Explore More

Subcription : Premium CDAC NOTES & MATERIAL @99

Contact to Join
Premium Group

Click to Join
Telegram Group

For More E-Notes

Join Our Community to stay Updated

TAP ON THE ICONS TO JOIN!

	codewitharrays.in freelance project available to buy contact on 8007592194	
SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

	freelance_Project available to buy contact on 8007592194	
21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+NextJs+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql

41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48	Train Ticket Booking Project	React+Springboot+MySql
49	Quizz Application Project	JSP+Springboot+MySql
50	Hotel Room Booking Project	React+Springboot+MySql
51	Online Crime Reporting Portal Project	React+Springboot+MySql
52	Online Child Adoption Portal Project	React+Springboot+MySql
53	online Pizza Delivery System Project	React+Springboot+MySql
54	Online Social Complaint Portal Project	React+Springboot+MySql
55	Electric Vehical management system Project	React+Springboot+MySql
56	Online mess / Tiffin management System Project	React+Springboot+MySql
57		React+Springboot+MySql
58		React+Springboot+MySql
59		React+Springboot+MySql
60		React+Springboot+MySql

Spring Boot + React JS + MySQL Project List

Sr.No	Project Name	YouTube Link
1	Online E-Learning Hub Platform Project	https://youtu.be/KMjyBaWmgzg?si=YckHuNzs7eC84-IW
2	PG Mate / Room sharing/Flat sharing	https://youtu.be/4P9clHg3wvk?si=4uEsi0962CG6Xodp
3	Tour and Travel System Project Version 1.0	https://youtu.be/-UHOBywHaP8?si=KHHfE_A0uv725f12
4	Marriage Hall Booking	https://youtu.be/VXz0kZQi5to?si=llOS-QG3TpAFP5k7
5	Ecommerce Shopping project	https://youtu.be/vJ_C6LkhrZ0?si=YhcBylSErvdn7paq
6	Bike Rental System Project	https://youtu.be/FIzsAmIBCbk?si=7ujQTJqEgkQ8ju2H
7	Multi-Restaurant management system	https://youtu.be/pvV-pM2Jf3s?si=PgvnT-yFc8ktrDxB
8	Hospital management system Project	https://youtu.be/IynIouBZvY4?si=CXzQs3BsRkjKhZCw
9	Municipal Corporation system Project	https://youtu.be/cVMx9NVyI4I?si=qX0oQt-GT-LR_5jF
10	Tour and Travel System Project version 2.0	https://youtu.be/_4u0mB9mHXE?si=gDiAhKBowi2gNUKZ

Sr.No	Project Name	YouTube Link
11	Tour and Travel System Project version 3.0	https://youtu.be/Dm7nOdpasWg?si=P_Lh2gcOFhlyudug
12	Gym Management system Project	https://youtu.be/J8_7Zrkg7ag?si=LcxV51ynfUB7OptX
13	Online Driving License system Project	https://youtu.be/3yRzsMs8TLE?si=JRI_z4FDx4Gmt7fn
14	Online Flight Booking system Project	https://youtu.be/m755rOwdk8U?si=HURvAY2VnizIyJlh
15	Employee management system project	https://youtu.be/lD1iE3W_GRw?si=Y_jv1xV_BljhrD0H
16	Online student school or college portal	https://youtu.be/4A25aEKfei0?si=RoVgZtxMk9TPdQvD
17	Online movie booking system project	https://youtu.be/Lfjv_U74SC4?si=fiDvrhhrjb4KSlSm
18	Online Pizza Delivery system project	https://youtu.be/Tp3izreZ458?si=8eWAOzA8SVdNwlyM
19	Online Crime Reporting system Project	https://youtu.be/0UlzReSk9tQ?si=6vN0e70TVY1GOwPO
20	Online Children Adoption Project	https://youtu.be/3T5HC2HKyT4?si=bntP78niYH802I7N

1) After Java 8, what do you think about Java? Is it still an object oriented language or it has turned into functional programming language?

Java is still an object oriented language where everything is done keeping objects (data) in mind. But, with the introduction of new features in Java 8, you can use Java as a functional programming language also. You can treat it as as an added advantage over the other languages which are either object oriented or functions oriented. From Java 8, you can use Java either in an object-oriented programming paradigm or in a functional programming paradigm. It supports both.

2) What are the three main features of Java 8 which make Java as a functional programming language?

Lambda expressions, functional interfaces and Stream API are the three main features of Java 8 which enables developers to write functional style of programming in Java also.

3) What are lambda expressions? How this feature has changed the way you write code in Java? Explain with some before Java 8 and after Java 8 examples?

Lambda Expressions can be defined as methods without names i.e anonymous functions. Like methods, they also have parameters, a body, a return type and possible list of exceptions that can be thrown. But unlike methods, neither they have names nor they are associated with any particular class.

Lambda expressions are used where an instance of functional interface is expected. Before Java 8, anonymous inner classes are used for this purpose. After Java 8, you can use lambda expressions to implement functional interfaces.

These lambda expressions have changed the style of programming in Java significantly. They have made the Java code more clear, concise and readable than before. For example,

Below code shows how `Comparator` interface is implemented using anonymous inner class before Java 8.

```
Comparator<Student> idComparator = new Comparator<Student>() {
            @Override
            public int compare(Student s1, Student s2) {
                return s1.getID()-s2.getID();
            }
        };
```

and after Java 8, above code can be written in a single line using Java 8 lambda expressions as below.

```
Comparator<Student> idComparator = (Student s1, Student s2) -> s1.getID()-s2.getID();
```

Another example,

Implementation of `Runnable` interface using anonymous inner class before Java 8 :

```
Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Runnable Implementation Using Anonymous Inner Class");
            }
        };
```

Implementation of *Runnable* interface using lambda expressions after Java 8 :

```
Runnable r = () -> System.out.println("Runnable Implementation Using Lambda Expressions");
```

4) How the signature of lambda expressions are determined?

The signature of lambda expressions are derived from the signature of abstract method of functional interface. For example,

`run()` method of `Runnable` interface accepts nothing and returns nothing. Then signature of lambda expression implementing `Runnable` interface will be `() -> void` .

`compare()` method of `Comparator` interface takes two arguments of type `Object` and returns `int`. Then signature of lambda expression for implementing `Comparator` interface will be `(Object, Object) -> int`.

5) How the compiler determines the return type of a lambda expression?

Compiler uses target type to check the return type of a lambda expression.

For example,

```
1  Runnable r = () -> System.out.println("Runnable Implementation Using Lambda Expressions");
```

In this example, target type of lambda expression is `Runnable`. Compiler uses `run()` method of `Runnable` interface to check the return type of lambda expression.

6) Can we use non-final local variables inside a lambda expression?

No. Only final local variables are allowed to use inside a lambda expressions just like anonymous inner classes.

7) What are the advantages of lambda expressions?

Lambda expressions let you to write more clear, concise and readable code.

Lambda expressions removes verbosity and repetition of code.

See More : Java 8 Lambda Expressions

8) What are the functional interfaces? Do they exist before Java 8 or they are the whole new features introduced in Java 8?

Functional interfaces are the interfaces which has exactly one abstract method. Functional interfaces provide only one functionality to implement.

There were functional interfaces exist before Java 8. It is not like that they are the whole new concept introduced only in Java 8. `Runnable`, `ActionListener`, `Callable` and `Comaprator` are some old functional interfaces which exist even before Java 8.

The new set of functional interfaces are introduced in Java 8 for writing lambda expressions. Lambda expressions must implement any one of these new functional interfaces.

9) What are the new functional interfaces introduced in Java 8? In which package they have kept in?

Below is the list of new functional interfaces introduced in Java 8. They have kept in `java.util.function` package.

Functional Interface And Its Abstract Method	Operation It Represents.	When To Use?	Related Functional Interfaces To Support Primitive Types
Predicate boolean test(T t)	Represents an operation which takes one argument and returns boolean.	Use this interface when you want to evaluate a boolean expression which takes an argument of type T.	IntPredicate LongPredicate DoublePredicate
Consumer void accept(T t)	Represents an operation that accepts single argument and returns nothing.	Use this interface when you want to perform some operations on an object.	IntConsumer LongConsumer DoubleConsumer
Function R apply(T t)	Represents an operation that accepts an argument of type T and returns a result of type R.	Use this interface when you want to extract a data from an existing data.	IntFunction LongFunction DoubleFunction ToIntFunction ToLongFunction ToDoubleFunction IntToLongFunction IntToDoubleFunction LongToDoubleFunction LongToIntFunction DoubleToIntFunction DoubleToLongFunction
Supplier T get()	Represents an operation which takes nothing but returns a result of type T.	Use this interface when you want to create new objects.	BooleanSupplier IntSupplier LongSupplier DoubleSupplier
BiPredicate boolean test(T t, U u)	Represents a predicate of two arguments.	Use this interface when you want to evaluate a boolean expression of two arguments.	
BiConsumer void accept(T t, U u)	Represents an operation that accepts two arguments and returns nothing.	Use this interface when you want to perform some operations on two objects.	ObjIntConsumer ObjLongConsumer ObjDoubleConsumer
BiFunction R apply(T t, U u)	Represents an operation which takes two arguments and produces a result.	Use this interface when you want to extract result data from two existing objects.	ToIntBiFunction ToLongBiFunction ToDoubleBiFunction
UnaryOperator (extends Function)	Same as Function but argument and result should be of same type.	Same as Function.	IntUnaryOperator LongUnaryOperatot DoubleUnaryOperator
BinaryOperator (extends BiFunction)	Same as BiFunction but argument and result should be of same type.	Same as BiFunction.	IntBinaryOperator LongBinaryOperator DoubleBinaryOperator

10) What is the difference between Predicate and BiPredicate?
Predicate is a functional interface which represents a boolean operation which takes one argument.
BiPredicate is also functional interface but it represents a boolean operation which takes two arguments.

11) What is the difference between Function and BiFunction?
Function is a functional interface which represents an operation which takes one argument of type T and returns result of type R.
BiFunction is also functional interface which represents an operation which takes two arguments of type T and U and returns a result of type R.

12) Which functional interface do you use if you want to perform some operations on an object and returns nothing?
Consumer

13) Which functional interface is the best suitable for an operation which creates new objects?
Supplier

14) When you use UnaryOperator and BinaryOperator interfaces?
UnaryOperator performs same operation as Function but it is used when type of the argument and result should be of same type.
BinaryOperator performs same operation as BiFunction but it is used when type of the arguments and result should be of same type.
15) Along with functional interfaces which support object types, Java 8 has introduced functional interfaces which support primitive types. For example, Consumer for object types and intConsumer, LongConsumer, DoubleConsumer for primitive types. What do you think, is it necessary to introduce separate interfaces for primitive types and object types?
Yes. If an input or output to an functional interface is a primitive type then using functional interfaces which support primitive types improves performance rather than using functional interfaces which support object types. Because it removes unnecessary boxing and unboxing of data.
16) How functional interfaces and lambda expressions are inter related?
Lambda expressions are introduced to implement functional interfaces in a simplest way and new functional interfaces are introduced to support lambda expressions in Java 8. Both together have given a new dimension to Java programming where you can write more complex data processing queries in a few lines of code.

See More : Java 8 Functional Interfaces

17) What are the method references? What is the use of them?
Java 8 method references can be defined as shortened versions of lambda expressions calling a specific method. Method references are the easiest way to refer a method than the lambdas calling a specific method. Method references will enhance the readability of your code.
18) What are the different syntax of Java 8 method references?

Method Type	Syntax
Static Method	ClassName::MethodName
Instance method of an existing object	ReferenceVariable::MethodName
Instance method of non-existing object	ClassName::MethodName
Constructor Reference	ClassName::new

See More : Java 8 Method References

19) What are the major changes made to interfaces from Java 8?
From Java 8, interfaces can also have concrete methods i.e methods with body along with abstract methods. This is the major change made to interfaces from Java 8 to help Java API developers to update and maintain the interfaces. The interfaces can have concrete methods either in the form of default methods or static methods.
20) What are default methods of an interface? Why they are introduced?
Default methods of an interface are the concrete methods for which implementing classes need not to give implementation. They inherit default implementation.
Default methods are introduced to add extra features to current interfaces without disrupting their existing implementations. For example, `stream()` is a default method which is added to `Collection` interface in Java 8. If `stream()` would have been added as abstract method, then all classes implementing `Collection` interface must have implemented `stream()` method which may have irritated existing users. To overcome such issues, default methods are introduced to interfaces from Java 8.
21) As interfaces can also have concrete methods from Java 8, how do you solve diamond problem i.e conflict of classes inhering multiple methods with same signature?
To solve the diamond problem, Java 8 proposes 3 rules to follow. They are,
Rule 1 : Select classes over interfaces
If your class inherit multiple methods with same signature then a method from super class is selected (Remember a class can inherit only one class).
Rule 2 : Select most specific interfaces than general interfaces.
If your class doesn't extend any class and inherit multiple methods with same signature from multiple interfaces which belong to same hierarchy, then a method from most specific interface is selected.
Rule 3 : InterfaceName.super.methodName()
If your class doesn't extend any class and inherit multiple methods with same signature from multiple interfaces which doesn't belong to same hierarchy, then override that method and from within body explicitly call desired method as `InterfaceName.super.methodName()`.
22) Why static methods are introduced to interfaces from Java 8?
Java API developers have followed the pattern of supplying an utility class along with an interface to perform basic operations on such objects.
For example, `Collection` and `Collections`. `Collection` is an interface and `Collections` is an utility class containing only static methods which operate on `Collection` objects.
But from Java 8, they have break this pattern by introducing static methods to interfaces. With the introduction of static methods to interface, such utility classes will disappear gradually and methods to perform basic operations will be kept as static methods in interface itself.

See More : Java 8 Interface Changes

23) What are streams? Why they are introduced?
streams can be defined as operations on data. They are the sequence of elements from a source which support data processing operations. Using Java 8 Streams, you can write most complex data processing queries without much difficulties.
Almost every Java application use Collections API to store and process the data. Despite being the most used Java API, it is not easy to write the code for even some common data processing operations like filtering, finding, matching, sorting, mapping etc using Collections

API . So, there needed Next-Gen API to process the data. So Java API designers have come with Java 8 Streams API to write more complex data processing operations with much of ease.

24) Can we consider streams as another type of data structure in Java? Justify your answer?

You can't consider streams as data structure. Because they don't store the data. You can't add or remove elements from the streams. They are the just operations on data. Stream consumes a data source, performs operations on it and produces the result. Source may be a collection or an array or an I/O resource. They don't modify the source.

25) What are intermediate and terminal operations?

The operations which return stream themselves are called intermediate operations. For example – `filter()` , `distinct()` , `sorted()` etc.

The operations which return other than stream are called terminal operations. `count()` . `min()` , `max()` are some terminal operations.

See More : Intermediate Vs Terminal Operations

26) What do you mean by pipeline of operations? What is the use of it?

A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation. Pipe-lining of operations let you to write database-like queries on a data source. Using this, you can write more complex data processing queries with much of ease.

27) "Stream operations do the iteration implicitly" what does it mean?

Collections need to be iterated explicitly. i.e you have to write the code to iterate over collections. But, all stream operations do the iteration internally behind the scene for you. You need not to worry about iteration at all while writing the code using Java 8 Streams API.

28) Which type of resource loading do Java 8 streams support? Lazy Loading OR Eager Loading?

Lazy Loading.

29) What are short circuiting operations?

Short circuiting operations are the operations which don't need the whole stream to be processed to produce a result. For example – `findFirst()` , `findAny()` , `limit()` etc.

30) What are selection operations available in Java 8 Stream API?

Operation	Description
filter()	Selects the elements which satisfy the given predicate.
distinct()	Selects only unique elements
limit()	Selects first *n* elements
skip()	Selects the elements after skipping first *n* elements

31) What are sorting operations available in Java 8 streams?

There is only one sorting operation available in Java 8 streams which is `sorted()` . It has two versions. One which takes no argument sorts the elements in natural order and another one which takes `Comparator` as an argument sorts the elements according to supplied `Comparator` .

32) What are reducing operations? Name the reducing operations available in Java 8 streams?

Reducing operations are the operations which combine all the elements of a stream repeatedly to produce a single value. For example, counting number of elements, calculating average of elements, finding maximum or minimum of elements etc.

Reducing operations available in Java 8 streams are,

Operation	Description
min()	Returns minimum element
max()	Returns maximum element
count()	Returns the number of elements
collect()	Returns mutable result container

33) What are the matching operations available in Java 8 streams?

Operation	Description
anyMatch()	Returns true if any one element of a stream matches with given predicate
allMatch()	Returns true if all the elements of a stream matches with given predicate
noneMatch()	Returns true only if all the elements of a stream doesn't match with given predicate.

34) What are searching / finding operations available in Java 8 streams?

Operation	Description
findFirst()	Returns first element of a stream
findAny()	Randomly returns any one element in a stream

35) Name the mapping operations available in Java 8 streams?

Operation	Description
map()	Returns a stream consisting of results after applying given function to elements of the stream.
flatMap()	

36) What is the difference between map() and flatMap()?
Java 8 map() and flatMap() are two important methods of `java.util.stream.Stream` interface used for transformation or mapping operations. Both are intermediate operations. The only difference is that `map()` takes `Stream<T>` as input and return `Stream<R>` where as `flatMap()` takes `Stream<Stream<T>` as input and return `Stream<R>` i.e `flatmap()` removes extra layer of nesting around input values.

See More : Differences Between Java 8 map() And flatMap()

37) What is the difference between limit() and skip()?
`limit()` is an intermediate operation in Java 8 streams which returns a stream containing first *n* elements of the input stream.
`skip()` is also an intermediate operation in Java 8 streams which returns a stream containing the remaining elements of the input stream after skipping first n elements.
38) What is the difference between findFirst() and findAny()?
`findFirst()` is a terminal operation in Java 8 streams which returns first element of the input stream. The result of this operation is predictable.
`findAny()` is also terminal operation in Java 8 streams which randomly returns any one element of the input stream. The result of this operation is unpredictable. It may select any element in a stream.
39) Do you know Stream.collect() method, Collector interface and Collectors class? What is the relation between them?
`collect()` method is a terminal operation in `Stream` interface. It is a special case of reduction operation which returns mutable result container such as `List` , `Set` or `Map` .
`Collector` is an interface in `java.util.stream` package.
`Collectors` class, also a member of `java.util.stream` package, is an utility class containing many static methods which perform some common reduction operations.
All the methods of `Collectors` class return `Collector` type which will be supplied to `collect()` method as an argument.
40) Name any 5 methods of Collectors class and their usage?

Method	Description
joining()	Concatenates input elements separated by the specified delimiter.
counting()	Counts number of input elements
groupingBy()	Groups the input elements according supplied classifier and returns the results in a *Map*.
partitioningBy()	Partitions the input elements according to supplied *Predicate* and returns a *Map<Boolean, List<T>>*
toList()	Collects all input elements into a new *List*

41) What are the differences between collections and streams?

Collections	Streams
Collections are mainly used to store and group the data.	Streams are mainly used to perform operations on data.
You can add or remove elements from collections.	You can’t add or remove elements from streams.
Collections have to be iterated externally.	Streams are internally iterated.
Collections can be traversed multiple times.	Streams are traversable only once.
Collections are eagerly constructed.	Streams are lazily constructed.
Ex : List, Set, Map…	Ex : filtering, mapping, matching…

See More : Collections Vs Streams

42) What is the purpose of Java 8 Optional class?
Java 8 Optional class is used represent an absence of a value i.e null. Before Java 8, if-constructs are used to check for null value. But, Optional class gives better mechanism to handle null vale or absence of a value.

See More : Java 8 Optional Class

43) What is the difference between Java 8 Spliterator and the iterators available before Java 8?

Iterator	Spliterator
It performs only iteration.	It performs splitting as well as iteration.
Iterates elements one by one.	Iterates elements one by one or in bulk.
Most suitable for serial processing.	Most suitable for parallel processing.
Iterates only collection types.	Iterates collections, arrays and streams.
Size is unknown.	You can get exact size or estimate of the size.
Introduced in JDK 1.2.	Introduced in JDK 1.8.
You can't extract properties of the iterating elements.	You can extract some properties of the iterating elements.
External iteration.	Internal iteration.

See More : Differences Between Iterator Vs Spliterator

44) What is the difference between Java 8 StringJoiner, String.join() and Collectors.joining()?

`StringJoiner` is a class in `java.util` package which internally uses `StringBuilder` class to join the strings. Using `StringJoiner`, you can join only the strings, but not the array of strings or list of strings.

`String.join()` method internally uses `StringJoiner` class. This method can be used to join strings or array of strings or list of strings, but only with delimiter not with prefix and suffix.

`Collectors.joining()` method can also be used to join strings or array of strings or list of strings with delimiter and it also supports prefix and suffix.

See More : Java 8 StringJoiner, String.join() And Collectors.joining()

45) Name three important classes of Java 8 Date and Time API?

`java.time.LocalDate`, `java.time.LocalTime` and `java.time.LocalDateTime`

46) How do you get current date and time using Java 8 features?

```
LocalDateTime currentDateTime = LocalDateTime.now();
```

Questions from 47 to 53 depends on the following `Student` class.

```
class Student
{
    String name;

    int id;

    String subject;

    double percentage;

    public Student(String name, int id, String subject, double percentage)
    {
        this.name = name;
        this.id = id;
        this.subject = subject;
        this.percentage = percentage;
    }

    public String getName()
    {
        return name;
    }

    public int getId()
    {
        return id;
    }

    public String getSubject()
    {
        return subject;
    }

    public double getPercentage()
    {
```

```
        return percentage;
    }

    @Override
    public String toString()
    {
        return name+"-"+id+"-"+subject+"-"+percentage;
    }
}
```

47) Given a list of students, write a Java 8 code to partition the students who got above 60% from those who didn't?

```
Map<Boolean, List<Student>> studentspartionedByPercentage = studentList.stream().collect(Collectors.partit
```

48) Given a list of students, write a Java 8 code to get the names of top 3 performing students?

```
List<Student> top3Students = studentList.stream().sorted(Comparator.comparingDouble(Student::getPercentage
```

49) Given a list of students, how do you get the name and percentage of each student?

```
Map<String, Double> namePercentageMap = studentList.stream().collect(Collectors.toMap(Student::getName, St
```

50) Given a list of students, how do you get the subjects offered in the college?

```
Set<String> subjects = studentList.stream().map(Student::getSubject).collect(Collectors.toSet());
```

51) Given a list of students, write a Java 8 code to get highest, lowest and average percentage of students?

```
DoubleSummaryStatistics studentStats = studentList.stream().collect(Collectors.summarizingDouble(Student:

System.out.println("Highest Percentage : "+studentStats.getMax());

System.out.println("Lowest Percentage : "+studentStats.getMin());

System.out.println("Average Percentage : "+studentStats.getAverage());
```

52) How do you get total number of students from the given list of students?

```
Long studentCount = studentList.stream().collect(Collectors.counting());
```

53) How do you get the students grouped by subject from the given list of students?

```
Map<String, List<Student>> studentsGroupedBySubject = studentList.stream().collect(Collectors.groupingBy(S
```


Questions from 54 to 61 are on the following `Employee` class.

```
class Employee
{
    int id;

    String name;

    int age;

    String gender;

    String department;

    int yearOfJoining;

    double salary;

    public Employee(int id, String name, int age, String gender, String department, int yearOfJoining, do
    {
        this.id = id;
```

```
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.department = department;
        this.yearOfJoining = yearOfJoining;
        this.salary = salary;
    }

    public int getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public String getGender()
    {
        return gender;
    }

    public String getDepartment()
    {
        return department;
    }

    public int getYearOfJoining()
    {
        return yearOfJoining;
    }

    public double getSalary()
    {
        return salary;
    }

    @Override
    public String toString()
    {
        return "Id : "+id
                +", Name : "+name
                +", age : "+age
                +", Gender : "+gender
                +", Department : "+department
                +", Year Of Joining : "+yearOfJoining
                +", Salary : "+salary;
    }
}
```


54) Given a list of employees, write a Java 8 code to count the number of employees in each department?

```
Map<String, Long> employeeCountByDepartment =
employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));
```

55) Given a list of employees, find out the average salary of male and female employees?

```
Map<String, Double> avgSalaryOfMaleAndFemaleEmployees=
                employeeList.stream().collect(Collectors.groupingBy(Employee::getGender, Collectors.averag
```

56) Write a Java 8 code to get the details of highest paid employee in the organization from the given list of employees?

```
Optional<Employee> highestPaidEmployeeWrapper=
employeeList.stream().collect(Collectors.maxBy(Comparator.comparingDouble(Employee::getSalary)));
```

57) Write the Java 8 code to get the average age of each department in an organization?

```
Map<String, Double> avgAgeOfEachDepartment =
                employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.av
```

58) Given a list of employees, how do you find out who is the senior most employee in the organization?

```
Optional<Employee> seniorMostEmployeeWrapper=
employeeList.stream().sorted(Comparator.comparingInt(Employee::getYearOfJoining)).findFirst();
```

59) Given a list of employees, get the details of the most youngest employee in the organization?

```
Optional<Employee> youngestEmployee =
                employeeList.stream().min(Comparator.comparingInt(Employee::getAge));
```

60) How do you get the number of employees in each department if you have given a list of employees?

```
Map<String, Long> employeeCountByDepartment=
employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));
```

61) Given a list of employees, find out the number of male and female employees in the organization?

```
Map<String, Long> noOfMaleAndFemaleEmployees=
employeeList.stream().collect(Collectors.groupingBy(Employee::getGender, Collectors.counting()));
```

See More : Solving Real Time Queries Using Java 8 Features -Employee Management System

62) What will be the output of the following statement?

```
System.out.println(IntStream.range(0, 5).sum());
```

Java Threads Interview Questions And Answers :

1) What is multithreaded programming? Does Java supports multithreaded programming? Explain with an example?
In a program or in an application, when two or more threads execute their task simultaneously then it is called multi threaded programming.
Yes, Java supports multithreaded programming.
For example, in the below code, main thread which is responsible for executing the main() method, creates two threads – t1 and t2. t1 prints numbers from 1 to 1000 and t2 prints numbers from 1001 to 2000. These two threads execute their task simultaneously not one after the other. This is called multi threaded programming.

```
//Thread1 : The task of this thread is to print numbers from 1 to 1000

class Thread1 extends Thread
{
    @Override
    public void run()
    {
        for (int i = 1; i <= 1000; i++)
        {
            System.out.println(i);
        }
    }
}

//Thread2 : The task of this thread is to print numbers from 1001 to 2000

class Thread2 extends Thread
{
    @Override
    public void run()
    {
        for (int i = 1001; i <= 2000; i++)
        {
            System.out.println(i);
        }
    }
}

public class JavaThreadsInterviewQuestions
{
    //Main Thread : The task of this thread is to execute main() method

    public static void main(String[] args)
    {
        //Creating and starting first thread

        Thread1 t1 = new Thread1();
        t1.start();

        //Creating and starting second thread

        Thread2 t2 = new Thread2();
        t2.start();

        //Both these two threads will be executed simultaneously
    }
}
```


Also Read : 30+ Java Exception Handling Interview Questions & Answers

2) In how many ways, you can create threads in Java? What are those? Explain with examples?
There are two ways two create threads in Java.
. By extending `java.lang.Thread` class
. By implementing `java.lang.Runnable` interface
1) Creating thread by extending java.lang.Thread class :
Your thread must extend Thread class and override run() method. Whatever the task which you want to be performed by this thread, keep that task in the overridden run() method.

```
class MyThread extends Thread
{
    @Override
    public void run()
```

```
    {
        //Keep the task to be performed here
    }
}
```

Where ever you want this task to be performed, create an object to your thread class and call start() method.

```
MyThread myThread = new MyThread();
myThread.start();
```

2) By implementing java.lang.Runnable interface
Runnable interface has only one methode i.e run() method. Your thread class must implement Runnable interface and override run() method and keep the task to be performed in this run() method.

```
class MyRunnable implements Runnable
{
    @Override
    public void run()
    {
        //Keep the task to be performed here
    }
}
```

Whenever you want this task to be performed, create an object to `java.lang.Thread` class by passing an object of your thread class which implements Runnable interface and call start() method.

```
Thread t = new Thread(new MyRunnable());
t.start();
```

Also Read : Extends Thread Vs Implements Runnable

3) How many types of threads are there in Java? Explain?
There are two types of threads in Java. They are,
. User Threads
. Daemon Threads
User threads are threads which are created by the application or user. They are high priority threads. JVM will not exit until all user threads finish their execution. JVM wait for user threads to finish their task. These threads are foreground threads.
Daemon threads are threads which are mostly created by the JVM. These threads always run in background. These threads are used to perform some background tasks like garbage collection. These threads are less priority threads. JVM will not wait for these threads to finish their execution. JVM will exit as soon as all user threads finish their execution.

Also Read : User Threads Vs Daemon Threads

4) What is the default daemon status of a thread? How do you check it?
Default daemon status of a thread is inherited from it's parent thread i.e a thread created by user thread will be a user thread and a thread created by a daemon thread will be a daemon thread.
`isDaemon()` method is used to check whether a thread is daemon thread or not.
5) Can you convert user tread into daemon thread and vice-versa? Explain with example?
Yes, you can convert user thread into daemon thread and vice-versa using `setDaemon()` method. But, it has to be done before starting the thread. If you call this method after starting the thread, you will get `java.lang.IllegalThreadStateException` .

```
class UserThread extends Thread
{
    @Override
```

```
    public void run()
    {
        System.out.println("Keep user thread task here...");
    }
}

public class JavaThreadsInterviewQuestions
{
    public static void main(String[] args)
    {
        UserThread userThread = new UserThread();

        userThread.setDaemon(true);

        userThread.start();
    }
}
```

6) Is it possible to give a name to a thread? If yes, how do you do that? What will be the default name of a thread if you don't name a thread?

Yes, it is possible to give a name to a thread. It can be done via `setName()` method or else you can pass the name while creating the thread itself.

```
class MyThread extends Thread
{
    public MyThread(String name)
    {
        super(name);
    }

    @Override
    public void run()
    {
        System.out.println("Keep the task to be performed here...");
    }
}

public class JavaThreadsInterviewQuestions
{
    public static void main(String[] args)
    {
        MyThread myThread = new MyThread("My_Thread");

        myThread.start();

        System.out.println(myThread.getName());   //Output : My_Thread

        myThread.setName("My_Thread_2.0");

        System.out.println(myThread.getName());   //Output : My_Thread_2.0
    }
}
```


If you don't name a thread, thread will get default name. Default name of the thread will consist of a word "Thread", followed by hyphen (-) and followed by an integer number starting from 0 like Thread-0, Thread-1, Thread-2.

7) Can we change the name of the main thread? If yes, How?

Yes, we can change the name of the main thread. Below code shows how to do it.

```
public class JavaThreadsInterviewQuestions
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();

        System.out.println(t.getName());       //Output : main

        t.setName("My_Main_Thread");

        System.out.println(t.getName());      //Output : My_Main_Thread
    }
}
```

8) Do two threads can have same name? If yes then how do you identify the threads having the same name?

Yes, two threads can have same name. In such scenarios, Thread ID can be used to identify the threads. Thread ID is a unique long number which remains unchanged through out the life of a thread. Thread ID can be retrieved using `getID()` method.

Also Read : 40+ Java String Interview Questions And Answers

9) What are MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY?
MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY are three constant fields in `java.lang.Thread` class which define lowest, normal and highest priority of a thread respectively.
`MIN_PRIORITY` : It defines the lowest priority that a thread can have and it's value is 1.
`NORM_PRIORITY` : It defines the normal priority that a thread can have and it's value is 5.
`MAX_PRIORITY` : It defines the highest priority that a thread can have and it's value is 10.
10) What is the default priority of a thread? Can we change it? If yes, how?
The default priority of a thread is same as that of it's parent. We can change the priority of a thread at any time using `setPriority()` method.
11) What is the priority of main thread? Can we change it?
The priority of a main thread, if explicitly not set, is always NORM_PRIORITY i.e 5.
Yes, we can change the priority of a main thread using `setPriority()` method.

```
public class JavaThreadsInterviewQuestions
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();

        System.out.println(t.getPriority());       //Output : 5

        t.setPriority(8);

        System.out.println(t.getPriority());      //Output : 8
    }
}
```

See More : Java Thread Priority

12) What is the purpose of Thread.sleep() method?
`Thread.sleep()` is used to pause the execution of current thread for a specified period of time.
13) Can you tell which thread is going to sleep after calling myThread.sleep(5000) in the below program? is it main thread or myThread?

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        for (int i = 0; i <= 10000; i++)
        {
            System.out.println(i);
        }
    }
}

public class JavaThreadsInterviewQuestions
{
    public static void main(String[] args)
    {
        MyThread myThread = new MyThread();

        myThread.start();

        try
        {
            myThread.sleep(5000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

It is the main thread which is going to sleep not myThread. Because, when you call sleep() method, it is currently executing thread which is going to sleep, not on which you have called it.
To sleep myThread in the above program, call `Thread.sleep()` inside the run() method of MyThread class.
14) Does the thread releases the lock it holds when it is going for sleep?

No. When the thread is going for sleep, it does not release the synchronized locks it holds.

See More : Thread.sleep() Method

15) What is the purpose of join() method? Explain with an example?

`join()` method can be used to apply the order of execution on threads. Using `join()` method, you can make the currently executing thread to wait for the some other threads to finish their task. For example, let's us assume that there are two threads – thread1 and thread2. You can make thread1 to hold it's execution for some time so that thread2 can finish it's task. After thread2 finishes it's task, thread1 resumes it's execution. For this to happen, you should call `join()` method on thread2 within thread1.

See More : join() Method With An Example

16) What do you mean by synchronization? Explain with an example?

Through synchronization, we can make the threads to execute particular method or block in sync not simultaneously. When a method or block is declared as synchronized, only one thread can enter into that method or block. When one thread is executing synchronized method or block, the other threads which wants to execute that method or block have to wait until first thread executes that method or block. Thus avoiding the thread interference and achieving the thread safeness.

```
class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName()+" : "+i);
        }
    }
}

public class ThreadsInJava
{
    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        Thread t1 = new Thread("Thread - 1")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        Thread t2 = new Thread("Thread - 2")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        t1.start();

        t2.start();
    }
}
```


In the above example, both threads t1 and t2 wants to execute sharedMethod() of s1 object. But, sharedMethod() is declared as synchronized. So, whichever thread enters first into sharedMethod(), it continues to execute that method. The other thread waits for first thread to finish it's execution of sharedMethod(). It never enters into sharedMethod() until first thread is done with that method. That means, both threads are executing sharedMethod() one by one not simultaneously.

17) What is object lock or monitor?

The synchronization in Java is built around an entity called object lock or monitor. Below is the brief description about lock or monitor.
Whenever an object is created to any class, an object lock is created and is stored inside the object.
One object will have only one object lock associated with it.
Any thread wants to enter into synchronized methods or blocks of any object, they must acquire object lock associated with that object and release the lock after they are done with the execution.

The other threads which wants to enter into synchronized methods of that object have to wait until the currently executing thread releases the object lock.
To enter into static synchronized methods or blocks, threads have to acquire class lock associated with that class as static members are stored inside the class memory.

18) I want only some part of the method to be synchronized, not the whole method? How do you achieve that?

This can be done using synchronized blocks.

19) What is the use of synchronized blocks?

Synchronization slows down the application. Because, at any given time, only one thread can enter into synchronized method. Other threads have to wait until first thread finishes it's execution of that method. This slows down the execution of whole application.
Instead of synchronizing the whole method, synchronizing the only that part which is to be monitored for thread safe saves the time. This can be done using synchronized blocks.

20) What is mutex?

synchronized block takes one argument and it is called mutex. If synchronized block is defined inside non-static definition blocks like non-static methods, instance initializer or constructors, then this mutex must be an instance of that class. If synchronized block is defined inside static definition blocks like static methods or static initializer, then this mutex must be like ClassName.class.

21) Is it possible to make constructors synchronized?

Not possible. Synchronized keyword can not be used with constructors. But, constructors can have synchronized blocks.

22) Can we use synchronized keyword with variables?

No, you can't use synchronized keyword with variables. You can use synchronized keyword only with methods but not with variables, constructors, static initializers and instance initializers.

23) As you know that synchronized static methods need class level lock and synchronized non-static methods need object level lock. Is it possible to run these two methods simultaneously?

Yes. It is possible.

24) If a particular thread caught with exceptions while executing a synchronized method, does executing thread releases lock or not?

Thread must release the lock whether the execution is completed normally or caught with exceptions.

25) Synchronized methods or synchronized blocks – which one do you prefer?

Synchronized blocks are better than synchronized methods. Because, synchronizing some part of a method improves the performance than synchronizing the whole method.

See More : Synchronization In Java

26) What is deadlock in Java?

Deadlock in Java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources(Locks) they hold.

See More : Deadlock In Java

27) How do you programatically detect the deadlocked threads in Java?


```
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadInfo;
import java.lang.management.ThreadMXBean;

public class JavaThreadsInterviewQuestions
{
    public static void main(String[] args)
    {
        ThreadMXBean bean = ManagementFactory.getThreadMXBean();

        long ids[] = bean.findMonitorDeadlockedThreads();

        if(ids != null)
        {
            ThreadInfo threadInfo[] = bean.getThreadInfo(ids);

            for (ThreadInfo threadInfo1 : threadInfo)
            {
                System.out.println(threadInfo1.getThreadId());     //Prints the ID of deadlocked thread

                System.out.println(threadInfo1.getThreadName());   //Prints the name of deadlocked thread

                System.out.println(threadInfo1.getLockName());     //Prints the string representation of a

                System.out.println(threadInfo1.getLockOwnerId());  //Prints the ID of thread which curren

                System.out.println(threadInfo1.getLockOwnerName());  //Prints name of the thread which cu
            }
        }
    }
}
```

28) What do you know about lock ordering and lock timeout?
Lock ordering and lock timeout are two methods which are used to avoid the deadlock in Java.
Lock Ordering : In this method of avoiding the deadlock, some predefined order is applied for threads to acquire the locks they need. For example, If there are three threads t1, t2 and t3 running concurrently and they needed locks A, B and C. t1 needs A and B locks, t2 needs A and C locks and t3 needs A, B and C locks. If you define an order to acquire the locks like, Lock A must be acquired before Lock B and Lock B must be acquired before Lock c, then deadlock never occurs.
Lock Timeout : It is another deadlock preventive method in which we specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time.
29) How do you avoid the deadlock? Tell some tips?
Below are some tips that can be used to avoid the deadlock in Java.
Try to avoid nested synchronized blocks. Nested synchronized blocks makes a thread to acquire another lock while it is already holding one lock. This may create the deadlock if another thread wants the same lock which is currently held by this thread.
If you needed nested synchronized blocks at any cost, then make sure that threads acquire the needed locks in some predefined order. It is called lock ordering.
Another deadlock preventive tip is to specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time. Such method of specifying time to acquire the lock is called lock timeout.
Lock the code where it is actually needed. For example, if you want only some part of the method to be thread safety, then lock only that part not the whole method.

Also Read : Java Array Interview Questions And Answers

30) How threads communicate with each other in Java?
Threads in Java communicate with each other using wait(), notify() and notifyAll() methods.
wait() : This method tells the currently executing thread to release the lock of this object and wait until some other thread acquires the same lock and notify it using either notify() or notifyAll() methods.
notify() : This method wakes up one thread randomly that called wait() method on this object.
notifyAll() : This method wakes up all the threads that called wait() method on this object. But, only one thread will acquire lock of this object depending upon the priority.

See More : Interthread Communication Using wait(), notify() and notifyAll()

31) What is the difference between wait() and sleep() methods in Java?

wait()	**sleep()**
The thread which calls wait() method releases the lock it holds.	The thread which calls sleep() method doesn't release the lock it holds.
The thread regains the lock after other threads call either notify() or notifyAll() methods on the same lock.	No question of regaining the lock as thread doesn't release the lock.
wait() method must be called within the synchronized block.	sleep() method can be called within or outside the synchronized block.
wait() method is a member of java.lang.Object class.	sleep() method is a member of java.lang.Thread class.
wait() method is always called on objects.	sleep() method is always called on threads.
wait() is a non-static method of Object class.	sleep() is a static method of Thread class.
Waiting threads can be woken up by other threads by calling notify() or notifyAll() methods.	Sleeping threads can not be woken up by other threads. If done so, thread will throw InterruptedException.
To call wait() method, thread must have object lock.	To call sleep() method, thread need not to have object lock.

See More : wait() Vs sleep()

32) What is the difference between notify() and notifyAll() in Java?
notify() : When a thread calls *notify()* method on a particular object, only one thread will be notified which is waiting for the lock or monitor of that object. The thread chosen to notify is random i.e randomly one thread will be selected for notification. Notified thread doesn't get the lock of the object immediately. It gets once the calling thread releases the lock of that object.
notifyAll() : When a thread calls *notifyAll()* method on a particular object, all threads which are waiting for the lock of that object are notified. All notified threads will move from WAITING state to BLOCKED state. All these threads will get the lock of the object on a priority basis. The thread which gets the lock of the object moves to RUNNING state. The remaining threads will remain in BLOCKED state until they get the object lock.

See More : notify() Vs notifyAll()

33) Though they are used for inter thread communication, why wait(), notify() and notifyAll() methods are included in java.lang.Object class not in java.lang.Thread class?
See this post to know why wait(), notify() and notifyAll() methods are included in java.lang.Object class not in java.lang.Thread class
34) What do you know about interrupt() method? Why it is used?

interrupt() method is used to interrupt sleeping or waiting thread. The whole thread interruption mechanism depends on an internal flag called interrupt status. The initial value of this flag for any thread is false. When you call interrupt() method on a thread, interrupt status of that thread will be set to true. When a thread throws `InterruptedException` , this status will be set to false again.

35) How do you check whether a thread is interrupted or not?

`isInterrupted()` or `interrupted()` method is used to check whether a particular thread is interrupted or not.

36) What is the difference between isInterrupted() and interrupted() methods?

Both, isInterrupted() and interrupted() methods are used to check whether a particular thread is interrupted or not. Both these methods return current interrupt status of a thread. isInterrupted() is a non-static method where as interrupted() is a static method of `java.lang.Thread` class. The main difference between these two methods is that isInterrupted() doesn't clear the interrupt status where as interrupted() clears the interrupt status of a thread.

37) Can a thread interrupt itself? Is it allowed in Java?

Yes, a thread can interrupt itself. It is very much legal in Java.

See More : Thread Interruption In Java

38) Explain thread life cycle? OR Explain thread states in Java?

There are six thread states. They are NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING and TERMINATED. At any point of time, thread will be in any one of these states.

. NEW : A thread will be in this state before calling start() method.
. RUNNABLE : A thread will be in this state after calling the start() method.
. BLOCKED : A thread will be in this state when a thread is waiting for object lock to enter into synchronized method/block or a thread will be in this state if deadlock occurs.
. WAITING : A thread will be in this state when wait() or join() method is called.
. TIMED_WAITING : A thread will be in this state when sleep() or wait() with timeOut or join() with timeOut is called.
. TERMINATED : A thread will be in this state once it finishes it's execution.

39) In what state deadlocked threads will be?

Deadlocked threads will be in BLOCKED state.

40) What is the difference between BLOCKED and WAITING states?

a thread will be in WAITING state if it is waiting for notification from other threads. A thread will be in BLOCKED state if it is waiting for other thread to release the lock it wants.

A thread enters into WAITING state when it calls `wait()` or `join()` method on an object. Before entering into WAITING state, thread releases the lock of the object it holds. It will remain in WAITING state until any other thread calls either `notify()` or `notifyAll()` on the same object.

Once the other thread calls `notify()` or `notifyAll()` on the same object, one or all the threads which are WAITING for lock of that object will be notified. All the notified threads will not get the object lock immediately. They will get the object lock on a priority basis once the current thread releases the lock. Until that they will be in BLOCKED state.

See More : BLOCKED Vs WAITING States In Java

41) What is the difference between WAITING and TIMED_WAITING states?

A thread enters into WAITING state when it calls `wait()` or `join()` method on an object. Before entering into WAITING state, thread releases the lock of the object it holds. It will remain in WAITING state until any other thread calls either `notify()` or `notifyAll()` on the same object.

A thread will be in TIMED_WAITING state when sleep() or wait() with timeOut or join() with timeOut is called. Thread doesn't release the lock it holds before entering into this state. It will remain in this state till specified time is over.

42) Can we call start() method twice?

No, start() method must be called only once. If you call start() method second time, it will throw `IllegalThreadStateException` as thread is already started.

43) What is the difference between calling start() method and calling run() method directly as anyhow start() method internally calls run() method?

When you call start() method, a new thread is created and that newly created thread executes the task kept in the run() method. If you call run() method directly, no new thread is created. Any task kept in run() method is executed by the calling thread itself.

If you are calling run() method directly, then you are not making use of the multi-threaded programming concept. Because, when you call run() method directly, no new thread is created. run() method is executed by the calling thread itself. It just acts as normal method invocation. You are not using the concept of multi-threading.

See More : start() Vs run()

44) How do you stop a thread?

As `stop()` method has been deprecated, there are two ways through which you can stop a thread in Java. One is using `boolean variable` and second one is using `interrupt()` method.

See More : How To Stop A Thread In Java?

45) Suppose there are two threads T1 and T2 executing their task concurrently. If an exception occurred in T1, will it effect execution of T2 or it will execute normally?

T2 will execute normally. Exception is thread wise not execution wise. i.e exception effects the thread in which it occurs. Other threads will execute normally.

46) Which one is the better way to implement threads in Java? Is it using Thread class or using Runnable interface?

when multiple threads need to execute same task, then use Runnable interface. If multiple threads need to execute different tasks, then go for Thread class.

See More : Extends Thread Vs Implements Runnable In Java

47) What is the difference between program, process and thread?
Program is an executable file containing the set of instructions written to perform a specific job on your computer. For example, chrome.exe, notepad.exe…
Process is an executing instance of a program. For example, When you double click on the Google Chrome icon on your computer, you start a process which will run the Google Chrome program. When you double click on a notepad icon on your computer, a process is started that will run the notepad program.
Thread is the smallest executable unit of a process. For example, when you run a notepad program, operating system creates a process and starts the execution of main thread of that process.

See More : Program Vs Process vs Threads

48) What are the differences between user threads and daemon threads?

User Threads	**Daemon Threads**
JVM waits for user threads to finish their work. It will not exit until all user threads finish their work.	JVM will not wait for daemon threads to finish their work. It will exit as soon as all user threads finish their work.
User threads are foreground threads.	Daemon threads are background threads.
User threads are high priority threads.	Daemon threads are low priority threads.
User threads are created by the application.	Daemon threads, in most of time, are created by the JVM.
User threads are mainly designed to do some specific task.	Daemon threads are designed to support the user threads.
JVM will not force the user threads to terminate. It will wait for user threads to terminate themselves.	JVM will force the daemon threads to terminate if all user threads have finished their work.

See More : User Threads Vs Daemon Threads

49) What is the use of thread groups in Java?
Thread groups in Java are used to group similar threads into one unit. A thread group can contain a set of threads or other thread groups. The main use of thread groups is that you can handle multiple threads simultaneously.
50) What is the thread group of a main thread?
main thread belongs to main thread group.
51) What activeCount() and activeGroupCount() methods do?
`activeCount()` returns the number of active threads in a specified group and it's subgroups. `activeGroupCount()` returns the numbers of active thread groups in a specified group and it's subgroups.

See More : Thread Group In Java

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/ccee2025notes

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project