All MNC Java Interview Questions

Simplified Java Cracked Interview.

Features:

- ✓ **Can refer freshers to 10 years of Experience.**
- ✓ **Questions collected from candidates, interviewers and multiple websites.**
- ✓ **550 Questions covered with programs.**
- ✓ **Covered Frequently Asked Questions.**
- ✓ **Covered Real Time Interview Questions.**
- ✓ **Covered 200+ Java 6 and Java 8 programs tested on Eclipse.**

YOGESH SANAS

All MNC Java Interview Questions

Simplified Java Cracked Interview.

Features:

- √ Can refer freshers to 10 years of Experience.
- √ Questions collected from candidates, interviewers and multiple websites.
- √ 550 Questions covered with programs.
- √ Covered Frequently Asked Questions.
- √ Covered Real Time Interview Questions.
- √ Covered 200+ Java 6 and Java 8 programs tested on Eclipse.

YOGESH

About this book and author.

I would like to thank my family who encourage me to write this book so I am dedicating this book to my family.

This book is all about how to crack java interviews. If you are an experienced person then you must have attended many interviews in multiple IT companies. If you are freshers then you will have to attend many interviews in future. Cracking the interview is a challenging thing for everyone. I have seen many people who got the job after rejecting in 8 to 10 companies. But some of them got a job offer at the first attempt. So what do you think why this happened with people despite they belong to the same branch, same college. Can you guess!!! Yes, You are right. PREPARATION!!! I am here to help you with interview preparation.

I would like to introduce myself, I am Yogesh Sanas. I have 7 plus years of experience in Java Technology. I had worked for many reputed companies like TCS and Cybage Softwares. Fortunately, I got many opportunities to conduct interviews on Java Technology. I have conducted more than 500 interviews by now.

Hope you are aware of which skills are required to crack interviews. You should have good communication skills, technical knowledge, a positive attitude and many more.

I have noticed all in the 7 years of experience where candidates are lacking, their common mistakes. I will cover all the points in upcoming chapters that need to be corrected, how to avoid common mistakes, how to answer everything that will be covered. Please go through all pages till last so you won't miss any important questions.

I have created this interview series, especially those people who are willing to make their career in Java Technology. I have covered interview questions asked in all reputed MNC like TCS, Infosys, Capgemini, Tech Mahindra, Cognizant, City Bank, Barclays, Amdocs, Mindtree etc.

I have collected all the questions from actual interviews attended by my friends, colleagues, college mate and classmate. I have covered frequently asked questions as well as challenging questions. I have included many programs to understand the concept thoroughly. I will try to explain the concept with the help of a real-time program in eclipse.

My purpose behind this book is, help you to get a job in your dream company. I can understand this is a struggling period for job seekers and freshers. Everybody must have gone from this phase. so never give up. Keep upgrading yourself. I am sure you will get a job in your dream company.

All the best!!!

INDEX

Interview Questions on Basic java and OOPs.

1. **What is object-oriented language? Is java 100% object-oriented language? Why?**

- Object-oriented programming is about creating an object which can have its data and methods.
- Java supports primitive data types like int, float, double etc. which is not an object, so Java is not a purely object-oriented language.

2. **What is platform-independent? Why java is platform-independent?**

- Platform independent means "write once run anywhere". In short, once you write a program on any platform can run it on any platform. e.g. If you wrote a program on Unix can run it on windows or any other platform or operating system.
  Once the Java program compiles that program converts into byte code. Byte code is not platform dependent so Java is platform-independent.

3. **What is a Class?**

- Class is a blueprint or template, from object can be created.
- E.g. I have a class with the name Person. I have created '10' objects of the Person class. Each object has its copy of each member variable.

```
package
simplifiedjava.crackedInterview;

public class Person {

        private int age;
        private int height;
        private int weight;
        private String name;
        private String education;
        // Getter and setter
methods.
}
```

4. **What is an Object?**

- Generally, an Object is a real-time entity. Objects are the behaviour of the class.
- Technically an Object is the combination of data and functions.

5. **What is JVM? What is the difference between JDK, JVM and JRE?**

- JVM stands for Java Virtual Machine.
- JVM provides a runtime environment in which byte code can be executed.

JDK	JVM	JRE
JDK Stands for Java Development Kit.	JVM Stands for Java Virtual Machine.	JRE Stands for Java Runtime Environment.
JDK Physically exist in System	JVM exist virtually but not physically.	JRE exists physically.

JDK = JRE + JVM	JVM is a run time environment in which bytecode can execute.	JRE is an Implementation of JVM.

6. **What is the class loader and what are the types of class loaders?**

- ClassLoader loads the classes dynamically into memory whenever required by Java Runtime Environment.
- ClassLoader is an abstract class in Java Runtime Environment.
- There are three types of class loaders available in java.

   a. **BootStrap Class Loader (Primordial ClassLoader):**

   This is a parent loader of all existing loaders. Bootstrap loaders load all JDK files from jre/lib/rt.jar.

   b. **Extension Class Loader :**

   Extension class loader loads the extensions of java core classes from jre/lib/ext directory. Extension class loader is a child loader of Bootstrap class loader.

   c. **System Class Loader :**

   System class loader loads the application level classes. System class loader is a child loader of Extension class loader.

7. **What is Heap memory and Stack Memory? What is the difference between Heap Memory Stack Memory?**

- **Heap Memory :**
    - Heap memory is used to store the Objects.
- **Stack Memory:**
    - Stack memory is used to store the local variables and function list.

	Heap Memory	**Stack Memory**
1.	Heap memory stores the objects.	Stack Memory stores local variables, reference variables and methods.
2.	Objects stores in a hierarchical manner.	Variables stores in a sequential manner.
3.	Objects can reside in heap memory until it points to reference variable.	Variables will be removed once the function is executed.
4.	Access speed is slow.	Access speed is high.
5.	If Heap memory gets full JVM immediately throw OutOfMemoryError.	If Stack gets full JVM immediately throws StackOverflowError.
6.	The Garbage collections mechanism is used to reclaim the memory.	There is no role of garbage collector because variables are automatically removed once the function is executed.
7.	Heap is a flexible memory.	Stack is not flexible memory.
8.	-Xms and –Xmx JVM options can be used to increase or decrease the size of the Heap.	-Xss option can be used to increase the size of the stack.

8. Explain the Term Association, Composition, and Aggregation? What is HAS-A and IS-A Relationship?

- **Association**: Creating the relationship between two objects is referred to as Association.
  - ➢ Association can be One-to-One, One-to-Many, Many-to-One, Many-to-Many.
  - ➢ Aggregation and Composition are two forms of Association.
  - ➢ One-to-One = One Student can have one Class.
  - ➢ One-to-Many = One Student can have multiple Subjects.
  - ➢ Many-to-One = Many developers can have one Project Manager.
  - ➢ Many-to-Many = Multiple Employees can have Multiple Tasks.
  - ➢ Example of Association. We have created an association between Company and Employees.

```
package simplifiedjava.crackedInterview;

public class Company {

        private String companyName;
        public Company(String companyName) {
        this.companyName = companyName;
        }
        public String getCompanyName() {
        return companyName;
        }
}

package simplifiedjava.crackedInterview;

public class Employee {

        private String empName;
        public Employee(String empName) {
        this.empName = empName;
        }
        public String getEmpName() {
        return empName;
        }
}

package simplifiedjava.crackedInterview;

public class AssociationExample {

        public static void main(String[] args) {
        Employee employee1 = new Employee("Rusty");
        Company company = new Company("Company");
        System.out.println(employee1.getEmpName() + " is working for "+
company.getCompanyName());
        }
}
```

OUTPUT : Rusty is working for Company.

- **Composition**:
  - ✓ The composition can be called the HAS-A relationship.

- ✓ When two objects are tightly coupled with each other or we can say when one object is dependent on another object is called Composition.
- ✓ E.g. Engine cannot survive without a car. If a car exists then an engine exists. If a car is destroyed automatically engine gets destroyed.

```
package simplifiedjava.crackedInterview;

public class Engine {

        private String engineName;
        private String power;
        private String modelNo;
        // getter and Setter methods.
}

package simplifiedjava.crackedInterview;

public class Car {

        private String carName;
        private String carModel;
        private String manufacturer;
        private Engine engine; // Car HAS-A reference of Engine.

        public Car(String carName, String carModel, String manufacturer, Engine engine) {
        super();
        this.carName = carName;
        this.carModel = carModel;
        this.manufacturer = manufacturer;
        engine = new Engine();
        }
        // getter and Setter
}
```

- Aggregation:

- Aggregation can be called the HAS-A relationship.
- When two objects are loosely coupled or we can say when one object is not dependent on another object is called Aggregation.
- E.g. We have two objects. The first one is College and the second one is Student. If college gets destroyed still Student objects can survive independently.

```
package simplifiedjava.crackedInterview;

public class Student {

        private int id;
        private String name;
        private String standard;
        // getters and Setters method.
}

package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class College {
```

```
            List<Student> studentList = new ArrayList<Student>();
            // add and remove Students.
}
```

9. What is the difference between Composition and Aggregation?

	Composition	Aggregation
2.	Tightly couple relationship.	Loosely coupled relationship.
3.	◆ Symbol ⟶	◇ Symbol ⟶

10. Is Java Pass by value or Pass by reference. How will you prove java is call by value?

- Java is Call by Value not a Pass by reference.
- We will illustrate with the examples below.

```
package simplifiedjava.crackedInterview;

public class Addition {

            int num1 = 100;
      int num2 = 0;
            public static void main(String[] args) {
            Addition add = new Addition();
            System.out.println("Before Adding Numbers : "+ add.num1);
            add.addNumber(add.num1);
            System.out.println("After Adding Numbers : "+ add.num1);
            }
            public void addNumber(int num2) {
            num2 = num2 + 100;
            }
}
```

OUTPUT:

```
Before Adding Numbers : 100
After Adding Numbers : 100
```

Explaination : In the above program, we have passed integer numbers and trying to add 100. But the change remained in addNumber() function not in main().

```
package simplifiedjava.crackedInterview;

public class Addition {

            int num1 = 100;

            public static void main(String[] args) {
            Addition add = new Addition();
            System.out.println("Before Addition " + add.num1);
            add.addNumberWithObject(add);
            System.out.println("After Addition " + add.num1);
            }
            public void addNumberWithObject(Addition addCopy) {
```

```
            addCopy.num1 = 200;
            }
}
```

OUTPUT :

```
Before Addition 100
After Addition 200
```

Explanation: We have passed the reference variable as a value so the value of num1 got changed.

11.
Which is the top most Parent class in Java? What the methods of Object Class?

- Java.lang.Object is the parent class of java hierarchy.
- Following are the defined methods in Object class.

1. **public final Class getClass()** : returns the Class class object.
2. **public String toString()**: Convert object to String.
3. **public boolean equals():** Comparing this object to another object.
4. **public int hashCode():** return the hash code of object.
5. **protected Object clone():** return the duplicate object.
6. **public final void wait():** put the thread into waiting area.
7. **public final void notify():** notify the waiting or sleeping thread.
8. **public final void notifyAll():** notify all the waiting and sleeping thread.

12.
What is the role of break and continue keywords in java?

- **break:** break keyword can be used to jump out of the loop.
- **continue:** continue keyword can be used to break one iteration for a specific condition and continue with the next iteration.
- We will look at examples for a break and continue.

Break Example:

```
package simplifiedjava.crackedInterview;

public class BreakStatementExample {

            public static void main(String[] args) {
            for(int i=0; i<10; i++) {
            System.out.println("Value of "+ i);
            if(i == 5) {
            System.out.println("Before Break");
            break;
            }
            }
```

```
		System.out.println("outside loop");
		}
}
```

OUTPUT :

```
Value of 0
Value of 1
Value of 2
Value of 3
Value of 4
Value of 5
Before Break
outside loop
```

Continue Example:

```
package simplifiedjava.crackedInterview;

public class ContinueStatementExample {

		public static void main(String[] args) {
		for(int i=0; i<10; i++) {
		System.out.println("Value of "+ i);
		if(i == 5) {
		System.out.println("Before continue = "+ (i + i));
		continue;
		}
		}
		System.out.println("outside loop");
		}
}
```

OUTPUT:

```
Value of 0
Value of 1
Value of 2
Value of 3
Value of 4
Value of 5
Before continue = 10
Value of 6
Value of 7
Value of 8
Value of 9
outside loop
```

13.
Is null keyword in java?

- No. null is not a keyword in java. Null is like literals like true and false in java.

14.
What is Actual Parameter and what is Formal Parameter?

- **Actual Parameters**: If we pass the parameters while calling a function are called Actual Parameter.
- **Formal Parameter**: If we are catching parameters in function are called formal parameters.

```
package simplifiedjava.crackedInterview;
```

```
public class ActualAndFormalParameters {

            public static void main(String[] args) {
            int a = 10;
            int b = 20;
            addition(a,b); // a and b are Actual Parameter
            }
            public static void addition(int num1, int num2) {
            // num1 and num2 are formal parameters
            }
}
```

15. What is the difference between Local variable and Instance variable?

	Local Variable	Instance Variable
1.	Local variable declares inside the method, block and constructor.	Instance variable declared outside the method, block and constructor but inside the class.
2.	We have to manually initialize the local variable before use.	When instance variable gets created default value assigned to that variable.
3.	Scope of local variable up to method execution.	Instance variable scope is wider than the local variable.

16. What is the difference between Instance variable and Class Variable?

	Instance Variable	Class Variable
1.	The instance variable belongs to the object.	The Class variable belongs to the class.
2.	Every object has its copy of the instance variable.	The class variable is a common variable among all objects.
3.	Instance variable declared with data type only.	A class variable is declared with data type and static keyword.
4.	Instance variable access with reference variable.	Class variable access with the class name.

17. What are the uses of this keyword? Where it is applicable and what are the rules to use this keywords?

- this keyword is used to refer to the current object.
- this keyword can be used to the invoked current class constructor.
- this keyword can be used to the invoked current method.
- this can be used to return the current class object.
- this can be used to pass as an argument in method as well as constructor.

18. What are the uses of the super keyword? Where it is applicable and what are the rules to use this keywords?

- The super keyword can be used to refer parent class instance variable.
- Super keyword can be used to invoke the parent class method.

- Super keyword can be used to invoke parent class constructor.

19.
What are the access modifiers and their scope?

- There are four access modifiers which are as follows.

1. Default
2. Private
3. Protected
4. Public

- Scope:

Access Modifier	Within Class	Within Package	Subclass in another package	Outside package
Default	Y	Y	N	N
Private	Y	N	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

20.
What are the main pillars of OOPs Concepts? Explain the OOPS Concepts?

- There are four pillars of OOPs Concepts which are as follows.

1. **Abstraction:**

Abstraction means hiding the implementation.

Real-Life Example: We can take an example of a Car. When we want to start a car we have to just turn keys in the ignition and the engine gets started. For the end-user, there is no need for how the engine is interconnected with different wheels. How the battery supply power. How petrol sprays in the engine. These all implementation is hidden. This is an abstraction.

Technical Implementation: Abstract class and Interface implements the abstraction. An abstract class is partial abstraction because abstract class can have concrete methods as well as abstract methods and the interface is full abstraction because the interface contained only abstract methods, not concrete methods.

2. **Encapsulation**:

Encapsulation means hiding the data.

Real Life Example: Capsule is the perfect example of encapsulation. The capsule has different types of small tablets inside a wrapper.

Technical Implementation: Bean class is an example of Encapsulation. We declare properties as private and provide public getter and setter methods for each variable.

3. **Inheritance:**

Inheritance means one object can access the properties of other objects. The child class can inherit the properties of the parent class.

Real-Life Example: Say suppose your grandfather has agricultural land, farmhouse and your father has flat, some gold and shares and you have racer bike, iPhone and laptop. Ultimately whatever property your grandfathers have will be inherited to your dad and your dad's property inherited to you. But remember reverse is not possible.

Technical Implementation: You can implement inheritance using extends and implementation keywords. If you are extending another class then you have to use extends keyword and If you implement the interface then you have to use the implement keyword.

4. **Polymorphism**:

Polymorphism means one name different form.
An entity can have different behaviour in different situations.

Real-Life Example: A person can have a different role at a time with a different location or context. The person can be a father in the home, the person can be a manager in the company, Person can be a dancer in the movie.

Technical Implementation: Polymorphism can be implemented by using Method overloading and method overriding.

21.
What is the difference between compile-time polymorphism and run-time polymorphism?

	Compile time Polymorphism	**Run time polymorphism**
1.	Method overloading is an example of Compile-time binding.	Method overriding is an example of Run-time binding.
2.	Compile-time polymorphism can be achieved by static binding.	Run-time polymorphism can be achieved by dynamic binding.
3.	Inheritance is not mandatory to implement compile-time polymorphism.	Inheritance is mandatory to implement Run-time polymorphism.

22.
How have you implemented Encapsulation in the Project?

- You can create POJO(Plain Old Java Object) class.
- You have to declare all members with private access modifier and provide public getters and setters method. So the original value of that variable cannot be change cause they are private and outsiders can use it cause we have provided public getters and setters methods.

```
package simplifiedjava.crackedInterview;

public class EncapsulationImplementation {

        private int id;
        private String name;
        public int getId() {
        return id;
        }
        public void setId(int id) {
        this.id = id;
        }
        public String getName() {
        return name;
        }
        public void setName(String name) {
        this.name = name;
        }
}
```

23. Can you explain the types of Inheritance supports in java?

- Java Supports 3 types of Inheritance.

1. Single Inheritance:

In Single Inheritance, only one class can extend at a time.
In single inheritance, there is only one child class and parent class.
Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Human {

}
public class Men extends Human{
}
```

2. Multilevel Inheritance:

When one class extends another class and that another class extends a subsequent class this type of inheritance is called Multilevel Inheritance.
Every child class becomes a parent class of its subsequent child class.
Please refer to an example below.

```
package simplifiedjava.crackedInterview;

public class Human {
}
public class Men extends Human{
}

public class Boy extends Men{
}
```

3. Hierarchical Inheritance:

When two or more classes extends one class is called Hierarchical Inheritance.
Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Human {

}
public class Men extends Human{
}
```

```
public class Women extends Human{
}
```

24. Why java doesn't support Multiple Inheritance?

- To avoid an ambiguity situation java doesn't support multiple inheritance.
- We can illustrate with an example. You have three classes. Class A, Class B and Class C. Class C extends Class A and Class B. Say suppose Class A and Class B has printHello() method. The compiler cannot decide whether which class printHello() method should inherit in class C. So avoiding this ambiguity situation java doesn't support multiple inheritance.
- But indirectly java supports multiple inheritance through interfaces.

25. How would you implement Multiple Inheritance in java?

- One way you can implement multiple interfaces.
- The second one is, one interface extends multiple interfaces and class implements the first interface.

26. Can you explain the public static void main(String[] args) Method?

- Main method is a starting point of java program execution.
- Main method components:

  1. **public:** Access modifier of main() is public so the method can be invoked from outside. This method must be public so the java runtime can invoke it. If you make the main method private or protected then you won't get a compile-time error but a runtime exception.
  2. **static:** We are aware of the use of static. Without creating an object we can invoke the method of the class. So java runtime can invoke the main method without instantiating the class.
  3. **void:** return type of the main method is void cause the main method doesn't return anything.
  4. **main:** This is the fixed name of the main method. You cannot change the name of the main method otherwise you will get a runtime exception.
  5. **(String args[]) :** These are the command line argument. We can pass these arguments to the java program.

27. List out which is wrong main method declaration.

- `public static void main(String[] args) {}` Valid
- `public static void main(String... args) {}` Valid
- `public final static void main(String args) {}` Invalid

- **public static void** main(String args...) {}   Invalid
- **private static void** main(String[] args) {}   Invalid
- **protected static void** main(String[] args) {}   Invalid

28. Why main method is static and public?

- Main () method is the entry point of program execution. JVM calls the main method internally.
- If the method is declared as static then there is no need to create an object to call that method. JVM does not need to create an object to call the main() method.
- If the method is declared as public then that method is accessible outside the class and package.

29. Can we overload or override the main method?

- We can overload the main() method but we can't override the main() method. The main () method can't override because the static method can't be overridden. The static method belongs to the class, not the object.

```
package simplifiedjava.crackedInterview;

public class Parent {

        public static void main(String[] args) {
        // TODO Auto-generated method stub
        }

        public static void main(String[] args, int num) {
        // TODO Auto-generated method stub
        }
}

package simplifiedjava.crackedInterview;

public class Child extends Parent{

        public static void main(String[] args) {
        // TODO Auto-generated method stub
        }
}
```

30. Can we declare the main method as a final? if not then why not?

- Yes. We can declare a main() method as a final.

```
package simplifiedjava.crackedInterview;

public class Child extends Parent{

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

```
		System.out.println("Hi");
		}
}
O/P : Hi
```

31. Can we create a method with the same name as the class name with return type? Will the code compile?

- Yes, we can create the method name the same as the class name. Code will also compile. But remember this is not a good practice. We have a constructor with the same class name.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Parent {

		public static void main(String[] args) {
		new Parent().Parent();
		}
		public void Parent() {
		System.out.println("Hello");
		}
}
Output: Hello
```

32. Can we override the overloaded method?

- Yes. We can override the overloaded method cause within a class overloading is possible but for overriding, you must have a child class where you defined the overriding method.
- For your reference, I have created one simple example.

```
package simplifiedjava.crackedInterview;

public class OverrideOverloadedMethodDemo {

		public static void main(String[] args) {
		Parent p = new Parent();
		Child c = new Child();
		c.check("Brother");
		}
}


package simplifiedjava.crackedInterview;

public class Parent {
		public void check() {
		System.out.println("Hello");
		}
		public void check(String relation) {
		System.out.println("Parent Class check() method.");
```

```
            }
}
package simplifiedjava.crackedInterview;

public class Child extends Parent{

            public void check(String relation) {
            System.out.println("Child Class check() method.");
            }
}

O/P :  Child Class check() method.
```

33. Can we execute a program without a main method?

- Yes, we can execute a program without a main method. We can simply write code inside static block but after execution completed of a static block, you will get an error java.lang.NoSuchMethodError: main. Java 7 onwards is not possible because JVM checks the presence of the main method before initializing the class.

34. What is the purpose of static method and static variable?

- **Static Method:**

  1. Without instantiating the class we can invoke the method with the class name.

- **Static Variable:**

  1. Static variable can be used to refer common property for all objects.

35. What is the difference between instance method and static method?

	Instance Method	**Static Method**
1.	No need to use any special keyword to declare Instance method.	The Static method must declare with the static keyword.
2.	Class must be instantiated to invoke an instance method.	No need to instantiate the class to invoke the static method.
3.	Instance method can be overloaded and overridden.	The static method can be overloaded but cannot overridden.
4.	The Instance method can access the static variables as well as instance variables.	The static method can access only Static variables but not instance variables.

36.

What is the difference between static block and instance block?

	Static Block	Non Static Block
1.	The Static block declared with the static keyword.	Non-Static block declared with no keyword.
2.	The Static block gets executed at the time of class loading.	Non-Static block gets executed at the time of object instantiation.
3.	The Static block executes before Non-static block.	Non-static block executes after the static block.

37. Can we access non-static members inside the static area either method or block?

- No. We cannot access non-static members inside static area either block or method.

38. What are method overriding and method overloading?

	Method Overriding	Method Overloading
1.	Method overriding means creating a method with the same signature in a parent class and child class.	Method overloading means creating multiple methods with a different signature and the same name in a class.
2.	Method overriding is run time polymorphism.	Method overloading is compiled time polymorphism.
3.	We can have a single method in both classes.	We can have multiple methods in a single class.
4.	Method overriding is dynamic binding.	Method overloading is static binding.
5.	Covariant return applicable in method overriding.	Covariant return cannot applicable in method overloading.
6.	A private and static method cannot be overridden.	A private and static method can be overloaded.

39. Will the code compile If the parent class method is not throwing any exception but the child class method is throwing a checked exception?

- No Class won't be compiled. We will get a compile-time error.
- We can see the below example.

```
package simplifiedjava.crackedInterview;

public class Parent {
		public void check(String relation) {
		System.out.println("Parent Class check() method.");
		}
}

package simplifiedjava.crackedInterview;
```

```
import java.io.FileNotFoundException;

public class Child extends Parent{

            public void check(String relation)throws FileNotFoundException {
            System.out.println("Child Class check() method.");
            }
}
Output : Compile time error:  Exception FileNotFoundException is not compatible with throws clause in
Parent.check(String)
```

40. Will the code compile If the Child class method throwing a broader checked exception than the parent class method?

- No. The class won't be compiled if your child class overriding method throws a broader exception.

```
package simplifiedjava.crackedInterview;

import java.io.IOException;

public class Parent {
            public void check(String relation) throws IOException{
            System.out.println("Parent Class check() method.");
            }
}

package simplifiedjava.crackedInterview;

import java.net.SocketException;
import java.sql.SQLException;

public class Child extends Parent{

            public void check(String relation)throws SQLException{
            System.out.println("Child Class check() method.");
            }
}
Output: Compile Time Error:  Exception SQLException is not compatible with throws clause in Parent.check(String)
```

41. Will the code compile If the parent class method is not throwing any exception but the child class method is throwing an unchecked exception or Runtime Exception?

- Yes. Code will compile if parent class method is not throwing any exception but child class method throws runtime exception or unchecked exception.

```
package simplifiedjava.crackedInterview;

public class OverrideOverloadedMethodDemo {

            public static void main(String[] args) {
            Parent p = new Parent();
```

```
        Child c = new Child();
        c.check("Brother");
        }
}

package simplifiedjava.crackedInterview;

import java.io.IOException;

public class Parent {
        public void check(String relation){
        System.out.println("Parent Class check() method.");
        }
}

package simplifiedjava.crackedInterview;

public class Child extends Parent{

        public void check(String relation)throws NullPointerException{
        System.out.println("Child Class check() method.");
        }
}

Output :  Child Class check() method.
```

42. Can I declare the final method inside an interface?

- We cannot declare the final method inside an interface.
- Methods declared inside the interface are by default public abstract.
- For your reference, I have given the below example.

```
package simplifiedjava.crackedInterview;

public interface MyInterface {

        public final void add(int num1, int num2);
}
Error:  Illegal modifier for the interface method add() only public, abstract, default, static(For Variable only, not a
method) and strictfp are permitted.
```

43. Can I declare a static method inside an interface?

- Interface cannot have any static method but can have static variable.

44. Why interface cannot declare a static method?

- Static method cannot override and a method declared inside the interface must override in implemented class.

45.

What is the role of finalize() method?

- The role of finalize() method is to perform clean up activity.
- Garbage collector calls finalize() method just before destroying an object. finalize() method releases the resources acquired by the objects which don't have any references.
- At last, all objects which don't have references will be deleted from memory.

46. Java.lang.Object class have equals() method then why there is a need to override equals() method in user-defined class?

- Object class equals() method compares references not the actual content of an object.
- In short, if two references pointing to the same object then only it will return true otherwise it will return false irrespective of content. That's why we have to override the equals() method in the user-defined class.
- Once we override the equals() method in the user-defined class then we can implement a logic to compare the actual content of the object.

47. Can you override the private method?

- No. The private method cannot override because the private method is not visible in the child class. If the method is not accessible or visible in child class how can we override the method?

48. Can you overload the private method?

- Yes. We can overload the private method. There is no need to have a parent-child relationship to overload the method.
- While method overloading we need to differentiate the type of parameter or number of the parameter so we can overload private methods.

49. Can you overload the static method?

- Yes. We can overload the static method.

50. Can you override the static method?

- No. We cannot override the static method.
- Static method belongs to class not instance.
- Only instance method can be overridden cause each object has its member variables and methods.

- If you are trying to override the static method that will be considered as method shadowing.

51. What is a constructor? What are the types of constructors?

- Constructor is a function that is used to initialize the object.
- Constructor doesn't have a return type.
- Constructor name must be the same as a class name.
- We can have multiple constructors in one class.
- We can declare the constructor as private, public and protected.
- There are **three types of constructors**.

  1. **Default Constructor**: If you don't provide any constructor compiler will provide you with, default constructor.
  2. **No-arg constructor:** If you have declared a constructor without a parameter. That constructor is called the no-arg constructor.
  3. **Parameterized Constructor:** If you declared a constructor with a parameter then it is called the parameterized constructor.

52. Will the parent class constructor be invoked while you have invoked the child class constructor without a super keyword?

- Yes. If you create an object of child class and call child class constructor. Child class constructor internally calls parent class constructor implicitly.

```
package simplifiedjava.crackedInterview;

public class OverrideOverloadedMethodDemo {

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

package simplifiedjava.crackedInterview;

public class Parent {
        public Parent() {
        System.out.println("Parent Class Constructor");
        }
}

package simplifiedjava.crackedInterview;

public class Child extends Parent{

        public Child() {
        System.out.println("Child Class Constructor...");
        }
}

output :
```

```
Parent Class Constructor
Child Class Constructor...
```

53. What is the difference between method and constructor?

	Method	Constructor
1.	The method is used to implement the logic.	Constructor is used to initializing the object.
2.	The method can have a return type.	A constructor cannot have a return type.
3.	Method name can be anything or can be class name as well.	The constructor name must be the same as the class name.
4.	The method must invoke explicitly.	A constructor can invoke automatically.
5.	If you don't have any method then the compiler won't provide any kind of method like a constructor.	If you don't have any constructor then the compiler will provide a default constructor.
6.	The method can be overridden or overloaded.	The constructor can be overloaded but cannot overridden.

54. Can we have a constructor in abstract class? If we can't instantiate abstract class then why there is a need for a constructor in abstract class?

- Yes. We can declare a constructor inside the abstract class.
- Absolutely right we cannot instantiate abstract class but when its Implementation class or child class instantiate that time abstract class constructor will be called.

55. Can I declare the constructor as an abstract?

- Constructors cannot be abstract.
- Abstract keyword can be applied to class and method, not constructor or blocks.

56. Can you declare constructor inside interface?

- Constructor cannot be declared inside interface.
- Constructor is mainly used to initialize the value. We cannot instantiate the interface so we cannot declare the constructor inside an interface.

57. Can you declare constructor static, final, abstract and synchronized?

- We cannot declare constructor static, final, abstract and synchronized.
- Purpose of all four keywords given above is different.

58. Why java doesn't allow static constructors?

- Static belongs to the class, not the object.
- Constructor belongs to object while initializing the value into it.

- When you create an object of child class then internally parent class constructor gets called automatically.
- If you declare the constructor as static then the child class cannot call the parent class constructor so it violates the inheritance mechanism. Hence we cannot declare the constructor as static.

59.
In your program, you have a static block, non-static block, constructor and main(). What will be the sequence of execution?

- The sequence of execution is as follows.

1. Static Block : will be executed at the time of class loading into memory.
2. Non-Static Block : Will be executed at the time of instantiation. But make a note it depends if you are instantiating the class before printing any statement in main. If you instantiate the class after printing the statement.
3. Constructor : Will be executed at the time of object initializing.
4. Main : Will execute last if you haven't instantiated the class.
5. We can see more scenarios.

Scenarios 1:

```
package simplifiedjava.crackedInterview;

public class BlockExecutionSequenceDemo {

        static {
        System.out.println("Static Block Excecution");
        }
        {
        System.out.println("Non Static Block Excecution");
        }
        public BlockExecutionSequenceDemo() {
        System.out.println("Constructor Excecution");
        }
        public static void main(String[] args) {
        System.out.println("Main Method Execution");
        }
}
```

Output:

```
Static Block Excecution
Main Method Execution
```

Scenarios 2:

```
package simplifiedjava.crackedInterview;

public class BlockExecutionSequenceDemo {

        static {
        System.out.println("Static Block Excecution");
        }
        {
        System.out.println("Non Static Block Excecution");
        }
        public BlockExecutionSequenceDemo() {
        System.out.println("Constructor Excecution");
        }
        public static void main(String[] args) {
        BlockExecutionSequenceDemo demo = new BlockExecutionSequenceDemo();
        System.out.println("Main Method Execution");
        }
}
Output :
Static Block Excecution
Non Static Block Excecution
Constructor Excecution
Main Method Execution
```

Scenarios 3:

```
package simplifiedjava.crackedInterview;

public class BlockExecutionSequenceDemo {

        static {
        System.out.println("Static Block Excecution");
        }
        {
        System.out.println("Non Static Block Excecution");
        }
        public BlockExecutionSequenceDemo() {
        System.out.println("Constructor Excecution");
        }
        public static void main(String[] args) {
        System.out.println("Main Method Execution");
        BlockExecutionSequenceDemo demo = new BlockExecutionSequenceDemo();
        }
}

Output:
Static Block Excecution
Main Method Execution
Non Static Block Excecution
Constructor Excecution
```

60.
What is the purpose to make a constructor private?

- Private constructor ensures that only one object can be created at a time.
- Cannot instantiate the class from outside the class.
- Generally for singleton design pattern uses a private constructor.

61.
What is constructor chaining?

- One constructor can call another constructor internally or explicitly is called constructor chaining.
- Constructor can call another constructor using this() and super().

62.
Can we override or overload the constructor?

- We cannot override the constructor but we can overload the constructor.
- Constructor overriding is not possible because the constructor must have the same name of the class and doesn't have any return type. But for overriding signature must be same. So ultimately overriding is not possible.

- Constructor overloading is possible because the constructor name is the same as class name and type of parameter and no of parameters are different so constructor overloading is allowed.

63.
Can a constructor have a return type? If no can you explain why?

- Constructor is specially used to initialize the object, not the business logic processing.
- If the constructor doesn't have any business logic and nothing is going to return so constructor doesn't have a return type.

64.
Can constructor have access specifiers? Which access specifiers are applicable to a constructor?

- Yes of course. Constructors can have access specifiers.
- Following access specifiers **are applicable** to the constructors.

1. Public
2. Private
3. Protected
4. Default

- Following access specifiers **are not applicable** to the constructors.

1. Abstract
2. Final
3. Native
4. Static
5. Synchronized.

65.
Can constructor declare exception with throws keyword?

- Yes we can declare an exception in the constructor definition with the throws keyword.
- We can declare both exceptions checked exceptions and unchecked exceptions in the constructor definition.
- For your reference, I have provided multiple scenarios in the below example.

```
package simplifiedjava.crackedInterview;

import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionThrowsInConstructor {

		public ExceptionThrowsInConstructor() throws FileNotFoundException{
		}
		public ExceptionThrowsInConstructor(int num) throws IOException{
		}
		public ExceptionThrowsInConstructor(String str) throws ArrayIndexOutOfBoundsException,
NullPointerException{
		}
}
```

66.

What is the default constructor and what is the difference between the default constructor and no-arg constructor and parameterized Constructor?

- Default constructor is a type of constructor provided by the compiler when there is no constructor declared in the class.
- Default constructor is a non-parameterized constructor.
- Default constructor doesn't have any parameters. Access specifiers are default.
- No-arg constructor doesn't have any parameter. But we can declare no-arg constructor as public, private, protected and default.
- Parameterized constructor must have at least one parameter. The parameterized constructor can be declared as public, private, protected and default.

67.
Can we declare multiple constructors in a single class?

- Yes. One class can have multiple constructors.
- If one class has multiple constructors then it is called constructor overloading.

68.
What is an abstract class? Can we instantiate abstract class?

- A class declared with an abstract keyword is called an abstract class.
- If you declare a class as an abstract then we can declare the abstract method and concrete method as well.
- If there is a single abstract method declared inside a class then it is mandatory to declare the class as an abstract class.
- It is not mandatory to declare at least a single abstract method inside an abstract class. In short, if you already declared class as abstract then it is not mandatory to the declared abstract method.

69.
Can I declare a final and abstract method in abstract class?

- You cannot declare final and abstract at a time. Either final or abstract method can be declared.
- You can declare the final method or abstract method inside the abstract class individually.

70.
Can we declare a class as Abstract without having a single abstract method?

- You can declare an abstract class without a single abstract method declared inside an abstract class.
- Make a note if you declared the already abstract method then it is mandatory to declare the class as abstract class.

71.
What is the difference between the final method and the abstract method?

- Final method cannot override.
- Abstract method must override in its implementation class.

72.
What is the difference between Interface and Abstract Class? Can you tell me one real-time example where Interface and abstract class

implementation is the best practice?

	Abstract Class	Interface
1.	Abstract class must be declared with abstract keyword.	An interface can be declared with the Interface keyword.
2.	An abstract class can have constructors.	An interface cannot have constructors.
3.	An abstract class can have a concrete method and an abstract method.	An interface can have only an abstract method. You cannot have a concrete method inside the interface.
4.	An abstract class can be declared private, protected, default and public methods.	An interface can declare only public methods.
5.	An abstract class can be declared private, protected and public variables.	An interface can declare only public variables.
6.	An Abstract class can declare non-final and local variables.	An interface can declare only final variables but not a local variable.
7.	An Abstract class can extend one class and implements multiple interfaces.	An interface can extend one or more interfaces only.
8.	An abstract class is a best practice when you have partial implementation and the rest of the implementation is remaining.	The interface is a best practice when there is no implementation at all.

73.
Will the code compile if I have declared a single abstract method in class but doesn't declare the class as an abstract class?

- No. Your class won't compile you will get a compile-time error.
- If you have declared already an abstract method inside a class then it is mandatory to declare that class as an abstract class.

74.
Can we declare static method and static variables in abstract class?

- Yes. You can declare static method and static variables inside an abstract class.

75.
Can we declare the concrete method in abstract class?

- Definitely, we can declare a concrete method inside an abstract class.
- Abstract class can have abstract methods as well as concrete methods.

76.
Can we use static members inside the non-static area?

- Yes. We can use static members inside the non-static area.
- We can observe this in the following demo.

```
package simplifiedjava.crackedInterview;

public class Parent {
        static int a = 10;
        public static void main(String[] args) {
        new Parent().check();
        }
        public void check(){
        System.out.println("Parent Class check() method = "+ a);
```

```
            }
}
Output :  Parent Class check() method = 10
```

77.
What is static import?

- Static import means we are importing static members from another class.
- Once we import static members then we can use those members without the class name.
- Generally, we use static members with the class name but once you import then there is no need to use the class name. You can directly access those members.

78.
Where the static variables are stored in java?

- The static variables were stored in the Perm-Gen space (also called the method area).
- The static variables are stored in the Heap itself.
- Java 8 onwards the Perm-Gen Space has been removed and a new space named Meta-Space is introduced which is not part of Heap memory anymore unlike the previous Perm-Gen Space.

79.
What is the diamond problem in Java? Why java doesn't support multiple inheritance?

- Diamond problem comes into the picture in multiple inheritance.
- When one class is inheriting the same property from multiple classes. This is called the diamond problem.
- E.g. There are 3 classes. Class A, Class B and Class C. Class A and Class B have add() method and when class C extends class A and class B. Class C is trying to override add() method then which method compiler will refer for overriding. There is ambiguity. This is a diamond problem. That's why java doesn't support multiple inheritance.

80.
What is a marker interface? Can you tell me the example of a marker interface?

- An empty interface is called a marker interface.
- Marker interface doesn't have any method or variable.
- List of marker interfaces.

  1. Serializable interface.
  2. Cloneable interface.
  3. Remote Interface.

81.
What are the wrapper classes in java? Why do we require wrapper classes?

- Wrapper class is a class whose objects are wraps or contains primitive data types.
- Wrapper class hold primitive values and is represented as an object.
- All primitives has its own wrapper class.
- E.g. int has Integer, float has Float, double has Double etc.
- In the collection framework if we wanted to represent primitive as an object for manipulation then wrapper class can be used.

82. Can we declare a class inside the interface? If yes can you invoke a method of the class which is declared inside an interface?

- Yes we can declare a class inside the interface and invoke any method from the class. Technically it is possible.
- For a demonstration please refer to the below example.

```
package simplifiedjava.crackedInterview;

public interface ClassInsideInterface {

            public class InnerClass{
            int i = 0;
            public void greetings() {
            System.out.println("Hello");
            }
            }
            public static void main(String[] args) {
            ClassInsideInterface.InnerClass in = new InnerClass();
            in.greetings();
            }
}
Output :  Hello
```

83. Can we declare an interface inside a class? If yes can you invoke a method of the class which is declared inside an interface?

- Yes we can declare an interface inside a class.
- For a demonstration please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class InterfaceInsideClass {

            interface MyInterface{
            public void greetings();
            }
            class InnerClass implements MyInterface{
            public void greetings() {
            System.out.println("Hi!! Interface Inside Class.");
            }
            }
            public static void main(String[] args) {
            InterfaceInsideClass.InnerClass inner = new InterfaceInsideClass().new InnerClass();
            inner.greetings();
            }
}
Output :  Hi!! Interface Inside Class.
```

84. Can we declare a class inside a class?

- Yes we can declare a class inside a class.
- For a demonstration please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class ClassInsideClass {

            class Inner{
            public void greetings() {
            System.out.println("Hello From Inner Class");
            }
            }
            public static void main(String[] args) {
            ClassInsideClass.Inner inner = new ClassInsideClass().new Inner();
            inner.greetings();
            }
}
Output:  Hello From Inner Class
```

85.
Can we declare interface inside an interface?

- Yes we can declare interface inside interface.
- For demonstration please refer below example.

```
package simplifiedjava.crackedInterview;

public interface OuterInterface {

            public interface InnerInterface{
            public void greetings();
            }
}

class MyClass implements OuterInterface.InnerInterface{
            public static void main(String[] args) {
            new MyClass().greetings();
            }
            public void greetings() {
            System.out.println("Implementation From Inner Interface");
            }
}
Output :  Implementation From Inner Interface
```

86.
What is variable shadowing?

- Shadowing means a variable declared within a certain scope (Declared inside method or block) has the same name as a variable declared in an outer scope.
- In short, we have the same variable name in block or method and class level variable.
- For a demonstration please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class VariavleShadowingDemo {

            int num = 10;
            public void printNum() {
            int num = 20;
            System.out.println("Num Value "+ num);
            }
            public static void main(String[] args) {
            new VariavleShadowingDemo().printNum();
            }
}
Output : 20. Because local variable has highest priority inside method.
```

87.
What is static method shadowing?

- The static method in the superclass will be hidden by the one that is declared in the subclass. This mechanism is known as method shadowing.
- Method shadowing is applicable only for a static method.

```
package simplifiedjava.crackedInterview;

public class Parent {
            public static void printGreetings() {
            System.out.println("Parent Class Static Method");
            }
}



package simplifiedjava.crackedInterview;

public class Child extends Parent{

            public static void printGreetings() {
            System.out.println("Child Class Static Method");
            }
            public static void main(String[] args) {
            Parent.printGreetings();
            Child.printGreetings();
            }
}
Output:
Parent Class Static Method
Child Class Static Method
```

88.
What is object cloning?

- Object cloning means creating a duplicate copy of an object.
- clone() can be used to creating the object clone.

89.
What is a co-variant return type?

- Covariant return means that when one overrides a method, the return type of the overriding method is allowed to be a subtype of the overridden method's return type.
- In the below example, There are two classes Human and Men. Men class extends the Human class.
- Both class has dressingStyle() method. This method is overriding in the Men class. Both methods have the same return type but if you observe then the parent class method returns the actual Human class object and the child class method returns the Child class(Men Class) object when the return type is Human. This is called covariant return type.

```
package simplifiedjava.crackedInterview;

public class Human {

	public Human dressingStyle() {
	return new Human();
	}
}
package simplifiedjava.crackedInterview;

public class Men extends Human {

	public Men dressingStyle() {
	return new Men();
	}
}
```

90.
What is the purpose of the final keyword where it can be applicable?

- Final keyword can be applied to the following.

1. **Variable:** Final variable value cannot change once assigned.
2. **Method:** Final method cannot override.
3. **Class:** Final class cannot be sub-classed.

91.
What is static binding and dynamic binding?

- Static binding occurs compile time.
- Dynamic binding occurs runtime
- Best example of static binding is method overloading.
- Best example of dynamic binding is method overriding.

92.
What is instanceOf in java? Where it can be used?

- instanceOf is an operator in java.
- instanceOf can be used to check the type of object.
- syntax for instanceOf operator is, (JavaInterview instanceOf String);

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class InstanceOfDemo {

	public static void main(String[] args) {
	List list = new ArrayList();
	list.add(new String("Java"));
	list.add(new Integer(100));
	list.add(new ITEmployee());
	for(Object obj : list) {
	if(obj instanceof String) {
	System.out.println((String)obj);
	}else if(obj instanceof Integer) {
```

```
            System.out.println((Integer)obj);
            }else if(obj instanceof ITEmployee) {
            System.out.println((ITEmployee)obj);
            }
            }
            }
}

class ITEmployee{
}
```

93.
What is the ternary operator in java?

- Ternary operator lets you write if statement in one line.
- Ternary operator evaluates either true or false.
- There are three parts of the ternary operator. The first part is expression end with a question mark and two options separated with a colon.
- Variable = condition ? expression1: expression2.

```
package simplifiedjava.crackedInterview;

public class TernaryOperatorDemo {

            public static void main(String[] args) {
            int num=20;
            String typeNum = (num > 0) ? "Positive" : "Negative";
            System.out.println(typeNum);
            }
}
Output :  Positive
```

94.
What is the difference between relative path and absolute path?

- **Absolute Path**: Absolute path specifies the location from the root directory.
- **E.g.** E:/drawings/images/sky.image
- **Relative Path**: Relative path specifies the location from the current directory.
- **E**.g. images/sky.image

95.
What is garbage collections? Who is responsible for garbage collection? How java reclaims a memory?

- Garbage collection is a process to delete unused objects and reclaim the memory.
- JVM is responsible to reclaim the memory.
- JVM finds the unused objects and no longer references objects that reside in memory and then JVM invokes finalize() method. This finalize() methods released the resources acquired by that objects.
- Once invoked the finalize() method will mark all targeted objects and simply removed them from memory.
- There are many GC algorithms that exist to implement GC. One of them is mark and sweep.

96.
How does an object become eligible for garbage collection?

- Object can be made eligible for garbage collection when there is no single reference variable pointing to it.
- There are three ways objects can make eligible for garbage collection.

  1. When an object goes out of scope.
  2. Objects reference variable explicitly assigned a null value.
  3. Reinitialize the reference variable.

97.
Can we ask JVM to collect the garbage or destroy the objects?

- Yes we can just recommend destroying unused objects but cannot force them to do it.
- System.gc() method can be used to recommend the garbage collection.

98.
What is the difference between equals() and ==?

- Equals() is the method and can be used for content comparison.
- == is an operator and can be used for reference comparison.

99.
What are Boxing and Unboxing?

- The conversion of primitive data type into its corresponding wrapper class is called boxing.
- The conversion of wrapper class type into its corresponding primitive type is called unboxing.
- Please refer to the below example for boxing and unboxing.

```
package simplifiedjava.crackedInterview;

public class BoxingUnboxingDemo {

        public static void main(String[] args) {
        int num = 100;
        Integer wrapperNum1 = new Integer(num);
        Integer wrapperNum2 = 200;
        System.out.println("Auto Boxing "+ wrapperNum1 + " and " + wrapperNum2);
        Integer wrapperNum3 = new Integer(300);
        num = wrapperNum3;
        System.out.println("Unboxing "+ num);
        }
}
Output :
Auto Boxing 100 and 200
Unboxing 300
```

100.
What are Upcasting and Downcasting?

- Upcasting means casting the child object into the parent object.
- Implicit upcasting is possible.
- UpCasting is also called Widening.
- Downcasting means casting the parent object into a child object.
- Implicit downcasting is not possible.

- Downcasting is also known as Narrowing.
- Please refer to the below example for upcasting and downcasting.

```
package simplifiedjava.crackedInterview;

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

package simplifiedjava.crackedInterview;

public class Child extends Parent{

		public void check(){
		System.out.println("Child Class check() method");
		}
}


package simplifiedjava.crackedInterview;

public class UpCastingDownCastingDemo {

		public static void main(String[] args) {
		Parent p = (Parent)new Child(); // Upcasting
		p.check();
		Parent p1 = new Child();
		Child c = (Child)p1;	//downcasting
		p1.check();
		}
}
Output :
Child Class check() method.
Child Class check() method.
```

101.
What is Widening?

- When we assign a value of a smaller data type to a bigger data type.
- It is possible when two data types are compatible.
- Widening Scope: **byte -> short -> int -> long -> float -> double.**

```
package simplifiedjava.crackedInterview;

public class WideningDemo {

		public static void main(String[] args) {
		int i=100;
		long j = i;
		float k = j;
		float k1 = i;
		System.out.println("Value of i = "+ i + " \nValue of j = "+ j + "\nValue of k = "+ k + "\nValue of k1 = 
"+ k1);
		}
}
```

102.

What is Narrowing?

- When we assign a value of a bigger data type to a smaller data type.
- It is possible when two data types are compatible.
- Narrowing Scope: **byte <- short <- int <- long <- float <- double.**

```
package simplifiedjava.crackedInterview;

public class WideningDemo {

	public static void main(String[] args) {
	float i=100;
	long j = (long) i;
	int k = (int) j;
	int k1 = (int) i;
	System.out.println("Value of i = "+ i + " \nValue of j = "+ j + "\nValue of k = "+ k + "\nValue of k1 = 
"+ k1);
	}
}
```

103.
What is equals() and hashcode() contract?

- If the two objects are equal then they should have the same hashcode and if two objects are not equal then they may or may not have the same hashcode.
- Objet1.equals(Object2) if true then hashcode must be same of both objects.
- Object1.equals(Object2) if false then hashcode may be or may not be same.
- As per the oracle documentation, both methods should be overridden to get the complete equality mechanism you must override equals() and hashcode() methods.

104.
What is Comparable and Comparator? What is the difference between Comparable and Comparator?

- Comparable and Comparator are interfaces that can be used for sorting purposes.
- Comparable interface can be used for natural sorting order.
- Comparator interface can be used to customize sorting orders.
- Comparable has compareTo() method.
- Comparator has compare() method.

105.
Can you write a code to customize Employee Sorting order by Salary or Designation?

```
package simplifiedjava.crackedInterview;

import java.util.Comparator;

public class Employee {

	private String empName;
	private String dept;
	private Double salary;
	private String designation;
	public Employee(String empName, String dept, Double salary, String designation) {
	super();
	this.empName = empName;
	this.dept = dept;
```

```
            this.salary = salary;
            this.designation = designation;
            }
            public Employee(String empName) {
            this.empName = empName;
            }
            public String getEmpName() {
            return empName;
            }
            public String getDept() {
            return dept;
            }
            public void setDept(String dept) {
            this.dept = dept;
            }
            public Double getSalary() {
            return salary;
            }
            public void setSalary(Double salary) {
            this.salary = salary;
            }
            public String getDesignation() {
            return designation;
            }
            public void setDesignation(String designation) {
            this.designation = designation;
            }
            public void setEmpName(String empName) {
            this.empName = empName;
            }
            @Override
            public String toString() {
            return "Employee [empName=" + empName + ", dept=" + dept + ", salary=" + salary + ",
designation=" + designation
            + "]";
            }
}

package simplifiedjava.crackedInterview;

import java.util.Comparator;

public class SortEmployeeBySalary implements Comparator<Employee>{

            public int compare(Employee e1, Employee e2) {
            return e1.getSalary().compareTo(e2.getSalary());
            }
}

package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class ComparableAndComparatorDemo {
```

```
            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee("Yogesh","IT",100000.00,"System Analyst"));
            empList.add(new Employee("Arpita","Mangement",1200000.00,"Trustee"));
            empList.add(new Employee("Shweta","DevOps",45000.00,"Jenkin Engineer"));
            empList.add(new Employee("Shruti","IT",65000.00,"DB Admin"));
            empList.add(new Employee("Priyanka","IT",35000.00,"Test Engineer"));

            System.out.println("=================== Employee Before Sorting
=====================");
            System.out.println(empList);
            System.out.println("=================== Employee After Sorting
=====================");
            empList.sort(new SortEmployeeBySalary());
            System.out.println(empList);
            }
}
```

Output:

```
=================== Employee Before Sorting =====================
[Employee [empName=Yogesh, dept=IT, salary=100000.0, designation=System Analyst], Employee
[empName=Arpita, dept=Mangement, salary=1200000.0, designation=Trustee], Employee [empName=Shweta,
dept=DevOps, salary=45000.0, designation=Jenkin Engineer], Employee [empName=Shruti, dept=IT,
salary=65000.0, designation=DB Admin], Employee [empName=Priyanka, dept=IT, salary=35000.0,
designation=Test Engineer]]
=================== Employee After Sorting =====================
[Employee [empName=Priyanka, dept=IT, salary=35000.0, designation=Test Engineer], Employee
[empName=Shweta, dept=DevOps, salary=45000.0, designation=Jenkin Engineer], Employee [empName=Shruti,
dept=IT, salary=65000.0, designation=DB Admin], Employee [empName=Yogesh, dept=IT, salary=100000.0,
designation=System Analyst], Employee [empName=Arpita, dept=Mangement, salary=1200000.0,
designation=Trustee]]
```

106.
What is ENUM? What is the purpose of ENUM?

- Enum stands for Enumeration.
- Enum is a data type that is a set of constants.
- Enum can be declared inside or outside the class.
- Please refer to the below example for enum implementation.

```
package simplifiedjava.crackedInterview;

public class EnumDemo {

            public enum months
{JANURAY,FEBRUARY,MAR,APRIL,MAY,JUNE,JULY,AUGUST,SEPTEBER,OCTOBER,NOVEMBER,DECEME
R};
            public static void main(String[] args) {
            for(MONTHS month : months.values()) {
            System.out.print(" "+ month);
            }
            }
}
Output:  JANURAY FEBRUARY MAR APRIL MAY JUNE JULY AUGUST SEPTEBER OCTOBER NOVEMBER DECEMER
```

107.

Can we declare constructor, methods and fields inside the ENUM definition?

- Yes. We can declare fields, methods and constructors inside ENUM.
- Please refer to the below example for enum implementation.
- Enum constructors must be declared as private if you don't declare private then by default enums constructor is private so cannot declare public or protected.

```
package simplifiedjava.crackedInterview;

public class EnumConstructorMethodsDemo {

            public enum Months{

JANURAY(1),FEBRUARY(2),MARCH(3),APRIL(4),MAY(5),JUNE(6),JULY(7),AUGUST(8),SEPTEBER(9),OCTOBI
(12);
            private int monthCounter;
            private Months(int monthCounter) {
            this.monthCounter = monthCounter;
            }
            }
            public static void main(String[] args) {
            for(Months month : Months.values()) {
            System.out.println(month +" = "+ month.monthCounter);
            }
            }
}
```

108.
When the constructor will be executed once you declare the constructor inside the enum?

- When constant enum values are passed to the constructor.

109.
What is the difference between enum, Enum and Enumeration?

- **enum :** enum is a keyword that can be used to define a group of named constants.
- **Enum**: Enum is a class in java, present.lang package, Every Enum in java should be a direct child class of Enum class. Hence this class is a base class.
- **Enumeration:** Enumeration is an interface in java that is present in java.util package. We can use Enumeration to get an object step by step.

110.
What is Assertion? What is the role of Assertion in java development?

- Assertion is a statement that can be used to test the assumption of your program.
- Assertion is used for debugging purposes.
- **Role of Assertion is:**

1. There is a very common way usage of System.out.println() statement for debugging but the problem of this statement degrades the performance of the application. To overcome this problem sun people introduced assertions.
2. The advantage of assertion is, we don't require removing all print statements you can simply enable or disable assertions.

Interview Questions on String

111.
What is an immutable class?

- Immutable class means the value or content cannot be changed once an object is created.
- Best example of immutable in java is the String class. The String object cannot be changed once it is created.
- All wrapper classes are immutable classes. Wrapper classes are Integer, Float, Double, Boolean, Byte, Short etc.

112.
Why String is immutable?

- String objects are immutable cause string objects are cached in SCP(String Constant pool).
- Cached String objects are shared between multiple clients.
- If anyone changed the value of the String object then all clients get affected. That's why String is immutable in java.

113.
Can we modify the existing String object? If I tried to change the existing String object then what will happen?

- No we cannot modify the existing String class object.
- If you are trying to modify an existing string object then internally new object will be created and if you pointing to a newly created object to reference variable then the old object will be eligible for garbage collection.

114.
What is the difference between StringBuilder and StringBuffer?

	StringBuilder	**StringBuffer**
1.	StringBuilder is not a thread-safe.	StringBuffer is a thread-safe.
2.	The performance of StringBuilder is good as compare to StringBuffer.	The performance of StringBuffer is lower as compare to StringBuilder cause StringBuffer is thread-safe.
3.	StringBuilder sb = new StringBuilder("ac");	StringBuffer sb = StringBuffer("ac");

115.
What is the use of the intern() method?

- Intern() method returns a canonical representation of String.
- In simple word, intern() method return String object from SCP(String Constant Pool).

116.
What is SCP(String Constant Pool)?

- String constant pool is a separate place in heap memory where all the String objects are stored.
- If you create an object using a new keyword then the object gets created on heap memory and keep the same string object into string constant pool.
- If you create an object without a new keyword then the object doesn't create on heap memory that object will be created on string constant pool.
- If an object exists then the reference variable pointing to an existing object instead of creating a new one.

117.

What is the difference between String s = "abc" and String s = new String("abc")?

- **String s=”abc”:** On this line String object will not create on heap memory. ‘abc’ string will be directly stored on the string constant pool.
- **String s = new String(“abc”):** new String(“abc”): On this line String object will be created on heap memory and ‘abc’ will be stored in string constant pool.

118.
List out some inbuilt methods of String class?

- String of inbuilt method of String class.

1. length()
2. substring()
3. contains()
4. equals()
5. concat()
6. replace()
7. isEmpty()
8. intern()
9. indexOf()
10. trim()
11. toUpperCase()
12. toLowerCase()

119.
What is the use of trim() method in the String class?

- Trim() of String class is used to remove the blank spaces of String.
- Trim() method removes blank spaces before and after the String.
- Trim() method doesn’t remove the blank spaces in between String.

120.
How to create an immutable class or What are the steps to create immutable objects?

- **Please refer to the below steps to create an immutable class.**

1. Create a final class so a subclass is not possible.
2. Make all instance variables final so no one can change the value.
3. Do not provide any setter method just provide a getter method.

- For your reference, I have created a new immutable class.

```
package simplifiedjava.crackedInterview;

public final class ImmutableClassDemo {

        private final String empCode="E001";

        public String getEmpCode() {
        return empCode;
        }
}
```

121.

What is the default size of StringBuffer and StringBuilder?

- The default size of StringBuffer and StringBuilder is **16.**

122.
What will be the size of StringBuffer or Builder in the following example?

String s = "abc"; StringBuffer sb = new StringBuffer(s); System.out.println(sb.capacity());

```
package simplifiedjava.crackedInterview;

public class StringBuilderSizeCalculationDemo {

        public static void main(String[] args) {
        String s = "abc";
        StringBuffer sb = new StringBuffer(s);
        System.out.println(sb.capacity());
        }
}
Output:  19
```

123.
What will happen if I am trying to keep adding new characters once the StringBuffer or StringBuilder gets fulled?

- The default size of StringBuffer and StringBuilder is 16.
- When StringBuilder or StringBuffer object gets full and we are still inserting new characters that time internally new StringBuilder or StringBuffer object gets created copy all content of the old object to new object.
- Old objects will be eligible for garbage collection.

124.
Can you specify the scenario where you can implement String, StringBuffer and StringBuilder?

- **String:** When content is fixed and never change throughout life then String is a good option.
- **StringBuffer:** When content is not fixed and the requirement is thread-safety then go for StringBuffer.
- **StringBuilder:** When content is not fixed and content would be changing multiple times and no need for thread safety then go for StringBuilder.

Interview Questions on Exceptions

125.
What are the exceptions?

- Exception is an event, can be occurred during the execution of the program that disrupts the normal flow of the program.
- You can say an Exception is an abnormal event that can be occurred during the execution of the program.
- Exception can stop the execution of the program if not handled.

126.
What is the difference between exception and error?

	Error	Exception
1.	Error is an invalid condition.	An exception is an abnormal condition.
2.	An error cannot be handled.	An exception can be handled.
3.	Error stops your program immediately.	If you handle exceptions then your program may execute smoothly.
4.	An error can occur only run time but not compile time.	An exception can occur compile time and run time.
5.	Error belongs to java.lang.Error	Exeception belongs to java.lang.Exception.
6.	e.g. OutOfMemoryError	e.g. NullPointerException, FileNotFoundException.

127.
What is the difference between checked exceptions and unchecked exceptions?

	Checked Exception	Unchecked Exception
1.	Checked Exceptions can occur at Compile time.	Unchecked Exceptions can occur at run time.
2.	If you don't handle checked exceptions then your code will not compile. You will get a compile-time error.	If you don't handle unchecked exceptions you will get an exception at run time but your program will be compiled.
3.	Checked Exceptions extends Throwable and Error but except RuntimeException.	Unchecked Exceptions extends RuntimeException.
4.	Examples : IOException, SQLException, FileNotFoundEx ception	Examples: NullPointerException, ArithmeticException, ArrayIndexOutofBoundException

128.
How many types we can handle the exception?

- We can handle exceptions in two ways.

1. Using try and catch block.
2. Using throws keyword.

- Please refer to the below example to try-catch block.

```
package simplifiedjava.crackedInterview;

public class Aeroplane {
        public void setLocation(String locationName) {
        try {
        if(locationName.equals(null)) {
        throw new NullPointerException();
```

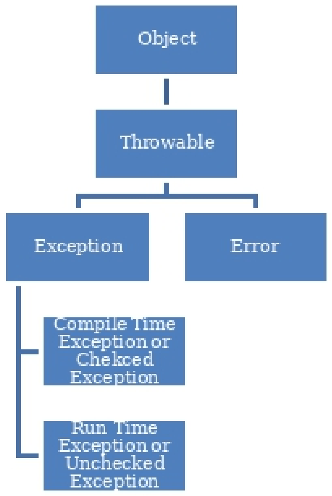
```
            }
            }catch(NullPointerException e) {
            e.printStackTrace();
            }
            }
}
```

- Please refer to the below example for the throws keyword.

```
package simplifiedjava.crackedInterview;
public class Aeroplane {

            public void setLocation(String locationName)throws NullPointerException {
            if(locationName.equals(null)) {
            throw new NullPointerException();
            }
            }
}
```

129. Which is the parent class of the Exception hierarchy?

- Parent Class of Exception is Throwable.


Object
Throwable
Exception
Error
Compile Time Exception or Chekced Exception
Run Time Exception or Unchecked Exception

130.
What is the difference between the throw and throws keyword?

	Throw	Throws
1.	Throw keyword can be used to throw an exception manually or explicitly.	Throws keyword can be used to declare the list of exceptions.
2.	Throw can be declared inside a method.	Throws can be declared in method definition followed by the parameter list.
3.	Only one Exception can be thrown with the throw keyword.	Multiple Exceptions can be declared with the throws keyword.
4.	The checked exception cannot throw with the throw keyword.	Checked and Unchecked exceptions can declare with the throws keyword.

131.
What is exception propagation?

- An Exception is thrown from the current method and if it is not handled then it drops to the previous method. If that exception is not handled in the previous method as well then it drops to the previous method and so on. If you never handle that exception then that exception reached to main method or base method which is called exception propagation.
- Please refer to the below diagram of exception propagation.

Current Method	Exception occurred in this function and it is not handled.
m4()	Exception not handled in current method so dropped in method m4().
m3()	Exception not handled in m4() method so dropped in method m3().
m2()	Exception not handled in m3() method so dropped in method m2().
m1()	Exception not handled in m2() method so dropped in method m1().
main() or base()	Exception not handled in m1() so dropped in main() method and the program terminates abruptly.

132.
Can you explain the exception hierarchy?

- **Compile Time Exception or Checked Exception.**

1. FileNotFoundException.
2. ClassNotFoundException.
3. NoSuchFieldException.
4. NoSuchMethodException.
5. InvalidClassException.
6. InvalidObjectException.
7. CloneNotSupportedException.
8. InterruptedException.

- **Runtime Exception or Unchecked Exception.**

1. NullPointerException.
2. ArithmeticException.
3. ClassCastException.
4. IndexOutOfBoundException.
5. ArrayIndexOutOfBoundException.
6. StringIndexOutOfBoundException.
7. IllegalStateException
8. IllegalMonitorStateException

- **Error.**

1. OutOfMemoryError.
2. StackOverFlowError.
3. IOError.
4. AssertionError.

133.
What is a multi-catch block? In which java version it has been introduced?

- In one catch block we can handle multiple exceptions is called a multi-catch block.

- Pipe Operator "|" can be used to segregate multiple exceptions.
- Multi catch block has been introduced in Java 7.
- Please refer to the below example for a multi-catch block.

```
package simplifiedjava.crackedInterview;

public class Aeroplane {

        public void setLocation(String locationName){
        try {
        if(locationName.equals(null)) {
        throw new NullPointerException();
        }
        }catch(NullPointerException | ArithmeticException e) {
        e.printStackTrace();
        }
        }
}
```

134.
What is the role of the printStackTrace() method? This method belongs to which class?

- printStackTrace() method is used to diagnosing the exceptions.
- This method points out exactly where the exception occurred. It will print the function name and line number as well.
- printStackTrace() method belongs to the Throwable class which is the super parent class of the Exception hierarchy.
- For reference to the printStackTrace() method you can refer to the previous example.

135.
Can we create a custom exception class? If yes then what are the steps to create a custom exception class?

- Yes we can create a custom exception class.
- Please follow the following steps to create your custom exception.

1. Create one Custom class and extends the Exception class.
2. Create one String type local variable and assign your custom message to that variable.
3. Pass the custom message in the Custom class constructor while instantiating the custom class.
4. Override toString() method to print the custom message.
5. We can simply throw a custom exception in any method and pass a custom message while creating a custom object.

- Please refer to the below example to create a custom exception.

```
package simplifiedjava.crackedInterview;

public class CustomException extends Exception {

        private String message;
        public CustomException(String message) {
        this.message = message;
        }
```

```
            public String toString() {
            return message;
            }
            public static void main(String[] args) {
            try {
            if(args.length >= -1) {
            throw new CustomException("No Argument Exception");
            }
            }catch(CustomException e) {
            System.out.println(e);
            }
            }
}
Output:  No Argument Exception
```

136.
Can you identify which try, catch and finally block combination is valid?

- Following are the combinations.

```
1.
try {
}catch() {
}
try {
}catch() {
}

2.
try {
}catch() {
}
try {

}finally {
}

3.
try {
}


4.
try {
}finally {
}catch() {
}

1 and 2 are valid.
3 and 4 are invalid.
```

137.
Can we declare an empty catch block?

- Yes you can declare an empty catch block. But it's mandatory to declare any exception class in the catch block parameter.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class CustomException extends Exception {
		public static void main(String[] args) {
		try {
		}catch(Exception e) {
		}
		}
}
```
It is mandatory to declare the Exception class name in the catch method signature.
But inside the catch block, you can keep it blank. If you missed to declared an Exception in the catch method signature then you will get a compile-time error.

138. Can you explain why NullPointerException occurred and how to handle it?

- Calling a method on null reference or trying to access a field of a null reference will trigger a NullPointerException.
- You can fix it either by making sure you are invoking the method non-null object.
- Make sure before invoking the method or modifying the field object is non-null.
- You can put a null check before invoking or performing any operation.
- You may use a try-catch block to handle NullPointerException.
- Please refer to the below scenario.

  a. Trying to find the length of an array when it is null.
  b. Calling the instance method on the null object.
  c. Trying to modify the field of the null object.
  d. Trying to count a length of String object when it is null.

Demo 1: Put null check before invoking or performing any operation.

```
package simplifiedjava.crackedInterview;

public class NullPointerExceptionDemo {

		String str;
		public static void main(String[] args) {
		new NullPointerExceptionDemo().calculateLength();
		}
		public void calculateLength() {
		if(str != null) {
		System.out.println(str.length());
		}else {
		System.out.println(" Variable is "+ str);
		}
		}
}
```
Output: Variable is null

Demo 2: Put code inside try catch block.

```
package simplifiedjava.crackedInterview;

public class NullPointerExceptionDemo {

		String str;
```

```
        public static void main(String[] args) {
        new NullPointerExceptionDemo().calculateLength();
        }
        public void calculateLength() {
        try {
        str.length();
        }catch(NullPointerException e) {
        System.out.println("Null Pointer Exception Handled");
        }
        }
}
Output:  Null Pointer Exception Handled
```

139. Can you explain why ClassCastException occurred and how to handle it?

- When you are trying to cast incompatible objects it triggers ClassClassException.
- You can say ClassCastException occurs when you are improperly converting a class from one type to another type.
- When you are trying to convert a Human object into an Animal object which is incompatible.
- You can use instanceof operator to avoid ClassCastException.
- Please refer to below example to handle ClassCastException.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class ClassCastExceptionHandleDemo {

        public static void main(String[] args) {
        List objList = new ArrayList();
        objList.add(new Integer(10));
        objList.add(new Integer(10));
        objList.add(new Double(10.00));
        objList.add(new String("abc"));

        for(Object obj: objList) {
        if(obj instanceof Integer) {
        System.out.println("Integer Type");
        }else if(obj instanceof Double) {
        System.out.println("Double Type");
        }else if(obj instanceof String) {
        System.out.println("String Type");
        }
        }
        }
}
Output:
Integer Type
Integer Type
Double Type
String Type
```

140.

Can you explain why OutOfMemoryError occurred and how to handle it?

- OutofMemoryError occurs when your program is expecting more memory than JVM has available.
- OutOfMemoryError normally occurs when you have a code with an infinite loop.
- You can handle OutOfMemoryError by avoiding infinite loop.
- Allocate more heap space to JVM. Perm-Gen Space must increase but recently it has changed to meta-space.

141.
Can you explain why StackOverFlowError occurred and how to handle it?

- StackOverFlowError occurs when the nesting function calls too deeply.
- Generally StackOverFlowError comes in recursion functions. Recursion means a function calls itself.
- To avoid StackOverFlowError you have to make sure recursive call must terminate somewhere.
- Please refer to the below example for StackOverFlowerror.

```
package simplifiedjava.crackedInterview;

public class StackOverflowErrorDemo {

        public static void main(String[] args) {
        recursiveFunction(1);
        }
        public static void recursiveFunction(int no) {
          System.out.println("Number: " + no);
          if (no == 0)
            return;
          else
            recursiveFunction(++no);
         }
}
Output:  Exception in thread "main" java.lang.StackOverflowError
```

142.
Can you explain why NoClassDefFoundError occurred?

- Java Virtual Machine is not able to find a particular class at runtime which was available at compile which triggers NoClassDefFoundError.
- If a class was present during compile time but not available in java classpath during runtime.

143.
Can you explain why ExceptionInInitializerError occurred?

- Whenever there is an error in the static block ExceptionInInitializerError will be thrown.
- Generally, this exception will be thrown at the time of class loading because static block executes at class loading time.

```
package simplifiedjava.crackedInterview;

public class ExceptionInInitializerErrorDemo {
        static {
```

```
        try {
        int no = 10/0;
        }catch(NullPointerException e) {
        e.printStackTrace();
        }
        }
        public static void main(String[] args) {
        System.out.println("Main");
        }
}
Output:  Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArithmeticException: / by zero     at
simplifiedjava.crackedInterview.ExceptionInInitializerErrorDemo.<clinit>(ExceptionInInitializerErrorDemo.java:6)
```

144. What's the difference between StackOverflowError and OutOfMemoryError in java?

- **StackOverflowError:** If there is no memory available in the stack for storing function call or local variable, JVM will throw StackOverflowError.
- **OutOfMemoryError:** If there is no memory available on heap memory while creating an object then JVM will throw OutOfMemoryError.

145. What is the difference between ClassNotFoundException and NoClassDefFoundError?

	ClassNotFoundException	NoClassDefFoundError
1.	ClassNotFoundException is checked exception.	NoClassDefFoundError is an unchecked Exception.
2.	It is a child class of Exception class.	It is a child class of Error class.
3.	This exception occurs when JVM is trying to load a class that is not present in the classpath at runtime.	This error occurs when JVM is trying to load a class that is not present in classpath at run time but was present at compile time.
4.	It occurs when the classpath is not updated with the required jar files.	It occurs when the required class definition is missing at runtime.
5.	This exception thrown by Class.forName() or loadClass().	This error is thrown by Java Runtime System.

146. Can we skip the finally block to execute? if yes then how?

- Yes we can skip the finally block to execute.
- We can skip using System.exit(0). Once you execute this method then your program terminates.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class SkipFinallyBlockDemo {

        public static void main(String[] args) {
        try {
        System.out.println("Inside try Block");
```

```
            System.exit(0);
            System.out.println("After Excecuting exit() method");
            }catch(Exception e) {
            System.out.println("Inside catch Block");
            e.printStackTrace();
            }finally {
            System.out.println("Inside finally Block");
            }
            }
}
Output:  Inside try Block
```

147.
Can we have multiple catch blocks?

- Yes. You can have multiple catch block but there is some condition to declared multiple catch blocks.

148.
How should be the sequence of List of Exception classes in the catch block?

- When you are declaring multiple catch blocks then the child class must declares first and then its parent and then parents parent and so on.
- Please refer to the below example for more clarity.

```
package simplifiedjava.crackedInterview;
public class MultipleCatchBlockDemo {

            public static void main(String[] args) {
            try {
            }catch(ArrayIndexOutOfBoundsException ae) {
            ae.printStackTrace();
            }catch(IndexOutOfBoundsException ie) {
            ie.printStackTrace();
            }catch(Exception e) {
            e.printStackTrace();
            }
            }
}
```

Explanation: ArrayIndexOutOfBoundsException is a child class of IndexOutOfBoundsException and IndexOutOfBoundsException is a child class of Exception class. If you shuffle then you will get a compile-time error. Most child classes should be first and then next child and then parent class.

149.
What is the difference between final, finally and finalize?

	Final	Finally	Finalize
1.	Final is a keyword.	Finally is block.	Finalize() is a method.
2.	Final keyword is applicable to class, method and variable.	Finally block is applicable in exception handling with try or try catch block.	Finalize() method can be used in garbage collection.
3.	Final class cannot be sub-classed.	Every time finally block will be executed exception is thrown or	Finalize() method can run to release the resources acquired

	Final method cannot be overridden. Final variable is not modifiable.	not. An Exception is caught or not. If you execute System.exit(0) then only the finally block won't be executed.	by objects. After running finalize() method unreachable objects will be eligible for garbage collection.

150.
Can we create a try block inside a try block?

- Yes. You can have any number of nested try-catch blocks.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class NestedTryCatchDemo {

        public static void main(String[] args) {
        try {
        try {
        try {
        }catch(Exception e) {
        e.printStackTrace();
        }
        }catch(Exception e) {
        e.printStackTrace();
        }
        }catch(Exception e) {
        e.printStackTrace();
        }
        }
}
```

151.
Can we write a try-catch block inside a catch block? What is the purpose to write try block inside catch block?

- Yes. We can write a try-catch block inside the catch block.
- Generally, we are required to try/catch block inside catch block to close the connections or close the statements or flush the data from statements and connections.

152.
What are the rules we must follow while throwing an exception in an overridden method?

- If the superclass method does not declare an exception, the subclass's overridden method cannot declare the checked exception.
- If the superclass method does not declare an exception, the subclass's overridden method can declare an unchecked or runtime exception.
- If the superclass method declares an exception, the subclass's overridden method can declare the same exception or child class of that exception but cannot declare the parent class of that exception.
- Please refer to the below examples for more clarity.

Parent Class Method	Child Class Method	Status
Calculate()	**Calculate() throws IOException**	Not Allowed. **Reason:** If the parent class method

		doesn't throw any exception then the child class method should not through any checked exception. You can throw unchecked exception.
Calculate()	**Calculate() throws NullPointerException**	Allowed.
Calculate() throws IOException	**Calculate()**	Allowed.
Calculate() throws NullPointerException	**Calculate()**	Allowed.
Calculate() throws ArithmeticException	**Calculate()throws Exception**	Not Allowed. **Reason:** Child class method cannot throw parent exception of Parent class.
Calculate() throws Exception	**Calculate() throws Exception**	Allowed.

153. What is an unreachable catch block error?

- Unreachable catch block error comes when we have multiple catch blocks in exception handling.
- Catch block of Exceptions sequence must follow. Most child Exception classes must be declared first and then its parent. Must be in ascending order.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.sql.Connection;

public class NestedTryCatchDemo {

        public static void main(String[] args) {
        try {
        }catch(ArrayIndexOutOfBoundsException e) {
        }catch(IndexOutOfBoundsException e) {
        }catch(Exception e) {
        }
        }
}
```

Explanation:

If you observe in the above program. We have declared ArrayIndexOutOfBoundsException exception first which is the child class of IndexOutOfBoundsException which is the child class of Exception.

154. Does the finally block get executed if the try or catch block has a return statement? How the control flows in this situation?

- Yes. finally{} block will be executed even though you have a return statement in either try block or catch block.
- finally{} block will be executed prior to the return statement execution.

- finally{} block will not be executed only in one condition. Once you invoke System.exit(0) your JVM will be crashed and the finally{} block will not be executed.

155.
What is a re-throwing exception in java?

- Exception throws from try block and catches it in the catch block. A caught exception that can be thrown from the catch block is called re-throwing the exception.
- Re-thrown exception must handle somewhere in the program otherwise your program will be terminated abruptly.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class NestedTryCatchDemo {

        public static void main(String[] args) {
        try {
        int no = 10 / 0;
        }catch(Exception e) {
        throw new NullPointerException();
        }
        }
}
Output:
Exception in thread "main" java.lang.NullPointerException
at
simplifiedjava.crackedInterview.NestedTryCatchDemo.main(NestedTryCatchDemo.java:10)
```

156.
What is the difference between NoSuchMethodError and NoSuchMethodException?

- NoSuchMethodException is thrown when you try and get a method that doesn't exist with the reflection.
- NoSuchMethodError is thrown when the virtual machine cannot find the method you are trying to call. This happens when a particular method was present at compile time but not available at run time.

157.
What is an AutoClosable resource?

- The resource is an object that must be closed after finishing the program.
- AutoClosable statement ensures that the resource is closed at the end of the statement execution.
- Please refer below example.

```
package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class AutoClosableResourceDemo {

        public static void main(String[] args) {
        Scanner scanner = null;
        try {
            scanner = new Scanner(new File("Resume.doc"));
            while (scanner.hasNext()) {
                System.out.println(scanner.nextLine());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            if (scanner != null) {
                scanner.close();
            }
        }
        }
}
```

Interview Questions on Multi-Threading

158. What is a Thread and what is the process?

- **Thread**: Thread is a single independent path of execution of the program.
- **Process**: Process is a program in execution.

159.
What are the advantages of Multithreading?

- Multithreading in java is a process of executing multiple threads simultaneously.
- Advantages of multithreading.

1. Perform multiple operations at a time.
2. The Performance will be improved if multiple threads are executing the shared resources.
3. It saves time if multiple threads are executing at a time.

160.
What are the multiple ways to create a thread? Which one is recommendable?

- There are two ways to create a thread.

1. **Thread Class**: You can extend the Thread class and create a thread.
2. **Runnable Interface**: You can implement a Runnable interface and create a thread.

- Second one is recommendable which is implementing a Runnable interface because if you implement a runnable interface then you will have an option to extends one more class.

- Please refer to the below example.

1. Extending Thread Class.

```
package simplifiedjava.crackedInterview;
public class CreatingThreadUsingThreadClassDemo extends Thread{

        Thread t = new Thread();
        public static void main(String[] args) {
        // TODO Auto-generated method stub
        }
}
```

2. **Implementing Runnable Interface.**

```
package simplifiedjava.crackedInterview;
public class CreatingThreadUsingThreadClassDemo implements Runnable{
        Thread t = new Thread();
        public static void main(String[] args) {
        // TODO Auto-generated method stub
        }

        @Override
        public void run() {
        // TODO Auto-generated method stub
        }
}
```

161.
Explain the thread life cycle?

- There are five states of the thread life cycle.

1. **New:** When a thread gets newly created.
2. **Runnable:** When you invoke the start() method then the thread enters into a runnable state.
3. **Running:** When the scheduler allocates time slots in CPU then that thread enters into a running state.
4. **Blocked:** When you invoke the wait() method then that thread enters into a waiting state.
5. **Terminate:** after execution of the thread enters into a dead state.

162.
What is the daemon thread?

- Daemon thread is a very lightweight thread that is running in the background.
- You may invoke the setDaemon() method to convert normal thread to daemon thread.
- isDaemon() method can be used to check whether the particular thread is daemon or not.
- When all non-daemon thread completes then the daemon thread terminates automatically.
- For example, the Garbage Collector thread is a daemon thread.

163.
What do you mean by thread priority?

- Thread priorities are nothing but a number between 1 to 10.
- Each thread has some priority.
- Default priority is 5.
- Scheduler schedules the thread according to their priority.

- There are 3 constants defined in thread priorities.

  1. MIN_PRIORITY
  2. NORM_PRIORITY
  3. MAX_PRIORITY

164.
Why wait(), notify() and notifyAll() methods are in Object class even these methods are used in multithreading?

- Basically, wait(), notify(), notifyAll() methods are using by threads to communicate with each other. But these methods belong to java.lang.Object class.
- Every Object has its lock or Monitor.
- One thread can hold only one lock or monitor at a time while executing a synchronized block or method.
- If wait(), notify() and notifyAll() methods are put in Thread class instead of Object class then we would have faced Thread communication problem, Synchronization on object won't be possible. If each thread will have a monitor we won't have any way of achieving synchronization.

165.
What is the reason to override the run method in multithreading? What will happen if I don't override the run method?

- Thread class has run() which doesn't have any implementation so you need to override the run() method and implement your logic into it.
- Compiler won't flash any error but nothing will be printed or executed.

166.
How to achieve Thread safety in java?

- Thread safety means multiple threads sharing the same resource without producing an unpredictable result.
- We can achieve thread-safety in multiple ways. Please refer to the below 8 options.

  1. Synchronized Methods.
  2. Synchronized Blocks.
  3. Synchronized Collection. (Using Collections Utility Class).
  4. Concurrent Collection.
  5. Immutable Implementation.
  6. Thread-Local Field.
  7. Volatile Field.
  8. Stateless Implementation.

Explanation in details.

1. **Synchronized Methods:**

   ➢ The method can be declared with a **synchronized** keyword.
   ➢ Once you declared the method with a **synchronized** keyword then only one thread can execute shared resources at a time. Other threads will be blocked until the first thread finishes its execution.
   ➢ **synchronized** method depends on either "Intrinsic Lock" or "Monitor Lock".

- ➢ When the thread calls the **synchronized** method, it acquires the intrinsic lock.
- ➢ The Monitor is just a reference to the role that the lock performs on the associated objects.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class SynchronizedMethodDemo {

		int counter = 0;
		public synchronized void getCounter() {
		counter =+ 1; // Similar to counter = counter +1;
		System.out.println("Counter is "+ counter);
		}
		public static void main(String[] args) {
		new SynchronizedMethodDemo().getCounter();
		}
}
```

2. Synchronized Block:

- ➢ A **synchronized** block can be called a statement block.
- ➢ Method synchronization is highly expensive so to avoid this situation we can perform synchronization at the statement level which is less expensive from a performance perspective.
- ➢ A **synchronized** block generally declares inside a method.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class SynchronizedBlockDemo {
		int counter = 0;
		public void getCounter() {
		synchronized(this) {
		counter =+ 1; // Similar to counter = counter +1;
		}
		System.out.println("Counter is "+ counter);
		}
		public static void main(String[] args) {
		new SynchronizedMethodDemo().getCounter();
		}
}
```

3. Synchronized Collection:

- ➢ We can create a thread-safe collection by using utility methods of the Collections utility class.
- ➢ Once you create the thread-safe collection object then only one thread can execute at a time. While other threads will be blocked until the collection object is unblocked by the first thread.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class SynchronizedCollectionDemo {

		public static void main(String[] args) {
		Thread t1 = new Thread(new MyClass1());
		Thread t2 = new Thread(new MyClass2());
		t1.start();
		t2.start();
		}
}

class MyClass1 implements Runnable{

		@Override
		public void run() {
		List<Integer> list1 = Arrays.asList(10,20,30,40,50);
		List<Integer> synchronizedList1 = Collections.synchronizedList(list1);
		for(Integer i : list1) {
		System.out.println(i);
		}
		}
}

class MyClass2 implements Runnable{

		@Override
		public void run() {
		List<Integer> list2 = Arrays.asList(60,70,80,90,100);
		List<Integer> synchronizedList2 = Collections.synchronizedList(list2);
		for(Integer i : list2) {
		System.out.println(i);
		}
		}
}
```

4. Concurrent Collection:

- Java provides java.util.concurrent package for thread safety.
- Java.util.concurrent package has some thread safe classes like ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentCollectionDemo {

		public static void main(String[] args) {
		Map<String,Integer> concurrentMap = new ConcurrentHashMap<>();
		concurrentMap.put("one", 1);
		concurrentMap.put("two", 2);
		concurrentMap.put("three", 3);
```

```
            }
}
```

5. Immutable Implementation:

- ➢ Immutable means its internal states can’t be changed once it is created.
- ➢ If we need to share the state between different threads, we can create thread-safe classes by making them immutable classes.
- ➢ If the state will be fixed forever then immutable objects can be used thread-safely.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public final class ImmutableClassDemo {

            private final String empCode="E001";

            public String getEmpCode() {
            return empCode;
            }
}
```

6. Thread-Local Field:

- ➢ We can easily create classes whose fields are thread-local by simply defining private fields in the Thread class.
- ➢ We can create thread-safe classes that don’t share states between threads by making their fields thread-local.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Arrays;
import java.util.List;

public class ThreadLocalDemo1 extends Thread {

            private final List<Integer> numList = Arrays.asList(10,20,30,40,50);
            @Override
            public void run() {
            for(int n : numList) {
            System.out.println(n);
            }
            }
}

package simplifiedjava.crackedInterview;

import java.util.Arrays;
import java.util.List;

public class ThreadLocalDemo2 extends Thread{

            private final List<String> wordList = Arrays.asList("Hi","Hello","GM","GA","GN");
            @Override
            public void run() {
            for(String str : wordList) {
            System.out.println(str);
```

```
        }
        }
}
```

7. **Volatile Field:**

- ➢ If you declare a variable as volatile, JVM will store those variables on main memory.
- ➢ We can make sure every time JVM reads the value from the main memory.
- ➢ Every time JVM writes the value it will write on the main memory.
- ➢ Volatile keyword ensures that variable will be visible to all thread and thread can read the value from main memory.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class ThreadLocalVariableDemo {

            private volatile int empId;
            private String name;
            public static void main(String[] args) {
            }
}
```

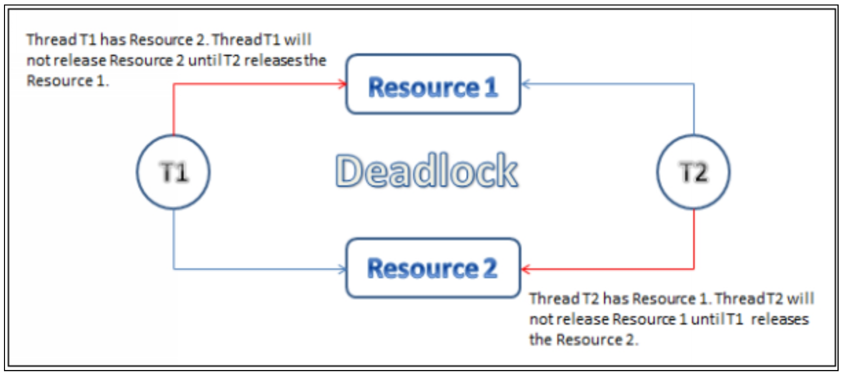
8. **Stateless Implementation:**

- ➢ Stateless implementation is the simplest way to achieve thread safety.
- ➢ Stateless implementation is nothing but function implementation which produces the same result every time. There is no change to change the state by any thread.
- ➢ Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class StateLessImplementationDemo {

            public static void main(String[] args) {
            int i = 10;
            int square = i * i ;
            System.out.println(square);
            }
}
```

167.
What is deadlock? How to avoid a situation of dead lock?

- Deadlocks are a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process.


Thread T1 has Resource 2. Thread T1 will not release Resource 2 until T2 releases the Resource 1.
Resource 1
T1
Deadlock
T2
Resource 2
Thread T2 has Resource 1. Thread T2 will not release Resource 1 until T1 releases the Resource 2.

- We can avoid the deadlock: Do not grant a resource request if this allocation has the potential to lead to deadlock.

168.
What happen if you don't override run() method on Thread class?

- If you don't override the run() method it will execute the run method of the Thread class that has empty implementation. The compiler won't throw any exception.
- It is recommended to override the run() method and provide your implementation.

169.
What is the difference between sleep() and wait() in java?

	Sleep()	**Wait()**
1.	The sleep () method can be used to pause the process for some time or the time we want to.	Wait() is a method that can be used to put the thread in a waiting state.
2.	Sleep() method never release the lock while sleeping the thread.	Wait() method can release the lock while a thread is waiting.

3.	Syntax:
	
	synchronized(LOCK) {
	  Thread.*sleep*(1000);
	}
		Syntax:
		
		synchronized(LOCK) {
		  LOCK.wait();
		}

170.
There are 3 Thread Thread1, Thread2 and Thread3 how would you ensure the thread execution for Thread1, Thread2 and then Thread3?

- If a thread wants to wait until completing some other thread then we should go for the join(); method.
- In above example, Thread1.join(Thread2) and Thread2.join(Thread3) in the above case Thread3 will execute first and then thread2 and then thread1.

171.
What is the difference between Runnable and Callable?

	Runnable	Callable
1.	The runnable interface has the run() method which needs to override an implemented class.	The callable interface has a call() method which needs to override an implemented class.
2.	Run() method cannot return a value.	Call() method can return a value.
3.	Runnable cannot throw checked Exception.	Callable can throw checked exception.

4.	Runnable cannot be used in Executor Framework.	Callable can use in Executor Framework.

5.	**public interface** Runnable {
	    void run();
	}
		public interface Callable<V> {
		    V call() **throws** Exception;
		}

172.
What is static synchronization?

- Static synchronized method will acquire a lock of the class instead of an object.
- Static belongs to the class, not the object so Thread acquires a lock from class.
- Suppose there are multiple static synchronized methods (m1, m2, m3, m4) in a class, and suppose one thread is accessing m1, then no other thread at the same time can access any other static synchronized methods.

173.
What is the difference between Synchronous and Asynchronous programming?

- Implies that tasks will be executed one by one. The next task is started only after the current task is finished. Task3 is not started until task2 is finished.
- Implies that task returns control immediately with a promise to execute a code and notify about the result later(e.g. callback, feature). Task 3 is executed even if Task 2 is not finished.
- Please refer to the Real-Time Example.
- **Synchronous**: You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.
- **Asynchronous**: You are in a restaurant with many other people. You order your food. Other people can also order their food, they don't have to wait for your food to be cooked and served to you before they can order. In the kitchen, restaurant workers are continuously cooking, serving, and taking orders. People will get their food served as soon as it is cooked.

174.
What is the difference between submit() and execute() in thread pool?

	Execute()	Submit()
1.	Return type of execute() method is void.	Return type of submit() is Future.
2.	execute() method is declared in Executor Interface.	submit() method is declared in ExecutorService interface.
3.	This method accepts only Runnable Task.	This method accepts Runnable as well as Callable tasks.

- Please refer below examples of each interface.

```
package simplifiedjava.crackedInterview ;
import java.util.concurrent.ExecutorService ;
import java.util.concurrent.Executors;

public class ThreadExecutorServiceUsingRunnableDemo {

        public static void main(String[] args) {

      ExecutorService executorService = xecutors.newSingleThreadExecutor();
      executorService.execute(new Runnable() {
      @Override
      public void run() {
      System.out.println("Executor Service Executed with Runnable...");
      for(int i=0; i<10; i++) {
      System.out.println("Child Thread...");
      }
      }
      });
              executorService.shutdown();
        }
}
```

Using Callable Interface.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ThreadExecutorServiceUsingCallableDemo {

        public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(1);
        Future obj = executorService.submit(new Callable() {

        @Override
        public Object call() throws Exception {
        //System.out.println("Executor Service Executed with Callable...");
        return "Executor Service Executed with Callable...";
        }
        });
        try {
        System.out.println(obj.get());
        } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
        }
        }
}
```

175. What is the difference between CyclicBarrier and CountDownLatch?

- CyclicBarrier and Countdown latch techniques can be used in a multi-threaded environment to manage resources for multiple threads.
- The difference is explained below.

	CountDownLatch	CyclicBarrier
1.	You cannot reuse the instance of CountDownLatch once the count reaches zero and the latch is open. CountDownLatch works like a count-down timer.	CyclicBarrier can be reused by resetting the barrier once the barrier is broken.
2.	CountDownLatch doesn't have any kind of Constructor.	CyclicBarrier has a constructor where Runnable instance can be provided.

3.	CountDownLatch maintains the count of Tasks.	CyclicBarrier maintains the count of Threads.
4.	The only current thread will throw InterruptedException. It will not impact other threads.	If one thread is interrupted while waiting then other threads will throw BrokenBarrierException.

176.
What is race condition?

- In a multithreaded environment when more than one thread tries to access shared resources at the same time.
- Please refer to the below real-life example to understand race conditions.
- You are planning to go to a movie at 5 pm. You inquire about the availability of the tickets at 4 pm. The representative says that they are available. You relax and reach the ticket window 5 minutes before the show. I'm sure you can guess what happens: it's a full house. The problem here was in the duration between the check and the action. You inquired at 4 and acted at 5. In the meantime, someone else grabbed the tickets.

177.
What is Synchronization?

- Synchronization is a technique used in a multi-threading environment to control the access of shared resources on multiple threads.
- For a detailed description please refer to question no. 166.

178.
What is the difference between Synchronization Method and Synchronized Block?

	Synchronization Method	Synchronized Block
1.	A method declared with a Synchronized keyword is called Synchronized Method.	Inside a method, if a chunk of code declared inside block with Synchronized keyword is called a Synchronized block.
2.	Performance is low as compared to Block.	Performance is high as compared to the method.

179.
What is the volatile keyword in java?

- If you declare a variable as volatile, JVM will store those variables on main memory.
- We can make sure every time JVM reads the value from the main memory.
- Every time JVM writes the value it will write on the main memory.
- Volatile keyword ensures that variable will be visible to all thread and thread can read the value from main memory.

180.
What will happen If I directly override the run() method instead of the start() method?

- First of all, when you invoke the start() method on a thread, New thread will create and start executing parallelly.
- Run() method will execute as it is in the same thread.
- Run() method will be treated as a normal overridden method of the Thread class.

181.
What will happen when an exception occurred in a thread?

- Exception behaviour won't change in the case of non-threading or multi-threading programming.
- Once an exception occurs and if it is not handled then your program will be terminated abruptly.
- In the case of a multi-threading environment if an exception occurs then the thread will die.
- If an uncaught exception handler is registered then it will get a callback.
- Nested interface **Thread.UncaughtExceptionHandler** is defined for handlers invoked when a Thread abruptly terminates due to an uncaught exception.
  When a thread is about to terminate due to an uncaught exception the Java Virtual Machine will query the thread for its UncaughtExceptionHandler using **Thread.getUncaughtExceptionHandler()** and will invoke the handler's **uncaughtException()** method, passing the thread and the exception as arguments.

182.
What is readwritelock in java?

- A ReadWriteLock maintains a pair of associated locks.
- One for read-only operations and one for writing.
- It allows various threads to read a specific resource but allows only one to write it at a time.
- The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.
- All ReadWriteLock implementations must guarantee that the memory synchronization effects of write-lock operations (as specified in the Lock interface) also hold with respect to the associated readLock. That is, a thread successfully acquiring the read lock will see all updates made upon the previous release of the write lock.
- **public interface** ReadWriteLock {}.

183.
What is Thread-Local class? What is the purpose to have this class?

- We can easily create classes whose fields are thread-local by simply defining private fields in the Thread class.
- We can create thread-safe classes that don't share states between threads by making their fields thread-local.
- Thread-Local is most commonly used to implement per-request global variables in app servers or servlet containers where each user or client request is handled by a separate thread.
- For example, Spring, Spring Security and JSF each have the concept of a "Context" through which functionality of the framework is accessed. And in each case, that context is implemented both via dependency injection (the clean way) and as a Thread-Local in cases where DI can't or won't be used.
- Please refer the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Arrays;
import java.util.List;

public class ThreadLocalDemo1 extends Thread {

        private final List<Integer> numList = Arrays.asList(10,20,30,40,50);
        @Override
        public void run() {
        for(int n : numList) {
        System.out.println(n);
        }
        }
}

package simplifiedjava.crackedInterview;

import java.util.Arrays;
import java.util.List;

public class ThreadLocalDemo2 extends Thread{

        private final List<String> wordList = Arrays.asList("Hi","Hello","GM","GA","GN");
        @Override
        public void run() {
        for(String str : wordList) {
        System.out.println(str);
        }
        }
}
```

184.
What is the ThreadGroup?

- Thread group represents a set of threads.
- ThreadGroup belongs to **java.lang.ThreadGroup.**
- A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.
- A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

185.
What is a Thread pool?

- Thread pool is a set of threads that resides in a single place.

186.
What is the difference between volatile and atomic variable?

- **Volatile Variable**: Volatile keyword is used in java synchronization technique to achieve thread-safety, basically to overcome the visibility problem. The variables marked as volatile gets stored in main memory rather than CPU cache.
- **Atomic Variable**: Atomic variables are used for performing atomic operations on a single variable instead of a compounded operation, So it solves the synchronization problem (race condition etc.)

187.
What will happen if I am trying to run a thread which is already in running the state?

- Yes, we can't start already running thread. It will throw IllegalThreadStateException at runtime - if the thread was already started.
- For your reference please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class RestartedThreadDemo extends Thread{
        @Override
        public void run() {
        }
        public static void main(String[] args) {
        Thread t = new Thread();
        t.start();
        t.start();
        }
}
```

```
Output:  Exception in thread "main" java.lang.IllegalThreadStateException
         at java.lang.Thread.start(Unknown Source)
         at
simplifiedjava.crackedInterview.RestartedThreadDemo.main(RestartedThreadDemo.java:14)
```

188.
Can we convert a normal thread into a daemon thread once the thread started or not started in both cases?

- Yes, we can convert a normal thread into a daemon thread.
- Daemon thread is a very lightweight thread.
- But, a Normal thread can convert into a daemon thread but it is not possible after the thread is running state. It is possible before the thread is running state.
- Normal thread can convert by invoking setDaemon() thread.

189.
How do you share the data between two threads?

- We should declare such variables as **static** and **volatile**.
- Volatile variables are shared across multiple threads. This means that individual threads won't cache their copy in the thread-local. But every object would have its own copy of the variable so threads may cache value locally.
- We know that static fields are shared across all the objects of the class, and it belongs to the class and not the individual objects. But, for static and non-volatile variables also, threads may cache the variable locally.

190.
What is the difference between notify and notifyAll()?

- Notify() method can be used to notify only one thread at a time which is waiting in a waiting state.
- NotifyAll() method can be used to notify all threads at a time that is waiting in a waiting state.

191.
What is the difference between preemptive scheduling and time slicing?

	Preemptive Scheduling	Time Slicing
1.	Executes highest priority task executes until it highest priority task comes into existence or thread enters into waiting state or dead state.	Task executes for a predefined slice of time and then reenters into the pool of ready tasks. The scheduler will decide which task needs to execute first according to priority.

192.

What is the difference between interrupted() and isInterrupted() method in java?

	Interrupted()	isInterrupted()
1.	interrupted() method is a static method of class thread checks the current thread and clear the interruption "flag". i.e. a second call to interrupted() will return false.	isInterrupted() method is an instance method; it reports the status of the thread on which it is invoked. it does not clear the interruption flag.
2.	Syntax: `if(t.interrupted()) { System.out.println("inside interrupted"); }`	Syntax: `if(t.isInterrupted()) { System.out.println("inside isInterrupted."); }`

193.
Why wait(), notify() and notifyAll() method must called/invoke from synchronized area? What will happen if I invoked outside synchronized block?

- Threads uses wait(), notify() and notifyAll() method for inter-thread communication.
- Thread can get a lock either synchronized method or synchronized block. Thread must get a lock to call wait(), notify() and notifyAll() method.
- If you invoke these methods outside the synchronized block or method then you will get IllegalMonitorStateException.

194.
What is context switching?

- Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system.
- When context switching performed in the system, it stores the old running process's status in the form of registers and assigns the CPU to a new process to execute its tasks.
- A context switching helps to share a single CPU across all processes to complete its execution and store the system's tasks status.
- Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time.

195.
What is blocking queue?

- A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
- A BlockingQueue does not accept null elements.
- Implementations throw NullPointerException on attempts to add, put or offer a null.
- A BlockingQueue may be capacity bounded.
- At any given time it may have a remaining Capacity beyond which no additional elements can be put without blocking. A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX_VALUE.
- BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface.
- BlockingQueue implementations are thread-safe.

196.
How would you check whether Thread holds a lock of object or not?

- You can identify the lock on the particular object by calling wait() or notify() method on that object. If the object does not hold the lock, then it will throw

llegalMonitorStateException.
- By calling holdsLock(Object o) method. This will return the boolean value.

197.
Which JVM Parameter is used to control stack size of thread?

- **Parameter**: -**Xssn** represents Stack Size;
- Every thread that is spawned during the execution of the program passed to java has n as its C stack size.

198.
What is the purpose of join() method?

- Thread class provides the join() method which allows one thread to wait until another thread completes its execution.
- If a thread wants to wait until completing some other thread then we should go for the join(); method.
- In above example, Thread1.join(Thread2) and Thread2.join(Thread3) in the above case Thread3 will execute first and then thread2 and then thread1.

199.
What is the difference between Synchronized and ReentrantLock?

	Synchronized	**ReentrantLock**
1.	Synchronized does not support fairness. Synchronized never offer chance longest waiting thread.	ReentrantLock support fairness. ReentrantLock has fairness property which provides a lock to the longest waiting thread.
2.	Synchronized doesn't provide any method like tryLock().	ReentrantLock provides the tryLock() method which acquires a lock only if it's available or not held by any other thread.
3.	In a Synchronized mechanism, a thread can be blocked for an infinite period.	In ReentrantLock thread cannot be blocked for infinite time because ReentrantLock provides a lockInterruptibly()method which can be used to interrupt the thread.

200.
Explain the busy spin technique?

- Busy-waiting or spinning is a technique in which a process repeatedly checks to see if a condition is true instead of calling the wait or sleep method and releasing the CPU.
- It is mainly useful in the multicore processor where the condition is going to be true quite quickly i.e. in a millisecond or microsecond.
- The advantage of not releasing CPU is that all cached data and instruction remain unaffected, which may be lost, had this thread is suspended on one core and brought back to another thread.

201.
What is Semaphore in Java? What are the types of Semaphore?

- A Semaphore is a variable that can be used in the Synchronization process.
- A Semaphore controls access to shared resources through the use of the Counter.
- If the counter is zero then access will be denied.
- If the counter is greater than zero then access will be allowed.
- A semaphore maintains the set of permits.
- **Types of Semaphores** are below.

1. **Binary semaphore:** A binary semaphore only takes only 0 and 1 as values and is used to implement mutual exclusion as well as synchronize concurrent processes.
2. **Counting semaphore:** The value of a counting semaphore at any point indicates the maximum number of processes that can enter the critical section at the same time.
3. **Bounded Semaphores:** Bounded semaphores can be used to set the upper bound limit. The Upper bound value denotes how many signals it can store. Counting semaphores do not contain any upper bound value so bounded semaphores can be used in a place of counting semaphores.
4. **Timed Semaphores:** The timed semaphores allow a thread to run for a specified period of time. After a particular time, the timer resets and releases all other permits.

202.
What is the difference between Deadlock and Starvation.

	Deadlock	Starvation
1.	Deadlock is a situation that occurs when one of the processes got blocked.	When all the low priority processes got blocked, while the high priority processes execute then this situation is termed as Starvation.
2.	There is starvation in every deadlock.	Not every starvation needs to be a deadlock.
3.	Deadlock is an infinite process.	Starvation is a long waiting but it is not an infinite process.

203.
What is Thread Starvation?

- Starvation is a situation where the thread is in a waiting state for a very long time because it is not getting access to shared resources because higher priority threads are coming.
- When all the low priority processes got blocked, while the high priority processes execute then this situation is termed Starvation.
- This happens when shared resources are made unavailable for long periods by "greedy" threads.

204.
What is the use of ExecutorService interface?

- The Java ExecutorService is the interface that allows us to execute tasks on threads asynchronously.
- The Java ExecutorService interface is present in the java.util.Concurrent package.
- Executors execute() method takes a runnable object and performs the task asynchronously.
- After making the call of execute() method we call the shutdown() method.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceDemo {

        public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
    executorService.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println("ExecutorService");
            }
        });
         executorService.shutdown();
    }
}
Ouput: ExecutorService
```

- Executors submit() method takes a runnable object and returns a Future object. Later on, this object can be used to check the status of Runnable whether it has completed execution or not.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;


public class ExecutorServiceDemo {

            public static void main(String[] args) {
             ExecutorService executorService = Executors.newSingleThreadExecutor();
                Future future = executorService.submit(new Runnable() {

                     @Override
                     public void run() {
                         //System.out.println("ExecutorService");
                     }
              });
                System.out.println(future.isDone());
    }
}
Output: false
```

- Executors invokeAny() method takes a collection of callable objects. This method returns the future object of the callable objects which are executed first successfully.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ExecutorServiceDemo {

            public static void main(String[] args) throws InterruptedException, Exception {
            // invoke method demo
            ExecutorService executorService = Executors.newSingleThreadExecutor();
            Set<Callable<String>> callableSet = new HashSet<Callable<String>>();
            callableSet.add(new Callable<String>() {
            public String call() throws Exception{
            return "Job 1";
            }
            });
            callableSet.add(new Callable<String>() {
            public String call()throws Exception{
            return "Job 2";
            }
            });
            callableSet.add(new Callable<String>() {
            public String call()throws Exception{
            return "Job 3";
            }
            });
            String executedJob = executorService.invokeAny(callableSet);
            System.out.println(executedJob);
            executorService.shutdown();
    }
}
Output: Job 2
```

- The invokeAll() method takes in a Collection of Callable objects having tasks and returns a list of Future objects containing the result of all the tasks.

```
package simplifiedjava.crackedInterview;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ExecutorServiceDemo {

            public static void main(String[] args) throws InterruptedException, Exception {
            // invoke method demo
            ExecutorService executorService = Executors.newSingleThreadExecutor();
            Set<Callable<String>> callableSet = new HashSet<Callable<String>>();
            callableSet.add(new Callable<String>() {
            public String call() throws Exception{
            return "Job 1";
            }
            });
            callableSet.add(new Callable<String>() {
            public String call()throws Exception{
            return "Job 2";
            }
            });
            callableSet.add(new Callable<String>() {
            public String call()throws Exception{
```

```
            return "Job 3";
            }
            });
            List<Future<String>> executedJob = executorService.invokeAll(callableSet);
            for(Future future: executedJob) {
            System.out.println(future.get());
            }
            executorService.shutdown();
    }
}
Output:
Job 2
Job 3
Job 1
```

205.
What is the role of Future interface in Multithreading?

- A Future represents the result of an asynchronous computation.
- Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.
- The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready.
- Cancellation is performed by the cancel method.
- Additional methods are provided to determine if the task was completed normally or was cancelled.
- Once a computation has been completed, the computation cannot be cancelled.
- If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.

Interview Questions on Serialization and Externalization

206.
Can you tell me what are the Serializable and Externalizable?

- Serializable and Externalizable are the Interfaces.

- Serializable interface Externalizable interface belongs to java.io. package.

207.
What are the advantages and disadvantages of Serialization?

- **Advantages of Serialization.**

    1. Convert the object onto the byte stream and transfer it through the network.
    2. A converted byte stream can save into a file or database.
    3. Third-party services do not require implementing serialization.
    4. Serialization allows java to perform Encryption, decryption, Authentication etc.

- **Disadvantages of Serialization.**

    1. Serialization is default serialization so unnecessary we have to implement full serialization.
    2. Serialization is inefficient when it comes to memory utilization.
    3. Sometimes the byte stream does not convert into objects completely which leads to error.

208.
What is Serialization? Explain the need of serialization?

- Serialization is a process of converting the state of object into byte stream.
- Serialization takes care by JVM.
- In Serialization it's mandatory to save total object.
- Need for the serialization is as follow:

    1. Transfer the state of object through the network.
    2. Can save the state of object into file.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Student implements Serializable {

        private int id;
        private static String name;
        private String dept;
        public Student(int id, String name, String dept) {
        super();
        this.id = id;
        this.name = name;
        this.dept = dept;
        }
        public int getId() {
        return id;
        }
        public void setId(int id) {
        this.id = id;
        }

        public String getName() {
        return name;
```

```
		}
		public void setName(String name) {
		this.name = name;
		}
		public String getdept() {
		return dept;
		}
		public void setdept(String dept) {
		this.dept = dept;
		}
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationDemo {

		public static void main(String[] args) throws IOException, ClassNotFoundException {
		Student s1 = new Student(100,"Yogesh","IT");
		File file1 = new File("stud1.ser");
		FileOutputStream fos = new FileOutputStream(file1);
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(s1);
		File file2 = new File("stud1.ser");
		FileInputStream fis = new FileInputStream(file2);
		ObjectInputStream ois = new ObjectInputStream(fis);
		Student s2 = (Student) ois.readObject();
		System.out.println("Student ID :" + s2.getId());
		System.out.println("Student Name :"+ s2.getName());
		System.out.println("Student Dept :"+ s2.getdept());
		}
}
Output:
Student ID :100
Student Name :Yogesh
Student Dept :IT
```

209.
What is deserialization?

- Deserialization is the reverse process of serialization.
- The reverse process of creating the object from a stored sequence of bytes.
- Please refer to the above example for deserialization.

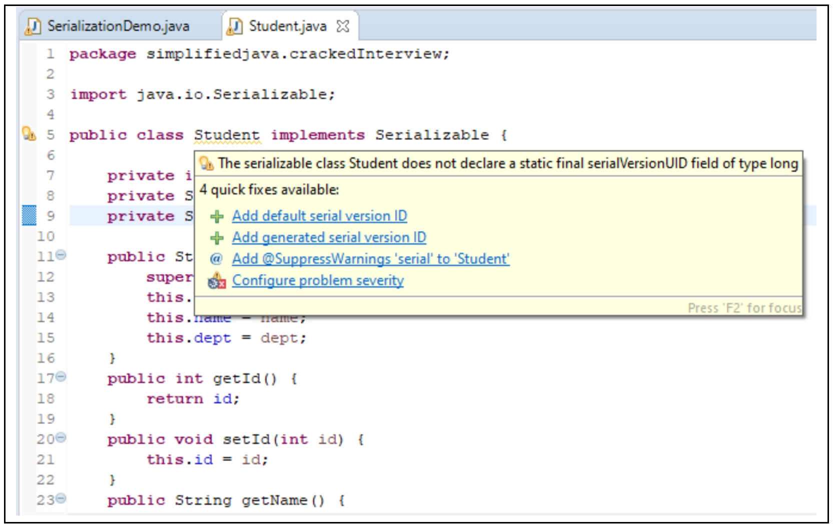
210.
What are the methods used for Serialization and Deserialization?

- For Serialization we have to use **writeObject()** method which belongs to ObjectOutputStream.

- For Deserialization we have to use **readObject()** method which belongs to ObjectInputStream.

211. What is SerialVersionUID? Who provides this SerialVersionUID? How will you create it?

- SerialVersionUID is a unique identifier of the class.
- SerialVersionUID represent the version of the class. This comes into the picture while serialization and deserialization.
- JVM uses it to compare the versions of the class ensuring that the same class was used during Serialization is loaded during Deserialization.
- the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization.
- You can refer the below image to generate the SerialVerionUID.



```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Student implements Serializable {

    private i
    private S
    private S

    public St
        super
        this.
        this.name = name;
        this.dept = dept;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
```

The serializable class Student does not declare a static final serialVersionUID field of type long

4 quick fixes available:

- Add default serial version ID
- Add generated serial version ID
- Add @SuppressWarnings 'serial' to 'Student'
- Configure problem severity

Press 'F2' for focus

212. What is the Externalization? What is the role of Externalizable interface?

- Externalization is customizing serialization.
- In Serialization, it's always possible to save the total object to the file and it's not possible to save part of the object which may create a performance issue. To cover come to this issue we should go for externalization.
- In externalization, everything takes care of by the programmer and JVM doesn't have any control over it.
- Please refer the blelow example.

```
package simplifiedjava.crackedInterview;

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

public class Account implements Externalizable{

        long accountId;
        String userName;
        int pin;
        int token;
        public Account() {
        super();
        }
        public Account(long accountId, String userName, int pin, int token) {
        super();
        this.accountId = accountId;
        this.userName = userName;
        this.pin = pin;
        this.token = token;
        }

        @Override
        public void writeExternal(ObjectOutput oos) throws IOException {
        oos.writeLong(accountId);
        oos.writeObject(userName);
        oos.writeInt(pin);
        }
        @Override
        public void readExternal(ObjectInput ois) throws IOException, ClassNotFoundException {
        long accountIdV1 = (long)ois.readLong();
        String userNameV1 = (String)ois.readObject();
        int pinV1 = (int)ois.readInt();
        }
}
package simplifiedjava.crackedInterview;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ExternalizationDemo {

        public static void main(String[] args) throws IOException, ClassNotFoundException {
        Account account = new Account(1001L,"Yogesh Sanas",5555,9800);
        FileOutputStream fos = new FileOutputStream("acc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(account);
        FileInputStream fis = new FileInputStream("acc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account accountV1 = (Account)ois.readObject();
        System.out.println(accountV1.accountId);
        System.out.println(accountV1.userName);
        System.out.println(accountV1.pin);
        System.out.println(accountV1.token);

        System.out.println(account);
        }
}
```

213.
What are the methods used for Externalization?

- There are couple of methods which are used in Externalization Interface.

1. **writeExternal():**This method can be used to write the custom object for serialization.
2. **readExternal():**This method can be used to read the custom object either from the network or file.

214.
What is marker interface? Can you tell me the examples of marker interface?

- Empty interfaces are called marker interfaces.
- Marker interfaces don't have any method, variable or constant.
- Marker interfaces are used to inform the JVM that a class implementing this interface will have some special behaviour.
- There are some marker interfaces in java which are as follows.

1. Serializable Interface.
2. Clonable Interface.
3. Remote Interface.
4. ActionListener Interface.

215.
What is the difference between Serialization and Externalization?

	Serialization	Externalization
1.	Serialization is default Serialization.	Externalization is customizing Serialization.
2.	Everything takes care of by JVM in serialization.	Everything takes care of by the Programmer in Externalization.
3.	Serialization is not capable to save the required part of the object.	It's also possible to save part of the object or full object in Externalization.
4.	Performance is low as compared to Externalization.	Performance is high as compared to serialization.
5.	If you want to serialize a full object then serialization is a good option.	If you want to serialize a partial object then externalization is a good option.
6.	For serialization, we have to implement a Serializable interface.	For externalization, we have to implement an Externalizable interface.
7.	The transient keyword is required if you don't want to make a particular field serialize.	The transient keyword is not required because this is customizing serialization.
8.	A No-arg constructor is not required.	A No-arg constructor is required at the time of deserialization otherwise we will get InvalidClassException.

216.
What are the steps for Serialization, Say suppose I want to serialize Employee

Object?

- Create Employee class.
- Implement a Serializable interface.
- Define some fields.
- Provide getters and setter methods.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Student implements Serializable {

        private int id;
        private static String name;
        private String dept;
        public Student(int id, String name, String dept) {
        super();
        this.id = id;
        this.name = name;
        this.dept = dept;
        }
        public int getId() {
        return id;
        }
        public void setId(int id) {
        this.id = id;
        }
        public String getName() {
        return name;
        }
        public void setName(String name) {
        this.name = name;
        }
        public String getdept() {
        return dept;
        }
        public void setdept(String dept) {
        this.dept = dept;
        }
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationDemo {

        public static void main(String[] args) throws IOException, ClassNotFoundException {
        Student s1 = new Student(100,"Yogesh","IT");
        File file1 = new File("stud1.ser");
        FileOutputStream fos = new FileOutputStream(file1);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(s1);
        File file2 = new File("stud1.ser");
        FileInputStream fis = new FileInputStream(file2);
        ObjectInputStream ois = new ObjectInputStream(fis);
```

```
          Student s2 = (Student) ois.readObject();
          System.out.println("Student ID :" + s2.getId());
          System.out.println("Student Name :"+ s2.getName());
          System.out.println("Student Dept :"+ s2.getdept());
          }
}
Output:
Student ID :100
Student Name :Yogesh
Student Dept :IT
```

217.
Can you serialize primitive data types?

- Yes, All primitives data types are part of Serialization.

218.
Can you serialize private and static variables?

- **Private Variable:** We can serialize private variables. There is no concern at all whether a variable is private, public or protected. It is simply used for serialization because it's a part of an object.
- **Static Variable:** Static variables belong to a class, not an object. So static variables are not a part of the object state. So static variables cannot consider for serialization.

219.
How can you prevent particular variable to be serialized?

- Transient keyword can be used to prevent a particular variable to be serialized.
- You can declare the variable as static. Static variables are not considered for serialization.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Student implements Serializable {

          private static final long serialVersionUID = 1L;
          private int id;
          private transient String name;
          private String dept;
          public Student(int id, String name, String dept) {
          super();
          this.id = id;
          this.name = name;
          this.dept = dept;
          }
          public int getId() {
          return id;
          }
          public void setId(int id) {
          this.id = id;
          }
          public String getName() {
          return name;
          }
```

```
            public void setName(String name) {
            this.name = name;
            }
            public String getdept() {
            return dept;
            }
            public void setdept(String dept) {
            this.dept = dept;
            }
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationDemo {

            public static void main(String[] args) throws IOException, ClassNotFoundException {
            Student s1 = new Student(100,"Yogesh","IT");
            File file1 = new File("stud1.ser");
            FileOutputStream fos = new FileOutputStream(file1);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(s1);
            File file2 = new File("stud1.ser");
            FileInputStream fis = new FileInputStream(file2);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Student s2 = (Student) ois.readObject();
            System.out.println("Student ID :" + s2.getId());
            System.out.println("Student Name :"+ s2.getName());
            System.out.println("Student Dept :"+ s2.getdept());
            }
}
Output:
Student ID :100
Student Name :null – This is not considered for serialization.
Student Dept :IT
```

220.
Can I apply transient keyword to primitive, static and reference variables?

- **Primitive Variables:** transient keyword is applicable to primitive variables.
- **Static Variables:** technically you can declare static variable as a transient but, it doesn't make a sense to make static variable as a transient because, transient key word is used when you don't want to consider a particular field for serialization. Static variables by default not considered for serialization.
- **Reference variable**: Reference variable can be null and this variable doesn't have any value it has an address. So transient keyword is not applicable to reference variables.

221. If your child class is Serializable and parent class is not and child class is inheriting some properties of parent class, in that case can you serialize child class?

- Yes, In that case, we can serialize the child class object.
- But while serialization, JVM will ignore the original value of parent class instance variables and save the default value to file.
- At the time of de-serialization, if any non-serializable superclass is present then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke the default(no-arg) constructor of that class. So every non-serializable superclass must necessarily contain a default constructor, otherwise, we will get InvalidClassException.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Parent {

            public int i;
            public Parent() {
            System.out.println("Parent Class no-arg constructor called.");
            }
            public Parent(int i) {
            this.i = i;
            }
            public void check(){
            System.out.println("Parent Class check() method");
            }
}

package simplifiedjava.crackedInterview;
import java.io.Serializable;

public class Child extends Parent implements Serializable{
            int j;
            public Child() {
            }
            public Child(int i,int j) {
            super(i);
            this.j = j;
            }
            public void check(){
            System.out.println("Addition in Child Class "+ (i + j));
            }
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationInHeritanceDemo {
            public static void main(String[] args) throws IOException, ClassNotFoundException {
            Child c1 = new Child(10,20);
            System.out.println(c1.i +" and "+ c1.j);
            FileOutputStream fos = new FileOutputStream(new File("abc.ser"));
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(c1);
            FileInputStream fis = new FileInputStream("abc.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            Child c2 =(Child) ois.readObject();
            System.out.println(c2.i +" and "+ c2.j);
            }
}
Output:
Values Before Serialization : 10 and 20
Parent Class no-arg constructor called.
Values After Deserialization : 0 and 20
```

222.
How will you prevent to be serialized child class when its parent class is Serializable?

- There is no direct way to prevent child class from serialization. You can override writeObject() and readObject() methods in your child class and then explicitly throw NotSerializableException. It will prevent child class from serialization.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Parent implements Serializable{

        public int i;
        public Parent() {
        System.out.println("Parent Class no-arg constructor called.");
        }
        public Parent(int i) {
        this.i = i;
        }
        public void check(){
        System.out.println("Parent Class check() method");
        }
}
package simplifiedjava.crackedInterview;

import java.io.NotSerializableException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Child extends Parent{
        int j;
        public Child() {
        }
        public Child(int i,int j) {
        super(i);
        this.j = j;
        }
        public void check(){
        System.out.println("Addition in Child Class "+ (i + j));
        }
        private void writeObject(ObjectOutputStream oos)throws NotSerializableException {
        throw new NotSerializableException();
        }
        private void readObject(ObjectInputStream ois) throws NotSerializableException {
        throw new NotSerializableException();
        }
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationInHeritanceDemo {
        public static void main(String[] args) throws IOException, ClassNotFoundException {
        Child c1 = new Child(10,20);
        System.out.println("Values Before Serialization : " + c1.i +" and "+ c1.j);
        FileOutputStream fos = new FileOutputStream(new File("abc.ser"));
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(c1);
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Child c2 =(Child) ois.readObject();
        System.out.println("Values After Deserialization : "+ c2.i +" and "+ c2.j);
        }
}
```

```
Values Before Serialization : 10 and 20
Exception in thread "main" java.io.NotSerializableException
        at simplifiedjava.crackedInterview.Child.writeObject(Child.java:26)
        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
simplifiedjava.crackedInterview.SerializationInHeritanceDemo.main(SerializationInHeritanceDemo.java:19)
```

223.
What is NotSerializableException? When it can be thrown?

- The NotSerializableException is thrown when attempting to serialize or deserialize an object that does not implement the java. io.Serializable interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Parent{

		public int i;
		public Parent() {
		System.out.println("Parent Class no-arg constructor called.");
		}
		public Parent(int i) {
		this.i = i;
		}
		public void check(){
		System.out.println("Parent Class check() method");
		}
}

package simplifiedjava.crackedInterview;

import java.io.NotSerializableException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Child extends Parent{
		int j;
		public Child() {
		}
		public Child(int i,int j) {
		super(i);
		this.j = j;
		}
		public void check(){
		System.out.println("Addition in Child Class "+ (i + j));
		}
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationInHeritanceDemo {

		public static void main(String[] args) throws IOException, ClassNotFoundException {
		Child c1 = new Child(10,20);
		System.out.println("Values Before Serialization : " + c1.i +" and "+ c1.j);
		FileOutputStream fos = new FileOutputStream(new File("abc.ser"));
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(c1);
		FileInputStream fis = new FileInputStream("abc.ser");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Child c2 =(Child) ois.readObject();
		System.out.println("Values After Deserialization : "+ c2.i +" and "+ c2.j);
		}
}

Output:
Values Before Serialization : 10 and 20
Exception in thread "main" java.io.NotSerializableException: simplifiedjava.crackedInterview.Child
		at java.io.ObjectOutputStream.writeObject0(Unknown Source)
		at java.io.ObjectOutputStream.writeObject(Unknown Source)
			at
simplifiedjava.crackedInterview.SerializationInHeritanceDemo.main(SerializationInHeritanceDemo.java:19)
```

224. What are the scenarios where InvalidClassException can be thrown?

- **InvalidClassException can be thrown in following scenarios.**

  1. The serial version of the class does not match that of the class descriptor read from the stream.
  2. The class contains unknown data types.
  3. The parent class does not have an accessible no-arg constructor while performing deserialization.

225. At the time of deserialization why do we must require default constructor in parent class if parent class is not Serializable?

- At the time of de-serialization, if any non-serializable superclass is present then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke the constructor of that class. So every non-serializable superclass must necessarily contain a default constructor, otherwise, we will get InvalidClassException.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

public class Parent{

		public int i;
		public Parent(int i) {
		this.i = i;
		}
		public void check(){
		System.out.println("Parent Class check() method");
		}
}

package simplifiedjava.crackedInterview;

import java.io.NotSerializableException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Child extends Parent implements Serializable{
		int j;
```

```
            public Child(int i,int j) {
            super(i);
            this.j = j;
            }
            public void check(){
            System.out.println("Addition in Child Class "+ (i + j));
            }
}

package simplifiedjava.crackedInterview;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationInHeritanceDemo {

            public static void main(String[] args) throws IOException, ClassNotFoundException {
            Child c1 = new Child(10,20);
            System.out.println("Values Before Serialization : " + c1.i +" and "+ c1.j);
            FileOutputStream fos = new FileOutputStream(new File("abc.ser"));
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(c1);
            FileInputStream fis = new FileInputStream("abc.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            Child c2 =(Child) ois.readObject();
            System.out.println("Values After Deserialization : "+ c2.i +" and "+ c2.j);
            }
}
```

226.
Can we customize serialization process? Can you override default serialization process if yes then how. Can you tell me the steps?

- Yes, We can customize the Serialization.
- We can use Externalization to customize the serialization.
- Steps needs to implement for Externalization.

1. A class must implement an Externalizable interface.
2. We have to override writeExternal(ObjectOutputStream ooo) and readExternal(ObjectOutputStream oos).
3. Inside the main method, we have to create a file object to store the serialised data.
4. Create FileOutputStream object to write the file. We have to pass the file object in the FileOutputStream class constructor.
5. Create ObjectOutputStream object to and pass fileOutputstream object to its constructor.
6. With the ObjectOutputStream object, you can write the data.

227.
Which changes are considered compatible and incompatible in serialization mechanism?

- **Compatible Changes are as follows.**

    1. **Changing access modifier:** If you change public, protected, default, private there will be no effect on serialization.
    2. **Adding New fields:** Adding new fields is also compatible. There is no restriction to adding new fields in serialization.
    3. **Changing the field from static to non-static:** It is kind of adding new fields in serialization.
    4. **Changing the field from transient to non-transient:** If you are removing the transient keyword then that field will be eligible for serialization. It's like adding a new field that is compatible with change.

- **Incompatible Changes are as follows.**

    1. **Changing the field from non-static to static:** It is equivalent to deletion of the filed.
    2. **Changing the field from non-transient to transient:** It is equivalent to deletion of the field.

228.
Can we transfer object through network if serialization mechanism is not available?

- Yes, we can transfer object through network if serialization mechanism is not available.

    1. **JSON:** We can convert the java object into JSON object. JSON object can be transferred through the network.
    2. **XML:** We can convert the java object into XML and then we can transfer through the network.

229.
Can we serialize the class if class has some collection objects like List, Map and Set?

- Yes, we can serialize the collection objects like List, Map and Set.
- All collections object implements a Serializable interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class MyCollection implements Serializable{

        List<String> myList;
        Set<String> mySet;
        Map<Integer,String> myMap;
        public MyCollection(List<String> myList, Set<String> mySet,Map<Integer,String> myMap) {
        this.myList = myList;
        this.mySet = mySet;
        this.myMap = myMap;
        }
```

```
		@Override
		public String toString() {
		return "MyCollection [myList=" + myList + ", mySet=" + mySet + ", myMap=" + myMap + "]";
		}
}

package simplifiedjava.crackedInterview;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class SerializeCollectionObjectsDemo {

		public static void main(String[] args) throws IOException, ClassNotFoundException {
		List<String> myList = new ArrayList<String>();
		Set<String> mySet = new HashSet<String>();
		Map<Integer,String> myMap = new HashMap<Integer,String>();
		myList.add("One");
		myList.add("Two");
		myList.add("Three");
		mySet.add("Four");
		mySet.add("Five");
		mySet.add("Six");
		myMap.put(1, "One");
		myMap.put(2, "Two");
		myMap.put(3, "Three");

		MyCollection collection = new MyCollection(myList, mySet, myMap);
		FileOutputStream fos = new FileOutputStream("coll.ser");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(collection);
		FileInputStream fis = new FileInputStream("coll.ser");
		ObjectInputStream ois = new ObjectInputStream(fis);
		MyCollection updatedCollection = (MyCollection)ois.readObject();
		System.out.println(updatedCollection);
		}
}
Output:  MyCollection [myList=[One, Two, Three], mySet=[Five, Six, Four], myMap={1=One, 2=Two, 3=Three}]
```

230.
Can you Serialize Singleton class and at the time of deserialization object should be in the same state as it was during serialization?

- Yes, we can serialize the singleton class.
- An object returned by deserialization process is in same state as it was during the serialization process.

- We can implement **readResolve()** method to ensure that we don't break singleton pattern during deserialization.
- Signature of readResolve() method.

```
private Object readResolve() throws ObjectStreamException {
      return INSTANCE;
}
```

- You have to define readObject() as well instead of creating new object.

```
private void readObject(ObjectInputStream ois) throws IOException,ClassNotFoundException{
      ois.defaultReadObject();
       synchronized (SingletonClass.class) {
        if (INSTANCE == null) {
               INSTANCE = this;
        }
      }
 }
```

231.
Can you tell me any purpose to Serialize Singleton Class?

- Serialization is one technique to break the singleton design pattern.
- Suppose you serialize the singleton object and then if you de-serialize that object, it will create a new instance, hence breaking the singleton design pattern.

232.
Why java.lang.Object class doesn't implement Serializable interface?

- Serializable is marker interface, which is empty, but when a class is marked Serializable that means its objects are Serializable.
- The reason the java.lang.Object didn't implement Serializable is because, what if you do not want to make certain fields as Serializable and you may missed to add transient to that field, then will be a havoc.
- By making the programmer implement Serializable for his class, it makes awareness among the programmer that he has consciously implemented it, and should take necessary steps to prevent anything to be serialized which should not be.

233.
What are the callback methods? What is the purpose of callback methods? Can you tell me the examples of callback methods?

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Interview Questions on Inner Classes.

234.
What is Inner Class?

- A class declared inside another class is called Inner class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;
```

```
public class Outer {

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

235.
What is the purpose of Inner classes?

- **It is a way of logically grouping classes that are only used in one place:** If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

236.
What are the advantages and disadvantages of Inner class?

- **Advantages:**

  1. An inner class is used to develop maintainable code because they logically group classes and interface in one place.
  2. Easy access as the inner classes is implicitly available in the outer class.
  3. Inner classes represent a special type of relationship which allows access to all members.

- **Disadvantages:**

  1. Code complexity increases.
  2. Multiple inner classes may decrease the performance.

237.
Can we declare a class inside a method?

- Yes, we can declare the class inside a method which is called method local inner class.

238.
Can we declare outer class is private?

- No, the Outer class cannot be private. An outer class should be public.

239.
Can we declare outer class as a Default or Strictfp or Final?

- Yes, all three keywords are applicable to the outer class.

240.
Can we declare inner class as a private?

- Inner class can be declared as a private but cannot declare outer class as a private class.

241.
Can we declare static inner class?

- We can declare an inner class as a static class.

242.
What is the difference between Normal class and Inner class?

	Normal Class / Outer Class	Inner Class
1.	An outer class can be instantiated without instantiating an inner class.	Non-static Inner class cannot be instantiated without instantiating an outer class.
2.	The outer class does not totally depend on the inner class.	An inner class totally depends on the outer class.
3.	An outer class cannot be private.	An inner class can be private.
4.	An outer class cannot be static.	An inner class can be static.
5.	An outer class cannot be protected.	An inner class can be protected.

243.
What is Anonymous Inner Class?

- We can declare inner classes without name such type of inner classes are called Anonymous Inner class.
- Anonymous inner class can use for instant use.
- Anonymous inner class can extend a class and implement an interface.
- Anonymous inner class can be declared as a method argument.
- Please refer to below example for an anonymous inner class.

```
package simplifiedjava.crackedInterview;

public class AnonymousInnerClassDemo {

	public static void main(String[] args) {
	AnonymousInnerClassDemo inner = new AnonymousInnerClassDemo() {

	};
	}
}
```

244.
What is method local inner class?

- We can declare classes inside a method such types of inner classes are called method local inner class.
- Method local inner class can be accessible only inside a method where it is declared.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class MethodLocalInnerClassDemo {
	public static void main(String[] args) {
	MethodLocalInnerClassDemo demo = new MethodLocalInnerClassDemo();
	demo.invokeMethodInnerClass();
	}
	public void invokeMethodInnerClass() {
	class Inner{
	public void sum(int num1, int num2) {
	System.out.println("Addition is "+ (num1+num2));
	}
	}
```

```
            Inner inner = new Inner();
            inner.sum(100, 200);
            }
}
```

245.
What is static nested class?

- We can declare an inner class with a static modifier such types of inner classes are called a static nested class.
- In the case of regular inner class without the existing outer class object there is no chance of existing inner class object i.e. inner class object is strongly associated with an outer class object.
- In the case of static nested classes without the existing outer class object there may be a chance of existing inner class object. Hence, Static nested class object is not strongly associated with an outer class object.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class StaticNestedClassDemo {

            public static void main(String[] args) {
            Nested n = new Nested();
            n.m1();
            }
            static class Nested{
            public void m1() {
            System.out.println("Nested Class.");
            }
            }
}
```

246.
Can we declare constructor inside inner class?

- Yes, we can declare a constructor inside the inner class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Outer {

            public static void main(String[] args) {
            }
            class Inner{
            int a;
            int b;
            public Inner(int a, int b) {
            this.a = a;
            this.b = b;
            }
            }
}
```

247.

Can we declare constructor inside anonymous inner class?

- No, we cannot declare a constructor inside an anonymous inner class.
- Anonymous inner class doesn't have any name and the constructor name should be the same as the class name. If the class name does not exist then how can we create a constructor?

248.
What is the difference between Regular Inner Class and Anonymous Inner Class?

	Regular Inner Class	Anonymous Inner Class
1.	Regular inner class can implement any number of interfaces at a time.	Anonymous inner class can implement only one interface at a time.
2.	Regular inner class can extend only one class and implements multiple interfaces.	Anonymous inner class either can extend a class or implement only one interface.
3.	Regular inner class can have multiple constructors.	Anonymous inner class cannot have any constructor.
4.	If the requirement is not temporary use then go for regular inner class.	If the requirement is temporary use then go for anonymous inner class.

249.
What is the difference between Regular Inner Class and Static Inner Class?

	Regular Inner Class	Static Inner Class
1.	Outer and Inner regular classes are strongly associated with each other.	Outer and nested static classes are not strongly associated with each other.
2.	Static members are not allowed. i.e. static methods and static fields. Still, if you want to declare a static field inside the inner class then you must declare with the final keyword as well and assign some value to it. e.g. **static final int** ***c*** = 10;	Static members are allowed.
3.	We cannot declare main() inside the inner class so we can't run directly through the command prompt.	We can declare the main method inside the inner class so we can call or run directly through the command prompt.

250.
Can we access non-static class members inside static inner class?

- No, non-static class members are not allowed inside static nested or inner class.
- Even it is applicable to the outer class as well. Inside the static area, only static members are allowed.

251.
How will you instantiate the Inner class?

- There are three ways to instantiate an inner class.

```
package simplifiedjava.crackedInterview;

public class Outer {

        static int i = 0;
        class Inner{
        public void m1() {
        System.out.println("Inner class m1() called. Counter = "+ ++i);
        }
        }
}

package simplifiedjava.crackedInterview;

public class InstantiatingInnerClassDemo {

        public static void main(String[] args) {
        // First way to create inner class instance.
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
        i.m1();
        // Second way to create inner class instance.
        Outer.Inner i2 = new Outer().new Inner();
        i2.m1();
        // Third way to create inner class instance.
        new Outer().new Inner().m1();
        }
}

Output:
Inner class m1() called. Counter = 1
Inner class m1() called. Counter = 2
Inner class m1() called. Counter = 3
```

252.
How many .class files will be created if there are 2 inner classes inside outer class?

- Total 3 .class files will be generated after successfully compiled a class which has two inner classes.

253.
Can we access all members of outer class including private in inner class?

- Yes, we can access all members including private members of the outer class in the inner class.

254.
Can you declare local inner class as a static class?

- No, we cannot declare local an inner class as a static class.

255.
Anonymous class means name without a class then how will you create an object without using name?

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class AnonymousClass {
            public void eat() {
             System.out.println("Ear First Time.");
             }
}
package simplifiedjava.crackedInterview;
public class AnnonymousInnerClassDemo {

            public static void main(String[] args) {
            AnonymousClass anonymous = new AnonymousClass() {
            public void eat() {
            System.out.println("Second Time");
            }
            };
            anonymous.eat();
            }
}
Output:  Second Time
```

256.
Can we declare local inner class with Public, Private and Protected?

- No. Public, Private and Protected are not applicable to local inner class.
- Only final, abstract and strictfp are applicable to local inner class.

257.
Can you declare class inside class?

- Without existing one type of object if there is no chance of existing another type of object then we can declare a class inside a class.
- E.g. University can have several departments, without existing university there is no chance of an existing departments. Hence, we have to declare department class inside university class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class University {
            public class Department{
            }
}
```

258.
Can you declare interface inside interface?

- Yes, we can declare interface inside interface.
- We can take the example of Map. A map is a group of key/value pairs.
- Each key/value pair is called Entry.
- Without an existing map object, there is no chance to exist an Entry object. Hence, interface Entry is defined inside Map.
- Please refer to the below example.

```
public interface Map  <K,V> {

            public interface Entry<K,V>{
            }
}
```

259.
Can you declare interface inside class?

- Inside a class, if we require multiple implementations of an interface and all these implementations are related to the class then we can define an interface inside a class.
- Please refer to the below Example.

```
package simplifiedjava.crackedInterview;

public class InterfaceDefinedInsideClassDemo {

        interface Vehicle{
        public void getNoOfWheels();
        }
        class Bus implements Vehicle{
        public void getNoOfWheels() {
        System.out.println("Six Wheels");
        }
        }
        class Pulsor implements Vehicle{
        public void getNoOfWheels() {
        System.out.println("Two Wheeler");
        }
        }
        public static void main(String[] args) {
        InterfaceDefinedInsideClassDemo demo = new InterfaceDefinedInsideClassDemo();
        Vehicle v = demo.new Bus();
        v.getNoOfWheels();

        v = demo.new Pulsor();
        v.getNoOfWheels();
        }
}
Output:
Six Wheels
Two Wheeler
```

260.
Can you declare class inside interface?

- If the functionality of the class is closely associated with the interface then it is highly recommended to declare a class inside the interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public interface EmailService {

        public void sendEmail(EmailDetails e);
        class EmailDetails{
        String toList;
        String ccList;
        String subject;
        String body;
        }
}
```

Interview Questions on Collections.

261.
What are Legacy Classes?

- A legacy class means old classes.
- Earlier versions of Java didn't include the Collection framework.
- From java version 1.2 started using these Legacy classes. These classes were reengineered to support the collection interface. These classes are known as Legacy classes.
- Following are the Legacy classes.

1. Vector.
2. Stack.
3. Dictionary.
4. HashTable.
5. Properties.

262.
What is the purpose of Arrays utility class?

- Array utility class contained in java.util.Arrays package.
- This class provides some utility methods for manipulating the arrays like sorting, searching, converting an array into a collection etc.

263.
What is the purpose of Collections Utility class? What are the utility methods are available in Collections Class?

- Collections is a utility class for the collection framework.
- Collections utility class has some methods which are as follows.

1. addAll()
2. binarySearch()
3. checkedCollection()
4. copy()
5. fill()
6. min()
7. max()
8. replaceAll()
9. reverse()
10. shuffle()
11. synchronizedCollection()
12. unmodifiableCollection()

264.
What is the difference between Collection and Collections?

	Collection	Collections
1.	The Collection is parent Interface.	Collections is a utility class.
2.	It extends the Object class.	It extends the Iterable interface.

265.

Is Collection object by default thread-safe?

- All collections are not thread-safe by default.
- Following collection objects are thread-safe rest are not thread-safe.

1. Vector.
2. HashTable.
3. ConcurrentHashMap.
4. CopyOnWriteArryList.
5. CopyOnWriteArraySet.

266.
How will you convert collection object to thread-safe collection object?

- We have a utility class that has some utility methods to convert non-thread-safe collections to the thread-safe collection.
- Each collection has its utility method to convert non-thread-safe to thread-safe.
- Following are the collections and its **utility methods**.

1. Collection has Collections.synchronizedCollection().
2. Map has Collections.synchronziedMap().
3. SortedMap has Collections.synchronizedSortedMap().
4. NavigableMap has Collections.navigableMap().
5. List has Collections.synchronizedList().
6. Set has Collections.synchronizedSet().
7. NavigableSet has Collections.synchronizedNavigableSet().
8. SortedSet has Collections.synchronizedSortedSet().

267.
How to remove the duplicates from list object without stream API?

- Couple of ways you can remove duplicates from the list.

1. By using HashSet.
2. By using LinkedHashSet.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

public class RemoveDuplicatesFromList {
        public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(10);
        list.add(20);
        list.add(30);
        list.add(30);
        list.add(10);
        System.out.println("----- Using HashSet-----");
```

```
		Set<Integer> listHashSet = new HashSet<Integer>(list);
		for(Integer i : listHashSet) {
		System.out.println(i);
		}
		System.out.println("----- Using LinkedHashSet-----");
		Set<Integer> listLinkedHashSet = new LinkedHashSet<Integer>(list);
		for(Integer i : listLinkedHashSet) {
		System.out.println(i);
		}
		}
}
Output:
----- Using HashSet-----
20
10
30
----- Using LinkedHashSet-----
10
20
30
```

268.
Is it possible to remove the duplicates and arrange all objects in sorting order in single step?

- Yes, It is possible to remove the duplicates and arrange all objects in sorting order using LinkedHashSet.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

public class RemoveDuplicatesFromList {

		public static void main(String[] args) {
		List<Integer> list = new ArrayList<Integer>();
		list.add(10);
		list.add(20);
		list.add(30);
		list.add(30);
		list.add(10);
		System.out.println("----- Using LinkedHashSet-----");
		Set<Integer> listLinkedHashSet = new LinkedHashSet<Integer>(list);
		for(Integer i : listLinkedHashSet) {
		System.out.println(i);
		}
		}
}
Output:
----- Using LinkedHashSet-----
10
20
30
```

269.
Can you make collection object ready only? If yes then how will you make sure your collection object is read only?

- We can make collection read-only by unmodifiableList() method which exists in the Collections utility class.
- If you apply the unmodifiableList() method on any list object then you are not allowed to modify the list object. Still, you try to modify it then you will get UnSupportedOperationException.
- Make sure once you invoked to unmodifiableList() method on the list then you have to assign the same list to the old reference variable.
- If you don't assign then you won't get UnSupportedOperationException and you will be able to modify the list.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class UnmodifiableListDemo {

            public static void main(String[] args) {
            List<Integer> list = new ArrayList<Integer>();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(30);
            list.add(10);

            System.out.println("Before Unmodified "+ list);

            list = Collections.unmodifiableList(list);

            list.remove(2);

            System.out.println("After Unmodified "+ list);
            }
}
Output:
Exception in thread "main" Before Unmodified [10, 20, 30, 30, 10]
java.lang.UnsupportedOperationException
            at java.util.Collections$UnmodifiableList.remove(Unknown Source)
                        at
simplifiedjava.crackedInterview.UnmodifiableListDemo.main(UnmodifiableListDemo.java:21)
```

270.
When you will get UnSupportedOperationException?

- UnSupportedOperationException is a runtime exception.
- When you are trying to modify the unmodifiable list then you will get UnSupportedOperationException.

271.
What is load factor in collection API? What is the default load factor?

- Load factor is a measure that decides when to increase the size of the collection.
- Default load factor is 75%. It means when your collection object gets full by 75% then collection size will increase the space internally.
- Every collection object has its own way or different formula to increase the size internally.

272.
What is the fill ratio in collection framework and what is the default fill ratio?

- Fill ratio is a measure that decides when to increase the size of the collection.
- Default fill ratio is 75%. It means when your collection object gets full by 75% then collection size will increase the space internally.
- Every collection object has its own way or different formula to increase the size internally.

273.
Why Collection interface doesn't extend Clonable and Serializable interface?

- Collection interface is a root interface of the collection hierarchy.
- A lot of concrete implementation classes implements the Collection interface.
- If Collection implements Clonable and Serializable interface then all concrete implementer classes mandatorily implement these two interfaces.
- If there is no requirement for these then every class has to implement that functionality unnecessarily.
- It may impact the performance or work redundancy may happen.
- To avoid all these problems Collections doesn’t implement Clonable and Serializable interface

274.
What are the characteristics of List Interface?

- List is a child interface of the Collection interface.
- Duplicates are allowed.
- Insertion order preserved via index.
- Index is used to retrieve elements.
- Auto growable array.

275.
What are the Implementation classes of List Interface?

- List interface is the child interface of the Collection interface.
- List of implementation classes of List interface.

  1. ArrayList.
  2. LinkedList.
  3. Vector.
     - Stack.

276.
What is ArrayList and what are the characteristics of ArrayList?

- ArrayList is a growable array. No need to declare the size like an array.
- ArrayList is introduced in Java 1.2.
- ArrayList is the best choice if you want to perform retrieval, sorting and searching operations.
- ArrayList is the worst choice if you want to perform an insertion and deletion operation.

- **Characteristics:**

  1. Resizable and Growable Array.
  2. Duplicate Allowed.
  3. Insertion Order Preserved.
  4. Heterogeneous (Different type of Object) Allowed.
  5. NULL insertion is possible.
  6. Not a Thread-safe.
  7. Arraylist is Serializable because this class implements Serializable interface.
  8. Arraylist is Clonable because this class implements Clonable interface.
  9. Randomly Accessible because Arraylist implements RandomAccess interface.

277.
What is the default size of ArrayList? What will happen if ArrayList gets fulled?

- Default size of ArrayList is **10**.
- Once ArrayList reaches its max capacity then a new ArrayList object will be created with the new size and copy all the elements from the old ArrayList to the new ArrayList. Reference variable pointing to new ArrayList and old ArrayList gets deleted.
- Old ArrayList will be eligible for garbage collection.
- Java people set some formula to create new space for array object which is newly created.
- **New Capacity = ((Current Capacity * 3) / 2)+ 1**

278.
How will you convert Array to ArrayList and vice a versa?

- Java has provided some readymade methods in utility class for such kinds of operations.
- We have Arrays and Collections utility classes for these types of operations.
- Arrays utility class has a couple of methods that convert the array to ArrayList and ArrayList to array.
- Convert Array to ArrayList we can use Arrays.asList() method.
- Convert ArrayList to Array we can use Arrays.toArray() method.
- Please refer to the below examples.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class ConvertArrayListToArrayDemo {

		public static void main(String[] args) {
		List<String> nameList = new ArrayList<String>();
		nameList.add("Yogesh");
		nameList.add("Arpita");
		nameList.add("Shweta");
		nameList.add("Shruti");
		nameList.add("Rusty");
		String[] updatedArray = new String[nameList.size()];
		updatedArray = nameList.toArray(updatedArray);
		for(String s : updatedArray) {
		System.out.println(s);
		}
		}
}
```

Output:

```
 Arpita
Yogesh
Shweta
Shruti
```

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class ConvertArrayListToArrayDemo {

		public static void main(String[] args) {
		List<String> nameList = new ArrayList<String>();
		nameList.add("Yogesh");
		nameList.add("Arpita");
		nameList.add("Shweta");
		nameList.add("Shruti");
		nameList.add("Rusty");
		String[] updatedArray = new String[nameList.size()];
		updatedArray = nameList.toArray(updatedArray);
		for(String s : updatedArray) {
		System.out.println(s);
		}
		}
}
```

Output:

```
Yogesh
Arpita
Shweta
Shruti
Rusty
```

279.
ArrayList is more flexible than Array but still where will you use Array over Arraylist?

- When you require the fixed size of homogeneous element then we must go for an Array instead of an ArrayList.
- Using Collection classes for primitives is appreciably slower since they have to use autoboxing and wrappers.
- We should prefer a more straightforward [] syntax for accessing elements over ArrayList's get(). This becomes more important when I need multidimensional arrays.
- ArrayLists usually allocate about twice the memory you need now in advance so that you can append items very fast. So there is wastage if you are never going to add any more items.
- ArrayList accesses are slower than plain arrays in general. The ArrayList implementation uses an underlying array, but all accesses have to go through the get(), set(), remove(), etc. methods which mean it goes through more code than simple array access.

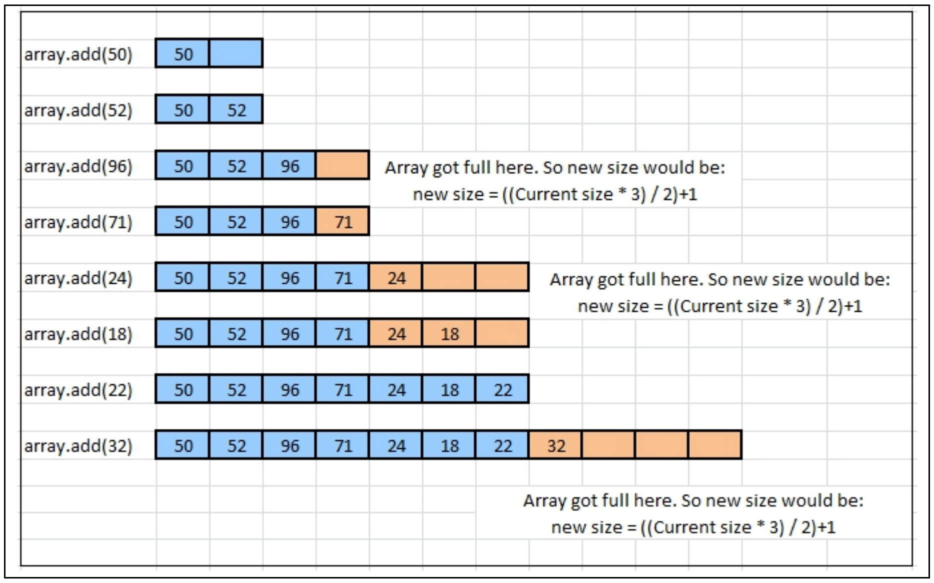
280.
Can you explain internal working of ArrayList?

- When you create an object of ArrayList default size of the ArrayList is 10.
- Means the initial size of the ArrayList is 10.
- When the ArrayList gets to fill 75% then internally JVM performs many operations.

- Internally JVM creates a new ArrayList, Copy all the elements of the old ArrayList into the new ArrayList.
- New ArrayList object size will be calculated by this formula.

    1. New Size = ((Current Size * 3) / 2) + 1.

- Old ArrayList objects will be eligible for garbage collection.
- Reference variable points to the new ArrayList object.
- Every time JVM performs the same process when an object gets full of 75%.
- Please refer below image to better understanding.


array.add(50)
50
array.add(52)
50 52
array.add(96)
50 52 96
Array got full here. So new size would be:
new size = ((Current size * 3) / 2)+1
array.add(71)
50 52 96 71
array.add(24)
50 52 96 71 24
Array got full here. So new size would be:
new size = ((Current size * 3) / 2)+1
array.add(18)
50 52 96 71 24 18
array.add(22)
50 52 96 71 24 18 22
array.add(32)
50 52 96 71 24 18 22 32
Array got full here. So new size would be:
new size = ((Current size * 3) / 2)+1

281.
What is LinkedList?

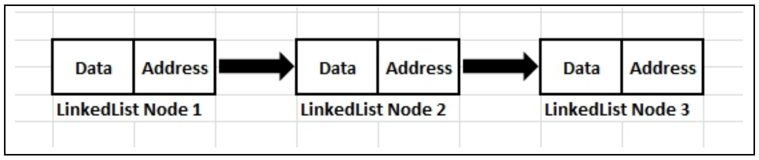
- Basically, LinkedList is a type of data structure that can be used to save the data and address of the next node.
- LinkedList is an implementation of the List interface.
- Linkedlist has three types.

    1. Singly LinkedList.
    2. Doubly LinkedList.
    3. Ciruclar LinkedList.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.LinkedList;

class SinglyLinkedList {

        public static void main(String[] args) {
        LinkedList<Integer> linkedList = new LinkedList<Integer>();
        linkedList.add(50);
        linkedList.add(100);
        linkedList.add(150);
        linkedList.add(200);
        linkedList.add(250);
        for(Integer i : linkedList) {
        System.out.println(i);
        }
        }
}
```

- Graphically representation of linked list.



282.
What are the characteristics of LinkedList?

- Insertion order preserved.
- Duplicates objects are allowed.
- Heterogeneous (Different types of Object) objects are allowed.
- The NULL insertion is possible.
- LinkedList is Clonable because LinkedList implements a Clonable interface.
- LinkedList is Serializable because LinkedList implements a Serializable interface.
- LinkedList is the worst choice if want to perform a serach or sort operation.
- LinkedList has not implemented RandomAccess.

283.
What is Vector? What are the characteristics of Vector?

- A vector can be defined as a dynamic array that can grow or shrink on its own.
- Vector will grow when more elements are added to it and will shrink when elements are removed from it.
- But similar to arrays, vector elements can be accessed using integer indices.
- Default size of vector is 10.
- Vector is a legacy class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Iterator;
import java.util.Vector;

public class VectorDemo {

        public static void main(String[] args) {
        Vector<Integer> v = new Vector<Integer>();
        v.add(10);
        v.addElement(20);
        v.add(30);
        v.addElement(40);
        // Using foreach.
        for(Integer i : v) {
        System.out.println(i);
        }
        // Using iterator
        Iterator<Integer> itr = v.iterator();
```

```
            while(itr.hasNext()) {
            Integer i = itr.next();
            System.out.println("Value of Vector "+ i);
            }
            }
}
```

284.
What are the characteristics of Vector?

- Resizable / Growable array.
- Insertion order preserved.
- Duplicated objects are allowed.
- Heterogeneous (Different types of Object) objects are allowed.
- The NULL insertion is allowed.
- Vector is thread-safe.
- Vector is Serializable because LinkedList implements a Serializable interface.
- Vector is the worst choice if want to perform serach or sort operation.
- Vector has implemented RandomAccess.

285.
What is the initial capacity of Vector?

- Vectors initial capacity is 10.

286.
What is Stack and what are the characteristics of Stack?

- Stack is a linear data structure.
- Stacks works with LIFO (Last In First Out).
- Stack is a child class of Vector.
- Real-Life example: We have observed many times in the cafeteria, Waiter always picks the top of the plate to serve the food to each customer.

287.
Which collection classes implements RandomAccess interface? What are the benefits of RandomAccess interface?

- ArrayList and Vector are implemented Random Access Interface.
- The presence of a marker interface is that it indicates (or expects) specific behaviour from the implementing class.
- So, in our case, ArrayList implements the RandomAccess marker interface.
- So, the expectation from the ArrayList class, is that it should produce RandomAccess behaviour to the clients of the ArrayList class when the client wants to access some element at some index.

288.
Which scenario are the best example to implement of LinkedList and ArrayList?

- **LinkedList:** If you want to perform addition and deletion kind of operation in between the list then we should go for LinkedList.
- **ArrayList:** If you want to perform a searching, sorting kind of operation then we should go for ArrayList.

289.
Which Collection objects implements Queue interface?

- LinkedList implements Queue interface.

290.
Explain Big-O notation with an example?

- The Big-O notation depicts the performance of an algorithm as the number of elements in ArrayList.
- A developer can use Big-O notation to choose the collection implementation.
- It is based on performance, time and memory.
- For example, ArrayList get (index i) is a method to perform a constant-time operation. It does not depend on the total number of elements available in the list. Therefore, the performance in Big-O notation is O (1).

291.
What is Set?

- Set is a child interface of Collection interface.
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order is not preserved.

292.
What are the implementation classes of Set Interface?

- Following are the Classes which implements Set interface.

1. HashSet.
2. LinkedHashSet.
3. TreeSet.

293.
What is HashSet? Can you explain internal working of Hashset?

- HashSet of Collection framework represents the group of unique objects.
- HashSet internally uses hashing technique to store the elements. It's not guaranteed in which sequence the elements will be stored or retrieved.
- Whenever you insert an element into HashSet using add() method, it creates an entry in the HashMap object with the element you have specified as its key and constant called "PRESENT" as its value. This "PRESENT" is defined in the HashSet class as below.
- `private static final Object PRESENT = new Object();`
- Please refer the add method.

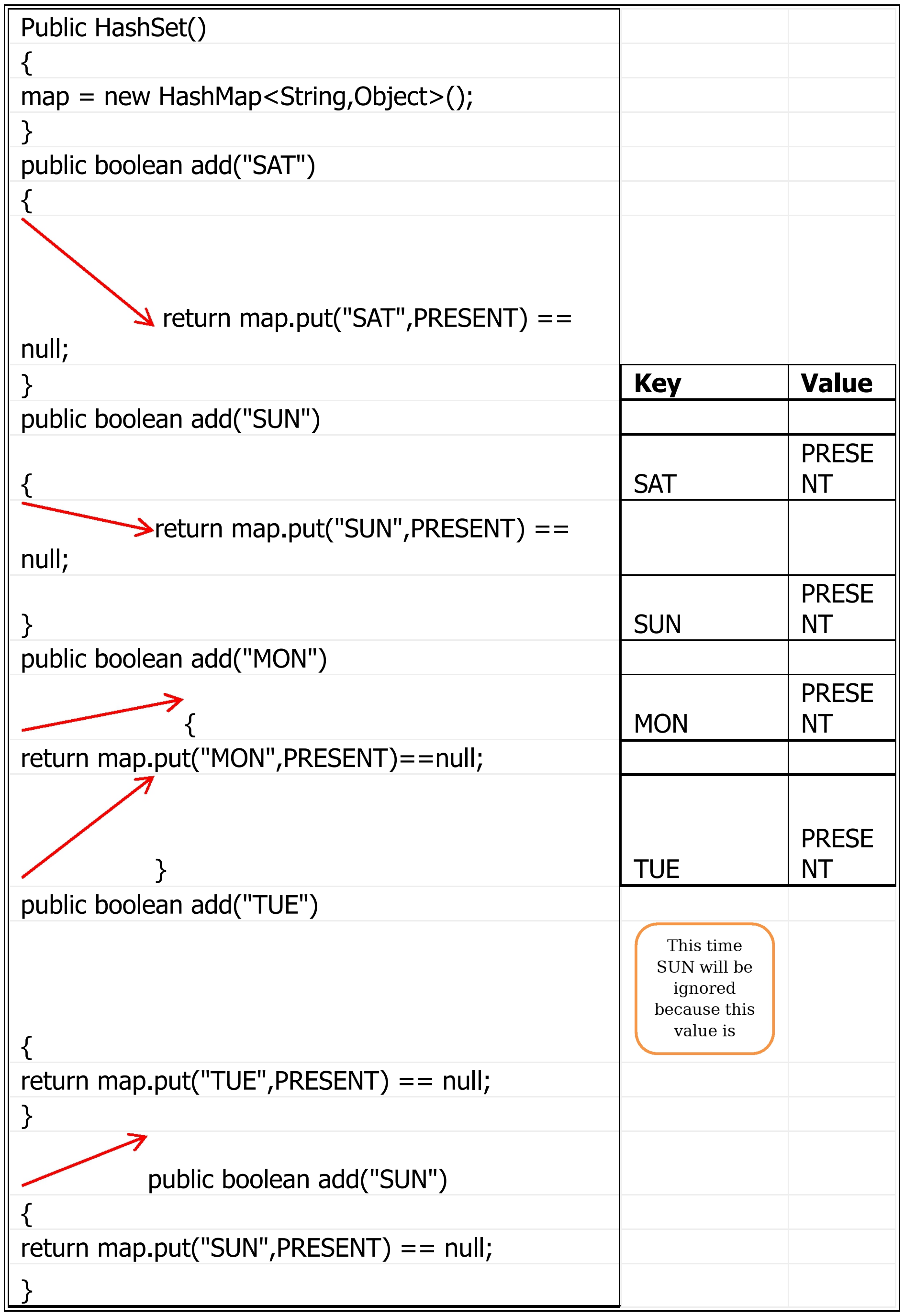
```
    public boolean add(E e) {
return map.put(e, PRESENT)==null;
-    }
```

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;
import java.util.HashSet;

public class HashSetDemo {

        public static void main(String[] args) {
        HashSet<String> dayList = new HashSet<String>();
        dayList.add("Sat");
        dayList.add("Sun");
        dayList.add("Mon");
        dayList.add("Tue");
        dayList.add("Sun");
```

```
            System.out.println(dayList);
              }
}
Output:  [Tue, Sat, Sun, Mon]
```

- Graphical Representation of Internal working of HashSet.



```
Public HashSet()
{
map = new HashMap<String,Object>();
}
public boolean add("SAT")
{
            return map.put("SAT",PRESENT) ==
null;
}
public boolean add("SUN")

{
           return map.put("SUN",PRESENT) ==
null;

}
public boolean add("MON")
               {
return map.put("MON",PRESENT)==null;

            }
public boolean add("TUE")



{
return map.put("TUE",PRESENT) == null;
}
            public boolean add("SUN")
{
return map.put("SUN",PRESENT) == null;
}
```

Key	Value
SAT	PRESENT
SUN	PRESENT
MON	PRESENT
TUE	PRESENT

294.
What are the characteristics of Hashset?

- Duplicate objects are not allowed.

- Insertion order not preserved.
- Based on Hashcode of Object.
- Only one null insertion is possible.
- Heterogeneous (Different types of Object) objects are allowed.
- HashSet is Serializable because HashSet implements a Serializable interface.
- HashSet is Clonable because HashSet implements a Clonable interface.
- HashSet has not implemented RandomAccess.
- Duplicate objects are not allowed and still, you are trying to add duplicate objects then you won't get any compile-time or run-time exception just simply return false if the object is present and return true when the object is not presented.

295.
What is the default size and load factor of HashSet?

- The default size of HashSet is 16.
- Load Factor of HashSet is 75%.

296.
Which data structure is implemented for HashSet?

- Internally HashMap has been implemented for HashSet.

297.
What is LinkedHashSet? What is the purpose of LinkedHashSet?

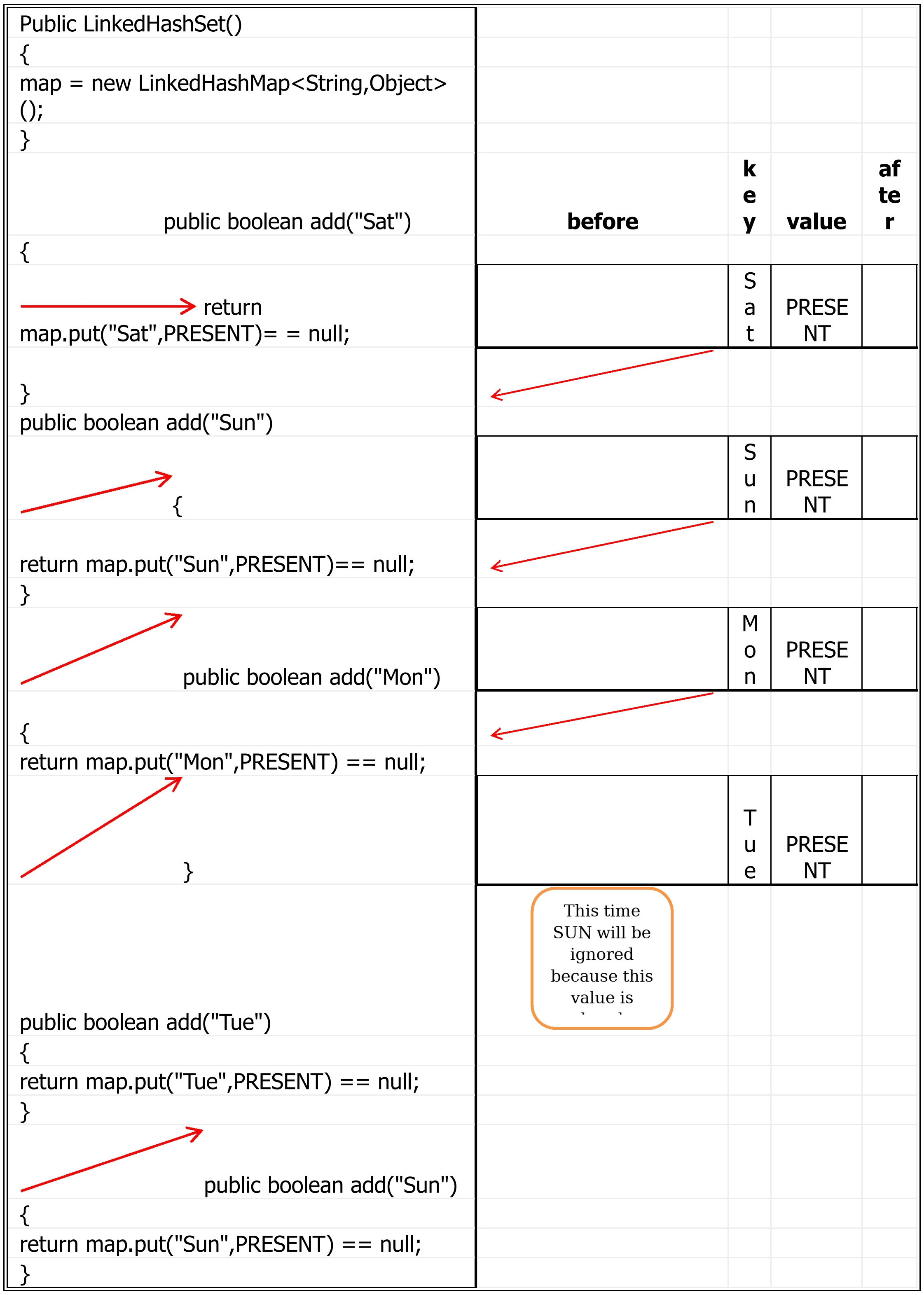
- LinkedHashSet is a child class of HashSet but, LinkedHashSet preserves the insertion order.
- LinkedHashSet is a combination of LinkedList and HashSet.
- You can say LinkedHashSet is the ordered version of HashSet.
- LinkedHashSet doesn't have any method, it inherits all the methods from its superclass i.e. HashSet.
- If you want to maintain the insertion order then you can go for LinkedHashSet.

298.
Explain the internal working of LinkedHashSet?

- LinkedHashSet internally uses the LinkedHashMap object to store the elements.
- The elements you inserted are stored as a key of the LinkedHashMap object.
- There are newly introduced two fields ***before*** and ***after*** which are responsible for maintaining the Insertion order in the LinkedHashSet object.
- LinkedHashMap internally used doubly linkedlist to store the elements.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.LinkedHashSet;

public class LinkedHashSetDemo {

        public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<String>();
        set.add("Sun");
        set.add("Mon");
        set.add("Tue");
        set.add("Wed");
        set.add("Sun");
        System.out.println(set);
        }
}
```

- Please refer Graphical representation below.



Public LinkedHashSet()				
{				
map = new LinkedHashMap<String,Object> ();				
}				
public boolean add("Sat")	before	key	value	after
{				
return map.put("Sat",PRESENT)= = null;		Sat	PRESENT	
}				
public boolean add("Sun")				
{		Sun	PRESENT	
return map.put("Sun",PRESENT)== null;				
}				
public boolean add("Mon")		Mon	PRESENT	
{				
return map.put("Mon",PRESENT) == null;				
}		Tue	PRESENT	
public boolean add("Tue")	This time SUN will be ignored because this value is			
{				
return map.put("Tue",PRESENT) == null;				
}				
public boolean add("Sun")				
{				
return map.put("Sun",PRESENT) == null;				
}				

299.
What is SortedSet? What is the purpose of SortedSet?

- If we want to represent a group of individual objects according to some sorting order without duplicate then we should go for SortedSet.
- SortedSet interface is a child interface of the Set interface.
- Null is not allowed in SortedSet implementation.
- All elements inserted into a sorted set must implement the Comparable interface (or be accepted by the specified comparator).
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo {

            public static void main(String[] args) {
            SortedSet<Integer> set = new TreeSet<Integer>();
            set.add(50);
            set.add(10);
            set.add(30);
            set.add(20);
            set.add(40);
            System.out.println(set);
            }
}
Output:  [10, 20, 30, 40, 50]
```

300.
What is NavigableSet?

- The Java NavigableSet interface, java.util.NavigableSet, is a subtype of the Java SortedSet interface.
- Therefore the NavigableSet behaves like a SortedSet, but with an additional set of navigation methods available in addition to the sorting mechanisms of the SortedSet.
- A NavigableSet may be accessed and traversed in either ascending or descending order.
- The descendingSet() method returns a view of the set with the senses of all relational and directional methods inverted.
- NavigableSet reverse = original.descendingSet(); also returns the reverse order.
- This interface additionally defines methods pollFirst and pollLast that return and remove the lowest and highest element, if one exists, else returning null.
- Methods subSet, headSet, and tailSet differ from the like-named SortedSet methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive.
- Implementation class of NavigableSet is java.util.TreeSet class.
- Subsets of any NavigableSet must implement the NavigableSet interface.
- Create a NavigableSet.

    1. NavigableSet navigableSet = new TreeSet();

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.NavigableSet;
import java.util.TreeSet;
```

```
public class NavigableSetDemo {

        public static void main(String[] args) {
        NavigableSet<Integer> set = new TreeSet<Integer>();
        set.add(50);
        set.add(10);
        set.add(30);
        set.add(20);
        set.add(40);
        System.out.println("Actual Set : " + set);
        System.out.println("Lower value of 30 using lower()      = "+ set.lower(30));
        System.out.println("Higher value of 30 using floor()       = "+ set.floor(30));
        System.out.println("Lowest value of 30 using ceiling()   = "+ set.ceiling(30));
        System.out.println("Lowest value of 30 using higher()    = "+ set.higher(30));
        System.out.println("Lowest value of 30 using pollFirst()    = "+ set.pollFirst());
        System.out.println("Lowest value of 30 using pollLast() = "+ set.pollLast());
        NavigableSet<Integer> reverse = set.descendingSet();
        System.out.println("Reverse Set : " + reverse);
        }
}
Output:
Actual Set : [10, 20, 30, 40, 50]
Lower value of 30 using lower()        = 20
Higher value of 30 using floor()        = 30
Lowest value of 30 using ceiling()   = 30
Lowest value of 30 using higher()    = 40
Lowest value of 30 using pollFirst() = 10
Lowest value of 30 using pollLast()  = 50
Reverse Set : [40, 30, 20]
```

301.
What is the TreeSet? How the object will be stored in Treeset?

- TreeSet stores the object in sorted and ascending order.
- TreeSets underlying data structure is a balanced tree.
- Duplicate objects are not allowed.
- Insertion order not preserved.
- Homogeneous objects are allowed.
- Heterogeneous (Different types of Object) objects are not allowed otherwise we will get ClassCastException.
- Null insertion is not possible otherwise you will get NullPointerException.
- TreeSet is Serializable because TreeSet implements a Serializable interface.
- TreeSet is Clonable because TreeSet implements a Clonable interface.
- TreeSet has not implemented RandomAccess.
- All objects will be inserted based on some sorting order. It may be a default sorting order or customized sorting order.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Iterator;
import java.util.TreeSet;

public class TreeSetDemo {

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

```
		TreeSet<Integer> set = new TreeSet<Integer>();
		set.add(50);
		set.add(10);
		set.add(30);
		set.add(20);
		set.add(40);
		System.out.println("Original Set : " + set);
		TreeSet<Integer> reverse = (TreeSet<Integer>) set.descendingSet();
		System.out.println("Reverse Set : " + reverse);
		// Reverse using iterator
		Iterator<Integer> itr = set.descendingIterator();
		 System.out.println("Using Iterator : ");
		while(itr.hasNext()) {
		int i = (int)itr.next();
		System.out.print(i+ ", ");
		}
		}
}
Output:
Original Set : [10, 20, 30, 40, 50]
Reverse Set : [50, 40, 30, 20, 10]
50, 40, 30, 20, 10,
```

302.
Can we insert null in TreeSet? If not then why it is not possible?

- We cannot insert null in TreeSet.
- TreeSet internally stores the elements according to some sorting order.
- While inserting records into the tree set JVM always compare the two values and while comparing you might get NullPointerException.

303.
What is importance of emptySet() method in collection framework?

- An emptySet() method can be used to get the set which is empty.

304.
What are the characteristics of Queue Interface?

- Queue interface is used to hold the element about to be processed in FIFO(First In First Out).
- A Queue is a collection for holding elements prior to processing.
- Queue is a child interface of the Collection interface.
- Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.
- Please refer the queue interface and its method.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

305.
What is Priority Queue?

- If we want to represent a group of individual objects prior to processing according to some priority then it is called Priority Queue.
- Priority can be natural sorting order or customized sorting order defined by the comparator.
- Insertion order is not preserved. It is based on some priority.
- Duplicate objects are not allowed.
- Null is not allowed.
- If we are depending on natural sorting order then objects must implement a comparable interface or it should be homogeneous. Otherwise, we will get ClassCastException.
- If we are defining our own Comparator then there is no need to have homogeneous object.
- Please refer to the below two examples.

```
package simplifiedjava.crackedInterview;

import java.util.Comparator;
import java.util.PriorityQueue;

public class PriorityQueueDemo {

	public static void main(String[] args) {
	PriorityQueue<String> q = new PriorityQueue<>(new DecendingQueueComparator());
	q.offer("Shweta");
	q.offer("Shruti");
	q.offer("Arpita");
	while(!q.isEmpty()) {
	System.out.println(q.poll().toString());
	}
	}
}

class DecendingQueueComparator implements Comparator<String>{

	@Override
	public int compare(String s1, String s2) {
	return s2.compareTo(s1);
	}
}
Output:
Shweta
Shruti
Arpita

// Ascending Order.

package simplifiedjava.crackedInterview;

import java.util.Comparator;
import java.util.PriorityQueue;

public class PriorityQueueDemo {


	public static void main(String[] args) {
	PriorityQueue<String> q = new PriorityQueue<>(new AscendingQueueComparator());
	q.offer("Shweta");
	q.offer("Shruti");
```

```
        q.offer("Arpita");
        while(!q.isEmpty()) {
        System.out.println(q.poll().toString());
        }
        }
}

class AscendingQueueComparator implements Comparator<String>{

        @Override
        public int compare(String s1, String s2) {
        return s1.compareTo(s2);
        }
}
```

306.
What is Blocking Queue? What are the types of Blocking Queue?

- A blocking queue is a queue that blocks when you try to dequeue from it.
- The queue is empty or if you try to enqueue objects to it and the queue is already full.
- A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
- A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate the failure of poll operations.
- BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally, support the Collection interface.
- For example, it is possible to remove an arbitrary element from a queue using remove(x). However, such operations are in general not performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.
- BlockingQueue implementations are thread-safe.
- All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.
- A BlockingQueue does not intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added.
- A BlockingQueue belongs to java.util.concurrent.BlockingQueue package.
- Following are the **types of BlockingQueue**.

1. PriorityBlockingQueue.
2. LinkedBlockingQueue.
3. ArrayBlockingQueue.

307.
What is CircularQueue? What are the advantages of CircularQueue?

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle.
- The last element is connected back to the first element to make a circle.
- CircularQueue is also called 'Ring Buffer'.
- Circular queue is used in memory management and Process Scheduling.
- **Advantages** of Circular Queue:

1. All operations occur in constant time.
2. Doesn’t use dynamic memory.

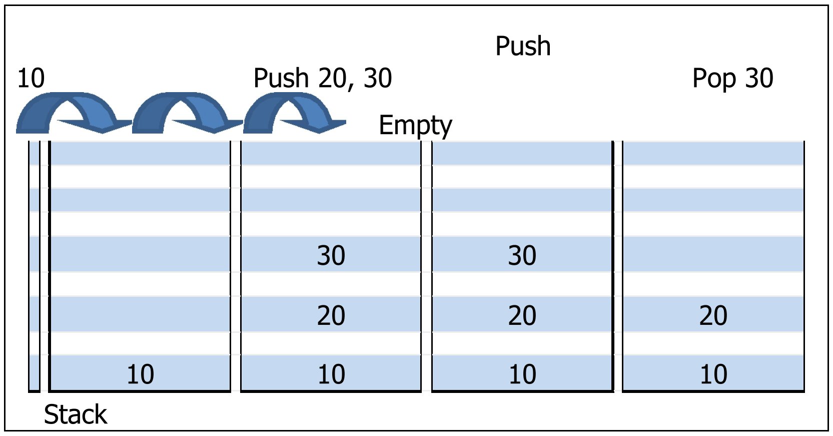
3. No Memory leaks.
4. Simple Implementation.
5. Plays an important role in memory management.

308.
Which data structure has implemented LIFO strategy?

- Stack data structure has implemented LIFO (Last In First Out) strategy.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Stack;

public class StackDemo {

		public static void main(String[] args) {
		Stack<Integer> s = new Stack<Integer>();
		s.push(10);
		s.push(20);
		s.push(30);
		System.out.println(s);
		s.pop();
		System.out.println(s);
		}
}
Output:
[10, 20, 30]
[10, 20]
```

- Graphical representation of Stack implementation.



309.
Explain the defined method in Stack?

- In Stack there are 5 main methods can be used to perform basic operations.

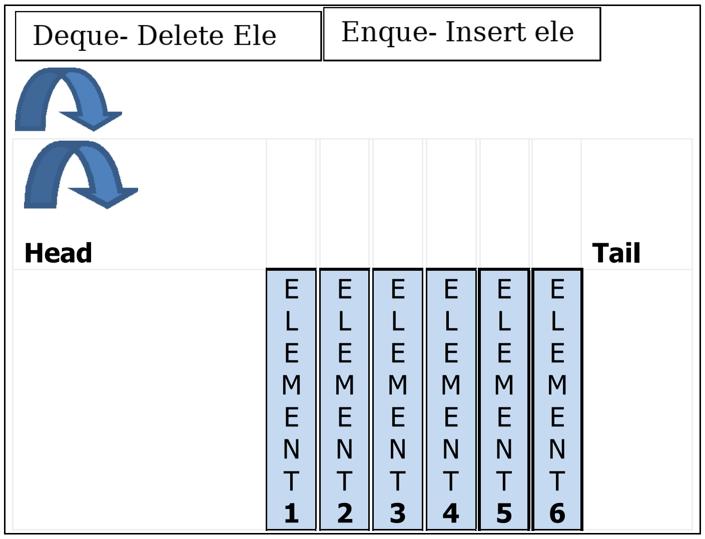
1. **Push(Object O):** To insert an object into stack.
2. **Pop():** To remove and return top of the stack.
3. **Peek():** To return top of the object without removing it.
4. **Empty():** return true if the stack is empty.
5. **Search():** returns offset if the element is available otherwise, returns -1.

310.
Which data structure has implemented FIFO strategy?

- Queue data structure has implemented FIFO (First In First Out) strategy.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.LinkedList;
import java.util.Queue;

public class QueueDemo {

        public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<Integer>();
        queue.offer(10);
        queue.offer(20);
        queue.offer(30);
        System.out.println(queue);
        queue.peek();
        System.out.println(queue);
        queue.poll();
        System.out.println(queue);
        }
}
Output:
[10, 20, 30]
[10, 20, 30]- Pick() just retrieved the value but not removed.
[20, 30] – Poll() retrieved the value and removed as well.
```

- Graphical Representation of Queue.



311.
Explain the defined method in Queue?

- There are 5 main methods defined in Queue.

1. **Offer(Object o):** Add object in queue.
2. **Peek():** return head element of queue. If queue is null then it will return null.

3. **Element():** return head element of queue. If the queue is null the it will throw NoSuchElementException.
4. **Poll():** returns head element of queue and remove. If the queue is empty then it will return null.
5. **Remove():** returns head element of queue and remove. If queue is empty then it will throw NoSuchElementException.

312.
Explain deque interface? What is the importance of deque interface?

- A Deque is a double-ended-queue.
- A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both endpoints.
- The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time.
- The Deque interface, defines methods to access the elements at both ends of the Deque instance.
- Methods are provided to insert, remove, and examine the elements.
- Predefined classes like ArrayDeque and LinkedList implement the Deque interface.
- Note that the Deque interface can be used both as last-in-first-out stacks and first-in-first-out queues.
- **Advantages** of Deque.

1. Deques are faster in adding/removing elements to the end or beginning.
2. Lists are faster in adding/removing elements to any other 'middle' position.
3. You can also use insert (index, value) on lists, while on deques you can only append left or right.

313.
What are the characteristics of Map Interface?

- Map represents the group of objects as a key-value pair.
- Map is not a child interface of the Collection interface.
- Both keys and values are objects only.
- **Characteristics** of the map are as follows.

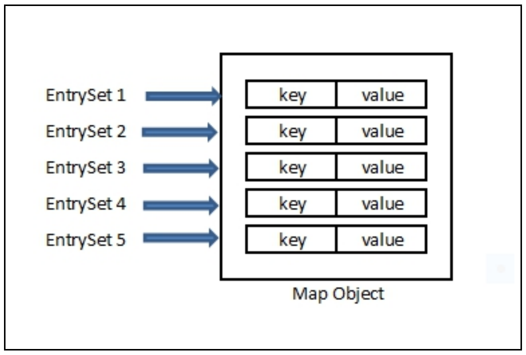
1. Underlying data structure is hashtable.
2. Insertion order is not preserved.
3. Inertion order is based on hashcode keys.
4. Duplicates keys are not allowed.
5. Values can be duplicated.
6. Heterogeneous (Different types of Object) objects are allowed for both keys and values.
7. Null is allowed for keys only once.
8. Null is allowed for values multiple times.
9. Hashmap is Serializable because Hashmap implements a Serializable interface.
10. Hashmap is Clonable because Hashmap implements a Clonable interface.
11. Hashmap has not implemented RandomAccess.

314.
What is the default size of Map?

- Default size of the map is 16.

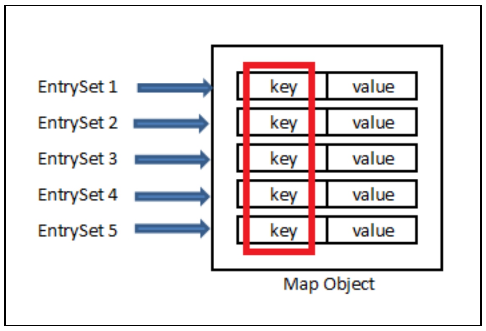
315.
What is EntrySet?

- A map is a group of key-value pair.Each key-value pair is called an Entry. Set of Entry is called EntrySet.
- Entry is an inner interface of Map Interface.
- Inside entry actual key and value pair will be stored.
- While iterating the map object we have to place iterator on Entry, not on a map.
- Without an existing Map object there is no chance of existing Entry object.
- Please refer to the below image for entry set.



316.
What is keySet?

- A map is a group of key-value pairs. The Set of keys are called keyset.
- Graphical representation of the key set.



317.
What is HashMap?

- HashMap is an implementation class of Map interface.
- Underlying data structure of HashMap is Hashtable.
- HashMap stores an object in the form of key and value pair.

318.
Explain the internal working of HashMap?

- Hashmap internally stores mapping in the form of Map.Entry object which contains key and value.
- Hashmap works on hashing principle.
- We use the put() method to insert elements in hashmap and get() method to retrieve elements from the hashmap object.

- Internal working of inserting an element in hashmap.

    1. When we use the put() method to insert key and value objects into a hashmap, internally hashmap uses hashing technique to identify the bucket reference.
    2. Once the bucket is identified object is stored in Map.Entry object in the form of key and value.
    3. It may happen multiple objects can have the same hashcode so multiple objects will be stored in the same bucket.
    4. If one bucket has multiple objects then those objects are stored in the form of a LinkedList.
    5. We must override hashcode() and equals() method.
    6. Hashcode() method calculates the hashcode as an integer value and the equals() method is used to identify the key if the key is matching then it will replace the existing element otherwise that element will be stored next to the current element.

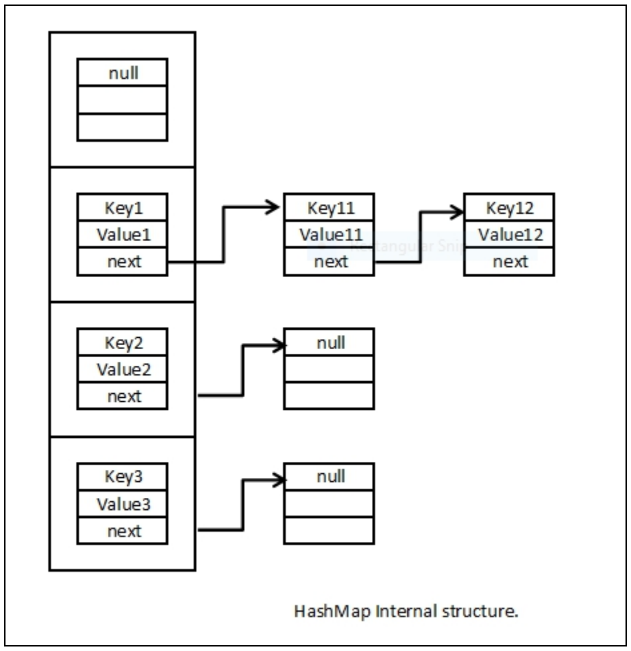
- Internal working of retrieving element in hashmap.

    1. When you want to retrieve an object from hashmap then you can call the get() method and pass the key object.
    2. Hashmap generates the hashcode for that key and finds the appropriate bucket.
    3. If there is only one element available in that bucket then simply that element will be returned.
    4. If we want to retrieve an object from this linked list, we need an extra check to search for the correct element, this is done by the equals() method.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.HashMap;

public class HashMapDemo {

        public static void main(String[] args) {
        HashMap<Integer, String> hMap = new HashMap<Integer, String>();
        hMap.put(1, "One");
        hMap.put(10, "Ten");
        hMap.put(100, "Hundred");
        hMap.put(1000, "Thousand");
        System.out.println(hMap);
        System.out.println(hMap.get(100));
        }
}
Output:
{1=One, 100=Hundred, 1000=Thousand, 10=Ten}
Hundred
```

- Graphical Representation of Hashmap.



319.
What will happen if two different objects have the same hashcode?

- If the two different object have same hashcode, bucket location would be same that time collision will occur in HashMap Since HashMap uses LinkedList to store object internally, this entry (object of Map.Entry contains key and value) will be stored in LinkedList.

320.
What happen if hashmap object gets full?

- Default size of the hashmap is 16 and the load factor is 75%.
- If the load factor is filled 75% then internally hashmap size gets changed like other data structures like ArrayList, vector etc.
- Hashmap re-size itself by creating a new bucket array of size twice the previous size of hashmap and then start putting all old elements into the new bucket array this process is called rehashing.

321.
What is rehashing?

- Hashmap re-size itself by creating a new bucket array of size twice the previous size of hashmap and then start putting all old elements into the new bucket array this process is called rehashing.

322.
Why we need to override hashcode() and equals() in HashMap implementation?

- If the two objects are equals by the .equals() method then their hashcode() must be equal. Two equivalent objects have the same hashcode.
- If(obj1.equals(obj2)) is true then obj1.hashCode() == obj2.hashCode() always true.
- If the two objects are not equal by the .equals() method then the hashcode of the two objects may or may not be equal.

- If(!obj1.equals(obj2)) is true then obj1.hashCode() == obj2.hashCode() may be true or may be false.
- The objects will be stored in the same bucket which has the same hashcode.
- If one bucket has multiple objects then internally it is stored in the form of LinkedList.
- If you override the hashcode() method then your object will be placed in the correct bucket and if you override the equals() method then hashmap will check all the keys and if the key exists then it will replace the new value and if the key is not present then that object will be placed next to the current object in the same bucket.
- If you don't override the hashcode() method then your object may place in the wrong bucket.
- If you don't override the equals() method then your object will not place at the correct location.

323.
What will be the impact if hashcode() method return constant value everytime? What will be the size of hashmap, In Employee class I have overridden equals and hashcode method and hashcode method always return 1. I have 5 employee objects in hashmap. What will be the size?

- You would lose any performance given by a hashmap, that can retrieve items from a collection in O(1) time for objects with different hashes, which is what we want to achieve when using HashMaps.
- If the equals method is implemented as per the contract and the hashcode method returns a constant value, then we will still be able to retrieve the value for the key from a hashMap, but the performance will be slow compared to the method returning a unique hashcode.
- The size would be 5.

324.
What will be the impact if I don't override equals() method?

- **Only Override HashCode, Use the default Equals:** Only the references to the same object will return true. In other words, those objects you expected to be equal will not be equal by calling the equals method.
- **Only Override Equals, Use the default HashCode:** There might be duplicates in the HashMap or HashSet. We write the equals method and expect {"xyz", "XYZ"} to be equals. However, when using a HashMap, they might appear in different buckets, thus the contains() method will not detect each other.

325.
Why String and wrapper classes are considered good keys?

- Strings and wrapper classes are immutable so immutable objects are really a good option for keys because once you insert the key's hashcode never change throughout life.
- String and wrapper classes also overrides the equals() and hashcode() methods.
- Immutable objects are thread-safe this is an advantage as well.

326.
Can we use ConcurrentHashMap in place of Hashtable?

- Yes, definitely we can use ConcurrentHashMap in place of Hashtable.
- Concurrent hashmap is more flexible than hashtable.
- Hashtable maintains the object level lock.
- ConcurrentHashMap maintains the bucket level lock.

327.
Can we use any custom object as a key in HashMap?

- We can set any custom object as a key in the hashmap.
- You have to override hashcode() and equals() methods into that class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Employee {

            private String empName;
            private String dept;
            private Double salary;
            private String designation;
            public Employee(String empName, String dept, Double salary, String designation) {
            super();
            this.empName = empName;
            this.dept = dept;
            this.salary = salary;
            this.designation = designation;
            }
            public Employee(String empName) {
            this.empName = empName;
            }
            public String getEmpName() {
            return empName;
            }
            public String getDept() {
            return dept;
            }
            public void setDept(String dept) {
            this.dept = dept;
            }

            public Double getSalary() {
            return salary;
            }
            public void setSalary(Double salary) {
            this.salary = salary;
            }
            public String getDesignation() {
            return designation;
            }
            public void setDesignation(String designation) {
            this.designation = designation;
            }
            public void setEmpName(String empName) {
            this.empName = empName;
            }
            @Override
            public String toString() {
            return "Employee [empName=" + empName + ", dept=" + dept + ", salary=" + salary + ",
designation=" + designation
            + "]";
            }
            @Override
            public int hashCode() {
```

```
		final int prime = 31;
		int result = 1;
		result = prime * result + ((dept == null) ? 0 : dept.hashCode());
		result = prime * result + ((designation == null) ? 0 : designation.hashCode());
		result = prime * result + ((empName == null) ? 0 : empName.hashCode());
		result = prime * result + ((salary == null) ? 0 : salary.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 (dept == null) {
		if (other.dept != null)
		return false;
		} else if (!dept.equals(other.dept))
		return false;
		if (designation == null) {
		if (other.designation != null)
		return false;
		} else if (!designation.equals(other.designation))
		return false;
		if (empName == null) {
		if (other.empName != null)
		return false;
		} else if (!empName.equals(other.empName))
		return false;
		if (salary == null) {
		if (other.salary != null)
		return false;
		} else if (!salary.equals(other.salary))
		return false;
		return true;
		}
}

package simplifiedjava.crackedInterview;

import java.util.HashMap;

public class HashMapEmpHashCodeDemo {

		public static void main(String[] args) {
		HashMap<Employee,Integer> empMap = new HashMap<Employee,Integer>();
		empMap.put(new Employee("Yogesh","IT",100000.00,"System Analyst"),101);
		empMap.put(new Employee("Arpita","Mangement",1200000.00,"Trustee"),102);
		empMap.put(new Employee("Shweta","DevOps",45000.00,"Jenkin Engineer"),103);
		empMap.put(new Employee("Shruti","IT",65000.00,"DB Admin"),104);
		empMap.put(new Employee("Priyanka","IT",35000.00,"Test Engineer"),105);
		System.out.println(empMap.size());
```

```
        }
}
Output:  5
```

328.
How null key is handled in HashMap?

- Hashmap handles the different ways to handle null keys while putting and retrieving.
- Null key always map to the 0th index.
- There are a couple of separate methods for that, putForNullKey (V value) and getForNullKey ().
- Equals() and hashcode() methods are not used in the case of null keys.
- For getting value by null key it also has a method getForNullKey which it calls from inside the get method.
- Please refer to the below example.

```
private V getForNullKey() {
        if (size == 0) {
          return null;
        }
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
          if (e.key == null)
              return e.value;
        }
        return null;
    }
```

329.
What is the time Complexity for map.get()?

- The time complexity of hashmap for get operation and put operation is generally O(1).
- O(1) certainly isn't guaranteed - but it's usually what you should assume when considering which algorithms and data structures to use.

330.
What is LinkedHashMap? Can you explain internal working of LinkedHashMap?

- LinkedHashMap is a data structure that preserves the insertion order and stores the objects in the form of key/value pair like a Hashmap.
- LinkedHashMap is a child class of Hashmap and implements a Map interface.
- LinkedHashMap is not a thread-safe.
- It can contain only one null key and multiple null values.
- It contains only unique elements.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.LinkedHashMap;

public class LinkedHashMapDemo {

        public static void main(String[] args) {
        LinkedHashMap<Integer, String> linkHm = new LinkedHashMap<Integer, String>();
        linkHm.put(100, "One Hundred");
        linkHm.put(200, "Two Hundred");
```

```
            linkHm.put(300, "Three Hundred");
            linkHm.put(500, "Five Hundred");
            linkHm.put(200, "Two Hundred");
            linkHm.put(100, "One Hundred");
            linkHm.put(null, "NULL Key");
            linkHm.put(600, null);
            linkHm.put(700, null);
            System.out.println(linkHm);
            }
}
Output:  {100=One Hundred, 200=Two Hundred, 300=Three Hundred, 500=Five Hundred, null=NULL Key,
600=null, 700=null}
```

331. What is WeakHashMap?

- In the case of hashmap even though the object doesn't have any reference still it is not eligible for garbage collection. If it is associated with HashMap i.e. Hashmap dominates garbage collector.
- In the case of WeakHashMap if the object doesn't contain any reference is eligible for garbage collection. Even though the object is associated with WeakHashMap. Garbage collector dominates WeakHashMap.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Temp {

            public String toString() {
            return "temp";
            }
            public void finalize() {
            System.out.println("Finalize() method called.");
            }
}
```

// Demo for HashMap

```
package simplifiedjava.crackedInterview;

import java.util.HashMap;

public class HashMapDemoForWeaknessCompare {

            public static void main(String[] args) throws InterruptedException {
            HashMap m = new HashMap();
            Temp t = new Temp();
            m.put(t, "Testing");
            System.out.println(m);
            t = null;
            System.gc();
            Thread.sleep(5000);
            System.out.println(m);
            }
}
```

```
Output:
{temp=Testing}
{temp=Testing}
```

// Demo for WeakHashMap

```java
package simplifiedjava.crackedInterview;

import java.util.WeakHashMap;

public class WeakHashMapDemo {

	public static void main(String[] args) throws InterruptedException {
	WeakHashMap m = new WeakHashMap();
	Temp t = new Temp();
	m.put(t, "Testing");
	System.out.println(m);
	t = null;
	System.gc();
	Thread.sleep(5000);
	System.out.println(m);
	}
}
```

```
Output:
{temp=Testing}
Finalize() method called.
{}
```

Please cross verify the output for both HashMap and WeakHashMap. In case of WeakHashMap finalize() method has been called means garbage collections performed his task and second time we are not able to get the object which has evacuated by GC.

332.
What is IdentityHashMap?

- In the case of IdentityHashMap JVM will use "==" to identify duplicate keys which are meant for reference comparison.
- For normal HashMap, JVM will use the .equals() method to identify duplicate keys which is meant for content comparison.
- Please refer couple of examples below.

```java
package simplifiedjava.crackedInterview;

import java.util.HashMap;
import java.util.IdentityHashMap;

public class HashMapDemoForIdentityHashMap {

	public static void main(String[] args) {
	HashMap<Integer, String> hm = new HashMap<Integer, String>();
	hm.put(new Integer(10), "Yogesh");
	hm.put(new Integer(10), "Arpita");
	System.out.println(hm);
	IdentityHashMap<Integer, String> ihm = new IdentityHashMap<Integer, String>();
	ihm.put(new Integer(10), "Yogesh");
	ihm.put(new Integer(10), "Arpita");
	System.out.println(ihm);
	}
}
```

```
{10=Arpita}
{10=Yogesh, 10=Arpita}
```

333. What is SortedMap? How objects will be stored in SortedMap bydefault?

- SortedMap is a child interface of Map interface.
- The map is ordered according to the natural sorting order of its keys but not the values.
- All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator).
- TreeMap is the implementation class of the SortedMap interface.
- Please refer to the below example. The reference is SortedMap and its implementation is TreeMap.

```
package simplifiedjava.crackedInterview;

import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapDemo {

        public static void main(String[] args) {
        SortedMap<Integer, String> sMap = new TreeMap<Integer, String>();
        sMap.put(1000, "Thousand");
        sMap.put(1, "One");
        sMap.put(10000, "Ten Thousand");
        sMap.put(100, "Hundred");
        sMap.put(10, "Ten");
        System.out.println(sMap);
        }
}
Output:  {1=One, 10=Ten, 100=Hundred, 1000=Thousand, 10000=Ten Thousand}
```

334. Can we store objects in decending order in SortedMap? If yes then, how can I store objects in decending order in SortedMap?

- We can store objects in descending order in SortedMap.
- There are a couple of ways we can do it in the case of SortedMap.

    1. The first way is to implement Comparator in TreeMap Constructor.
    2. The second way is you can use the Collections class reverseOrder() method.

- Please refer to the below two examples.

1. Using Comparator Interface.

```
package simplifiedjava.crackedInterview;

import java.util.Collections;
import java.util.Comparator;
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapDemo {

        public static void main(String[] args) {
        SortedMap<Integer, String> sMap = new TreeMap<>(new DecendingOrder());
        sMap.put(1000, "Thousand");
        sMap.put(1, "One");
        sMap.put(10000, "Ten Thousand");
        sMap.put(100, "Hundred");
        sMap.put(10, "Ten");
        System.out.println(sMap);
        }
}

class DecendingOrder implements Comparator<Integer>{

        @Override
        public int compare(Integer i1, Integer i2) {
        return i2.compareTo(i1);
        }
}
Output:  {10000=Ten Thousand, 1000=Thousand, 100=Hundred, 10=Ten, 1=One}
```

2. Using reverseOrder() of Collections class.

```
package simplifiedjava.crackedInterview;

import java.util.Collections;
import java.util.Comparator;
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapDemo {

            public static void main(String[] args) {
            SortedMap<Integer, String> sMap = new TreeMap<>(Collections.reverseOrder());
            sMap.put(1000, "Thousand");
            sMap.put(1, "One");
            sMap.put(10000, "Ten Thousand");
            sMap.put(100, "Hundred");
            sMap.put(10, "Ten");
            System.out.println(sMap);
            }
}
Output:  {10000=Ten Thousand, 1000=Thousand, 100=Hundred, 10=Ten, 1=One}
```

335.
What is NavigableMap? What is the purpose of NavigableMap?

- NavigableMap is a child class of SortedMap.
- NavigableMap has added some new methods to navigate throughout the collection object.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.NavigableMap;
import java.util.TreeMap;

public class NavigableMapDemo {

            public static void main(String[] args) {
            NavigableMap<Integer, String> nMap = new TreeMap<Integer, String>();
```

```
		nMap.put(50,"Fifty");
		nMap.put(10,"Ten");
		nMap.put(30,"Thirty");
		nMap.put(20,"Twenty");
		nMap.put(40,"Forty");
		System.out.println("Actual nMap : " + nMap);
		System.out.println("Lower value of 30 using lower()     = "+ nMap.lowerKey(30));
		System.out.println("Higher value of 30 using floor()       = "+ nMap.floorKey(30));
		System.out.println("Lowest value of 30 using ceiling()   = "+ nMap.ceilingKey(30));
		System.out.println("Lowest value of 30 using higher()    = "+ nMap.higherKey(30));
		System.out.println("Lowest value of 30 using pollFirst()    = "+ nMap.pollFirstEntry());
		System.out.println("Lowest value of 30 using pollLast() = "+ nMap.pollLastEntry());
		NavigableMap<Integer, String> reverse = nMap.descendingMap();
		System.out.println("Reverse nMap : " + reverse);
		}
}
Output:
Actual nMap : {10=Ten, 20=Twenty, 30=Thirty, 40=Forty, 50=Fifty}
Lower value of 30 using lower()          = 20
Higher value of 30 using floor()          = 30
Lowest value of 30 using ceiling()     = 30
Lowest value of 30 using higher()      = 40
Lowest value of 30 using pollFirst()   = {10=Ten}
Lowest value of 30 using pollLast() = {50=Fifty}
Reverse nMap : {40=Forty, 30=Thirty, 20=Twenty}
```

336.
What is TreeMap? Can you explain how the objects will be stored in TreeMap?

- TreeMap is a data structure where elements are stored in some natural sorting order.
- Underlying data structure is the RED-BLACK tree.
- Insertion order never maintains. Maintain some natural sorting order.
- Duplicates keys are not allowed but duplicate values are allowed.
- If we are depending on natural sorting order then keys should be homogeneous and comparable otherwise it will through ClassCastException.
- If we are defining our comparator then keys need not be homogeneous or comparable.
- We can insert homogeneous as well as Heterogeneous (Different types of Object) objects in the Treemap.
- Null is strictly now allowed after java 7.
- Till java 6, For non-empty Treemap if we are trying to insert an entry with the null key then we will get NullPointerException. For empty tree map at first entry with null key is allowed but after inserting new key you will get NullPointerException.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.TreeMap;

public class TreeMapDemo {

		public static void main(String[] args) {
		TreeMap<Integer, String> tMap = new TreeMap<Integer, String>();
		tMap.put(100, "Hundred");
		tMap.put(1, "One");
		tMap.put(1000, "Thousand");
```

```
            tMap.put(10, "Ten");
            System.out.println(tMap);
            }
}
Output:  {1=One, 10=Ten, 100=Hundred, 1000=Thousand}
```

337.
Is it possible to store elements in decending order in TreeMap?

- Yes, it is possible to store elements in treeMap in descending order.
- There are a couple of ways we can store elements in descending order which is as follows.

1. You can provide a Comparator object in the TreeMap constructor.
2. You can use Collections.reverse() method to it.

```
package simplifiedjava.crackedInterview;

import java.util.Collections;
import java.util.Comparator;
import java.util.TreeMap;

public class TreeMapDemo {

            public static void main(String[] args) {
            TreeMap<Integer, String> tMap = new TreeMap<Integer, String>(new ReverseOrderTreeElements());
            tMap.put(100, "Hundred");
            tMap.put(1, "One");
            tMap.put(1000, "Thousand");
            tMap.put(10, "Ten");
            System.out.println(tMap);
            }
}

class ReverseOrderTreeElements implements Comparator<Integer>{

            @Override
            public int compare(Integer o1, Integer o2) {
            return o2.compareTo(o1);
            }
}
Output :  {1000=Thousand, 100=Hundred, 10=Ten, 1=One}

package simplifiedjava.crackedInterview;

import java.util.Collections;
import java.util.Comparator;
import java.util.TreeMap;

public class TreeMapDemo {

            public static void main(String[] args) {
            TreeMap<Integer, String> tMap = new TreeMap<Integer, String>(Collections.reverseOrder());
            tMap.put(100, "Hundred");
            tMap.put(1, "One");
            tMap.put(1000, "Thousand");
            tMap.put(10, "Ten");
            System.out.println(tMap);
            }
}
Output:  {1000=Thousand, 100=Hundred, 10=Ten, 1=One}
```

338. What is Hashtable? What are the characteristics of Hashtable?

- Hashtable is a thread-safe version of HashMap.
- Hashtable also maintain the elements in the form of Key/value pair.
- Underlying data structure of Hashtable is Hashtable.
- Insertion order never maintains.
- Insertion depends on the hashcode of the key.
- Duplicate keys are not allowed but duplicate values are allowed.
- Heterogeneous (Different types of Object) objects are allowed.
- Null is not allowed for both keys as well as value. Otherwise, it will through NullPointerException.
- Every method defined inside HashTable is synchronized.
- Hashtable is Clonable because Hashtable implements a Clonable interface.
- Hashtable is Serializable because Hashtable implements a Serializable interface.
- Hashtable is the worst choice if want to perform search or sort operation.
- Hashtable has not implemented RandomAccess.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Hashtable;

public class HashTableDemo {

            public static void main(String[] args) {
            Hashtable<Integer, String> tMap = new Hashtable<Integer, String>();
            tMap.put(100, "Hundred");
            tMap.put(1, "One");
            tMap.put(1000, "Thousand");
            tMap.put(10, "Ten");
            System.out.println(tMap);
            }
}
Output:  {10=Ten, 1000=Thousand, 1=One, 100=Hundred}
```

339. What is Properties Class? What is the role of properties class?

- Properties class is used to read the values from multiple sources like config file, properties file etc.
- The main advantage of this approach is if there is a change in the properties file to reflect that change in java class then just re-deployment is required, Not required to rebuild the project.
- In our project, if any changes are required frequently like username, password, the port number is not recommended to hardcode in the java program, If there is any change to reflect that change requires recompilation, rebuild and redeployment. Even sometimes server restart may require which may impact the business.
- To overcome this problem we can set values in the properties file and then just redeploy is required if any change is there.

- Please refer to the below examples.

  1. In the first example, we are reading values from the properties file.
  2. In the second example, we are setting values to the properties file.

1. First example we are reading values from properties file.

```
package simplifiedjava.crackedInterview;

import java.io.FileInputStream;
import java.util.Enumeration;
import java.util.Properties;

public class PropertiesFileDemo {

        public static void main(String[] args){
        Properties properties = new Properties();
        try {
        FileInputStream fis = new FileInputStream("G:\\Core
Java\\InterviewPrograms\\src\\javaproperties.properties");
        properties.load(fis);
        } catch (Exception e) {
        System.out.println("File Not Found");
        e.printStackTrace();
        }
        Enumeration enumeration = properties.propertyNames();
        while(enumeration.hasMoreElements()) {
        String propName = (String)enumeration.nextElement();
        String value = properties.getProperty(propName);
        System.out.println(propName + " Value is = " + value);
        }
        }
}
```

Output:

```
port Value is = 8080
password Value is = yogi@12345
username Value is = yogi
```

2. Second example we are setting values to properties file.

```
package simplifiedjava.crackedInterview;

import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

public class PropertiesFileDemo {

        public static void main(String[] args){
        Properties properties = new Properties();
        try {
        FileOutputStream fos = new FileOutputStream("G:\\Core
Java\\InterviewPrograms\\src\\javaproperties.properties");
        properties.setProperty("url", "thin:driver:oracle:4848");
        properties.store(fos, "URL added");
        } catch (IOException e) {
            e.printStackTrace();
            }
            }
}
Output:
javaproperties.properties:
#URL added
#Mon Nov 01 14:08:17 IST 2021
url=thin\:driver\:oracle\:4848
```

340.
What is Dictionary class?

- A java Dictionary is an abstract class that stores elements in the form of key-value pairs.
- The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
- Every key and every value is an object.
- In any one Dictionary object, every key is associated with at most one value.
- Any non-null object can be used as a key and as a value.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Dictionary;
import java.util.Hashtable;

public class DictionaryClassDemo {

            public static void main(String[] args) {
            Dictionary<Integer, String> dictionary = new Hashtable<Integer, String>();
            dictionary.put(10, "Ten");
            dictionary.put(100, "Hundred");
            dictionary.put(1000, "Thousand");
            dictionary.put(1, "One");
            System.out.println(dictionary);
            }
}
Output:  {1000=Thousand, 10=Ten, 1=One, 100=Hundred}
```

341.
Why map doesn't extends Collection interface?

- Map doesn’t extend the Collection interface because they are not compatible.
- Internal data structure of Collection implementations are like a list of objects and the internal data structure of the map is key-value pair.
- Collection implementations are index-based collections.
- Map is not an indexed based collection but it is key-value based collection.
- For adding objects to the collection we use add(Object o) method.
- For adding objects to the map we use the put(key, value) method.
- In the case of iterator, we directly put the iterator on the collection implementation class.
- But, In the case map, we have to put the iterator on the entry set.

342.
Can you write a code to iterate a hashMap object?

- We can iterate Hashmap couple of ways.

    1. The first way is we can collect keyset and iterate the map and print the vales.
    2. The second way is we can collect the entry set and iterate map and print the values.

```
package simplifiedjava.crackedInterview;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class HashMapIteratorDemo {

		public static void main(String[] args) {
		HashMap<Integer, String> hMap = new HashMap<Integer, String>();
		hMap.put(1, "One");
		hMap.put(10, "Ten");
		hMap.put(100, "Hundred");
		hMap.put(1000, "Thousand");
		System.out.println("=============== Using KeySet===============");
		Iterator itr1 = hMap.keySet().iterator();
		while(itr1.hasNext()) {
		Integer key = (Integer) itr1.next();
		System.out.println("Key : "+ key + "\t Value : "+ hMap.get(key));
		}
		System.out.println("=============== Using EntrySet===============");
		Iterator itr2 = hMap.entrySet().iterator();
		while(itr2.hasNext()) {
		Map.Entry<Integer, String> pair = (Map.Entry<Integer, String>)itr2.next();
		System.out.println("Key : " + pair.getKey()+ "\tValue : "+ pair.getValue());
		}
		}
}
Output:
=============== Using KeySet===============
Key : 1 Value : One
Key : 100	Value : Hundred
Key : 1000	Value : Thousand
Key : 10	Value : Ten
=============== Using EntrySet===============
Key : 1Value : One
Key : 100	Value : Hundred
Key : 1000	Value : Thousand
Key : 10	Value : Ten
```

343.
Which data structure has implemented for HashMap?

- Hashtable data structure has been implemented for HashMap.

344.
Which data structure has implemented for TreeMap?

- RED-BLACK Tree data structure has been implemented for TreeMap.

345.

Can I convert Map into List?

- We can convert the map into a List.
- But we have to create a different list for the key set and value set.
- Please refer to the below examples.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class ConvertMapToList {

        public static void main(String[] args) {
        HashMap<Integer, String> hMap = new HashMap<Integer, String>();
        hMap.put(1, "One");
        hMap.put(10, "Ten");
        hMap.put(100, "Hundred");
        hMap.put(1000, "Thousand");
        List<Integer> keyList = new ArrayList<Integer>(hMap.keySet());
        List<String> valueList = new ArrayList<String>(hMap.values());
        System.out.println("Key List "+ keyList);
        System.out.println("Value List "+ valueList);
        }
}
Output:
Key List [1, 100, 1000, 10]
Value List [One, Hundred, Thousand, Ten]
```

346.
What is hash-collision in Hashtable and how will you handled in Collection API?

- Two different keys with the same hash value are known as hash-collision.
- Two separate entries will be kept in a single hash bucket to avoid the collision.
- There are two ways to avoid hash-collision. Separate Chaining and Open Addressing.

1. **Separate Chaining:**
    - Keys are stored inside and outside the hashtable.
    - **Extra space is required for the pointers to store the keys outside the hashtable.**
    - **Some buckets of the hash table are never used which leads to wastage of space.**
    - **Cache performance is poor. This is because of linked lists which store the keys outside the hash table.**
2. **Open Addressing:**
    - All the keys are stored inside only in the hash table. No key is present outside the Hashtable.
    - **No extra space is required.**
    - **Buckets may be used even if non-mapped keys to those particular buckets.**
    - **Cache performance is better. This is because there is no linked lists are used.**

347.
What is the difference between Array and ArrayList?

	Array	ArrayList
1.	An array is a fixed length of data structure. It's not dynamic.	ArrayList is not a fixed type of data structure Arraylist is dynamic.
2.	An array can store only homogeneous objects.	ArrayList can store homogeneous as well as heterogeneous objects.
3.	While declaring an array it is mandatory to declare the size of an array.	While declaring an ArrayList it is not mandatory to declare the size of ArrayList.
4.	You cannot add a new element once the array gets full.	You can easily add a new element once the ArrayList gets full.
5.	An array is not a part of the collection framework.	ArrayList is a part of the Collection framework.
6.	An array can be single or multidimensional.	ArrayList cannot be multidimensional.
7.	For iterating array, we can use while loop, for loop, for each loop.	For iterating ArrayList, we can use an iterator.
8.	Arrays are by default type-safe.	ArrayList is not by-default type-safe. We have to make it type-safe explicitly.
9.	Performance of array is best than ArrayList.	The performance of ArrayList is a little low as compared to an array.

348.
What is the difference between ArrayList and Vector?

	ArrayList	Vector
1.	ArrayList is not thread-safe.	Vector is thread-safe.
2.	Methods of ArrayList are not thread-safe.	All methods of Vector are thread-safe.
3.	The performance of ArrayList is best than Vector.	The performance of Vector is low as compared to ArrayList.
4.	ArrayList is not a legacy class. Introduced in Java 1.2	Vector is a legacy class introduced in 1.0

349.
What is the difference between ArrayList and LinkedList?

	ArrayList	LinkedList
1.	ArrayList is an index base collection.	LinkedList is a node base collection.
2.	ArrayList implements RandomAccess Interface.	LinkedList doesn't implement the RandomAccess interface.
3.	ArrayList elements are stored on consecutive memory.	LinkedList elements are not stored on consecutive memory.
4.	There is no singly or doubly concept for ArrayList. Each ArrayList has only one index.	LinkedList can be Singly LinkedList or Doubly LinkedList.
5.	ArrayList is the best option if you want to perform searching, sorting and retrieval operations.	LinkedList is the best option if you want to perform insertion, deletion operations in between of LinkedList.

350.
What is the difference between List and Set?

	List	**Set**
1.	Duplicate elements are allowed in the list.	Duplicates are strictly not allowed.
2.	Insertion order of elements are preserved.	Insertion order of elements are not preserved.
3.	Implementation classes are as Follows. 1. ArrayList 2. LinkedList 3. Vector 4. Queue	Implementation classes are as Follows. 1. HashSet 2. LinkedHashSet 3. TreeSet 4. Queue

351.
What is the difference between Stack and Queue?

	Stack	**Queue**
1.	The stack data structure is based on LIFO (Last In First Out) principle.	The queue data structure is based on FIFO (First In First Out) principle.
2.	Stack has only one pointer points to elements which is always point on top.	The queue has two pointers, Front Pointer which is pointing to the first element and the rear pointer points to the last element.
3.	Insertion operation is called Push.	Insertion operation is called Enqueue.
4.	The deletion operation is called Pop.	The deletion operation is called Deque.
5.	Insertion and deletion in stacks take place only from one place called Top.	Insertion operation takes place at the rear of the queue and deletion takes place at the front of the queue.

352.
What is the difference between peek() and poll() method of Queue?

	Peek()	**Poll()**
1.	Only return the head element of the queue without removing it.	Only return the head element of the queue and removed it as well.

353.
What is the difference between HashSet and LinkedHashSet?

	HashSet	**LinkedHashSet**
1.	HashSet implements using a HashTable.	LinkedHashSet implements using LinkedList + HashSet.
2.	Insertion order never preserved.	LinkedHashSet insertion order preserved.
3.	Unique elements without insertion order.	Unique elements with insertion order.
4.	Introduced in Java 1.2	Introduced in Java 1.4

354.
What is the difference between HashSet and SortedSet?

	HashSet	**SortedSet**
1.	HashSet is a concrete class.	SortedSet is an interface.
2.	HashSet doesn't preserve insertion order or doesn't insert order according to some type of sorting.	SortedSets implementation classes insert the records according to some natural sorting order.

3.	The underlying data structure is HashTable.	The underlying data structure is a RED-BLACK tree.
4.	The complexity of HashSet is O(1) meaning it will do basic operations independent of the size of input data in a constant time.	The complexity of SortedSet is log(N) meaning depending on the size of input it will do the basic operations logarithmic.

355.
What is the difference between HashSet and TreeSet?

	HashSet	**TreeSet**
1.	The data Structure of HashSet is HashTable.	Data Structure of TreeSet is a Balanced Tree.
2.	Heterogeneous (Different types of objects) objects are allowed.	Heterogeneous (Different types of objects) objects are not allowed.
3.	HashSet doesn't maintain any sorting order.	TreeSet maintains natural sorting order.

356.
What is the difference between SortedSet and TreeSet?

	SortedSet	**TreeSet**
1.	SortedSet is an interface.	TreeSet is a concrete class.
2.	SortedSet is an interface so cannot be instantiated.	TreeSet can be instantiated.

357.
What is the difference between HashMap and LinkedHashMap?

	HashMap	**LinkedHashMap**
1.	The Data Structure of HashMap is HashTable.	The Data Structure of LinkedHashMap is based on LinkedList + HashTable.
2.	HashMap doesn't preserve the Insertion order.	LinkedHashMap preserves the insertion order.
3.	Introduced in Java 1.2	Introduced in Java1.4

358.
What is the difference between HashMap and SortedMap?

	HashMap	**SortedMap**
1.	HashMap is a concrete Class.	SortedMap is an interface.
2.	HashMap never preserves the insertion order.	SortedMap implementation class preserves the natural sorting order.

359.
What is the difference between HashMap and TreeMap?

	HashMap	**TreeMap**
1.	The underlying data structure of HashMap is HashTable.	The underlying data structure of TreeMap is a RED-BLACK tree.
2.	HashMap allows only one null key and multiple null values.	TreeMap doesn't allow a single null key but allows multiple null values.
3.	HashMap doesn't maintain any order.	TreeMap maintains natural sorting order which is ascending.
4.	HashMap allows heterogeneous objects.	TreeMap allows only homogeneous objects because TreeMap inserts the elements

		according to some natural sorting order.
5.	Performance of HashMap is higher than TreeMap because it provides constant-time performance that is O(1) for basic operation.	The performance of TreeMap is lower than HashMap because it provides the performance of O(log(n)) for most operations.
6.	HashMap uses equals() to compare the keys.	TreeMap uses the compareTo() method to compare the keys.
7.	HashMap implements Map interface.	TreeMap implements NavigableMap interface.
8.	HashMap provides some basic methods for operation.	TreeMap provides some special methods for operation like firstKey(), lastKey(), tailMap() etc.
9.	HashMap is the best option when you don't want the elements in some type of sorting order.	Hashtable is the best option when you want the elements in some type of sorting order.

360.
What is the difference between TreeMap and SortedMap?

	TreeMap	SortedMap
1.	TreeMap is a concrete class.	SortedMap is an interface.

361.
What is the difference between HashMap and HashTable?

	HashMap	HashTable
1.	All methods of HashMap are not synchronized.	All methods of HashTable are synchronized.
2.	HashMap is not thread-safe.	Hashtable is thread-safe.
3.	The performance of HashMap is relatively high as compared to Hashtable.	The performance of Hashtable is relatively low as compared to HashMap.
4.	Only one NULL key is allowed and multiple NULL values are allowed.	Null is not allowed either key or value otherwise it will throw NullPointerException.
5.	HashMap is not a legacy class.	Hashtable is a legacy class.
6.	HashMap introduced in Java 1.2 version.	Hashtable was introduced in Java 1.0 version.

362.
What is the difference between Fail-Fast and Fail-Safe?

	Fail-Fast	Fail-Safe
1.	While traversing the collection object if someone is trying to modify the value of the collection object, the iterator gets failed immediately which is called Fail-Fast.	While traversing the collections object if someone is trying to modify the value of the collection object, the iterator never failed immediately which is called Fail-Safe.
2.	It throws ConcurrentModificationException while modifying the collection object during the iteration process.	It doesn't throw ConcurrentModificationException while modifying the collection object during the iteration process.
3.	Performance is higher as compared to Fail-safe iterator.	Performance is low as compared to Fail-fast iterator.

4.	Memory utilization of Fail-Fast iterator is lower.	Memory utilization of Fail-Safe iterator is higher.
5.	E.g. ArrayList, LinkedList, HashSet.	E.g. ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet.

363.
What is the difference between Iterator and ListIterator?

	Iterator	ListIterator
1.	Iterator is universal iterator.	ListIterator is not universal iterator.
2.	Iterator can iterate all collection objects.	ListIterator can iterate only the list.
3.	Iterator moves only forwards direction.	ListIterator can move bidirectionally.
4.	Iterator can perform only read and remove operations.	ListIterator can perform read, write, update and remove operations.
5.	Iterator has 3 methods for iteration. 1. hasNext(); 2. next(); 3. remove();	ListIterator has 9 methods. 1. hasNext(); 2. next(); 3. nextIndex(); 4. hasPrevious(); 5. previous(); 6. previousIndex(); 7. remove(); 8. add(); 9. set();

364.
What is the difference between Iterator and Enumarator?

	Iterator	Enumerator
1.	Iterator is applicable to collection objects which not legacy classes.	An enumerator is applicable to all legacy classes which are released Java 1.0.
2.	Iterator can perform read-only and remove operations.	The enumerator can perform only read-only operations.
3.	Iterator has three methods for iteration. 1. hasNext(); 2. next(); 3. remove();	Enumeration has two methods. 1. HasMoreElement(); 2. nextElement();

365.
What is the difference between Iterator and Iterable?

	Iterator	Iterable
1.	Iterator has three methods to iterate. 1. hasNext(); 2. next(); 3. remove();	Iterable has only one method. 1. Iterator();
2.	Return type of 1. hasNext() method is Boolean.	Return type of iterator() method is Iterator<T> (T for Type).

	2. Next() method is E. 3. Remove method is void.	

366.
What is the Iterator and why do we require iterator in Collection framework?

- Basically, Iterator is an interface.
- Iterator can be used to traverse throughout the collection object.
- Iterator is a universal iterator that can apply to all types of collection like list, set, map.
- The iterator has three methods to perform the operation.

1. hasNext();
2. next();
3. remove();

367.
What is the ListIterators and why do we require ListIterators in Collection framework?

- ListIterator is a child interface of the Iterator interface.
- ListIterator is an iterator can be applicable to list
- List iterator can move bidirectionally.
- There are 9 methods present inside ListIterator which are as follows.

1. hasNext();
2. next();
3. nextIndex();
4. hasPrevious();
5. previous();
6. previousIndex();
7. remove();
8. add();
9. set();

```
package simplifiedjava.crackedInterview;
import java.util.ArrayList;
import java.util.ListIterator;

public class ListIteratorDemo {
		public static void main(String[] args) {
		ArrayList<Integer> list = new ArrayList<Integer>();
		list.add(10);
		list.add(20);
		list.add(30);
		list.add(40);
		list.add(50);
		ListIterator<Integer> itr = list.listIterator();
		while(itr.hasNext()) {
		int no = itr.next();
		System.out.println(no);
		boolean result = itr.hasNext();
		System.out.println(result);
		}
```

```
	}
}
Output:
10
true
20
true
30
true
40
true
50
false
```

368.
What is the Enumarator and why do we require Enumarator in Collection framework?

- Enumeration can be used to iterate the legacy classes.
- Enumeration is also a legacy interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Enumeration;
import java.util.Vector;

public class EnumerationDemo {

	public static void main(String[] args) {
	Vector<Integer> v = new Vector<Integer>();
	v.addElement(10);
	v.addElement(20);
	v.addElement(30);
	v.addElement(40);
	Enumeration<Integer> enu = v.elements();
	while(enu.hasMoreElements()) {
	int element = enu.nextElement();
	System.out.println(element);
	}
	}
}
Output:
10
20
30
40
```

Interview Questions on Concurrent Collection.

369.

What is Concurrency? What is the need of Concurrency?

- Concurrency is the ability to run multiple programs at a time.
- For concurrency threads are responsible.
- Concurrency package provides more flexible classes than synchronized methods or blocks.
- Concurrency belongs to java.util.Concurrent package.
- Needs of Concurrency are as follows.

1. Traditional collection object ArrayList, LinkedList, HashSet and Hashmap access by multiple threads so data inconsistency problem may occur hence, these are not thread-safe. But, Concurrency classes are thread-safe.
2. Already existing thread-safe collection Vector, Stack, SynchronizedMap, SynchronizedSet are not good performance-wise. But, Concurrent package classes are very good performance-wise.
3. Only one thread can execute thread-safe objects so other threads must wait until the first thread completes its task so performance goes down. Concurrent classes overcome these problems.
4. Another big problem is, when one thread is iterating collection objects, the other thread can not modify the same collection object. When the first tread is iterating the collection object at the same time another thread is trying to modify the collection object then we will get ConcurrentModificationException.

370.
What is the difference between Traditional Collection and Concurrent Collection?

	Traditional Collection	Concurrent Collection
1.	All traditional collections are not thread-safe.	All Concurrent collections are thread-safe.
2.	Thread-safe traditional collections performance is low as compared to Concurrent collections.	Concurrent Collections performance is high as compared to the traditional collection.
3.	While one thread is executing the collection object no other thread is allowed to access the same collection object.	In Concurrent collection, while one thread is executing a collection object another thread is allowed to access the same collection object.
4.	ConcurrentModificationException can be thrown when multiple threads execute the same resources.	ConcurrentModificationException cannot occur when multiple threads executing the same resources.
5.	Traditional Thread-safe classes are follows. 1. Vector 2. Stack 3. Hashtable 4. Properties	Concurrent collections are as follows. 1. ConcurrentHashMap 2. CopyOnWriteArrayList 3. CopyOnWriteArraySet

371.
What is ConcurrentHashMap? What are the Characteristics of ConcurrentHashMap?

- ConcurrentHashMap is the concurrent class with key and value pairs belonging to java.util.Concurrent package.
- ConcurrentHashMap is a concurrent version of HashMap. We can say it is a thread-safe HashMap.

- Characteristics of ConcurrentHashMap are as follows.

  1. Characteristics of ConcurrentHashMap are as follows.
  2. The underlying data structure is Hashtable.
  3. Null is not allowed either key or value.
  4. ConcurrentHashMap allows any number of reading operations at a time and only 16 write operations at a time when the concurrency level is 16.
  5. Instead of getting a whole map object lock concurrency divides map objects into 16 parts because the default size of ConcurrentHashMap is 16.
  6. To perform read operation would not require any type of lock but for write operation thread must get Bucket level lock or Segment level lock.
  7. ConcurrentHashMap allows concurrent read and thread-safe write operations.
  8. While one thread performs read operations other 16 threads can perform write operations at a time.
  9. ConcurrentHashMap never throw ConcurrentModificationException.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapDemo {

		public static void main(String[] args) {
		ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
		map.put(101, "Yogesh");
		map.put(102, "Arpita");
		map.putIfAbsent(103, "Shweta");
		System.out.println(map);
		map.putIfAbsent(102, "Shruti");
		map.putIfAbsent(104, "Shivani");
		System.out.println(map);
		map.replace(104, "Prajakta");
		System.out.println(map);
		map.remove(101);
		System.out.println(map);
		}
}
Output:
{101=Yogesh, 102=Arpita, 103=Shweta}
{101=Yogesh, 102=Arpita, 103=Shweta, 104=Shivani}
{101=Yogesh, 102=Arpita, 103=Shweta, 104=Prajakta}
{102=Arpita, 103=Shweta, 104=Prajakta}
```

372.
What do you mean by Concurrency Level?

- Concurrency divided map objects into 16 parts that smaller parts are known as Concurrency level.

373.
What is the default concurrency level of ConcurrentHashMap?

- Default concurrency level is 16 because the default size of the HashMap is 16.

374.
What is the difference between HashMap and ConcurrentHashMap?

	HashMap	ConcurrentHashMap
1.	HashMap is not thread-safe.	ConcurrentHashMap is thread-safe.
2.	Only one null key and multiple null values are allowed.	ConcurrentHashMap doesn't allow null keys or values either. It will throw NullPointerException at runtime.
3.	The performance of HashMap is high as compared to ConcurentHashMap because multiple threads can access the same object.	The performance of ConcurentHashMap is relatively low because it is thread-safe.
4.	Thread gets a lock of the whole object and then execute.	In ConcurrentHashMap thread won't take whole objects lock, Thread will get Segment lock or bucket lock.
5.	If multiple threads are trying to perform write or read operations on the same collection object then we will get ConcurrentModifcationException.	Multiple threads are allowed to perform a read operation and 16 or more threads are allowed to perform the write operation.
6.	HashMap has only one lock available.	ConcurrentHashMap can have one or multiple locks we call it a Concurrency level.
7.	Iterator is Fail-Fast.	Iterator is Fail-Safe.
8.	When multiple threads access the same resource at a time then iterator fails immediately is called Fail-Fast.	When multiple threads access the same resources at a time, Iterator won't fail is called Fail-Safe Iterator.

375.
What is the difference between HashTable and ConcurrentHashMap?

	HashTable	ConcurrentHashMap
1.	Thread will get whole object lock.	Thread will get a bucket level or Setment level lock instead of whole object lock.
2.	Only one thread is allowed to perform thread safe operation.	Multiple threads are allowed to perform thread-safe operation at a time.
3.	Every read and write operation require object level lock.	Read operation can perform without lock and write operation can perform by getting bucket level or Segment level lock.
4.	When one thread is performing read operation no other threads are allowed to perform any operation on same object otherwise you will get ConcurrentModificationException.	When one thread is performing read operation other threads are allowed to perform write operation on same object still you won't get ConcurrentModificationException.
5.	Iterator is Fail-Fast	Iterator is Fail-Safe.
6.	Introduced in Java 1.0	Introduced in Java 1.5.

376.

What is the difference between SynchronizedHashMap and ConcurrentHashMap?

	SynchronizedHashMap	ConcurrentHashMap
1.	The thread will get the whole object locked.	The thread will get a bucket level or Segment level lock instead of a whole object lock.
2.	Only one thread is allowed to perform the thread-safe operation.	Multiple threads are allowed to perform the thread-safe operation at a time.
3.	Every read and write operation require a full object lock.	Read operation can perform without lock and write operation can perform by getting bucket level or Segment level lock.
4.	When one thread is performing a read operation no other thread is allowed to perform any operation otherwise we will get ConcurrentModificationException.	When one thread is performing a read operation other threads are allowed to perform a write operation on the same object still you won't get ConcurrentModificationException.
5.	Iterator is Fail-Fast.	Iterator is Fail-Safe.
6.	Introduced in Java 1.2	Introduced in Java 1.5.

377.
What the the new methods introduced in ConcurrentHashMap to performing read and write operation?

- There are 3 methods has introduced in ConcurrentHashMap which are as follows.

1. **putIfAbsent():** If the key is present then there won't be any change in collection.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapPutIfAbsentDemo {

        public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
        map.put(101,"Shweta");
        map.put(102,"Shruti");
        map.put(103,"Yogesh");
        map.putIfAbsent(103,"Arpita"); // In this case 103 key was present
        System.out.println(map);
        map.putIfAbsent(104,"Ananya"); // In this case 104 key was not present.
        System.out.println(map);
        }
}
Output:
{101=Shweta, 102=Shruti, 103=Yogesh}
{101=Shweta, 102=Shruti, 103=Yogesh, 104=Ananya}
```

2. **remove():** This method won't remove elements until and unless key and value match.

```
package simplifiedjava.crackedInterview;
```

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapRemoveDemo {

		public static void main(String[] args) {
		ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
		map.put(101,"Shweta");
		map.put(102,"Shruti");
		map.put(103,"Yogesh");
		map.remove(103,"Arpita"); // In this case 103 key is present but value was present.
		System.out.println(map);
		map.remove(103,"Yogesh"); // In this case key 103 and value Yogesh was present so removed.
		System.out.println(map);
		}
}

Output:
{101=Shweta, 102=Shruti, 103=Yogesh}
{101=Shweta, 102=Shruti}
```

3. **replace():** This method won't replace value until and unless key and value match.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapReplaceDemo {

		public static void main(String[] args) {
		ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
		map.put(101,"Shweta");
		map.put(102,"Shruti");
		map.put(103,"Yogesh");
		map.replace(103,"Arpita");
		System.out.println(map);
		map.replace(103,"Arpita","Updated Arpita");
		System.out.println(map);
		}
}
Output:
{101=Shweta, 102=Shruti, 103=Arpita}
{101=Shweta, 102=Shruti, 103=Updated Arpita}
```

378.
What is CopyOnWriteArrayList? How does it work internally?

- CopyOnWriteArrayList is Concurrent implementation of List interface.
- CopyOnWriteArrayList implements List interface so, all basic functionality of List is applicable to CopyOnWriteArrayList.
- It is a very costly object because every update operation creates a clone copy of the same object.
- When one thread is iterating CopyOnWriteArrayList same time another thread is allowed to perform writes operation still we won't get ConcurrentModificationException.
- Iterator of ArrayList performs remove operation. Iterator of CopyOnWriteArrayList can not perform the remove operation. If we try to remove then it will throw UnSupportedOperationException.

- Iterator of CopyOnWriteArrayList is Fail-Safe.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListDemo {

		public static void main(String[] args) {
		CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<Integer>();
		list.add(10);
		list.add(20);
		list.add(30);
		list.add(40);
		list.add(50);
		System.out.println(list);
		}
}
Output:  [10, 20, 30, 40, 50]
```

379.
What is ConcurrentModificationException?

- When one thread is performing a read operation other threads are not allowed to perform any operation on the same object still you are attempting the same then it will throw ConcurrentModificationException.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class ConcurrentModificationExceptionDemo {

		public static void main(String[] args) {
		HashMap<Integer,String> m = new HashMap<Integer,String>();
		m.put(101,"Shweta");
		m.put(102,"Shruti");
		Iterator itr = m.entrySet().iterator();
		while(itr.hasNext()) {
		m.put(103, "Anjali");
		Map.Entry<Integer, String> pair = (Map.Entry<Integer, 
String>)itr.next();
		System.out.println("Key = "+ pair.getKey()+ "\t Value = "+ pair.getValue());
		}
		}
}
Output:
Exception in thread "main" java.util.ConcurrentModificationException
		at java.util.HashMap$HashIterator.nextNode(Unknown Source)
		at java.util.HashMap$EntryIterator.next(Unknown Source)
		at java.util.HashMap$EntryIterator.next(Unknown Source)
		at 
simplifiedjava.crackedInterview.ConcurrentModificationExceptionDemo.main(ConcurrentModificationExceptionDemo.jav
17)
```

380.
Will it throw ConcurrentModificationException while iterating ConcurrentHashMap? If No Why?

- While iterating ConcurrentHashMap it won't throw ConcurrentModificationException.
- Thread will get a bucket level or Segment level lock instead of a whole object lock.
- Multiple threads are allowed to perform a thread-safe operation at a time.
- Read operation can perform without lock and the write operation can perform by getting bucket level or Segment level lock.
- When one thread is performing a read operation other threads are allowed to perform a write operation on the same object still you won't get ConcurrentModificationException.
- Please refer to the below example. Please cross verify with the above example as well.

```
package simplifiedjava.crackedInterview;

import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentModificationExceptionDemo {

		public static void main(String[] args) {
		ConcurrentHashMap<Integer,String> m = new ConcurrentHashMap<Integer,String>();
		m.put(101,"Shweta");
		m.put(102,"Shruti");
		Iterator itr = m.entrySet().iterator();
		while(itr.hasNext()) {
		m.put(103, "Anjali");
		Map.Entry<Integer, String> pair = (Map.Entry<Integer,
String>)itr.next();
		System.out.println("Key = "+ pair.getKey()+ "\t Value = "+ pair.getValue());
		}
		}
}
Output:
Key = 101	Value = Shweta
Key = 102	Value = Shruti
Key = 103	Value = Anjali
```

381.
What will happen while iterating any collection object including map trying to perform some operations like read or write on the object?

- It will throw ConcurrentModificationException.
- For more details, you may refer last 5 questions.

382.
Iterator of Concurrent Collection object is Fail Safe or Fail Fast?

- Iterators of Concurrent Collections are Fail-Safe.
- Fail-safe means iterator doesn't fail if you are trying to modify the element at the time of iteration.
- The Iterator will not fail while updating Concurrent Collection objects at the time of iteration.

383.

Can I perform remove operation while iterating CopyOnWriteArrayList object?

- We cannot perform the remove operation while iterating the CopyOnWriteArraylist object.
- It is possible for List implementations.

384.
What will happen if I am trying to perform remove operation while iterating CopyOnWriteArrayList?

- If we are trying to perform a remove operation while iterating CopyOnWriteArrayList, it will throw UnSupportedOperationException.
- Please refer to the bleow example.

```
package simplifiedjava.crackedInterview;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListDemo {

	public static void main(String[] args) {
	CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<Integer>();
	list.add(10);
	list.add(20);
	list.add(30);
	list.add(40);
	list.add(50);
	Iterator itr = list.iterator();
	while(itr.hasNext()) {
	System.out.println(itr.next());
	itr.remove();
	}
	}
}
Output:
10
Exception in thread "main" java.lang.UnsupportedOperationException
	at java.util.concurrent.CopyOnWriteArrayList$COWIterator.remove(Unknown Source)
		at
simplifiedjava.crackedInterview.CopyOnWriteArrayListDemo.main(CopyOnWriteArrayListDemo.java:19)
```

385.
What is the difference between ArrayList and CopyOnWriteArrayList?

	ArrayList	CopyOnWriteArrayList
1.	ArrayList is not thread-safe.	CopyOnWriteArrayList is thread-safe.
2.	In ArrayList, clone copy doesn't create for every update operation.	In CopyOnWriteArrayList clone copy creates for each update operation.
3.	While executing one thread no other thread allowed accessing the same resources. If you	While executing one thread other threads are allowed to perform an operation on the same

	try to modify then it will throw ConcurrentModificationException.	resources it won't throw ConcurrentModificationException.
4.	Iterator is Fail-Fast.	Iterator is Fail-Safe.
5.	While traversing the list using iterator we can perform a remove operation.	While traversing CopyOnWriteArrayList using iterator we cannot perform removes operation otherwise, it will throw UnSupportedOperationException.

386. What is the difference between CopyOnWriteArrayList and SynchronizedList?

	CopyOnWriteArrayList	SynchronizedList
1.	A new clone copy will be created for every update operation.	A new clone copy won't create for every update operation.
2.	Multiple threads are allowed to perform read/write operations at a time.	Only one thread is allowed to perform read/write operations.
3.	Iterator is Fail-Safe.	Iterator is Fail-Fast.
4.	Never throw ConcurrentModificationException.	Possible to throw ConcurrentModification Exception.
5.	While iterating CopyOnWriteArrayList by iterator we cannot perform remove () operation.	While iterating Synchronized list by iterator we can perform remove () operation.

387. What the the new methods introduced in CopyOnWriteArrayList to performing read and write operation?

- CopyOnWriteArrayList is is concurrent version of list implementation.
- All methods of the list will be by default available in CopyOnWriteArrayList.
- There are a couple of methods that have been introduced by CopyOnWriteArrayList.

1. **addIfAbsent():**This method will add an element if and only if an element is not present in the list. Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListDemo {

        public static void main(String[] args) {
        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<Integer>();
        list.add(10);
        list.add(20);
        list.add(30);
        list.addIfAbsent(20);// This element will be ignored because its already in list.
        list.addIfAbsent(40);// This element will be added because it was not in list.
        System.out.println(list);
        }
}
Output:
[10, 20, 30, 40]
```

2. **addAllAbsent():** This method will add all elements if and only if elements are not present in the list. Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListDemo {

		public static void main(String[] args) {
		CopyOnWriteArrayList<Integer> list1 = new CopyOnWriteArrayList<Integer>();
		list1.add(10);
		list1.add(20);
		list1.add(30);
		System.out.println("List 1 = "+ list1);
		CopyOnWriteArrayList<Integer> list2 = new CopyOnWriteArrayList<Integer>();
		list2.add(10);
		list2.add(20);
		list2.add(40);
		System.out.println("List 2 "+ list2);
		list1.addAllAbsent(list2);
		System.out.println("Updated List = "+ list1);
		}
}
Output:
List 1 = [10, 20, 30]
List 2 [10, 20, 40]
Updated List = [10, 20, 30, 40]
```

388.
What is CopyOnWriteArraySet? How does it work internally?

- It is a thread-safe version of Set.
- Internally implemented with CopyOnWriteArrayList.
- Insertion order is always preserved.
- Duplicates are not allowed.
- Null is allowed.
- Multiple threads can perform read operation without lock but for every update, a clone copy will be created.
- When one thread is iterating a set another thread is allowed to access the same resources still we won't get ConcurrentModificationException.
- You can't perform the remove operation while traversing the set object using an iterator otherwise it will throw UnSupportedOperationException.
- Iterator is Fail-safe.

389.
Can CopyOnWriteArraySet preserve insertion order?

- Yes, CopyOnWriteArraySet preserves the insertion order.

Interview Questions on Generics.

390.
What is Generics? Can you list out some advantages of Generics?

- Generic provides type safety and resolve type casting problems.
- Advantages of Generics.

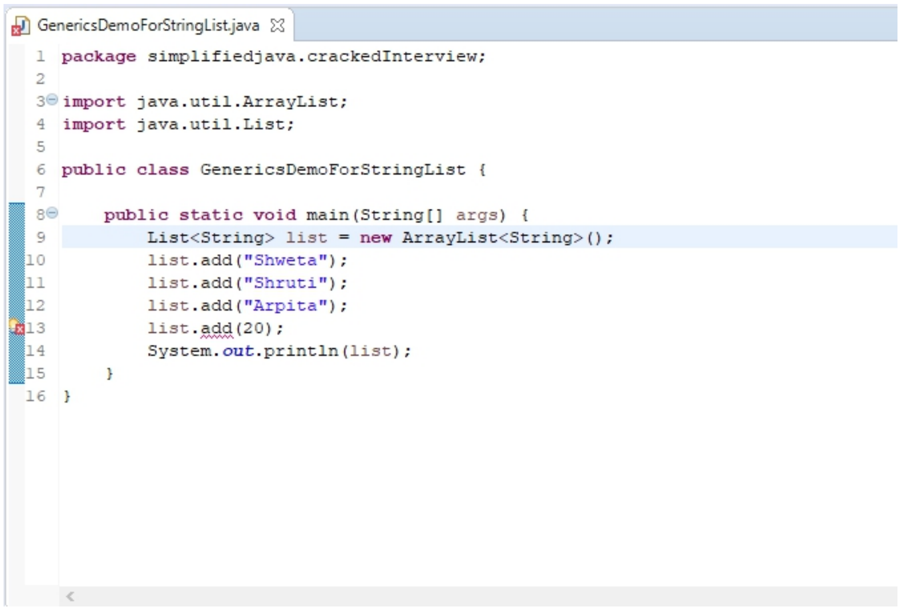
1. No explicit type casting is required.
2. Reduce chances to throw ClassCastException.
3. With the help of generics, we can identify which type of list is there.
4. If you are trying to add a heterogeneous object then you will get a compile-time error.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class GenericsDemoForStringList {

        public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Shweta");
        list.add("Shruti");
        list.add("Arpita");
        System.out.println(list);
        }
}
```

Output: [Shweta, Shruti, Arpita]

If you are trying to add other thanString you will get compile time error.



GenericsDemoForStringList.java

```
package simplifiedjava.crackedInterview;

import java.util.ArrayList;
import java.util.List;

public class GenericsDemoForStringList {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Shweta");
        list.add("Shruti");
        list.add("Arpita");
        list.add(20);
        System.out.println(list);
    }
}
```

391.
What is Generic Type Parameter?

- Type parameter is an identifier that specifies a generic type name.
- Type parameter is also known as the Type variable.
- Type parameter can be used to declare the return type.
- Please refer to the syntax.

```
class ArrayList<T>{
        add(T t);
        T get(int index);
}

class ArrayList<String>{
        add(String s);
        String get(int index);
}
```

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;
public class GenericTemplate<T> {

        T obj;
        public GenericTemplate(T obj) {
        this.obj = obj;
        }
        public T getobj() {
        return obj;
        }
        public void show() {
        System.out.println("Type of Object "+ obj.getClass().getName());
        }
}

package simplifiedjava.crackedInterview;

public class GenericDemo {

        public static void main(String[] args) {
        GenericTemplate<String> str = new GenericTemplate<String>("Yogesh");
        str.show();
        System.out.println(str.getobj());
        GenericTemplate<Integer> numbers = new GenericTemplate<Integer>(100);
        numbers.show();
        System.out.println(numbers.getobj());
        GenericTemplate<Double> dblNum = new GenericTemplate<Double>(25.50);
        dblNum.show();
        System.out.println(dblNum.getobj());
        }
}

Output:
Type of Object java.lang.String
Yogesh
Type of Object java.lang.Integer
100
Type of Object java.lang.Double
25.5
```

What is bounded type parameter?

- We can bound type parameters to the particular range by using extends keyword such types are called bounded types.
- Please refer to the below example.

```
class Test <T extends Number>{
}
```

393.
What is type interface?

- Type safety means that the compiler will validate types while compiling, and throw an error if you try to assign the wrong type to a variable.

394.
What is type Erasure?

- Generic provides compile-time java files all generics syntax will be removed.
- At runtime generics syntax that won't be available is called Type Erasure.

395.
What is the difference between Generic type and Generic method?

- Upper-bound is when you specify **(? extends Field**) means argument can be any Field or subclass of Field.
- Lower-bound is when you specify **(? super Field**) means argument can be any Field or superclass of Field.

396.
What is unbounded wildcard type?

- The unbounded wildcard type is specified using the wildcard character (?).
- For example, List<?>. This is called a list of unknown types.
- There are two scenarios where an unbounded wildcard is a useful approach:

  1. If you are writing a method that can be implemented using functionality provided in the Object class.
  2. When the code is using methods in the generic class that doesn't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.util.Arrays;
import java.util.List;

public class UnboundedWildCardDemo {

        public static void main(String[] args) {
        List<Integer> li = Arrays.asList(1, 2, 3);
        List<String>  ls = Arrays.asList("one", "two", "three");
        printList(li);
        printList(ls);
        }
        public static void printList(List<?> list) {
                for (Object elem: list)
                    System.out.print(elem + " ");
                System.out.println();
            }
}
Output:
1 2 3
one two three
```

397.
What is difference between List<? extends T> and List <? super T> ?

- List<? extends T> - Example of Lower Bound.
- List <? super T> - Example of Upper Bound.

398.
How will you crate a parameterized class using generics?

- Please refer to the below example.

```
package simplifiedjava.crackedInterview;


public class GenericTemplate<T> {

        T obj;
        public GenericTemplate(T obj) {
        this.obj = obj;
        }
        public T getobj() {
        return obj;
        }
        public void show() {
        System.out.println("Type of Object "+ obj.getClass().getName());
        }
}

package simplifiedjava.crackedInterview;

public class GenericDemo {

        public static void main(String[] args) {
        GenericTemplate<String> str = new GenericTemplate<String>("Yogesh");
        str.show();
        System.out.println(str.getobj());
        GenericTemplate<Integer> numbers = new GenericTemplate<Integer>(100);
        numbers.show();
        System.out.println(numbers.getobj());
        GenericTemplate<Double> dblNum = new GenericTemplate<Double>(25.50);
        dblNum.show();
        System.out.println(dblNum.getobj());
        }
}
Output:
Type of Object java.lang.String
Yogesh
```

```
Type of Object java.lang.Integer
100
Type of Object java.lang.Double
25.5
```

399.
Will you consider Array is bydefault generics? If yes then why?

- Yes, I will consider Array as by default generic class.
- Array never accepts heterogeneous objects, Array always accepts homogeneous objects.
- If any collection object has the same type of object then it will never through ClassCastException.

400.
What is the difference between ArrayList and ArrayList<?> in Java?

- ArrayList – This version is non-generic version.
- ArrayList<?> - This version is generic version of ArrayList.

401.
Can I add other object than String in following snippet?

ArrayList<String> list = new ArrayList<String>();
- We have declared ArrayList with generic type safety.
- We already declared the list will be only for String. On other objects are allowed to insert it.
- You may refer to question no. 390 for more detail.

402.
List out the valid and invalid combinatins of Generic definition?

1.
class Test<T extends Runnable>
2.
class Test<T extends Runnable & Comparable>
3.
class Test<T extends Number & Runnable & Comparable>
4.
class Test<T extends Runnable & Number>
5.
class Test<T extends Number & Thread>
6.
class Test<T extends Number>
7.
class Test<T implements Runnable>
8.
class Test<T extends Runnable>
9.
class Test<T super String>

- Following are the valid combinations.

Definition	Valid/Invalid
class Test<T extends Runnable>	Valid
class Test<T extends Runnable & Comparable>	Valid

class Test<T extends Number & Runnable & Comparable>	Valid
class Test<T extends Runnable & Number>	Invalid
class Test<T extends Number & Thread>	Invalid
class Test<T extends Number>	Valid
class Test<T implements Runnable>	Invalid
class Test<T extends Runnable>	Valid
class Test<T super String>	Invalid

403.
What is type inference in generics?

- Type inference means we can create a template method for multiple data types.
- In below example printArray()method is Type Inference.
- We can print both arrays.

```
package simplifiedjava.crackedInterview;

public class InferenceTypeDemo {

        public static void main(String[] args) {
        Integer[] intArray = {10,20,30,40,50};
        Character[] charArray = {'A','B','C','D','E'};
        printArray(intArray);
        printArray(charArray);
        }
        public static <E> void printArray(E[] elements){
        System.out.print("{");
        int count = 0;
        for(E element : elements) {
        if(elements.length > count) {
        System.out.print(element + ",");
        }
        ++count;
        }
        System.out.println("}");
        }
}
Output:
{10,20,30,40,50,}
{A,B,C,D,E,}
```

Interview Questions on JVM

404.
What is bytecode?

- Java bytecode is the instruction set for the Java Virtual Machine.
- Bytecode is the intermediate representation of a Java program.

405.
What is JIT compiler?

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

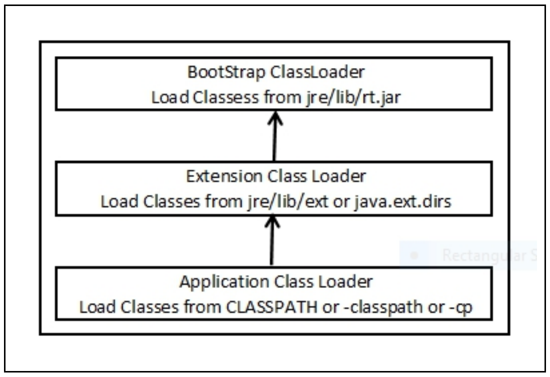
406.
What is the different between Linkers and Loaders?

	Linkers	Loaders
1.	The main functionality of Linkers is to generate the executable file.	The main functionality of Loaders is to load the executable files into the main memory.
2.	The process of combining source code to obtain executable code is called Linkers.	The process of loading executable code to the main memory for further execution is called Loaders.
3.	The linker takes input from the Compiler or assembler.	Loaders take input from Linkers.
4.	There are two types of Linker available. 1. Dynamic Linker. 2. Linkage Editor.	There are four types of loader available. 1. BootStrap Loaders. 2. Absolute Loaders. 3. Direct Linking Loaders. 4. Relocating Loaders.

407.
What is ClassLoaders? What are the types of ClassLoaders available?

- ClassLoader are used to load the classes at run-time in main memory.
- ClassLoader is an abstract class.
- When JVM request for a class to load the class in memory, it invokes loadClass() method which belongs to java.lang.ClassLoader class.
- The loadClass() method internally calls the findLoadedClass() method to verify whether requested class is already loaded or not.
- There are three types of class loaders which are as follows.

1. **BootStrapClassLoader:** It loads JDK class files from jre/lib/rt.jar and other core classes. It is a parent of all class loaders. If we call String.class.getClassLoader() it returns null.
2. **ExtensionsClassLoader:** It delegates the class loading request to its parent. If the loading of the class is unsuccessful, it loads classes from jre/lib/ext directory or any other directory like java.ext.dirs.
3. **SystemClassLoader:** It loads the application classes from CLASSPATH environment variable. It is a child class of Extension class loader.


BootStrap ClassLoader
Load Classess from jre/lib/rt.jar
Extension Class Loader
Load Classes from jre/lib/ext or java.ext.dirs
Application Class Loader
Load Classes from CLASSPATH or -classpath or -cp

408.
What are the PC Registers? What is the role of PC registers in JVM?

- PC stands for Program Counter.
- If that method is not native, the pc register contains the address of the Java Virtual Machine instruction currently being executed.
- If the method currently being executed by the thread is native, the value of the Java Virtual Machine's pc register is undefined.
- The Java Virtual Machine's pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.

409.
What is JVM?

- JVM stands for Java Virtual Machine.
- Implementation of JVM is JRE(Java Runtime Environment).
- JVM is responsible to manage memory, creating an object, destroying objects, instantiating classes etc.

410.
What is the difference between JDK, JRE and JVM?

	JDK	**JRE**	**JVM**
1.	JDK stands for Java Development Kit.	JRE stands for Java Runtime Environment.	JVM stands for Java Virtual Machine.
2.	JDK is superset of JRE.	JRE is a subset of JDK.	JVM is a subset of JRE
3.	JDK is a software development kit to develop applications in java.	It is a software bundle that provides java class libraries with the necessary components to run java code.	JVM executes java byte code and provides an environment for executing it.
4.	JDK enables developers to create a java program that can be executed and run by the JRE and JVM.	JRE is a part of java that creates the JVM.	It is the java platform component that executes source code.
5.	JDK containes the tools to debugging, developing and monitoring the code.	It contains the class libraries and other supporting files that JVM requires to execute the program.	Software development tools are not included in JVM.

411.
How JVM reclaims memory?

- Memory reclaims means simply deleting the unreferenced objects.
- JVM identifies the unreferenced objects and marks them.
- After marking the objects JVM runs finalize() method to release the resources acquired by unreferenced objects.
- Once releases all the resources acquired by unreferenced objects JVM simply swipes them from memory.

412.
Can we trigger the garbage collection from java code?

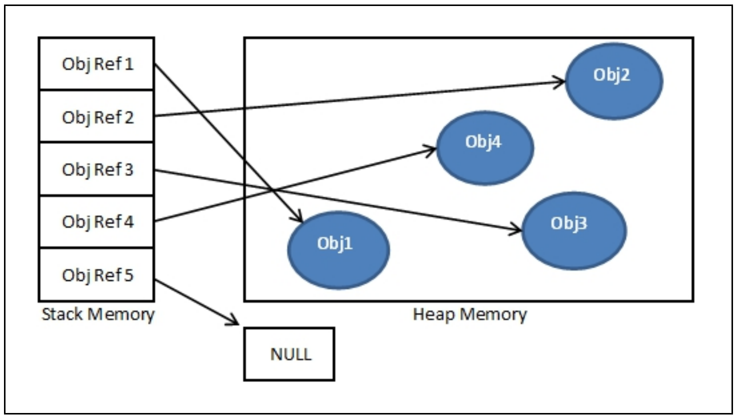
- Yes, we can just recommend to JVM for garbage collections. But we cannot force JVM.
- We can recommend invoking the System.gc() method. But, it's not guaranteed garbage collectors will execute.

413.
What is the permsize? What is the importance of permsize? Where we can modify the permsize?

- Permsize can be defined as Permanent Space.
- Permsize is an initial value of permanent space which is allocated at JVM startup.
- java -XX:PermSize=512 MyProgram.

414.
How Stack and Heap correlated to each other while creating object?

- JVM has bifurcated memory into two parts. The first one is stack memory and the second one is heap memory.
- Stack memory is mainly used for storing the order of method execution and local variables.
- Heap memory is mainly used for storing actual objects created with a new keyword.
- Stack and heap memory are correlated while managing the objects life cycle. Once created object on heap memory will be referenced to the reference variable which is stored on stack memory.
- Please refer to the below diagram.



415.
What is the difference between Stack Memory and Heap Memory?

	Stack Memory	Heap Memory
1.	Stack memory is mainly used for storing the order of method execution and local variables.	Heap memory is mainly used for storing actual objects created with a new keyword.

2.	Stack memory works with the LIFO mechanism.	There is no mechanism to allocate and deallocate the memory in heap.
3.	Stack variables are visible only to the owner thread.	Objects are visible to all threads.
4.	We can increase stack memory size by using –XSS parameter.	We can modify the size of heap memory by using Xms and –Xmx.
5.	If the stack gets full and still trying to add more elements in it then it will throw StackOverFlowError.	If JVM gets full then it will throw OutOfMemoryError.

416.
What the Strong reference, Weak reference, phantom reference and soft reference?

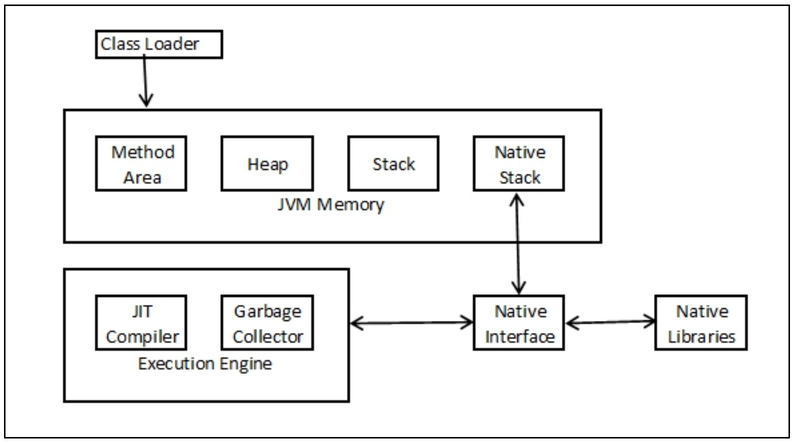
- **Strong Reference** - is a kind of reference, which makes the referenced object not eligible for GC. builder classes. eg - StringBuilder
- **Weak Reference** - is a reference that is eligible for GC.
- **Soft Reference** - is a kind of reference whose object is eligible for GC until memory is available. Best for image cache. It will hold them till the memory is available.
- **Phantom Reference** - is a kind of reference whose object is directly eligible for GC. Used only to know when an object is removed from memory.

417.
What are the steps JVM needs to take while loading class in memory?

- **Load** - The class file is loaded to JVM memory.
- **Verify** - Verifies that the class follows the Java language and JVM specifications.
- **Resolve** - Change symbolic references to direct references.
- **Initialize** - Initializes class variable and static variables.

418.
What are the components of JVM or Explain Java Architecture?

- Please refer to the below diagram first.
- JVM's prime responsibility is to convert byte code into machine code.
- JVM load the code into memory and verifies it and execute the code and provides a run-time environment.
- Please refer to the below diagram for java architecture.



- Following are the details of JVM components.

1. **ClassLoader:** ClassLoader is used to load the classes in the main memory. JVMs primary task is to load the classes into memory.
2. **Method Area:** In the method area class data is stored during the code's execution. Class method area holds the information of static variables, static methods, static blocks and instance methods.
3. **Heap:** Heap memory is mainly used for storing actual objects created with a new keyword. There is no mechanism to allocate and deallocate the memory in heap. We can modify the size of heap memory by using Xms and –Xmx. If JVM gets full then it will throw OutOfMemoryError.
4. **Stack:** Stack memory is mainly used for storing the order of method execution and local variables. Stack memory works with the LIFO mechanism. We can increase stack memory size by using –XSS parameter. If the stack gets full and still trying to add more elements in it then it will throw StackOverFlowError.
5. **Native Stack:** This contains the data about native methods. The Methods which are written other than java language are called Native methods.
6. **Execution Engine:** This is the main part of JVM. Its main task is to execute the byte code and execute the java classes. There are three components in the Execution engine.
   - **Interpreter:** The interpreter sequentially executes the code. The interpreter interprets the code continuously line by line.
   - **JIT Compiler:** JIT stands for Just-In-time compiler. The performance of the JIT compiler is superior to Interpreter. We can replace the JIT compiler with an Interpreter.
   - **Garbage Collector:** Garbage collector is used to removing the unreferenced objects and reclaim the memory.
7. **Native Interface:** Native interface is a mediator between java methods and native libraries.

Miscellaneous Interview Questions.

419.
What is Singleton Class? Write a program for the same?

- Singleton means only one object can be created.
- Following are the steps that need to take to create a Singleton class.

1. Create a class variable with private and static.
2. Create a private constructor.

- Please refer to the below example.

1. Lazy Loading

```
package simplifiedjava.crackedInterview;

public class Singleton {
	private static Singleton INSTANCE;
	private Singleton() {
	}
	public static Singleton getInstance() {
	if(INSTANCE == null) {
	INSTANCE = new Singleton();
	}
```

```
		return INSTANCE;
		}
		public void m1() {
		System.out.println("m1() method called.");
		}
}

package simplifiedjava.crackedInterview;

public class SingletonDemo {

		public static void main(String[] args) {
		Singleton obj = Singleton.getInstance();
		System.out.println(obj1.hashCode());
		System.out.println(obj2.hashCode());
		obj1.m1();
		}
}
Output:
2018699554
2018699554
m1() method called.
```

2. Eagar loading.

```
package simplifiedjava.crackedInterview;

public class Singleton {
		private static Singleton INSTANCE = new Singleton();
		private Singleton() {
		}
		public static Singleton getInstance() {
		if(INSTANCE != null) {
		return INSTANCE;
		}
		return null;
		}
		public void m1() {
		System.out.println("m1() method called.");
		}
}

package simplifiedjava.crackedInterview;

public class SingletonDemo {

		public static void main(String[] args) {
		Singleton obj = Singleton.getInstance();
		System.out.println(obj1.hashCode());
		System.out.println(obj2.hashCode());
		obj1.m1();
		obj.m1();
		}
}
Output:
2018699554
2018699554
m1() method called.
```

420.
Can we break Singleton? How many ways we can break the singleton?

- Yes, we can break the singleton.
- There are three ways we can break the Singleton.

1. Using Reflection API.
2. Using Deserialization.
3. Using Cloning.

421.
WAP to break Singleton using Reflection? How to prevent Singleton to break with Reflection?

```
package simplifiedjava.crackedInterview;

public class Singleton {
        private static Singleton INSTANCE = new Singleton();
        private Singleton() {
        }
        public static Singleton getInstance() {
        if(INSTANCE != null) {
        return INSTANCE;
        }
        return null;
        }

        public void m1() {
        System.out.println("m1() method called.");
        }
}

package simplifiedjava.crackedInterview;

import java.lang.reflect.Constructor;

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

        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = null;
        try {
        Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        obj2 = (Singleton)constructor.newInstance();
        }catch(Exception e) {
        e.printStackTrace();
        }
        System.out.println("HashCode For First Object "+ obj1.hashCode());
        System.out.println("HashCode For Second Object "+ obj2.hashCode());
        }
}
Output:
HashCode For First Object 2018699554
HashCode For Second Object 1311053135
```

- We can prevent the singleton pattern to break with the following steps.

1. One of the best solutions is to throw any customize run-time exception in the constructor if the instance already exists.
2. You can refer to the code snippet for it. You have to change the Singleton class.

```
package simplifiedjava.crackedInterview;

import javax.activity.InvalidActivityException;

public class Singleton {
		private static Singleton INSTANCE = new Singleton();
		private Singleton() {
		if(INSTANCE != null) {
		try {
		throw new InvalidActivityException("Can not create multiple objects.");
		} catch (InvalidActivityException e) {
		e.printStackTrace();
		}
		}
		}
		public static Singleton getInstance() {
		if(INSTANCE != null) {
		return INSTANCE;
		}
		return null;
		}
		public void m1() {
		System.out.println("m1() method called.");
		}
}
Output:
javax.activity.InvalidActivityException: Can not create multiple objects.
		at simplifiedjava.crackedInterview.Singleton.<init>(Singleton.java:12)
				at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
HashCode For First Object 1311053135
HashCode For Second Object 118352462
```

422. WAP to break singleton using Deserialization? How to prevent Singleton to break with Deserialization?

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

import javax.activity.InvalidActivityException;

public class Singleton implements Serializable{
		private static Singleton INSTANCE = new Singleton();
		private Singleton() {
		}
		public static Singleton getInstance() {
		if(INSTANCE != null) {
		return INSTANCE;
		}
		return null;
		}
		public void m1() {
		System.out.println("m1() method called.");
```

```
                }
}

package simplifiedjava.crackedInterview;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SingletonBreakUsingDeserialization {


                public static void main(String[] args) throws FileNotFoundException, IOException,
ClassNotFoundException {
                Singleton obj1 = Singleton.getInstance();
                ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("file.ser"));
                oos.writeObject(obj1);
                oos.flush();
                oos.close();
                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file.ser"));
                Singleton obj2 = (Singleton)ois.readObject();
                ois.close();
                System.out.println("HashCode for Original Object "+ obj1.hashCode());
                System.out.println("HashCode for Desrialized Object "+ obj2.hashCode());
                }
}
Output:
HashCode for Original Object 1442407170
HashCode for Desrialized Object 990368553
```

- To prevent this issue we have to override the **readResolve()** method in the Singleton class and return the same Singleton Instance.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

import java.io.Serializable;

import javax.activity.InvalidActivityException;

public class Singleton implements Serializable{
                public static Singleton INSTANCE = new Singleton();
                private Singleton() {
                }
                protected Object readResolve() {
            return INSTANCE;
  }
                public void m1() {
                System.out.println("m1() method called.");
                }
}

package simplifiedjava.crackedInterview;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SingletonBreakUsingDeserialization {

		public static void main(String[] args) throws FileNotFoundException, IOException,
ClassNotFoundException {
		Singleton obj1 = Singleton.INSTANCE;
		ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("file.ser"));
		oos.writeObject(obj1);
		oos.flush();
		oos.close();
		ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file.ser"));
		Singleton obj2 = (Singleton)ois.readObject();
		ois.close();
		System.out.println("HashCode for Original Object "+ obj1.hashCode());
		System.out.println("HashCode for Desrialized Object "+ obj2.hashCode());
		}
}
Output:
HashCode for Original Object 1442407170
HashCode for Desrialized Object 1442407170
```

423. WAP to break Singleton using Cloning? How to prevent Singleton to break with Cloning?

- The class implements a Clonable interface then only the object of that class is a Clonable object.
- Clone means creating a copy of an original object.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Singleton implements Cloneable{
		public static Singleton INSTANCE = new Singleton();
		private Singleton() {
		}
		public static Singleton getInstance() {
		if(INSTANCE == null) {
		INSTANCE = new Singleton();
		}
		return INSTANCE;
		}
		@Override
    protected Object clone() throws CloneNotSupportedException  {
        return super.clone();
    }
		public void m1() {
		System.out.println("m1() method called.");
		}
}
```

```
package simplifiedjava.crackedInterview;

public class SingletonBreakUsingCloning {

            public static void main(String[] args) throws CloneNotSupportedException {
            Singleton obj1 = Singleton.getInstance();
            Singleton obj2 = (Singleton)obj1.clone();
            System.out.println("HashCode of obj1"+ obj1.hashCode());
            System.out.println("HashCode of obj2"+ obj2.hashCode());
            }
}
Output:
HashCode of obj1 : 2018699554
HashCode of obj2 : 1311053135
```

- We can prevent breaking the singleton in a Clonable pattern. Inside clone() method we can simply throw CloneNotSupportedException explicitly.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview;

public class Singleton implements Cloneable{
            public static Singleton INSTANCE = new Singleton();
            private Singleton() {
            }
            public static Singleton getInstance() {
            if(INSTANCE == null) {
            INSTANCE = new Singleton();
            }
            return INSTANCE;
            }
            @Override
    protected Object clone() throws CloneNotSupportedException  {
        throw new CloneNotSupportedException();
    }
            public void m1() {
            System.out.println("m1() method called.");
            }
}

package simplifiedjava.crackedInterview;

public class SingletonBreakUsingCloning {

            public static void main(String[] args) throws CloneNotSupportedException {
            Singleton obj1 = Singleton.getInstance();
            Singleton obj2 = (Singleton)obj1.clone();
            System.out.println("HashCode of obj1 : "+ obj1.hashCode());
            System.out.println("HashCode of obj2 : "+ obj2.hashCode());
            }
}
Output:
Exception in thread "main" java.lang.CloneNotSupportedException
            at simplifiedjava.crackedInterview.Singleton.clone(Singleton.java:20)
            at
simplifiedjava.crackedInterview.SingletonBreakUsingCloning.main(SingletonBreakUsingCloning.java:7)
```

424.
What is the difference between JAR, WAR and EAR?

	JAR	WAR	EAR
1.	JAR stands for Java Archive.	WAR stands for Web Application Archive or Web Application Resources.	EAR stands for Enterprise Application Archive.
2.	Extension of the JAR file is .jar	Extension of WAR file is .war	Extension of EAR file is .ear
3.	The JAR file contains java classes, resources such as text, images aggregated into one file.	WAR file contains JSP, servlets, XML, static web pages.	EAR file represents the modules of the application and meta-data directory called META-INT which has one or more deployment descriptors.

425.
What is the difference between Web Application and Enterprise Application?

	Web Application	Enterprise Application
1.	The web application can be deployed on the web as well as Enterprise applications.	An enterprise application cannot deploy on a web application it can only deploy on an Enterprise application.
2.	Servet, JSP, HTML can be deployed on Web application.	EJB, JMS can be deployedon Enterprise application.

426.
What is the difference between Web Server and Application Server?

	Web Server	Application Server
1.	EJB and JMS cannot be deployed on a Web Server.	EJB and JMS can be deployed on the Application Server.
2.	Web server is one component of Application Server.	The application server is a parent of the web server.
3.	An example of a Web Server is Apache Tomcat.	An example of an Application Server is JBoss.

427.
List out some web servers and application servers?

- Web Servers.

  1. Apache Tomcat.
  2. Lightpd.
  3. Microsoft Internet Information Service.
  4. Sun Java System Web Server.
  5. Nginx.
  6. Jagsaw.

- Application Servers.

1. JBoss
2. WebLogic
3. WebSphere
4. Glassfish
5. Oracle OC4J

428.
What is the difference between Path and class path?

- **Path:** Path is used to define where the system can find executables(.exe) files.
- **Classpath:** Classpath is used to specify the location of the .class file.

429.
Can you explain the following command java, javaw and javaws?

- **java:** Java application executor which is associated with a console to display output/errors
- **javaw:** (Java windowed) application executor not associated with the console. So no display of output/errors. It can be used to silently push the output/errors to text files. It is mostly used to launch GUI-based applications.
- **javaws:** (Java web start) to download and run the distributed web applications. Again, no console is associated.

430.
What is interpreter?

- The computer program that converts high-level language into Assembly level language is called Interpreter.
- It is designed to read the source code and interpret the source code line by line.

431.
What are SOLID Priciple?

Principle	Description
Single Responsibility Principle	Each class should be responsible for a single part or functionality of the system.
Open-Closed Principle	Software components should be open for extension, but not for modification.
Liskov Substitution Principle	Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
Interface Segregation Principle	No client should be forced to depend on methods that it does not use.
Dependency Inversion Principle	High-level modules should not depend on low-level modules, both should depend on abstractions.

432.
Eclise Dubugging Process?

- Start the application on Debug mode.
- Put debugger points into java classes.
- Hit API or appropriate URL.
- You will be able to see a green highlighter at your debugger line.
- F6 key can be used to move for the next step.
- F8 key can be used to move to the next debugger point or a full run.

433. Eclipse Short Cuts?

Ctrl Buttons	Description
Ctrl + ↑ Shift + R	Open a resource. No Need to know the path of the resource.
Ctrl + ↑ Shift + T	Open type. It will show all interfaces, Classes, libraries etc. Wide range than above shortcut.
Ctrl + space	Type Assist. It will populate all the function names and variable names.
Ctrl + ↑ Shift + F	Format the code.
Ctrl + ↑ Shift + O	Organize all imports. I mean it will import all necessary files or classes.
Ctrl + ↑ Shift + U	Find reference in file.
Alt + ↑ Shift + Z	Enclose block in try-catch. Select the code area and press these keys.
Ctrl + ↑ Shift + P	Go to the matching parenthesis.
Ctrl + ↑ Shift + G	Search for current cursor positioned word reference in workspace
F3	Go to Declaration.
F4	Show type hierarchy of the class.

F5	Step into
F6	Step over.
F8	Resume.
F12	Focus on current editor.
Ctrl + Q	Inspect.
Ctrl + F11	Repeatedly Run last run program.
Ctrl + 1	Quick fix code.
Ctrl + H	Java search in workspace
Ctrl + .	Navigate to next or previous error.
Ctrl + L	Go to line number.
Ctrl + D	Delete a line.

Ctrl + M	Maximize editor.
Ctrl + T	Show inheritance tree of current token.
Ctrl + O	List all methods of the class including inherited methods.
Ctrl + Page Down	Move to next file in Editor.
Ctrl + Page Up	Move to previous file in Editor.
Alt + ->	Move to next edit position from the editor history list.
Alt + <-	Move to previous edit position from the editor history list.
↑ Shift + F2	Show Javadoc for current element.
Ctrl + E	Display the list of all open files on editor. You can move through arraow and then either click or press enter on required file.

Interview Questions on Java 8

434.
What are the features introduced in Java 8?

- Following are the features introduced in Java 8.

1. Lambda Expression.
2. Functional Interface.
3. Default methods in Interface.
4. Static methods in Interface.
5. Predicate functional interface.

6. Function functional interface.
7. Consumer functional interface.
8. Method reference and constructor reference (::).
9. Stream API.
10. Date and Time (JODA API).

435.
What is Lambda Expression?

- Lambda expression is an anonymous function without having return type, modifier and name.
- To enable functional programming in Java we have to use a lambda expression.
- We can pass the procedure and function directly.
- →For lambda expression, we have to use this symbol.

Normal Method.	Lambda Expression.

public void **m1() {**
    System.*out*.println("Hello");
}
	~~**public void** m1~~() {
	    System.***out***.println("Hello");
	}
	Lambda Expression will be like,
	→() {
	    System.***out***.println("Hello");
	}

public void **m1(int a, int b) {**
    System.*out*.println(a+b);
}
	~~**public void** m1~~(int a, int b) {
	    System.***out***.println("Hello");
	}
	Lambda Expression will be like,
	→(int a, int b) {
	    System.***out***.println(a+b);
	}

public int **m1(String s) {**
    return s.length();
}
	~~**public int** m1~~(String s) {
	    return s.length();
	}
	Lambda Expression will be like,
	→ (String s) {
	    return s.length();
	}

- If your lambda expression has only one line of code then it is allowed to skp the curly braces {}.

    1. →E.g. (int a, int b) System.out.println(a+b);

- If your lambda expression has only one parameter then it is allowed to skip parenthesis as well.

    1. →E.g. s s.length();

- If lambda expression return something then return keyword is also optional.

    1. E.g. please refer above example.

- Lambda expression can have body inside curly braces if there are multiple statements.

1. →E.g. s {

Statement 1;
Statement 2;
Statement 3;
};

436.
How will you identify lambda expression?

- We can identify the the lambda expression with following points.

1. →This arrow symbol should be there to implement lambda expression.
2. If we are passing parameters without parenthesis if there is only one parameter.
3. If we are returning something without a return statement.

437.
Please identify valid lambda expression?

- () -> System.out.println("Hello");
- (s) -> System.out.println("Java");
- (a,10) -> {System.out.println(a); System.out.println(10);}
- (a,b) -> System.out.println(a+b);
- s -> {System.out.println(s);}
- (int a, int b) -> {System.out.println(a+b);}
- s -> s.length();
- (s) -> {s.length();}

Expression	Valid / Invalid
() -> System.out.println("Hello");	Valid
(s) -> System.out.println("Java");	Valid
(a,10) -> {System.out.println(a); System.out.println(10);}	Valid
(a,b) -> System.out.println(a+b);	Valid
s -> {System.out.println(s);}	Valid
(int a, int b) -> {System.out.println(a+b);}	Valid
s -> s.length();	Valid
(s) -> {s.length();}	Valid

438.
What is functional Interface?

- An Interface with only one abstract method is called a functional interface.
- @FunctionalInterface annotation can be used to indicate.
- If an interface has more than one abstract method then it is not a functional interface it is a regular interface.
- Functional interface can have multiple default and static methods but only one abstract method.
- Lambda expression can work with a functional interface only. Lambda expression cannot work with a regular interface that has multiple abstract methods.
- Please refer to the below example for the functional interface.

```
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface FunctionalInterfaceDemo {
            public void add();
}
```

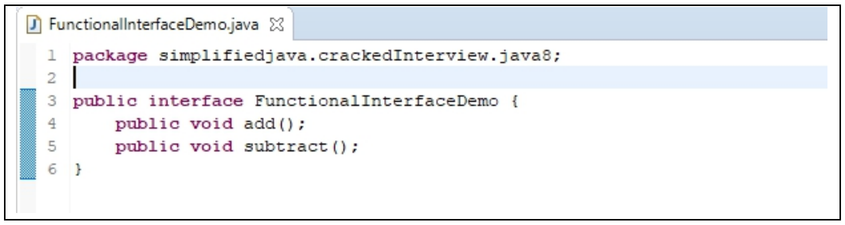
439.
Can you list out some functional interface used in java 6?

- FunctionalInterface must have only one abstract method.
- We have used 4 functional interfaces.

1. **Runnable:** Runnable interface has run() method.
2. **Callable:** Callable interface has call() method.
3. **ActionListener:** ActionListener has actionPerformed() method.
4. **Comparable:** Comparable has compareTo() method.

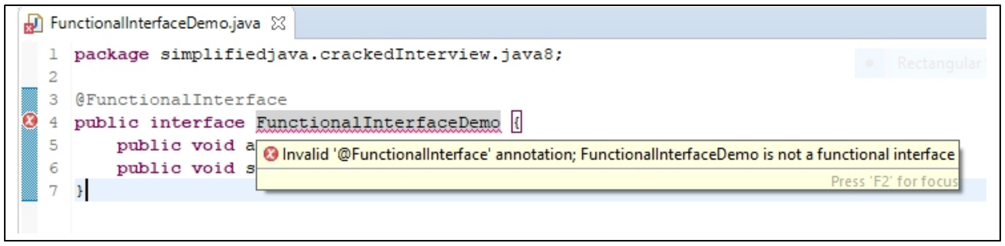
440.
Can I have multiple abstract methods in functional interface?

- No, we cannot have multiple abstract methods in the functional interface.
- If you declare multiple abstract methods inside an interface and an interface is not annotated with @FunctionalInterface then it will become a regular interface.


FunctionalInterfaceDemo.java

```
package simplifiedjava.crackedInterview.java8;

public interface FunctionalInterfaceDemo {
    public void add();
    public void subtract();
}
```

- If you declare multiple abstract methods inside an interface and an interface is annotated with @FunctionalInterface then it will give a compile-time error.


FunctionalInterfaceDemo.java

```
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface FunctionalInterfaceDemo {
    public void a
    public void s
}
```

Invalid '@FunctionalInterface' annotation; FunctionalInterfaceDemo is not a functional interface

Press 'F2' for focus

- If you declare only one abstract methods and multiple default and static method but not annotated with @FunctionalInterface then it is also functional interface.

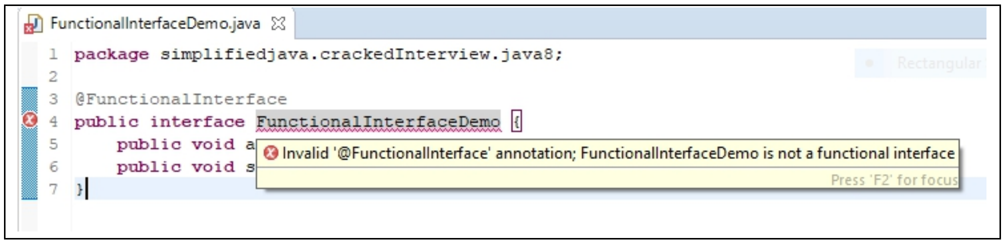
441.
Which annotation can be used to make functional interface?

- @FunctionalInterface Annotation can be used to make any Interface as functional interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface FunctionalInterfaceDemo {
            public void add();
}
```

442.
What will happen if I declared multiple abstract methods in one interface?

- If you declare multiple abstract methods inside interface and interface is annotated with @FunctionalInterface then it will give a compile-time error.

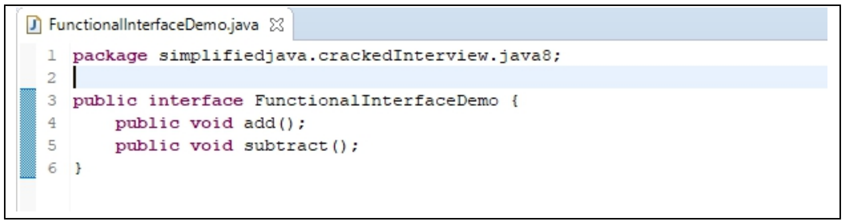


```
FunctionalInterfaceDemo.java
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface FunctionalInterfaceDemo {
    public void a
    public void s
}
```

Invalid '@FunctionalInterface' annotation; FunctionalInterfaceDemo is not a functional interface

Press 'F2' for focus

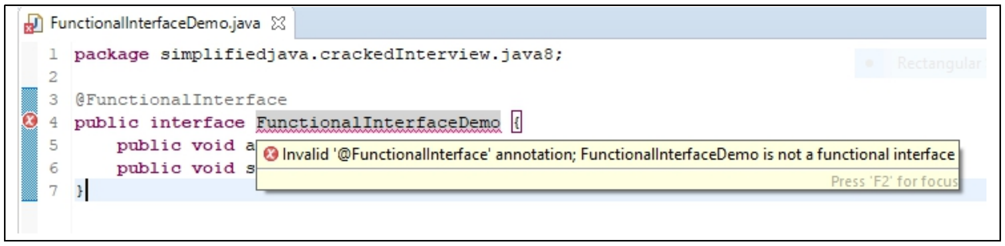
- If you declare multiple abstract methods inside interface and interface is not annotated with @FunctionalInterface then it will become a regular interface.



```
FunctionalInterfaceDemo.java
package simplifiedjava.crackedInterview.java8;

public interface FunctionalInterfaceDemo {
    public void add();
    public void subtract();
}
```

443.
What will happen if I declared multiple abstract methods in a fuctional interface?

- If you declare multiple abstract methods inside interface and interface is annotated with @FunctionalInterface then it will give a compile-time error.



```
FunctionalInterfaceDemo.java
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface FunctionalInterfaceDemo {
    public void a
    public void s
}
```

Invalid '@FunctionalInterface' annotation; FunctionalInterfaceDemo is not a functional interface

Press 'F2' for focus

444.
How lambda expression and functional interface are related?

- Functional interface must have only one abstract method.
- Lambda expression can call an abstract method of functional interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface FunctionalInterfaceDemo {
        public void add(int no1,int no2);
}

package simplifiedjava.crackedInterview.java8;
```

```
public class FunctionalInterfaceImplDemo {

        public static void main(String[] args) {
        FunctionalInterfaceDemo f = (no1,no2) -> System.out.println("Addition of Two nos = "+ (no1+no2));
        f.add(100, 200);
        }
}
Output:
Addition of Two nos = 300
```

445.
Can you create your own functional interface?

- Yes, we can create our functional interface.
- You have to just create normal interface.
- Create only one abstract method into it and annotated with @FunctionalInterface.

446.
Will the child interface functional interface?

```
1. @FunctionalInterface
public interface Parent {
        public void add();
}
@FunctionalInterface
public interface Child extends Parent {
}

2.@FunctionalInterface
public interface Parent {
        public void add();
}
@FunctionalInterface
public interface Child extends Parent {
        public void add();
}

3.@FunctionalInterface
public interface Parent {
        public void add();
}
@FunctionalInterface
public interface Child extends Parent {
        public void substract();
}

4.@FunctionalInterface
public interface Parent {
        public void add();
}
public interface Child extends Parent {
        public void substract();
}
```

- **Case 1:** If an interface extends functional interface and the child interface doesn't contain any abstract method, then the child interface is always a functional interface.

- **Case 2:** In the child interface we can define exactly the same parent interface abstract method.
- **Case 3:** In the child interface we can't define any new abstract method then we will get a compile-time error.
- **Case 4:** Child interface doesn't declare FunctinalInterface then code will compile.

447.
Can you compare Inner classes and lambda expression?

	Inner class	Lambda Expression
1.	An inner class can have a name.	Lambda expression doesn't have any name.
2.	An inner class can extend the abstract class.	Lambda expression cannot extend the abstract class.
3.	An inner class can extend the concrete class.	Lambda expression cannot extend the concrete class.
4.	An inner class can implement an interface that has multiple abstract methods.	Lambda expression can't implement an interface that has multiple abstract methods.

448.
What is the difference between Annonymous Inner class and Lambda Expression?

	Annonymous Inner class	Lambda Expression
1.	It is a class without a name.	It is a function without a name.
2.	An anonymous inner class can extend an abstract class or concrete class.	Lambda expression cannot extend an abstract class or concrete class.
3.	An anonymous inner class can implement an interface that has multiple abstract methods as well as a functional interface.	Lambda expression can implement only a functional interface.
4.	Inside the inner class, we can declare an instance variable.	Inside lambda expression, we can declare only local variables.
5.	An anonymous inner class can be instantiated.	Lambda expression cannot be instantiated.
6.	An anonymous interface is the best choice if we want to handle multiple methods.	Lambda expression is the best choice if we want to handle the interface with a single abstract method.
7.	Inside the Anonymous inner class, 'this' always refers current anonymous inner class object but not the outer class object.	Inside lambda expression, this always refers current outer class object.
8.	For an anonymous inner class at the time of compilation, a separated .class file will be generated.	For the lambda expression, separate .class file won't be generated.

449.
What are default methods? What is the basic purpose of default methods?

- Methods defined inside interface with default keyword is called default method.

- Since, Java 8 you can have method implementation inside the interface.
- Default methods can be used for backward compatibility.
- In the regular case, if you declare or add a new method in the interface then million of code may break so avoiding this situation, we can add a default method inside the interface so no code will be affected.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

public interface InterfaceForDefaultMethod {

        default void m1() {
        System.out.println("m1() from DefaultMethodDemo interface is called");
        }
}

package simplifiedjava.crackedInterview.java8;

public class DefaultMethodDemo implements InterfaceForDefaultMethod{

        public static void main(String[] args) {
        InterfaceForDefaultMethod demo = new DefaultMethodDemo();
        demo.m1();
        DefaultMethodDemo demo1 = new DefaultMethodDemo();
        demo1.m1();
        }
}
Output:
m1() from DefaultMethodDemo interface is called
m1() from DefaultMethodDemo interface is called
```

450.
Can you override default method?

- Yes, we can override the default method in the implementation class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

public interface InterfaceForDefaultMethod {

        default void m1() {
        System.out.println("interface m1() method called.");
        }
}

package simplifiedjava.crackedInterview.java8;

public class DefaultMethodDemo implements InterfaceForDefaultMethod{

        public void m1() {
        System.out.println("Class m1() method called.");
        }
        public static void main(String[] args) {
        InterfaceForDefaultMethod demo = new DefaultMethodDemo();
        demo.m1();
        DefaultMethodDemo demo1 = new DefaultMethodDemo();
        demo1.m1();
        }
}
Output:
Class m1() method called.
Class m1() method called.
```

451.
How will you uniquely call same methods declared in multiple interfaces? If Inteface A has add() and interface B also have add() then how will you differentiate both methods?

- Please refer below example.

```
package simplifiedjava.crackedInterview.java8;

public interface Left {

        default void m1() {
        System.out.println("Left m1() called.");
        }
}

package simplifiedjava.crackedInterview.java8;

public interface Right {
        default void m1() {
        System.out.println("Right m1() called.");
        }
}

package simplifiedjava.crackedInterview.java8;

public class LeftRightInterfaceDemo implements Left, Right{

        public static void main(String[] args) {
        LeftRightInterfaceDemo demo = new LeftRightInterfaceDemo();
        demo.m1();
        }

        @Override
        public void m1() {
        Left.super.m1();
        Right.super.m1();
        }
}
Output:
Left m1() called.
Right m1() called.
```

452.
What is the difference between Abstract class and Interface with Default methods?

	Abstract class	Interface with Default methods
1.	An abstract class can be declared with an abstract keyword.	Interface with default method declared with interface keyword and default keyword for method.
2.	An abstract class can have a constructor.	An interface cannot have a constructor. Doesn't matter interface has a default method

		or not.
3.	An abstract class can have a static block or a non-static block.	Interface with the default method cannot have a static or non-static block.
4.	Abstract class can't refer to lambda expression.	Interface with default methods refers to lambda expression.
5.	Inside the abstract class, we can override the objects class method.	We cannot override objects class method in interface with default methods.
6.	An abstract class can be declared protected members.	Interface with default methods cannot declare protected members.

453.
In functional interface how many default methods can be declared?

- There is no limit on declaring default methods inside the interface.
- You can have n number of methods inside the interface.

454.
How will you identify particular method is default method?

- All default methods must be declared inside the interface. You cannot declare the default method inside the class.
- All default methods are declared with the default keyword.

455.
How will you invoke default methods from class?

- First, you need to declare the default method in the interface.
- Second, you have to create one class and implement that interface that has the default method.
- Then, simply instantiate the class and with the class reference variable, you can call default methods.
- Please refer to the below example.

```java
package simplifiedjava.crackedInterview.java8;

public interface InterfaceForDefaultMethod {

        default void m1() {
        System.out.println("interface m1() method called.");
        }
}

package simplifiedjava.crackedInterview.java8;

public class DefaultMethodDemo implements InterfaceForDefaultMethod{

        public void m1() {
        System.out.println("Class m1() method called.");
        }
        public static void main(String[] args) {
        InterfaceForDefaultMethod demo = new DefaultMethodDemo();
        demo.m1();
        DefaultMethodDemo demo1 = new DefaultMethodDemo();
        demo1.m1();
        }
}
```

```
Output:
Class m1() method called.
Class m1() method called.
```

456.
What are the static methods in functional interface?

- Static method declared inside the interface is called the static method.
- We can have the implementation of static class inside the interface.
- Static methods can invoke with the Interface name. No need to instantiate the class.

457.
What is the purpose of static methods in functional interface?

- Just to define utility methods in the interface then we can go for static methods.
- The main purpose of the static method is, we can invoke static methods like the utility class method. No need to create an instance of that class.

458.
How will you invoke static methods from class?

- We can invoke the static method using the interface name. No need to create an instance of the class.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

public interface InterfaceForStaticMethod {

		public static void m1() {
		System.out.println("m1() method called.");
		}
}

package simplifiedjava.crackedInterview.java8;

public class StaticMethodDemo {

		public static void main(String[] args) {
		InterfaceForStaticMethod.m1();
		}
}
Output:
m1() method called.
```

459.
Can you declare public static void main() method inside functional interface?

- Yes, we can declare the main() method inside the functional interface and invoke the main() method from class with the interface name.
- But, It will be treated as a normal method.
- JVM always invoked the main method of a class not declared in the interface.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

public interface InterfaceForStaticMethod {
```

```
            public static void m1() {
            System.out.println("m1() method called.");
            }
            public static void main(String[] args) {
            System.out.println("Main method called.");
            }
}

package simplifiedjava.crackedInterview.java8;

public class StaticMethodDemo {

            public static void main(String[] args) {
            InterfaceForStaticMethod.m1();
            InterfaceForStaticMethod.main(args);
            }
}
Output:
m1() method called.
Main method called.
```

460.
What are the pre-defined functional interfaces?

- Java 8 provides some functional interfaces inbuilt are called pre-defined functional interfaces.
- Following are the pre-defined functional interfaces.

  1. Predicate functional interface.
  2. Function functional interface.
  3. Consumer functional interface.
  4. Supplier functional interface.
  5. Bi-Predicate functional interface.
  6. Bi-Function functional interface.
  7. Bi-Consumer functional interface.

461.
What is Predicate functional interface?

- Predicate is used to perform a particular condition check.
- Predicate can be used like if condition. It will always return a Boolean value.
- Please refer to the below example.

```
@FunctionalInterface
public interface Predicate<T> {

}
```

462.
What is the abstract method defined in predicate functional interface?

- Predicate has a defined test() method which is only one abstract method.
- Test method accepts only one parameter to test it.
- Please refer to the below syntax for Predicate.

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

463.
What is the return type of test method in Predicate functional interface?

- Return type of test() method is Boolean.
- Please refer below example.

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

464.
What are the default methods defined in Predicate functional interface?

- There are 3 default methods declared inside Predicate functional interface.

1. **Negate():** negate() returns the logical negation of existing result.

```
package simplifiedjava.crackedInterview.java8;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateDefaultMethodDemo {

		public static void main(String[] args) {
		List<String> list = 
Arrays.asList("Yogesh","Arpita","Shweta","Shruti","Ananya","Asmita","Yogita");

		Predicate<String> p1 =  s -> s.startsWith("S");
		Predicate<String> p3 =  p1.negate();

		for(String s : list) {
		if(p1.test(s)) {
		System.out.println(s);
		}
		}
		}
}
```

2. And(): And() works like logical && just combine two conditions.

```
package simplifiedjava.crackedInterview.java8;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateDefaultMethodDemo {

		public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10,20,30,40,50,60,70,80,90,100);

        Predicate<Integer> p1 =  n -> n > 50;
        Predicate<Integer> p2 =  n -> n < 100;
        Predicate<Integer> p3 =  p1.and(p2);

        for(Integer s : list) {
        if(p3.test(s)) {
        System.out.println(s);
        }
        }
        }
}
Output:
60
70
80
90
```

3. Or(): Or() works like logical || just combine tow conditions.

```
package simplifiedjava.crackedInterview.java8;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateDefaultMethodDemo {

        public static void main(String[] args) {
        List<String> list =
Arrays.asList("Yogesh","Arpita","Shweta","Shruti","Ananya","Asmita","Yogita");

        Predicate<String> p1 =  s -> s.startsWith("S");
        Predicate<String> p2 =  s -> s.startsWith("Y");

        Predicate<String> p3 =  p1.negate();
        Predicate<String> p4 =  p1.or(p2);

        for(String s : list) {
        if(p4.test(s)) {
        System.out.println(s);
        }
        }
        }
}
Output:
Yogesh
Shweta
Shruti
Yogita
```

What are the static methods defined in Predicate functional interface?

- Predicate has defined isEqual () static method inside it.
- Please refer to the below example for syntax.

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.Predicate;

public class PredicateStaticMethodDemo {

        public static void main(String[] args) {
        Predicate<String> p = Predicate.isEqual("Java");
        System.out.println(p.test("java"));
        System.out.println(p.test("Java"));
        System.out.println(p.test("Databse"));
        }
}
Output:
false
true
false
```

466. Wap to implement Predicate interface?

```
package simplifiedjava.crackedInterview.java8;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FindPositiveOrNegativeNoUsingPredicate {

        public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(-10);
        list.add(20);
        list.add(30);
        Predicate<Integer> p = no -> no > 0;
        for(Integer i : list) {
        if(p.test(i)) {
        System.out.println("Positive Number");
        }else {
        System.out.println("Negative Number");
        }
        }
        }
}
Output:
Negative Number
Positive Number
Positive Number
```

467.
What is Function functional interface?

- Function interface takes one argument and returns any type of object. As we have seen Predicate return only Boolean but Function returns any type of object.
- Please refer to the below example for syntax.

```
@FunctionalInterface
public interface Function<T, R> {

}
```

468. What is the abstract method defined in Function functional interface?

- Function interface has defined apply() as an abstract method.
- Please refer to the below example for syntax.

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

469. What is the return type of apply method in Function functional interface?

- The return type of apply() method is any object.
- Java doc denoted as R(Return Type).
- Please refer to the below example.

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

470. Wap to implement Function interface?

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.Function;

public class FunctionInterfaceDemo {

            public static void main(String[] args) {
            Function<String, Integer> f = s -> s.length();
            System.out.println(f.apply("Simplify Java"));
            }
}
Output: 13
```

471. What are the default methods defined in Function functional interface?

- There are a couple of default methods inside the Function interface.

1. **andThen():** f1 will be applied first and then f2.

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.Function;
```

```
public class FunctionInterfaceDemo {

        public static void main(String[] args) {
        Function<String, Integer> f1 = s -> s.length();
        Function<String, String> f2 = s -> s.toUpperCase();
        Function<String, String> f3 = s -> s.toLowerCase();
        Function<String, String> f4 = f3.andThen(f2);
        System.out.println(f1.apply("Simplify Java"));
        System.out.println(f2.apply("Simplify Java"));
        System.out.println(f3.apply("Simplify Java"));
        System.out.println(f4.apply("Simplify Java"));
        }
}
Output:
13
SIMPLIFY JAVA
simplify java
SIMPLIFY JAVA
```

2. **compose():** f2 will be applied fist and then f1.

```
package simplifiedjava.crackedInterview.java8;

import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

public class FunctionInterfaceDemo {

        public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10,20,30,40,50);
        Function<Integer, Integer> f1 = n -> n + 10;
        Function<Integer, Integer> f2 = n -> n * 2;
        Function<Integer, Integer> f3 = f1.compose(f2);// f2 will execute first and then f1
        for(Integer i : list) {
        System.out.println(f3.apply(i));
        }
        }
}
Output:
30  [10 = (10*2) + 10 = 30]
50  [20 = (20*2) + 10 = 50]
70  [30 = (30*2) + 10 = 70]
90  [40 = (40*2) + 10 = 90]
110 [50 = (50*2) + 10 = 100]
```

472.
What is the static method defined in Function functional interface?

- There is only one static method defined inside the Function interface.
- Identity() static method is defined in Function.
- Returns a function that always returns its input argument.
- You will get the same output as your input.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

public class FunctionStaticMethodDemo {

        public static void main(String[] args) {
        List<String> list = Arrays.asList(".Net","Php","Java","C++","C","Database");
        Function<String, String> f = Function.identity();
        for(String s: list) {
        System.out.println(f.apply(s));
        }
        }
}
Output:
.Net
Php
Java
C++
C
Database
```

473.
What is the difference between Predicate and Function interface?

	Predicate	Function
1.	The predicate can return only a Boolean value.	The function can return any type of object.
2.	Predicate defined one abstract method. i.e. test() method.	The function defined one abstract method. i.e. apply() method.
3.	Public Boolean test(T t);	Public R apply(T t, R r);
4.	The predicate can take only one type of parameter which represent the input argument.	Function interface takes two parameters. The first one represents input type and the second one represent return type.
5.	Predicate<T>	Function<T,R>

474.
What is Consumer functional interface?

- Consumer interface function just consumes a value, perform a certain operation and doesn't return anything.
- Syntax for Consumer interface is,

```
@FunctionalInterface
public interface Consumer<T> {

}
```

475.
What is the abstract method defined in Consumer functional interface?

- Consumer interface has only one abstract method.
- Consumer interface has accept() method which is abstract.
- Please refer to the below example.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

476.
What is the return type of apply method in Consumer functional interface?

- The return type of accept method is void.
- Accept() method only consumes the input and process it.

477.
Wap to implement Consumer interface?

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.Consumer;

public class ConsumerInterfaceDemo {

        public static void main(String[] args) {
        Consumer<String> c = s ->  System.out.println("List Item " +s);
        c.accept("Java");
        c.accept("Interview");
        }
}
Output:
List Item Java
List Item Interview
```

478.
What are the default methods defined in Consumer functional interface?

- There is only one default method declared inside the Consumer interface.
- The abstract method defined inside consumer is andThen().
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.Consumer;

public class ConsumerInterfaceDemo {

        public static void main(String[] args) {
        Consumer<String> c1 = s -> System.out.println("List 1 " +s);
        c1.accept("Java");
        Consumer<String> c2 = s -> System.out.println("List 2 " + s);
        c2.accept(".Net");
        Consumer<String> c3 = c2.andThen(c1);//c2 calls first & then c1
        c3.accept("Both");
        }
}
Output:
List Item 1 Java
List Item 2 .Net
List Item 2 Both
List Item 1 Both
```

479.
What are the static methods defined in Consumer functional interface?

- There is no static method declared inside the Consumer interface.
- Consumer interface has only one abstract method and default method.

480.
What is Supplier functional interface?

- The Supplier interface represents an operation that takes no argument and returns a result.
- Please refer to the below example.

```
@FunctionalInterface
public interface Supplier<T> {

}
```

481.
What is the abstract method defined in Supplier functional interface?

- Supplier interface has defined only one abstract method.
- Get() abstract method has defined the Supplier interface.
- Please refer to the below syntax.

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

482.
What is the return type of get() method in Supplier functional interface?

- The return type of Supplier interface can be any type of object.

483.
What are the static method defined in Supplier functional interface?

- There is no static method declared inside the Supplier interface.

484.
Wap to implement Supplier interface?

```
package simplifiedjava.crackedInterview.java8;

import java.util.Date;
import java.util.function.Supplier;

public class SupplierInterfaceDemo {

        static String technology = "Java";
        public static void main(String[] args) {
        Supplier<Integer> s1 = ()-> technology.length();
        Supplier<String> s2 = () -> technology.toUpperCase();
```

```
            Supplier<Date> s3 = () -> new Date();
            System.out.println(s1.get());
            System.out.println(s2.get());
            System.out.println(s3.get());
            }
}
Output:
4
JAVA
Sat Nov 13 21:26:04 IST 2021
```

485.
What is the difference between Consumer and Supplier?

	Consumer	Supplier
1.	Consumer interface always takes some input and perform a certain operation and never return anything.	The Supplier interface represents an operation that takes no argument and returns a result.

2.	`@FunctionalInterface`
	`public interface Consumer<T> {`
	`    void accept(T t);`
	`}`
		`@FunctionalInterface`
		`public interface Supplier<T> {`
		`    T get();`
		`}`

3.	Default method is andThen()	There is no default method in Supplier.

486.
What is Bi-Predicate functional interface?

- Bi-Predicate is used to perform a particular condition check.
- Predicate can be used like if condition. It will always return a Boolean value.
- Bi-Predicate accepts two arguments and return a Boolean value.
- Please refer to the below example.

```
@FunctionalInterface
public interface BiPredicate<T, U> {

}
```

487. Wap to implement Bi-Predicate interface?

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.BiPredicate;

public class BiPredicateInterfaceDemo {

            public static void main(String[] args) {
            BiPredicate<Integer, Integer> bp = (num1,num2) -> num1 > num2;
            System.out.println(bp.test(10, 25));
            System.out.println(bp.test(100, 25));
            }
}
Output:
false
true
```

488.
What is the difference between Predicate and Bi-Predicate?

	Predicate	Bi-Predicate
1.	Predicate accepts only one argument.	Bi-Predicate accepts two parameters.

2.	@FunctionalInterface
	public interface Predicate<T> {
	}
		@FunctionalInterface
		public interface BiPredicate<T, U>{
		}

489.
What is Bi-Function functional interface?

- Bi-Function interface takes one argument and returns any type of object. As we have seen Bi-Predicate returns only Boolean but Function return any type of object.
- Please refer below example for syntax.

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

}
```

490. Wap to implement Bi-Function interface?

```
package simplifiedjava.crackedInterview.java8;

import java.util.function.BiFunction;

public class BiFunctionInterfaceDemo {

        public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> bf = (num1,num2) -> num1 + num2;
        Integer total = bf.apply(100, 200);
        System.out.println("Total = "+ total);
        }
}
Output:  Total = 300
```

491.
What is the difference between Function and Bi-Function?

	Function	Bi-Function
1.	The Function accepts two parameters.	Bi-Function accepts three parameters.

2.	@FunctionalInterface
	public interface Function<T, U> {
	}
		@FunctionalInterface
		public interface BiFunction<T,U,R>{
		}

492.
What is Bi-Consumer functional interface?

- Bi-Consumer interface function just consumes a couple of values, perform the certain operation and doesn't return anything.
- Syntax for Consumer interface is,

```
@FunctionalInterface
```

```java
public interface BiConsumer<T, U> {

}
```

493. Wap to implement Bi-Consumer interface?

```java
package simplifiedjava.crackedInterview.java8;

import java.util.function.BiConsumer;

public class BiConsumerInterfaceDemo {

		public static void main(String[] args) {
		BiConsumer<String, String> bc = (str1, str2) -> System.out.println(str1+str2);
		bc.accept("Java", " Interview");
		}
}
Output:
Java Interview
```

494. What is the difference between Consumer and Bi-Consumer?

	Consumer	Bi-Consumer
1.	The consumer interface function just consumes a single value, perform a certain operation and doesn't return anything.	Bi-Consumer interface function just consumes a couple of values, perform a certain operation and doesn't return anything.

2.	@FunctionalInterface
	public interface Consumer<T>{
	
	}
		@FunctionalInterface
		public interface BiConsumer<T, U>{
		
		}

495. What are the primitive functions?

- Primitives cannot use generic type argument, Java 8 has provided some primitives functions for performance improvement.
- There are 15 primitive functions.

1. **IntFunction<R> :**
   - If your input is always int primitive then we should use this function.
   - **Method: R apply(int value);**
2. **LongFunction<R>:**
   - If your input is always long primitive then we should use this function.
   - **Method: R apply(long value);**
3. **DoubleFunction<R>:**
   - If your input is always double primitive then we should use this function.
   - **Method: R apply(double value);**
4. **ToIntFunction<T>:**
   - Input type can be anything but return type is always int.
   - **Method: int applyAsInt(T value);**
5. **ToLongFunction<T>:**
   - Input type can be anything but return type is always long.
   - **Method: long applyAsLong(T value);**

6. **ToDoubleFunction<T>:**
   - Input type can be anything but return type is always double.
   - **Method: double applyAsDouble(T value);**
7. **IntToLongFunction<T>:**
   - Input type must Int and output type must long then we should use this function.
   - **Method: long applyAsLong(int value);**
8. **IntToDoubleFunction<T>:**
   - Input type must Int and output type must double then we should use this function.
   - **Method: double applyAsDouble(int value);**
9. **longToIntFunction<T>:**
   - Input type must long and output type must int then we should use this function.
   - **Method: int applyAsInt(long value);**
10. **longToDoubleFunction<T>:**
    - Input type must long and output type must double then we should use this function.
    - **Method: double applyAsDouble(long value);**
11. **doubleToIntFunction<T>:**
    - Input type must double and output type must int then we should use this function.
    - **Method: int applyAsInt(double value);**
12. **doubleToLongFunction<T>:**
    - Input type must double and output type must long then we should use this function.
    - **Method: long applyAsLong(double value);**
13. **ToIntBiFunction<T,U>:**
    - Method: **int** applyAsInt(T t, U u);
14. **ToLongBiFunction<T,U>:**
    - Method: **long** applyAsLong(T t, U u);
15. **ToIntBiFunction<T,U>:**
    - Method: **int** applyAsInt(T t, U u);

496. What is the difference between IntFunction, ToIntFunction and IntToDoubleFunction?

	IntFunction	ToIntFunction	IntToDoubleFunction
1.	If your input is always int primitive then we should use this function.	Input type can be anything but return type is always int.	Input type must be Int and output type must double then we should use this function.
2.	**Method:** R apply(int value);	**Method:** int applyAsInt(T value);	**Method:** double applyAsDouble(int value);

497.
What is method Reference? How will you implements method reference?

- Method reference can be used to use the existing method with :: symbol.
- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface MyInterface {
        public void display();
}

package simplifiedjava.crackedInterview.java8;

public class MyClass{

        public static void main(String[] args) {
        MyClass myClass = new MyClass();
        MyInterface myInterface = myClass::myMethod;
        myInterface.display();
        }
        public void myMethod() {
        System.out.println("My Method Displayed.");
        }
}
Output:  My Method Displayed.
```

498.
What is the purpose of Method Reference?

- Code reusability can be achieved with method reference.
- Method reference means we use the readymade method.

499.
What is Constructor Reference? How will you implements Constructor reference?

- Constructor References in Java 8 can be created using the Class Name and the keyword new keyword.
- Syntaxes for Constructor reference.

1. Integer::**new**;
2. String::**new**;
3. ArrayList::**new**;
4. Employee::**new**;

- Please refer to the below example.

```
package simplifiedjava.crackedInterview.java8;

@FunctionalInterface
public interface StudentInterface {
		public Student get(String name);
}

package simplifiedjava.crackedInterview.java8;

public class Student {

		private String name;

		public Student(String name) {
		super();
		this.name = name;
		System.out.println("Student Name is : "+ name);
		}
		public static void main(String[] args) {
		StudentInterface stuInt = Student :: new;
		stuInt.get("Shruti");
		}
}
Output:  Student Name is : Shruti
```

500.
Can you call static method with method reference?

- Surely, we can call the static method with method reference.

501. Write a program to call static methods with method reference?

```
package simplifiedjava.crackedInterview.java8;

public class Addition {

		public static int add(int num1, int num2) {
		return num1 + num2;
		}
}

package simplifiedjava.crackedInterview.java8;

import java.util.function.BiFunction;

public class AdditionDemo {

		public static void main(String[] args) {
		BiFunction<Integer, Integer, Integer> bf = Addition::add;
		int result = bf.apply(100, 200);
		System.out.println("Result "+ result);
		}
}
Output: Result 300
```

502.
What is Optional? What is the purpose of Optional?

- An optional is a public final class that is used to deal with NullPointerException in java applications.
- Optional class is mostly used in Stream API where we are applying some filters and retrieved the filtered data.
- Optional class is mainly used for handling NullPointerException.

```
package simplifiedjava.crackedInterview.java8;

import java.util.Optional;

public class OptionalClassDemo {

        public static void main(String[] args) {
        String[] strArray = new String[5];
        strArray[0] = "Core";
        strArray[1] = "Java";
        strArray[2] = "Spring";
        Optional<String> opt = Optional.ofNullable(strArray[3]);
        if(opt.isPresent()) {
        System.out.println("Value is Present");
        }else {
        System.out.println("Value is not Present");
        }
        }
}
Output:  Value is not Present
```

503.
What is Stream API?

- Stream API is mostly used to process collection objects.
- Stream API can be used to filter out some elements based on some sort of conditions and map some values according to some process.
- Stream is an interface present in java.util.stream package.

504.
How does it different from Collection API?

- Collection can be used for storing and manipulating a group of data.
- Stream API is only used for processing groups of data.

505.
What are the intermidiate operations? Can you list out some intermediate operations?

- The operations which return another stream, as a result, are called intermediate operations.
- Following are the intermediates operations.

	Operation	Processing way
1.	**filter()**	Returns another stream of elements which satisfy given condition inside filter.
2.	**map()**	Returns a stream consisting of results after applying given function to elements of the stream.

3.	**sorted()**	Returns a stream consisting of elements sorted according to natural sorting order.
4.	**distinct()**	Returns a stream of unique elements.
5.	**skip()**	Returns a stream after skipping first n elemetns.
6.	**limit()**	Returns a stream containing first n elements.

506.
What are the terminal operations? Can you list out some terminal operations?

- The operations which return non-stream values like null or collection object or objects or primitive is called terminal operations.
- Following are the intermediates operations.

	Operation	**Processing way**
1.	**collect()**	Returns processed collection object like list, set or map.
2.	**forEach()**	Perform an action on all elements of stream.
3.	**min()**	Returns lowest element in the stream wrapped in optional object.
4.	**max()**	Returns highest element in the stream wrapped in optional object.
5.	**count()**	Returns the number of elements in a stream.
6.	**findFirst()**	Returns the first element of a stream wrapped in Optional object.
7.	**findAny()**	Returns anyone element randomly from the stream.
8.	**reduce()**	Performs reduction operation on elements of the stream using initial value and binary operation.
9.	**toArray()**	Returns an array containing elements of a stream.
10.	**anyMatch()**	Returns true if anyone element of a stream matches with given predicate.
11.	**allMatch()**	Returns true if all elements of a stream matches with given predicate.
12.	**noneMatch()**	Returns true only if all the elements of a stream doesn't match with given predicate.

507.
What is the difference between Intermidiate operations and Terminal operations?

	Intermidiate operations	**Terminal operations**
1.	These operations return the processed stream.	Terminal operations can return any type of object.
2.	Intermidiate operations are lazy loading opeartions.	Terminal operations are eagar loading operations.
3.	Intermidiate operations can contain any number of operations.	Terminal operations can have only one operaton.
4.	Intermidate operations cannot create final result.	Terminal operation can create final operation.
5.	E.g. 1. filter() 2. map() 3. sorted() 4. distinct() 5. skip()	E.g. 1. collect() 2. forEach() 3. min() 4. max() 5. count()

	6. limit()	6. findFirst() 7. findAny() 8. reduce() 9. toArray() 10. anyMatch() 11. allMatch() 12. noneMatch()

508.
What is the purpose of StringJoiner class?

- StringJoiner class can be used to join the list or array with a specified delimiter.
- Please refer to the below example.

```
package org.practice.staticmethoddemo.streamapi;

import java.util.Arrays;
import java.util.List;
import java.util.StringJoiner;
import java.util.stream.Collectors;

public class StringJoinerDemo {

	public static void main(String[] args) {
	StringJoiner joiner = new StringJoiner(",");
	joiner.add("Arpita")
	.add(" Yogesh")
	.add(" Sanas");
	System.out.println(joiner.toString());
	StringJoiner joiner1 = new StringJoiner(":","[","]");
	joiner1.add("Arpita")
	.add(" Yogesh")
	.add(" Sanas");
	System.out.println(joiner1.toString());
	List<String> list = Arrays.asList("Arpita","Yogesh","Sanas");
	String joiner2 = list.stream()
	.collect(Collectors.joining(","));
	System.out.println(joiner2.toString());
	}
}
Output:
Arpita, Yogesh, Sanas
[Arpita: Yogesh: Sanas]
Arpita,Yogesh,Sanas
```

509.
What is Metaspace introduced in java 8?

- **PermGen:**

1. PermGen (Permanent Generation) is a special heap space separated from the main memory.
2. The JVM keeps track of class metadata in the PermGen. Also, the JVM stores all the static content in this.
3. Due to limited memory size, PermGen can throw OutOfMemoryError.

- **Metaspace:**

  1. Metaspace is new memory space.
  2. It has replaced the older PermGen memory space.
  3. It can now handle memory allocation.
  4. Metaspace grows automatically by default.

Java 8 - Practicle Examples.

510.
WAP to find out odd and even numbers from the given list.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class FindOddEvenNumberDemo {
		public static void main(String[] args) {
		List<Integer> numList = Arrays.asList(5,25,10,15,40,80,100);
		List<Integer> evenList = numList.stream()
		.filter(num -> num % 2 == 0)
		.collect(Collectors.toList());
		List<Integer> oddList = numList.stream()
		.filter(num -> num % 2 != 0)
		.collect(Collectors.toList());
		System.out.println("Even List "+ evenList);
		System.out.println("Odd List "+ oddList);
		}
}
Output:
Even List [10, 40, 80, 100]
Odd List [5, 25, 15]
```

511. **WAP to print double value of each value in the list.**

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class PrintDoubleValueOfEachValueDemo {

		public static void main(String[] args) {
		List<Integer> numList = Arrays.asList(5,25,10,15,40,80,100);
```

```
            List<Integer> updatedList = numList.stream()
            .map(num -> num * 2)
            .collect(Collectors.toList());
            System.out.println("Double List "+ updatedList);
            }
}
Output:
Double List [10, 50, 20, 30, 80, 160, 200]
```

512. WAP to Sort Interger list in ascending order.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class SortListAscendingOrderDemo {

            public static void main(String[] args) {

            List<Integer> numList = Arrays.asList(5,25,100,80,10,15,40);

            System.out.println("List Before Sorting "+ numList);

            List<Integer> sortedList = numList.stream()
            .sorted()
            .collect(Collectors.toList());
            System.out.println("List After Sorting "+ sortedList);
            }
}
Output:
List Before Sorting [5, 25, 100, 80, 10, 15, 40]
List After Sorting [5, 10, 15, 25, 40, 80, 100]
```

513. WAP to Sort Interger list in descending order.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class SortListDescendingOrderDemo {

            public static void main(String[] args) {
            List<Integer> numList = Arrays.asList(5,25,100,80,10,15,40);
            System.out.println("List Before Sorting "+ numList);
            List<Integer> sortedList = numList.stream()
                                        .sorted((n1,n2) -> n2.compareTo(n1))
            .collect(Collectors.toList());
            System.out.println("List After Sorting "+ sortedList);
            }
}
Output:
List Before Sorting [5, 25, 100, 80, 10, 15, 40]
List After Sorting [100, 80, 40, 25, 15, 10, 5]
```

514. WAP to Sort Interger list in descending order by using comparator.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class SortListDescendingOrderByComparatorDemo {

		public static void main(String[] args) {
		List<Integer> numList = 
Arrays.asList(5,25,100,80,10,15,40);
		System.out.println("List Before Sorting "+ numList);
		List<Integer> sortedList = numList.stream()
		.sorted(new IntDescComparator())
		.collect(Collectors.toList());
		System.out.println("List After Sorting "+ sortedList);
		}
}

class IntDescComparator implements Comparator<Integer>{
		@Override
		public int compare(Integer num1, Integer num2) {
		return num2.compareTo(num1);
		}
}
Output:
List Before Sorting [5, 25, 100, 80, 10, 15, 40]
List After Sorting [100, 80, 40, 25, 15, 10, 5]
```

515. WAP to find highest value from the list.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;

public class FindHighestValueListDemo {

		public static void main(String[] args) {
		List<Integer> numList = Arrays.asList(5,25,100,80,10,15,40);
		int highestNum = numList.stream()
			.max((num1,num2)-> num1.compareTo(num2)).get();
		System.out.println("Highest Value = "+ highestNum);
		}
}
Output:  Highest Value = 100
```

516. WAP to find lowest value fromt the list.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
```

```
public class FindLowestValueListDemo {

        public static void main(String[] args) {
        List<Integer> numList = Arrays.asList(5,25,100,80,10,15,40);
        int lowestNum = numList.stream()
           .max((num1,num2)-> num2.compareTo(num1)).get();
        System.out.println("Lowest Value = "+ lowestNum);
        }
}
Output:  Lowest Value = 5
```

517. WAP to calculate square of each element of list using forEach.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;

public class FindSquareEachElementDemo {

        public static void main(String[] args) {
        List<Integer> numList = Arrays.asList(5,25,100,80,10,15,40);
        numList.stream()
        .forEach(num -> System.out.println(num +" Square = "+ num * num));
        }
}

Output:
5 Square = 25
25 Square = 625
100 Square = 10000
80 Square = 6400
10 Square = 100
15 Square = 225
40 Square = 1600
```

518. WAP to print the values using Stream.of() method?

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.stream.Stream;

public class PrintValuesUsingStreamOfDemo {

        public static void main(String[] args) {
        Integer[] numList = {10,20,30,40,50};
        Stream<Integer> stream = Stream.of(numList);
        stream.forEach(System.out::println);
        }
}
Output:
10
20
30
40
50
```

519. WAP to iterate a map using java 8?

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.HashMap;
import java.util.Map;

public class IterateMapUsingJava8Demo {

		public static void main(String[] args) {
		Map<Integer, String> map = new HashMap<Integer, String>();
		map.put(10, "Ten");
		map.put(100, "Ten");
		map.put(1000, "Ten");
		map.put(10000, "Thousand");
		map.put(100000, "Lakh");
		map.forEach((k,v) -> System.out.println("Key : "+ k + "\t Value : "+ v));
		}
}
Output:
Key : 10000	Value : Thousand
Key : 100000	Value : Lakh
Key : 100	Value : Ten
Key : 1000	Value : Ten
Key : 10	Value : Ten
```

520. WAP to filter null values and print non-null values.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class FilterNullandPrintNonNullDemo {

		public static void main(String[] args) {
		List<String> list = new ArrayList<String>();
		list.add("One");
		list.add("Two");
		list.add(null);
		list.add("Four");
		list.add("Five");
		list.stream()
		.filter(Objects :: nonNull)
		.forEach(System.out::println);
		}
}
Output:
One
Two
Four
Five
```

521. WAP to convert List into Map.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

public class Employee {

		private int empNo;
```

```
        private String name;
        private String dept;
        public Employee(int empNo, String name, String dept) {
        super();
        this.empNo = empNo;
        this.name = name;
        this.dept = dept;
        }
        public int getEmpNo() {
        return empNo;
        }
        public void setEmpNo(int empNo) {
        this.empNo = empNo;
        }
        public String getName() {
        return name;
        }
        public void setName(String name) {
        this.name = name;
        }

        public String getDept() {
        return dept;
        }
        public void setDept(String dept) {
        this.dept = dept;
        }
}

package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ConvertListToMapDemo {

        public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee(101, "Yogesh", "IT"));
        empList.add(new Employee(102, "Arpita", "Management"));
        empList.add(new Employee(103, "Shweta", "IT"));
        empList.add(new Employee(104, "Shruti", "IT"));
        empList.add(new Employee(105, "Rusty", "Automation"));
        Map<Integer, String> map = empList.stream()
                                .collect(Collectors.toMap(Employee::getEmpNo, Employee::getName));
        map.forEach((k,v) -> System.out.println("Employee ID : " + k + "\tEmployee Name : "+ v));
        }
}
```

Output:

```
Employee ID : 101   Employee Name : Yogesh
Employee ID : 102   Employee Name : Arpita
Employee ID : 103   Employee Name : Shweta
Employee ID : 104   Employee Name : Shruti
Employee ID : 105   Employee Name : Rusty
```

522. WAP to convert list to map incase you hace duplicates keys.

```java
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ConvertLisToMapForDuplicateKeys {

		public static void main(String[] args) {
		List<Employee> empList = new ArrayList<Employee>();
		empList.add(new Employee(101, "Yogesh", "IT"));
		empList.add(new Employee(102, "Arpita", "Management"));
		empList.add(new Employee(103, "Shruti", "IT"));
		empList.add(new Employee(103, "Shweta", "IT"));
		empList.add(new Employee(105, "Rusty", "Automation"));
		Map<Integer, String> map = empList.stream()
							.collect(Collectors.toMap(Employee::getEmpNo, Employee::getName,
(oldValue,newValue)->newValue));
		map.forEach((k,v) -> System.out.println("Employee ID : " + k + "\tEmployee Name : "+
v));
		}
}
Output:
Employee ID : 101   Employee Name : Yogesh
Employee ID : 102   Employee Name : Arpita
Employee ID : 103   Employee Name : Shweta
Employee ID : 105   Employee Name : Rusty
```

523. WAP to convert list to map and put all elements in sorted order.

```java
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class ConvertLisToMapWithSortedKeys {

		public static void main(String[] args) {
		List<Employee> empList = new ArrayList<Employee>();
		empList.add(new Employee(101, "Yogesh", "IT"));
		empList.add(new Employee(102, "Arpita", "Management"));
		empList.add(new Employee(103, "Shweta", "IT"));
		empList.add(new Employee(104, "Shruti", "IT"));
		empList.add(new Employee(105, "Rusty", "Automation"));
		Map<Integer,String> map = empList.stream()
					.sorted(Comparator.comparing(Employee::getName))
pNo, Employee::getName,(oldValue,newValue)-> newValue, LinkedHashMap::new));
```

```
            map.forEach((k,v) -> System.out.println("Employee ID : " + k + "\tEmployee Name : "+ v));
            }
}
Output:
Employee ID : 102   Employee Name : Arpita
Employee ID : 105   Employee Name : Rusty
Employee ID : 104   Employee Name : Shruti
Employee ID : 103   Employee Name : Shweta
Employee ID : 101   Employee Name : Yogesh
```

524. WAP to sort and print unique values.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;

public class UniqueValueSortAndPrint {

            public static void main(String[] args) {
            List<Integer> numList = Arrays.asList(30,10,20,50,20,40,40);
            numList.stream()
            .distinct()
            .sorted()
            .forEach(System.out::println);
            }
}
Output:
10
20
30
40
50
```

525. WAP to sort unique value and skip first 3 elements and print remaining list.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;

public class SkipElementsFromList {

            public static void main(String[] args) {
            List<Integer> numList = Arrays.asList(30,10,20,50,20,40,40);
            numList.stream()
            .distinct()
            .sorted()
            .skip(3)
            .forEach(i -> System.out.println(" "+ i));
            }
}
Output:
 40
50
```

526. **WAP to demonstrate the peek() method of Stream API.**

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;


public class PeekMethodDemo {

	public static void main(String[] args) {
	List<Integer> numList = Arrays.asList(30,10,20,50,20,40,40);
	numList.stream()
	.distinct()
	.peek(n -> System.out.println("Before Sorting "+ n))
	.sorted()
	.peek(n -> System.out.println("After Sorting "+ n))
	.collect(Collectors.toList())
	.forEach(n -> System.out.println("Final List "+ n));
	}
}
Output:
Before Sorting 30
Before Sorting 10
Before Sorting 20
Before Sorting 50
Before Sorting 40
After Sorting 10
After Sorting 20
After Sorting 30
After Sorting 40
After Sorting 50
```

527. WAP to convert Integer Array into Stream.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.stream.IntStream;

public class ConvertArrayToStreamDemo {

	public static void main(String[] args) {
	int[] array = {30,20,10,50,40};
	IntStream s = Arrays.stream(array);
	s.forEach(System.out::println);
	}
}
Output:
30
20
10
50
40
```

528. WAP to combine two Streams remove duplicates and print in ascending sorted manner.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;
```

```
import java.util.stream.Stream;

public class CombineTwoStreamAndSortedUnique {

        public static void main(String[] args) {
        Stream<Integer> s1 = Stream.of(10,20,50,40,60);
        Stream<Integer> s2 = Stream.of(20,40,50,70,80);
        Stream.concat(s1, s2)
        .distinct()
        .sorted()
        .forEach(System.out::println);
        }
}
Output:
10
20
40
50
60
70
80
```

529. WAP to combine two Streams remove duplicates and print in decending sorted manner.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.stream.Stream;

public class CombineTwoStreamAndSortedUnique {

        public static void main(String[] args) {
        Stream<Integer> s1 = Stream.of(10,20,50,40,60);
        Stream<Integer> s2 = Stream.of(20,40,50,70,80);
        Stream.concat(s1, s2)
        .distinct()
        .sorted((n1,n2) -> n2.compareTo(n1))
        .forEach(System.out::println);
        }
}
Output:
80
70
60
50
40
20
10
```

530. WAP to print the particular range like 1-10, 5-10.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.stream.IntStream;

public class PrintRange {

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

```
		IntStream stream1 = IntStream.range(10,15);
		stream1.forEach(n -> System.out.println(" "+ n));
		}
}
Output:
No 10
No 11
No 12
No 13
No 14
```

531. WAP to print the particular range and then its square and cube.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.stream.IntStream;

public class PrintParticularSeriesWithSquAndCub {


		public static void main(String[] args) {
		IntStream stream1 = IntStream.range(10,15);
		stream1.forEach(n -> System.out.println("No = "+ n + " Square = "+ n*n + " Cube = "+ n*n*n));
		}
}
Output:
No = 10 Square = 100 Cube = 1000
No = 11 Square = 121 Cube = 1331
No = 12 Square = 144 Cube = 1728
No = 13 Square = 169 Cube = 2197
No = 14 Square = 196 Cube = 2744
```

532. WAP to calculate the total of 1-10 using reduce() method.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.OptionalInt;
import java.util.stream.IntStream;

public class CalculateTotalUsingReduceMethod {

		public static void main(String[] args) {
		IntStream stream = IntStream.range(1, 11);
		OptionalInt oi = stream.reduce((n1,n2) -> (n1+n2));
		System.out.println(oi.getAsInt());
		}
}
Output:  55
```

533. WAP to convert set into map.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class ConvertSetToMapDemo {

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

```
		Set<Integer> set = new HashSet<Integer>();
		set.add(10);
		set.add(20);
		set.add(30);
		set.add(40);
		set.add(50);
		Map<Integer,Integer> map = set.stream()
							.collect(Collectors.toMap(n -> n, n -> n*n));
		System.out.println(map);
		}
}
Output:
{50=2500, 20=400, 40=1600, 10=100, 30=900}
```

534. WAP to calculate average and summary of list.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class CalculateAverage {

		public static void main(String[] args) {
		List<Integer> numList = new ArrayList<Integer>();
		numList.add(10);
		numList.add(20);
		numList.add(30);
		numList.add(40);
		numList.add(50);
		Double avg = numList.stream()
				.collect(Collectors.averagingInt(n -> n));
		System.out.println("Average = "+ avg);
		}
}
Output:
Average = 30.0
```

535. WAP to calculate the summary of list.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;

public class CalculateTheSummaryOfList {

		public static void main(String[] args) {
		List<Integer> numList = new ArrayList<Integer>();
		numList.add(10);
		numList.add(20);
		numList.add(30);
		numList.add(40);
		numList.add(50);
		IntSummaryStatistics summary = numList.stream()
```

```
                .collect(Collectors.summarizingInt(n -> n));
                System.out.println(summary);
                }
}
Output:
IntSummaryStatistics{count=5, sum=150, min=10, average=30.000000, max=50}
```

536. WAP to print odd numbers and print x, x*10 and x*100 format using FlatMap.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class PrintODDAndItsMultipleValues {

                public static void main(String[] args) {
                List<Integer> numList = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
                List<Integer> filteredList = numList.stream()
{                                                                                  if(n%2 == 0)
{                                                                                    return
Stream.empty();                                                                  }else
{                                                                                          return
n*100);                                                  }
ist());
                System.out.println(filteredList);
                }
}
Output:
[1, 10, 100, 3, 30, 300, 5, 50, 500, 7, 70, 700, 9, 90, 900]
```

537. WAP to concat the two different streams, remove duplicate and sort it.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.stream.Stream;

public class CombineStreamsSortDemo {

                public static void main(String[] args) {
                Stream<Integer> stream1 = Stream.of(10,20,30,40,50);
                Stream<Integer> stream2 = Stream.of(30,40,50,60,70);
                Stream.concat(stream1, stream2)
                .distinct()
                .sorted()
                .forEach(n -> System.out.println(" "+ n));
                }
}
Output:
 10
20
30
40
```

```
50
60
70
```

538. WAP to create stream and sort it in decending order.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.Comparator;
import java.util.stream.Stream;

public class CombineStreamsSortDecending {

	public static void main(String[] args) {
	Stream<Integer> stream1 = Stream.of(10,20,30,40,50);
	Stream<Integer> stream2 = Stream.of(30,40,50,60,70);
	Stream.concat(stream1, stream2)
	.distinct()
	.sorted(new DecendingSort())
	.forEach(n -> System.out.println(" "+ n));
	}
}

class DecendingSort implements Comparator<Integer>{

	@Override
	public int compare(Integer o1, Integer o2) {
	return o2.compareTo(o1);
	}
}
Output:
 70
60
50
40
30
20
10
```

539. WAP to count number of Male and Female employees using stream API.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CountMaleFemaleEmployeesDemo {

	public static void main(String[] args) {
	List<Employee> empList = new ArrayList<Employee>();
	empList.add(new Employee(101, "Yogesh","IT","Male"));
	empList.add(new Employee(102, "Arpita","Management","Female"));
	empList.add(new Employee(103, "Shruti","IT","Female"));
	empList.add(new Employee(104, "Shweta","IT","Female"));
	empList.add(new Employee(105, "Rusty","Automation","Male"));
```

```
            Map<String,Long> empCounter = empList.stream()
                                        .collect(Collectors.groupingBy(Employee::getGender,
Collectors.counting()));
            System.out.println(empCounter);
            }
}
Output:
{Female=3, Male=2}
```

540. **WAP to count number of departments in organization.**

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CountNumberOfDepts {

            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee(101, "Yogesh","IT","Male"));
            empList.add(new Employee(102, "Arpita","Management","Female"));
            empList.add(new Employee(103, "Shruti", "IT","Female"));
            empList.add(new Employee(104, "Shweta", "IT","Female"));
            empList.add(new Employee(105, "Rusty", "Automation","Male"));

            Map<String,Long> map = empList.stream()
                                        .collect(Collectors.groupingBy(Employee::getDept, Collectors.counting()));
            System.out.println(map);
            System.out.println("Total No. of Departments = "+ map.size());
            }
}
Output:
{Automation=1, Management=1, IT=3}
Total No. of Departments = 3
```

541. **WAP to print all department Names.**

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;

public class PrintAllDeptNames {

            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee(101, "Yogesh","IT","Male"));
            empList.add(new Employee(102, "Arpita","Management","Female"));
            empList.add(new Employee(103, "Shruti","IT","Female"));
            empList.add(new Employee(104, "Shweta","IT","Female"));
            empList.add(new Employee(105, "Rusty","Automation","Male"));
            empList.stream()
            .map(Employee::getDept)
            .distinct()
            .forEach(System.out::println);
            }
}
Output:
IT
Management
Automation
```

542. **WAP to calculate average age of Male and Female Employee.**

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class AverageAgeMaleAndFemale {

            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee(101, "Yogesh","IT","Male",35));
            empList.add(new Employee(102,"Arpita","Mgmt","Female",22));
            empList.add(new Employee(103, "Shruti","IT","Female",29));
            empList.add(new Employee(104, "Shweta","IT","Female",36));
            empList.add(new Employee(105, "Rusty","Automation","Male",25));
            Map<String, Double> map = empList.stream()
            .collect(Collectors.groupingBy(Employee::getGender, Collectors.averagingDouble(Employee::getAge)));
            System.out.println(map);
            }
}

Output:
{Female=29.0, Male=30.0}
```

543. **WAP to find out maximum salary.**

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class FindMaximumSalary {

            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee(101,"Yogesh", "IT","Male",35,100000.00));
            empList.add(new Employee(102, "Arpita", "Management","Female",22,150000.00));
            empList.add(new Employee(103, "Shruti", "IT","Female",29,65000.00));
            empList.add(new Employee(104, "Shweta", "IT","Female",36,75000.00));
            empList.add(new Employee(105, "Rusty", "Automation","Male",25,35000.00));

            Optional<Employee> emp = empList.stream().collect(Collectors.maxBy(
Comparator.comparingDouble(Employee::getSalary)));
            System.out.println("Employee Name = " +emp.get().getName() + " Salary = "+ emp.get().getSalary());
            }
}
Output:
Employee Name = Arpita Salary = 150000.0
```

544. WAP to print all employee names belongs to IT dept.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;

public class PrintAllEmpBelongsToITDept {

            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee(101, "Yogesh", "IT","Male",35,100000.00));
            empList.add(new Employee(102, "Arpita", "Management","Female",22,150000.00));
            empList.add(new Employee(103, "Shruti", "IT","Female",29,65000.00));
            empList.add(new Employee(104, "Shweta", "IT","Female",36,75000.00));
            empList.add(new Employee(105, "Rusty", "Automation","Male",25,35000.00));
            empList.stream()
            .filter(e -> e.getDept().equals("IT"))
            .map(Employee::getName)
            .forEach(System.out::println);
            }
}
Output:
Yogesh
Shruti
Shweta
```

545. WAP to print all employees with details belongs to IT dept.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class PrintAllEmpWithDetailsBelongsToITDept {

            public static void main(String[] args) {
            List<Employee> empList = new ArrayList<Employee>();
            empList.add(new Employee(101, "Yogesh", "IT","Male",35));
            empList.add(new Employee(102, "Arpita", "Management","Female",22));
            empList.add(new Employee(103, "Shruti", "IT","Female",29));
            empList.add(new Employee(104, "Shweta", "IT","Female",36));
            empList.add(new Employee(105, "Rusty","Automation","Male",25));
            List<Employee> itEmp = empList.stream()
                                                   .filter(e ->
e.getDept().equals("IT"))                                                .collect(Collectors.toList());
            System.out.println(itEmp);
            }
}
```

```
Output:
[Employee [empNo=101, name=Yogesh, dept=IT, gender=Male, age=35],
 Employee [empNo=103, name=Shruti, dept=IT, gender=Female, age=29],
 Employee [empNo=104, name=Shweta, dept=IT, gender=Female, age=36]
]
```

546. WAP to find out average salary department wise.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CalculateAvgSalaryDeptWise {

		public static void main(String[] args) {
		List<Employee> empList = new ArrayList<Employee>();
		empList.add(new Employee(101, "Yogesh", "IT","Male",35,100000.00));
		empList.add(new Employee(102, "Arpita", "Management","Female",22,150000.00));
		empList.add(new Employee(103, "Shruti", "IT","Female",29,65000.00));
		empList.add(new Employee(104, "Shweta", "IT","Female",36,75000.00));
		empList.add(new Employee(105, "Rusty", "Automation","Male",25,35000.00));
		Map<String, Double> deptSal = empList.stream()
						.collect(Collectors.groupingBy(Employee::getDept,
ry)));
		System.out.println(deptSal);
		}
}
Output:
{Automation=35000.0, Management=150000.0, IT=80000.0}
```

547. WAP to find out youngest employee of the company.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class FindYoungestEmployee {

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

		empList.add(new Employee(101, "Yogesh", "IT","Male",35,100000.00));
		empList.add(new Employee(102, "Arpita", "Management","Female",22,150000.00));
		empList.add(new Employee(103, "Shruti", "IT","Female",29,65000.00));
		empList.add(new Employee(104, "Shweta", "IT","Female",36,75000.00));
		empList.add(new Employee(105, "Rusty", "Automation","Male",25,35000.00));
		Optional<Employee> emp = empList.stream()
						.collect(Collectors.minBy(Comparator.comparingInt
ge)));
		System.out.println(emp);
```

```
            }
}
Output:
Optional[Employee [empNo=102, name=Arpita, dept=Management, gender=Female, age=22]]
```

548. WAP to most experience employee in company.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class PrintAllEmpNamesDeptWise {

        public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee(101, "Yogesh", "IT","Male",35));
        empList.add(new Employee(102, "Arpita", "Management","Female",22));
        empList.add(new Employee(103, "Shruti", "IT","Female",29));
        empList.add(new Employee(104, "Shweta", "IT","Female",36));
        empList.add(new Employee(105, "Rusty", "Automation","Male",25));
        Map<String, List<Employee>> empListDeptWise = empList.stream()
                                                .collect(Collectors.groupingBy
        (Employee::getDept,Collectors.toList()));
            System.out.println(empListDeptWise);
        }
}
Output:
{Automation=[Employee [empNo=105, name=Rusty, dept=Automation, gender=Male, age=25]],

Management=[Employee [empNo=102, name=Arpita, dept=Management, gender=Female, age=22]],

IT=[Employee [empNo=101, name=Yogesh, dept=IT, gender=Male, age=35], Employee [empNo=103,
name=Shruti, dept=IT, gender=Female, age=29], Employee [empNo=104, name=Shweta, dept=IT,
gender=Female, age=36]]}
```

549. WAP to calculate the total no of salary of company.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class CalculateTotalSalary {

        public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee(101, "Yogesh", "IT","Male",35,100000.00));
        empList.add(new Employee(102, "Arpita", "Management","Female",22,150000.00));
        empList.add(new Employee(103, "Shruti", "IT","Female",29,65000.00));
        empList.add(new Employee(104, "Shweta", "IT","Female",36,75000.00));
        empList.add(new Employee(105, "Rusty", "Automation","Male",25,35000.00));
        double salaryTotal = empList.stream()
                .collect(Collectors.summingDouble(Employee::getSalary));

        System.out.println("Total Salary "+ salaryTotal);
        }
}
Output:
Total Salary 425000.0
```

550. WAP to calculate total salary, min salary, max salary and average salary.

```
package simplifiedjava.crackedInterview.java8.streams.exmaples;

import java.util.ArrayList;
import java.util.DoubleSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;

public class CalculateSalarySummary {

        public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee(101, "Yogesh", "IT","Male",35,100000.00));
        empList.add(new Employee(102, "Arpita", "Management","Female",22,150000.00));
        empList.add(new Employee(103, "Shruti", "IT","Female",29,65000.00));
        empList.add(new Employee(104, "Shweta", "IT","Female",36,75000.00));
        empList.add(new Employee(105, "Rusty", "Automation","Male",25,35000.00));
        DoubleSummaryStatistics summary = empList.stream()
        .collect(Collectors.summarizingDouble(Employee::getSalary));
        System.out.println("Total Salary   = "+ summary.getSum());
        System.out.println("Average Salary = "+ summary.getAverage());
        System.out.println("Maximum Salary = "+ summary.getMax());
        System.out.println("Minimum Salary = "+ summary.getMin());
        System.out.println("Total Count    = "+ summary.getCount());
        }
}
Output:
Total Salary   = 425000.0
Average Salary = 85000.0
Maximum Salary = 150000.0
Minimum Salary = 35000.0
Total Count    = 5
```