks a method as after throwing advice.

After (finally) advice : An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. @After annotation is used to create an after advice.

Around advice : this advice surrounds a join point. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception. @Around annotation is used to create around advice methods.

Question 155: Have you used Spring Security in your application

Answer: If you are giving a Spring/SpringBoot interview, then this question will definitely be asked that how did you secure your application APIs? Every application uses some type of security to protect the unwanted access, so if you are working in a project and even if you have not implemented the security part yourself, then I suggest, you should try to understand how it is implemented and read more about it. This will be better than not answering the question. You can always say you worked on it as a team or you have implemented it in some of your personal projects

Question 156: What do you know about Spring Batch framework?

Answer: Spring batch framework can be used when a task requires batch processing, for example, generating financial reports at end of day or month. Batch processing includes tasks like reading and writing to files, transforming data, reading and writing to databases etc. These steps are often chained together and can also be executed sequentially or in parallel. If an error occurs in a batch job step, then it can be found out that in which step, the error has occurred and it is also possible to resume that job execution from that failed step.

Question 157: Difference between SOAP and REST

Answer: The differences are:

- SOAP stands for Simple Object Access Protocol and REST stands for Representational State Transfer
- SOAP is a protocol whereas REST is an architectural style
- SOAP cannot use REST because it is a protocol whereas REST can use SOAP web services as the underlying protocol, because it is just an architectural pattern
- SOAP uses service interfaces to expose its functionality to client applications whereas REST uses URI to expose its functionality
- SOAP defines standards that needs to be strictly followed for communication to happen between client and server whereas REST does not follow any strict standard
- SOAP requires more bandwidth and resources than REST because SOAP messages contain a lot of information whereas REST requires less bandwidth than SOAP because REST messages mostly just contains a simple JSON message
- SOAP only works with XML format whereas REST allows different data formats like Plain text, HTML, XML, JSON etc.
- SOAP defines its own security whereas REST inherits the security

Question 158: What is Restful api?

Answer: For an api to be Restful, it has to satisfy 6 constraints, they are:

Client-Server : This constraint operates on the concept that the client and the server should be separate from each other and allowed to evolve individually.

Stateless : REST is stateless, it means the client request should contain all the information that is needed for a server to respond.

Cacheable : Cache constraint requires that response returned from a server should be labelled as cacheable or non-cacheable either implicitly or explicitly. If the response is defined as Cacheable then the client can re-use

the response data for future requests because a stateless API can increase request overhead by handling large loads of incoming and outgoing calls.

Uniform Interface : The key to decoupling the client from server is having a uniform interface that allows independent evolution of the application without having the application's services, or models and actions, tightly coupled to the API layer itself.

Layered System : REST APIs have different layers of their architecture working together to build a hierarchy that helps create a more scalable and modular application.

Code on Demand (Optional) : this is an Optional constraint. It allows servers to temporarily extend the functionality of a client by downloading and executing code in the form of applets or scripts.

Question 159: What is a stateless object?

Answer: An instance of a class without any instance fields is a stateless object. The class can have fields but they can only be constants, *static final* .

Question 160: What is the difference between Web Server and Application Server?

Answer: The differences are:

- Web server provides environment to run only web related applications, whereas Application Server provides environment to run Java J2EE applications (enterprise applications)
- Web server is used to deliver static contents like static html pages, whereas through Application server, you can deliver dynamic content
- Web servers can be used for Servlets, JSPs whereas Application servers can be used for Servlets, JSPs, EJBs, JMS etc. (Application servers have an internal web server inside it to serve web applications)
- Web servers support HTTP protocol, whereas Application servers support HTTP as well as RPC/RMI protocols
- Web server consume less resources than Application servers
- Tomcat, Jetty are examples of Web Servers, whereas GlassFish, JBoss, WebLogic, WebSphere are some examples of Application servers

***Question 161: What do you know about
CommandLineRunner and ApplicationRunner?***

Answer: SpringBoot provides us with two interfaces, CommandLineRunner and ApplicationRunner. Both these runners have a run() method, which gets executed just after the application context is created and before SpringBoot application startup.

You just have to register these runners as beans in application context. Then Spring will automatically pick them up.

Both of these interfaces provide the same functionality and the only difference between them is CommandLineRunner.run() method accepts String arr[], whereas ApplicationRunner.run() method accepts ApplicationArguments as argument.

Multiple runners can be defined and we can order them as well, either by extending the interface org.springframework.core.Ordered or via the @Order annotation.

Knowing these topics well should be enough to crack a Spring/SpringBoot interview, if you want to really impress the interviewer, you can prepare the below topics as well:

- Spring Cloud, how different environment properties are stored in Git
- Eureka Naming Server
- Zuul
- Zipkin
- Hystrix

All the above topics are advanced but I will explain about them a little, so you can go out and read more about it.

Question 162: What do you know about Eureka Naming Server?

Answer:

To know about the need of Eureka Naming Server and what it does, let's consider an example. Suppose you have 5 micro-services which are experiencing heavy traffic and you want to deploy multiple instances of these 5 micro-services and use a load balancer to distribute the traffic among these instances. Now, when you create new instances of your micro-service, you have to configure these in your load balancer, so that load balancer can distribute traffic properly. Now, when your network traffic will reduce then you will most likely want to remove some instances of your micro-service, means you will have to remove the configuration from your load balancer. I think, you see the problem here.

This manual work that you are doing can be avoided by using Eureka naming server. Whenever a new service instance is being created/deleted, it will first register/de-register itself to the Eureka naming server. Then you can simply configure a **Ribbon client** (Load Balancer) which will talk with Eureka Naming server to know about the currently running instances of your service and Ribbon will properly distribute the load between them.

Also, if one service, serviceA wants to talk with another service, serviceB then also Eureka Naming server will be used to know about the currently running instances of serviceB.

You can configure Eureka Naming Server and Ribbon client in SpringBoot very easily.

Question 163: What do you know about Zuul?

Answer:

Zuul is an API gateway server. It handles all the requests that are coming to your application. As it handles all the requests, you can implement some common functionalities of your micro-services as part of Zuul server like Security, Monitoring etc. You can monitor the incoming traffic to gain some insights and also provide authentication at a single place rather than repeating it in your services. Using Zuul, you can dynamically route the incoming requests to the respective micro-services. So, the client doesn't have to know about the internal architecture of all the services, it will only call the Zuul server and Zuul will internally route the request.

Question 164: What do you know about Zipkin?

Answer:

To understand Zipkin use-case, let's consider an example. Suppose you have a chain of 50 micro-services where first micro-service is calling the second and second calling third and so on. Now, if there is an error in, say 35th micro-service, then how you will be able to identify your request that you made to the first micro-service, from all the logs that gets generated in all 35 micro-services. I know this is an extreme example

Zipkin helps in distributed tracing, especially in a micro-service architecture. It assigns an 'id' to each request and gives you a dashboard, where you can see the complete request and a lot more details, like the entire call-chain, how much time one micro-service took and which service failed etc.

Question 165: What do you know about Hysterix?

Answer:

Hysterix is a library that makes our micro-service, fault-tolerant. Suppose, you have a chain of 10 micro-services calling each other and the 6th one fails for some reason, then your application will stop working until the failed micro-service is fixed.

You can use Hysterix here and provide a fallback method in case of a service failure.

```
@GetMapping("/getByName/{name}")
@HystrixCommand(fallbackMethod = "handlerMethod")
public String getMapping(@PathVariable String name) {
    return "Get";
}

public String handlerMethod() {
    return "Service is down";
}
```

If the GET service is getting failed, then the fallback method will be executed.

JPA / Hibernate

Question 166: What is JPA?

Answer: JPA stands for Java Persistence API. It is a specification which gives a standard API for accessing databases within java applications. As JPA is just a specification, it does not perform any operation by itself. It requires an implementation, there are many JPA implementations available like Hibernate, iBatis, TopLink, EclipseLink etc. Hibernate ORM is the most popular implementation of JPA.

Question 167: What is Hibernate?

Answer: Hibernate is an Object Relational Mapping tool (ORM tool), that maps the Java objects to the database tables and vice-versa.

Some points to remember:

- Hibernate framework provides the facility to create database tables automatically
- Hibernate framework provides us object-oriented version of SQL known as HQL (Hibernate Query Language). It generates the database independent queries. So, even if our database gets changed, we don't have to change our SQL queries according to the new database
- Using Hibernate, we can define relationships between our Entities (tables), that makes it easy to fetch data from multiple tables
- Hibernate supports Caching, that improves the performance of our application
- Using Hibernate, we can generate the Primary key of our tables automatically

Question 168: Difference between JPA and Hibernate

Answer: JPA is just a specification i.e. it defines a set of concepts that can be implemented by any tool or framework, and Hibernate is one of the

implementation of JPA.

Question 169: What is @Entity?

Answer: @Entity annotation defines that a class can be mapped to a database table. The class fields will be mapped to the columns of the table.

Question 170: How to give a name to a table in JPA?

Answer: @Table annotation can be used to give name to a table

```
import javax.persistence.Entity;
import javax.persistence.Table;
```

```
@Entity
@Table(name="USER")
public class User {
```

Question 171: What is @Id, @GeneratedValue?

Answer: @Id annotation defines the primary key of a table and @GeneratedValue annotation is used to specify the primary key generation strategy to use. If the strategy is not specified, the default strategy AUTO will be used.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private int userId;
```

Question 172: How to use a custom database sequence in Hibernate to generate primary key values?

Answer: JPA specification provides a set of annotations to define the primary key generation strategy. To use a custom sequence, we have to set

the *GenerationType* to SEQUENCE in @GeneratedValue annotation, this tells Hibernate to use a database sequence to generate the primary key value. If we don't provide any other information, Hibernate will use its default sequence.

To define the name and schema of the database sequence, there is an annotation @SequenceGenerator.

See the code snippet for doing all this:

```
@Id  
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "userSeq_generator")  
@SequenceGenerator(name = "userSeq_generator", sequenceName = "user_seq")
```

If you persist a new entity, then Hibernate will use 'user_seq'.

Question 173: How to give names to the columns of a JPA Entity

Answer: @Column annotation can be used to give names to a column

```
@Column(name="address")  
private String userAddress;
```

Question 174: How to define a @OneToMany relationship between entities

Answer: Here the interviewer will ask you about the syntax, so to define a One to Many relationship, consider a class *Cart* that can have multiple items represented by class *Item*. So, one Cart -> many Items.

Cart Entity:

```
@Entity
public class Cart {

    @OneToMany(mappedBy = "cart")
    private List<Item> items;
```

In mappedBy attribute, the field name of Item entity is passed, see below,

Item Entity:

```
@Entity
public class Item {

    @ManyToOne
    @JoinColumn(name = "CART_ID")
    private Cart cart;
```

@JoinColumn annotation is used to define the name of foreign key column that represents the entity association.

Question 175: Why annotations should be imported from JPA and not from Hibernate?

Answer: If you see @Entity annotation, it is present in *javax.persistence* package and also in *org.hibernate.annotations* package:

`@Entity`

 **Entity** - javax.persistence

 **Entity** - org.hibernate.annotations

You should choose the one from `javax.persistence` package, because if you choose the Hibernate one, then in future, if by any chance, you have to remove Hibernate and use some other JPA implementation, like iBatis, then you will have to change your code (imports). As you remember, JPA is just a specification, like an interface. You can plug any of its implementations as long as you use the imports from `javax.persistence` package.

Question 176: What is the difference between get() and load() method of Hibernate Session?

Answer: Hibernate Session class provides two methods to access object, `session.get()` and `session.load()`

The differences are:

- `get()` method involves a database hit, if the object does not exist in Session cache and it returns a fully initialized object which may involve several database calls, whereas `load()` method returns a proxy object and it only hit the database if any method other than `getId()` is called on the entity object
- `load()` method results in slightly better performance as it can return a proxy object, it will only hit the database when a non-identifier getter method is called, whereas `get()` method returns a fully initialized object when it does not exist in Session cache which may involve multiple database calls based on entity relationships
- `get()` method returns null if the object is not found in the cache as well as the database whereas `load()` method will throw `ObjectNotFoundException` but never return null
- If you are not sure whether the object exists or not, then use `get()` as it will return null but if you are sure that the object exists, then use `load()` method as it is lazily initialized

Question 177: What is the difference between save(), saveOrUpdate() and persist() method of Hibernate Session?

Answer: Hibernate Session class provides various methods to save an object into the database, like save(), saveOrUpdate() and persist()

The difference between save() and saveOrUpdate() method is that save() method saves the record into the database by INSERT query, generates a new identifier and returns the Serializable identifier back, while saveOrUpdate() method either INSERT the record or UPDATE the record if it already exists, so it involves extra processing to find whether the record already exists in the table or not.

Similar to save(), persist() method is also used to save the record into the database table.

The differences between save() and persist() are:

- Return type of persist() method is void while return type of save() method is Serializable object
- Both persist() and save() methods makes a transient instance persistent. But persist() method does not guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time
- Both behave differently when they are executed outside the transaction boundaries. persist() method ensures that it will not execute an INSERT when it is called outside of a transaction boundary whereas save() method does not guarantee this, it returns an identifier and if an INSERT query has to be executed to get the identifier then this INSERT happens immediately and it does not matter if the save() is called inside or outside of a transaction
- persist() method is useful in long-running conversation with an extended Session context because it does not execute an INSERT outside of a transaction. On the other hand, save() method is not good in a long-running conversation with an extended Session context

Question 178: What is Session and SessionFactory in Hibernate?

Answer: SessionFactory creates and manages the Session objects.

Some points about SessionFactory:

- it is one instance per datasource/database
- it is thread-safe
- it is an immutable and heavy-weight object as it maintains Sessions, mappings, hibernate configurations etc.
- SessionFactory provides second level cache in hibernate also called application-level cache

Some points about Session:

- Session objects are created using sessionFactory.openSession()
- It is one instance per client/thread/transaction
- It is not thread-safe
- It is light-weight
- Session provides first level cache, which is short-lived

Question 179: What is First Level and Second Level Cache in Hibernate?

Answer: Hibernate framework provides caching at two levels, first-level cache which is at the Session level and second-level cache which is at the application level.

The **first level cache** minimizes the database access for the same object if it is requested from the same Session. The first level cache is by default enabled. When you call session.get() method then it hits the database, and while returning, it also saves this object in the first-level cache. So, the subsequent requests for this same object from the same session will not hit the database and the object from cache will be used.

But, since this cache is associated with the Session object, which is a short-lived object in Hibernate, as soon as the session is closed, all the information held in the cache is also lost. So, if we try to load the same object using the get() method, Hibernate will go to the database again and fetch the record.

This poses a significant performance challenge in an application where multiple sessions are used, Hibernate provides second-level cache for this and it can be shared among multiple sessions.

The **second level cache** is maintained at the SessionFactory level, this cache is by default disabled, to enable second level cache in hibernate, it needs to be configured in hibernate configuration file, i.e. *hibernate.cfg.xml* file. There are various providers of second level cache, like EhCache, OSCache etc.

Once second level cache is configured, then object request will first go to the first-level cache, if it is not found there, then it will look for this object in second-level cache, if found then it will be returned from the second-level cache and it will also save a copy in first-level cache.

But, If the object is not found in the second-level cache also, then it will hit the database and if it present in database, this object will be put into both first and second level cache, so that if any other session requests for this object then it will be returned from the cache.

Question 180: What is `session.flush()` method in Hibernate?

Answer: Flushing the session forces Hibernate to synchronize the in-memory state of the Session with the database. By default, Hibernate will flush changes automatically for you:

- before some query executions
- when a transaction is committed
- when `session.flush` is called explicitly

It is also possible to define a flushing strategy, by using *FlushMode* class. It provides below modes:

- ALWAYS: the session is flushed before every query
- AUTO: the Session is sometimes flushed before query execution in order to ensure that queries never return stale state
- COMMIT: the Session is flushed when Transaction.commit() is called
- MANUAL: the Session is only ever flushed when Session.flush() is explicitly called by the application

Question 181: How can we see the SQL query that gets generated by Hibernate?

Answer: If you are using *hibernate.cfg.xml* file, then use the below property:

```
< property name = "show_sql" > true</ property >
```

If you are using Spring Data JPA, then you can set this property in *application.properties* file, like:

```
spring.jpa.show-sql=true
```

Question 182: What is Hibernate Dialect and why we need to configure it?

Answer: The Dialect specifies the type of database that our application is using. As we know, Hibernate is database agnostic and it can work with many databases. However, each database has some variations and standard implementations. We need to tell Hibernate about it, so that Hibernate can generate the database specific SQL wherever it is necessary.

You can configure this dialect in *hibernate.cfg.xml* file, like:

```
< property name = "dialect" >
org.hibernate.dialect.PostgreSQLDialect</ property >
```

And in SpringBoot, you can configure it in *application.properties* file, like:

```
spring.jpa.database-
platform=org.hibernate.dialect.MySQL5InnoDBDialect
```

Question 183: What do you know about hibernate.hbm2ddl.auto property in Hibernate?

Answer: *hibernate.hbm2ddl.auto* automatically validates or exports schema DDL to the database when the SessionFactory is created.

This property takes various values:

- create: creates the schema, but previous data will be lost
- validate: validates the schema
- update: updates the schema. This does not drop any existing table, so we don't lose any existing data. If any column is added in hibernate entity, it will add this column in database table or if any new entity has been added, it will create a new table in database
- create-drop: when this value is given, then the schema is first created at SessionFactory startup and gets dropped at SessionFactory shutdown
- none: it does nothing with the Schema, makes no changes to the database

You can configure this property in hibernate.cfg.xml file, like:

```
< property name = "hbm2ddl.auto" > create</
property >
```

And in SpringBoot, you can configure it in application.properties file, like:

```
spring.jpa.hibernate.ddl-auto=update
```

Maven

Question 184: What is Maven?

Answer: Maven is a tool that is used for building and managing any Java based project. It is a powerful project management tool that is based on POM (Project Object Model). It simplifies the build process.

Question 185: What is pom.xml?

Answer: POM stands for Project Object Model, it is an xml file which contains the configuration information related to the project. Maven uses this file to build the project. We specify all the dependencies that are needed for a project, the plugins, goals etc. By using <packaging> tag, we can specify whether we need to build the project into a JAR/WAR etc.

Question 186: What is local repo and central repo?

Answer: **Local repo** : Local repository is a directory on the developer machine. This repository contains all the dependencies that are downloaded by Maven. The dependencies are downloaded only once even if it is used in multiple projects. By default, local repository location is, C:/Users/USER_NAME/.m2

Central Repo : if any dependency that is required by a project is not found in local repository then Maven looks in central repository for this dependency, then Maven downloads this dependency into the local repository.

We also have one **Remote repository**, which resides on a server from which Maven can download the dependencies into the local repository. It is mainly used in organizations, to share the dependencies within the organization teams.

Question 187: Where we define our local repo path?

Answer: settings.xml file is used to define a local repository location. This file is also used to define proxies, remote repository server locations,

plugin groups, profiles etc. By default, it is present in `~/.m2/settings.xml`

```
<!-- localRepository
| The path to the local repository maven will use to store artifacts.
|
| Default: ~/.m2/repository -->
<localRepository>/path/to/local/repo</localRepository>
```

Question 188: Where do we define proxies so that maven can download jars from the internet in a corporate environment?

Answer: `settings.xml` file is used to define proxies which helps in connecting to a network while working in a corporate environment.

```
<proxies>
  <!-- proxy
  | Specification for one proxy, to be used in connecting to the network.-->
  <proxy>
    <id>optional</id>
    <active>true</active>
    <protocol>http</protocol>
    <username>proxyuser</username>
    <password>proxypass</password>
    <host>proxy.host.net</host>
    <port>80</port>
    <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
  </proxy>
</proxies>
```

Question 189: Explain Maven build life-cycle

Answer: Maven build life-cycle is made up of below phases,

- validate: validate the project is correct and all necessary information is available
- compile: compile the source code of the project
- test: test the compiled source code using a suitable unit testing framework. These tests should not require the code to be packaged or deployed
- package: take the compiled code and package it in its distributable format, such as a JAR
- verify: run any checks on results of integration tests to ensure quality criteria's are met
- install: install the package into the local repository, for using as a dependency in other projects locally
- deploy: done in the build environment, copies the final package to the remote repository for sharing with other developers and projects

Maven will first validate the project, then it will try to compile the sources, run the tests against the compiled code, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository and then deploy the installed package to a remote repository.

mvn command can be used to execute these build life-cycle phases. If you run *mvn verify*, then it will execute all the phases in order, validate, compile, test, package before calling the *verify*. We only need to call the last build phase.

mvn clean command is used to delete all the project jars that are built by Maven (/target directory of a project). Generally, this clean command is used with install/deploy phase, like *mvn clean deploy* to cleanly build and deploy artifacts into the shared repository.

Database

Question 190: What do you know about SQL Joins?

Answer: SQL joins are used to combine rows from two or more tables, based on a related column between them.

There are 4 types of Joins in SQL:

1. **Inner join** : Inner join selects all records from Table A and Table B, where the join condition is met.

Syntax:

```
SELECT Table1.column1, Table1.column2, Table2.column1, ....  
FROM Table1  
INNER JOIN Table2  
On Table1.MatchingColumnName = Table2.MatchColumnName;  
(Note: Use either INNER JOIN or JOIN for this operation)
```

2. **Left Join** : Left join selects all records from Table A along with records of Table B for which the join condition is met.

Syntax:

```
SELECT Table1.column1, Table1.column2, Table2.column1, ....  
FROM Table1  
LEFT JOIN Table2  
On Table1.MatchingColumnName = Table2.MatchColumnName;
```

3. **Right Join** : Right join selects all records from Table B along with records of Table A for which the join condition is met.

Syntax:

```
SELECT Table1.column1, Table1.column2, Table2.column1, ....  
FROM Table1  
RIGHT JOIN Table2  
On Table1.MatchingColumnName = Table2.MatchColumnName;
```

4. **Full Join** : Full join selects all records from Table A and Table B, regardless of whether the join condition is met or not.

Syntax:

```
SELECT Table1.column1, Table1.column2, Table2.column1, ....  
FROM Table1  
FULL JOIN Table2  
On Table1.MatchingColumnName = Table2.MatchColumnName;
```

Question 191: Difference between TRUNCATE & DELETE statements

Answer: The differences are:

- TRUNCATE is a DDL command, whereas DELETE is a DML command
- TRUNCATE removes all rows from a table, whereas DELETE can also remove all rows but it can be used with a 'WHERE' clause to remove specific rows from a table
- TRUNCATE command records very little entry in the transaction log, whereas DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row, because of this reason TRUNCATE is faster than DELETE
- To use TRUNCATE on a table, you need at least ALTER permission on the table, whereas to use DELETE, you need DELETE permission on the table
- TRUNCATE uses less transaction space than DELETE
- TRUNCATE operation cannot be rolled back, whereas DELETE operation can be rolled back

TRUNCATE command Syntax:

TRUNCATE TABLE employee;

DELETE command Syntax:

DELETE FROM employee; -- delete all rows of the table

DELETE FROM employee WHERE name = 'Mark'; -- delete record where emp_name is Mark

Question 192: Difference between Function and Stored Procedure

Answer: The differences are:

- Function must return a value, whereas Stored Procedure can return zero or n values
- A Function can have only input parameters, whereas a Stored Procedure can have both input and output parameters
- Functions can be called from a Stored Procedure, but a Stored Procedure cannot be called from a Function
- In a Function, DML statements cannot be used, whereas DML statements can be used in a Stored Procedure
- Functions does not allow the usage of try-catch blocks, whereas in Stored Procedure, try-catch block can be used for exception handling
- Transactions are not allowed within Functions, whereas transactions can be used within Stored Procedure
- In a Function, we can only use the table variables, it does not allow using the temporary tables, whereas in a Stored Procedure, both table variables and temporary tables can be used
- Functions can be used in SELECT statement, WHERE clause, HAVING clause, whereas Stored Procedure cannot be used with these
- A Function can be used in JOIN clause as a result set, whereas a Stored Procedure cannot be used in JOIN clause

Question 193: What is DDL, DML statements?

Answer: DDL: DDL stands for Data Definition Language. These statements are used to define the database structure or schema. DDL commands are auto-committed.

Examples of DDL commands are: CREATE, ALTER, DROP, TRUNCATE

DML: DML stands for Data Manipulation language. These statements allows us to manage the data stored in the database. DML commands are not auto-committed, so they can be rolled back.

Examples of DML commands are: INSERT, UPDATE, DELETE, SELECT

Question 194: How to find the nth highest salary from Employee table

Answer: The query to find nth highest salary is:

```
SELECT name, salary  
FROM Employee e1  
WHERE N-1 = (SELECT COUNT(DISTINCT salary) FROM Employee e2  
WHERE e2.salary > e1.salary);
```

Here, to find the 3rd highest salary, replace N with 3, for 5th highest salary, replace N with 5 and so on.

The DISTINCT keyword is used to deal with the duplicate salaries in the table. The highest salary means no salary is higher than it, the second highest salary means only one salary is higher than it, and similarly Nth highest salary means N-1 salaries are higher than it. This is a generic solution and works in all databases, however it is a little slow because the inner query will run for every row processed by the outer query.

Question 195: Difference between UNION and UNION ALL commands in SQL

Answer: Both UNION and UNION ALL are used to combine results of two separate queries, it could be on a same table or a different table but number of columns should be same in both queries.

The Key difference between them is UNION removes duplicates, whereas UNION ALL keeps the duplicates. Because of this, UNION ALL takes less time, as there is no extra step of removing duplicate rows.

Question 196: Difference between Unique Key and Primary Key in SQL

Answer: Both Unique and Primary keys uniquely identifies each row of a table.

The differences between them are:

- There can be only one primary key in a table, whereas there can be multiple unique keys in the table
- Primary key cannot be null, whereas Unique Keys can be null
- In Primary key, default index is clustered, whereas in Unique key, default index is non-clustered

Question 197: What is the difference between Primary and Foreign key in SQL?

Answer: **Primary key** is used to uniquely identify a row in the table. A table can have only one primary key. Primary key is of two types, simple and composite primary key. A Simple Primary key is made up of just one column, whereas a composite primary key is made up of more than one column.

Primary key also enforces some constraints, like UNIQUE and NOT NULL, which means a table cannot have duplicate primary keys and the key cannot be null.

A **Foreign key** in a table is the primary key of another table. For example, consider 2 tables, Employee & Department. Department table have a primary key dept_id and this primary key can be used as foreign key in Employee table to identify that this employee belongs to this department.

The differences between Primary key and Foreign key are given below:

- Primary key uniquely identify a record in the table, whereas Foreign key is the field in the table that is the primary key of another table
- By default, a clustered index is created on primary key, whereas foreign key do not automatically create an index
- We can have only one primary key in a table, whereas we can have more than one foreign key in a table
- Primary keys does not allow duplicate or Null values, whereas Foreign keys allows both

Question 198: What is the difference between clustered and non-clustered index?

Answer: Indexes are used to speed-up the data retrieval performance. There are 2 types of indexes, clustered and non-clustered index and the difference between them,

- A clustered index defines the order in which data is physically sorted in a table, whereas non-clustered index does not sort the physical data inside the table
- By default, clustered index is automatically created on primary key, whereas non-clustered index can be created on any key
- There can be only one clustered index in a table, whereas there can be any number of non-clustered index in a table
- Data Retrieval is faster using Clustered index than non-clustered index
- Data Update is faster using Non-clustered index than clustered index
- Clustered index does not need any extra space, whereas non-clustered index requires extra space to store the index separately
- The size of clustered index is quite large as compared to non-clustered index

Syntax of creating a custom clustered index:

CREATE CLUSTERED INDEX index_name

ON table_name (column_name ASC);

For example, creating a clustered index on Employee table, where data should be stored in ascending order of age column:

CREATE CLUSTERED INDEX employee_asc_age_index

ON employee(age ASC);

Syntax of creating a custom non-clustered index:

CREATE NONCLUSTERED INDEX index_name

ON table_name (column_name ASC);

For example, creating a non-clustered index on Employee table, where data should be stored in ascending order of name column:

CREATE NONCLUSTERED INDEX employee_asc_name_index

ON employee(name ASC);

Question 199: What is the difference between WHERE and HAVING clause in SQL

Answer: The differences are:

- WHERE clause can be used with SELECT, INSERT, UPDATE and DELETE statements, whereas HAVING clause can only be used with SELECT statement
- WHERE clause is used for filtering the rows and it applies on each and every row, whereas HAVING clause is used to filter groups
- WHERE clause is used before GROUP BY clause, whereas HAVING clause is used after GROUP BY clause. It means that WHERE clause is processed before GROUP BY clause while HAVING clause is executed after groups are created
- Aggregate functions cannot be used in WHERE clause, whereas we can use aggregate functions in HAVING clause

Question 200: How to change the gender column value from Male to Female and Female to Male using single Update statement

Answer: This is also a very common interview question. Mostly, this question is asked in a telephonic round. The question is like, you are given a table, say Employee which has a column named ‘Gender’ having only “Male” or “Female” strings as values. You have to swap these values like wherever the value is Male, it should become Female, and wherever the value is Female, it should become Male. And you have to do this by writing only one Update statement.

The Update query for this is:

UPDATE Employee SET Gender =

CASE Gender WHEN ‘Male’ THEN ‘Female’ WHEN ‘Female’ THEN ‘Male’ ELSE Gender END;

Other than these common questions, you can be asked to write a lot of queries which mostly contains Joins, so you should also prepare for those types of database queries.

Question 201: Find first 3 largest numbers in an array

In this question, the interviewer will most probably ask you to not use sorting and pick the first/last 3 numbers from the array. Instead he will ask you to use one “for loop” to solve this problem.

```
public class FindLargestThree {  
    //method to find the largest three numbers from an array  
    public static void findLargestThree(int arr[]) {  
  
        if(arr.length < 3) {  
            System.out.println("Invalid input, Array size is less than 3");  
        }  
  
        int first = Integer.MIN_VALUE;  
        int second = Integer.MIN_VALUE;  
        int third = Integer.MIN_VALUE;  
  
        for(int i=0; i<arr.length; i++) {  
            int current = arr[i];  
  
            if(current > first) {  
                third = second;  
                second = first;  
                first = current;  
            } else if(current > second) {  
                third = second;  
                second = current;  
            } else if (current > third) {  
                third = current;  
            }  
        }  
        System.out.println("Three largest elements are: " + first  
                           + ", " + second + ", " + third);  
    }  
  
    public static void main(String[] args) {  
        int arr[] = {19, 5, 78, 1, 33, 11, 20};  
        findLargestThree(arr);  
    }  
}
```

Output:

Three largest elements are: 78, 33, 20

Here, the idea is to have 3 numbers and then iterating through the array and finding where the current element of array fits in.

At first, we check whether the current element is greater than first, if true, assign the current element to first number by swapping the values of first, second and third.

When the first condition is not true, then we compare the current element with second largest number to find whether the current number is second largest or not, same goes for third condition.

Question 202: Move all negative numbers at the beginning of an array and all positive numbers at the end

Here, also the interviewer will ask not to use any additional data structure like an extra array and this question can be asked in two ways, whether the sequence of original array elements should be maintained or not, so let's see the program where the sequence is not maintained:

```
public class RearrangeArrayElements {  
  
    static void rearrange(int arr[]) {  
        int j=0, temp;  
        for(int i=0; i<arr.length; i++) {  
            if(arr[i] < 0) {  
                if(i != j) {  
                    temp = arr[i];  
                    arr[i] = arr[j];  
                    arr[j] = temp;  
                }  
                j++;  
            }  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    int arr[] = {-9, 5, 1, -2, -15, 7, 12, -3, 2};  
    rearrange(arr);  
  
    for(int i=0; i<arr.length; i++) {  
        System.out.print(arr[i] + " ");  
    }  
}
```

Output:

-9 -2 -15 -3 1 7 12 5 2

Here, the idea is to iterate through the array and when a negative number is found, then bring that number to the beginning of the array by swapping it with the first positive number.

As you can see that the output is not maintaining the original sequence of array elements.

Now, let's take a look at the solution which maintains the element sequence:

```

public class RearrangeArrayElements {

    static void rearrange(int arr[]) {
        int j, current;
        for(int i=0; i<arr.length; i++) {
            current = arr[i];
            //if current element is positive then do nothing
            if(current > 0) {
                continue;
            }
            //if current element is negative, then shift positive
            //numbers of arr[0...i-1], one position to their right
            j = i-1;
            while(j >= 0 && arr[j] > 0) {
                arr[j+1] = arr[j];
                j = j-1;
            }

            arr[j+1] = current;
        }
    }

    public static void main(String[] args) {
        int arr[] = {-9, 5, 1, -2, -15, 7, 12, -3, 2};

        rearrange(arr);

        for(int i=0; i<arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}

```

Output:

-9 -2 -15 -3 5 1 7 12 2

Now, the output is maintaining the original sequence of array elements.

There are many programmatic, puzzle problems that the interviewer can ask. If you are a beginner, then prepare for programs like Palindrome, Fibonacci, Array problems, String problems, Linked list programs etc. First,

try to solve them with whatever brute-force solution that comes to your mind, then try to find its time and space complexity, then try to optimize it. If you are not able to think of a better solution, no problem, look for the optimal solution on the internet.

About the Author

Jatin Arora is a Computer Science graduate. He holds an expertise in Java & SpringBoot. He has worked on a variety of interesting projects across different domains like Inventory management, DevOps, cloud & financial domain.