Java

Ques) Solid Principal

It's a design principal encourage us to create more maintainable, understandable and flexible software.

S – Single responsibility :

➔ A class should have one responsibility . It should have only one reason to change.

O- Open/Closed Model :

➔ The Open/Closed Principle states that software entities should be open for extension but closed for modification.

L – Liskov's Substitution :

➔ The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. **(Electrical Car Engine and Normal Car Engine example)**

I – Interface Segregation :

➔ The Interface Segregation Principle states that no client should be forced to depend on methods it does not use.
➔ Instead of one large interface, many smaller and more specific interfaces are preferred. **(Cook and waiter example)**

D- Dependency Inversion Principle :

➔ The Dependency Inversion Principle states that high-level modules should not depend on low-level modules.
➔ Both should depend on abstractions. **(Debit card and Credit card Example)**

Ques) Garbage Collection

- ➔ Garbage collection in Java is an automatic memory management process that helps Java programs run efficiently.
- ➔ The garbage collector automatically finds and removes objects that are no longer needed, freeing up memory in the heap.
- ➔ It runs in the background as a daemon thread, helping to manage memory efficiently .
- ➔ Java garbage collection is an automatic process that manages memory in the heap.
- ➔ It identifies which objects are still in use **(referenced)** and which are not in use (**unreferenced).**
- ➔ Unreferenced objects can be deleted to free up memory**.**

Suppose ,
we have **s1 = "data"** and **s2 = "data"** // defined with string literals
and suppose we make **s1 = null**.

If you set **s1 = null**, it means that the reference s1 no longer points to the string "data".
However, **the string "data" is still present in the string pool** and **remains reachable via s2**, which still references it.
The garbage collector will not collect the string "data" because it is still reachable through s2.

➔ A **weakly referenced object** is **cleared** by the **Garbage Collector** when it's weakly reachable.

Ques) Metaspace ?

Structure of the Heap

The Java heap is subdivided into three main areas:

- **Young Generation:** This is where all new objects are allocated. The Young Generation is further divided into one 'Eden' space and two 'Survivor' spaces. Objects initially reside in Eden and move to a Survivor space if they remain alive after a garbage collection event.
- **Old Generation:** Objects that have survived several garbage collection cycles in the Young Generation are promoted to the Old Generation. It is designed for objects with a longer lifecycle. The threshold of survival is set by the garbage collector's policies and can be tuned by the developer.
- **Permanent Generation (only in JVMs before Java 8):** This part of the heap holds metadata such as classes and methods, which do not change frequently.

➔ With Java 8 and later, the introduction of **Metaspace** has replaced the **Permanent Generation.**

Metaspace

Metaspace is a non-heap memory area that came into existence with Java 8, replacing the Permanent Generation. It is used to store metadata such as class definitions, method data, and field data. Unlike the heap, Metaspace is allocated out of the native memory, and its size is not fixed but can increase dynamically, which helps prevent the OutOfMemoryErrors that were possible with the Permanent Generation.

Monitoring and Managing Metaspace

Since Metaspace can grow dynamically, monitoring its size and usage is crucial to prevent system memory from being exhausted. Java provides several options for monitoring and managing the size of Metaspace, such as `-XX:MetaspaceSize` and `-XX:MaxMetaspaceSize`, which allow developers to set initial and maximum Metaspace sizes, respectively.

Importance of Metaspace

The introduction of Metaspace enhances performance and scalability by dynamically adjusting memory usage based on application demands, thereby allowing Java applications to manage class metadata more efficiently. This is particularly beneficial in environments where many classes are loaded and unloaded.

Ques) Explain 4 important pillar of OOPS concept ?

- ➔ Encapsulation
- ➔ Polymorphism
- ➔ Inheritance
- ➔ Abstraction

Ques) Explain finalize keyword and garbage collection in java ?

- ➔ Finalize() is a method of Object class.
- ➔ Garbage collectors calls it before destroying object from memory.
- ➔ This method helps the Garbage collectors close all the resources the object uses and helps JVM in memory optimization.
- ➔ Externally program can call System.gc() for the same operation.

Ques) Can we create abstract class .

- ➔ Yes

Ques) Can we create abstract class with non abstract method.

- ➔ Yes

Ques) Can we create object of an abstract class.

- ➔ No . **Reflection** and **cloning** can be used to create instances of classes, but the rules around abstract classes still apply.
- ➔ Reflection can be used to instantiate an abstract class if the abstract class has a concrete subclass.
- ➔ But you cannot directly instantiate an abstract class itself. You would **still need a concrete class** to create an object.

Ques) Can we serialize an abstract class.

➔ In Java, an abstract class can be serialized, but you can't instantiate it directly. You serialize objects of concrete subclasses that extend the abstract class.

Ques) Abstract class can have constructor ?

➔ In Java, an abstract class can have a constructor. Subclasses can call the abstract class constructor using the super() keyword.

Ques) Difference b/w abstract class and interface.

➔ Both are a mechanism to achieve abstraction.
➔ We can load multiple interfaces in a class but not the abstract class.
➔ Interfaces are lightweight as it does not have static block ,constructor and member variable just like a class.
➔ Abstract class are suitable when we have a group of related class that need to share the common code ,while interfaces are better for defining a contract that unrelated class can implement.

Ques) How we solve multiple inheritance problem using java8 .

➔ Multiple inheritance is a situation where a single child class inherits properties from multiple parent classes.
➔ Java rejects multiple inheritance relationships because they create ambiguities.

```
class C implements A, B {

    // Method 1
    // m1() method of class C (This class)
    public void m1()
    {

        // Super keyword called over m1() method
        // for interface 2/B
        B.super.m1();
    }
```

Ques) What is Marker Interface ?

- ➔ A marker interface is an empty interface without any fields or methods
- ➔ It is used to signal to the JVM or frameworks that an object of this class will have some special behavior.
- ➔ It provides **run-time type information about objects** .

Custom Marker Interface

```
public interface Deletable {
}
```

```
public class Entity implements Deletable {
    // implementation details
}
```

```
public class ShapeDao {

    // other dao methods

    public boolean delete(Object object) {
        if (!(object instanceof Deletable)) {
            return false;
        }

        // delete implementation details

        return true;
    }
}
```

Let's say that we have a DAO object with a method for removing entities from the database. We can write our *delete()* method so that **only objects implementing our marker interface** can be deleted:

In the previous example, we could get the same results by modifying our DAO's *delete()* method to test whether our object is a *Shape* or not, instead of testing whether it's a *Deletable*:

```
public class ShapeDao {

    // other dao methods

    public boolean delete(Object object) {
        if (!(object instanceof Shape)) {
            return false;
        }

        // delete implementation details

        return true;
    }
}
```

So why create a marker interface when we can achieve the same results using a typical interface?

Let's imagine that, in addition to the *Shape* type, we want to remove the *Person* type from the database as well. In this case, there are two options to achieve that.

But what if we have more types that we want to remove from the database as well? Obviously, this won't be a good option because we have **to change our method for every new type**.

The second option is **to make the *Person* type implement the *Shape* interface**, which acts as a marker interface. But is a *Person* object really a *Shape*? The answer is clearly no, and that makes the second option worse than the first one.

Consequently, although **we can achieve the same results by using a typical interface as a marker, we'll end up with a poor design.**

Marker Interface using annotation

Let's make Annotations to replace Serializable and see how it works —

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface SerializableClass {

  /* If you are considering doing this, then you actually have to
    write the complete logic for Serialization */
}
```

```
//Now, here instead of "public class User implements Serializable"
 // we use annotation @SerializableClass created above

@SerializableClass
public class User {
    private String username;

    public User(String username) {
        this.username = username;
    }
}
```

Ques) Transient Keyword and Volatile keyword

Transient

➔ Serialization is the process of converting an object into a byte stream, and de-serialization is the opposite of it.
➔ When we mark any variable as transient, then that variable is not serialized.
➔ Since ***transient*** fields aren't present in the serialized form of an object, the **de-serialization process** would use the **default values.**
➔ It is useful for fields that do not represent the state of the object
➔ When we want to store sensitive information and don't want to send it through the network

Behavior With Final Keyword

```
public class Book implements Serializable {
    // existing fields

    private final transient String bookCategory = "Fiction";

    // getters and setters
}
```

➔ The final modifier makes no difference when it has literal initialization.
➔ When a variable of type String is declared as final and transient, its value is determined at compile-time and is stored in the class's constant pool.
➔ Since it is final, it's value can't be change after it's initialization. Hence, its value will be taken **from the class and not null.**

Volatile

- ➔ The volatile keyword indicates that a variable's value will be modified by different threads.
- ➔ ***volatile*** is used to make sure that the value of a variable is always **read from the main memory** and **not** from the **thread's cache**.
- ➔ Ensures that changes made in one thread are immediately visible to other threads.
- ➔ Useful in concurrent programming to make class thread-safe by ensuring the visibility of variables across threads.

Ques) If Parent is Serializable but I don't want my child to be Serialized ? What Should we do it ?

- ➔ We will declare field as transient.

```
class Parent implements Serializable {
    private static final long serialVersionUID = 1L;

    String parentField;

    public Parent(String parentField) {
        this.parentField = parentField;
    }
}

class Child extends Parent {
    private static final long serialVersionUID = 1L;

    transient String childField; // This will not be serialized

    public Child(String parentField, String childField) {
        super(parentField);
        this.childField = childField;
    }
}
```

Ques) If we have parametrized constructor , then is it required to implement default constructor ?

- ➔ No, it is not required to implement a default constructor if your class already has a parameterized constructor.
- ➔ However, there are cases where you might need to implement a default constructor explicitly.
- ➔ Depending on how the class is used, such as when working with serialization, frameworks like Hibernate, or if your code needs to be compatible with certain libraries that require a no-argument constructor.

Here's a breakdown:

1. **Parameterized Constructor:** If you define a constructor with parameters, you can create instances of the class by passing arguments to this constructor. In this case, a default (no-argument) constructor is not required.

2. **No-Argument Constructor (Default Constructor):** If you don't define any constructor in your class, Java automatically provides a default constructor (a no-argument constructor). However, if you define any constructor (parameterized or otherwise), Java will **no longer automatically provide** the default constructor.

   If your class requires a no-argument constructor, you must explicitly define one:

3. **When you might need the default constructor:**
    - **Serialization:** Some frameworks or libraries (like Java's default serialization mechanism or frameworks like Hibernate) may require a no-argument constructor to instantiate objects via reflection. In such cases, if you only have a parameterized constructor, you will need to explicitly provide a no-argument constructor.
    - **Reflection-based libraries:** Some libraries that use reflection to instantiate objects may require a no-argument constructor. For example, many frameworks (like Spring or JAXB) use reflection and might need a default constructor to create objects.

4. **When is it not needed?**
    - If you don't use serialization frameworks or reflection-based libraries that need a default constructor, and your use cases are fine with parameterized constructors, then there is no need to implement a default constructor.

Ques)What is aggregation ,composition, association. ? Which one is stronger .

Aggregation

➔ A specific type of association that signifies **“has-A”** relationship.

➔The **contained object** can **exist independently** of the **container object**.

Composition

➔ A stronger form of aggregation, where one object (the whole) contains another object (the part), and the part cannot exist without the whole.

➔“Has-A” relationship and have strong coupling.

Ques) Create Immutable class without breaking.

➔ An immutable class in Java is a class whose instances cannot be modified after they are created.

➔ Immutable objects are inherently thread-safe and can be shared freely between threads without synchronization.

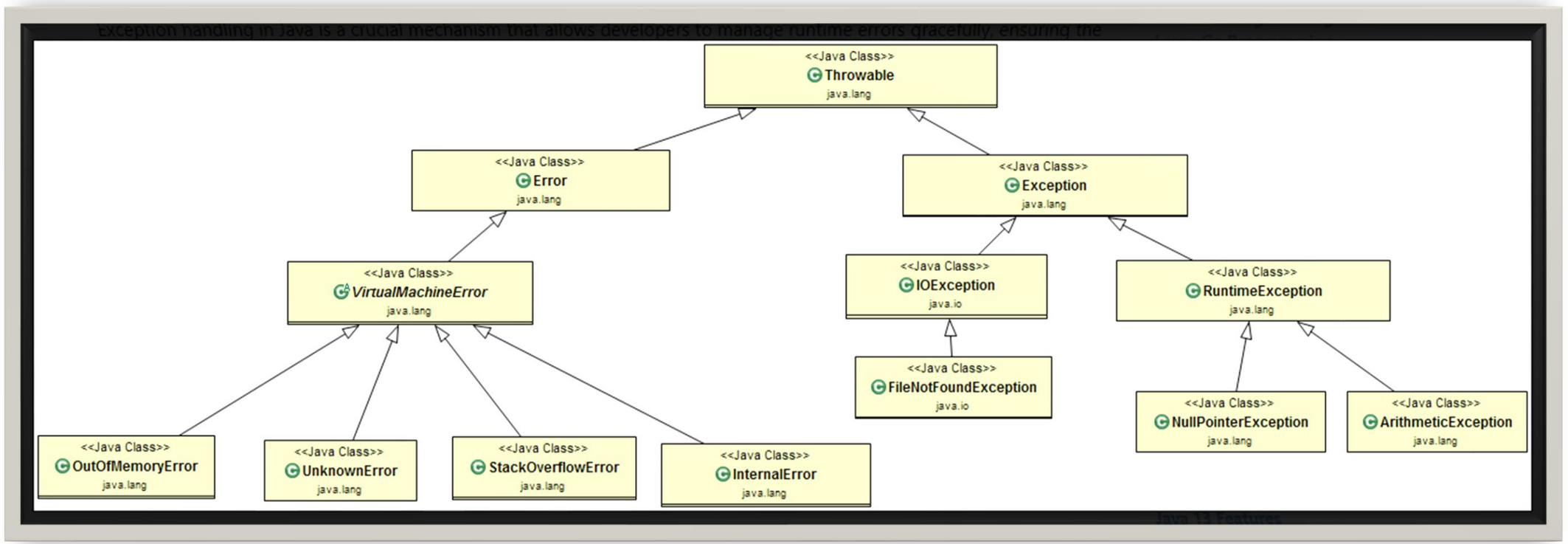
Key Points:

- **Immutability**: Once an object is created, its state cannot be changed.
- **Thread-Safety**: Immutable objects are inherently thread-safe.
- **Design Principles**: Follow specific rules to ensure immutability.

Steps to Create an Immutable Class

1. Declare the class as final.
2. Make all fields private and final.
3. Initialize all fields in the constructor.
4. Do not provide setter methods.
5. Perform defensive copying of mutable fields in the constructor and getter methods.

Ques) Custom Exception Handling



Checked Exception

➔ Checked exceptions are exceptions that are checked at compile-time.
➔ These exceptions must be either caught or declared in the method signature using the **throws** keyword.

Example :

IOException , SQLException , FileNotFoundException

Unchecked Exceptions

➔ These exception not checked at compile time.
➔ Sub Classes of Runtime Exception.
➔ It represents programming error, such as logic mistakes

Example :

- **NullPointerException**
- **ArrayIndexOutOfBoundsException**
- **ArithmeticException**

4. Exception Hierarchy

The exception hierarchy in Java is as follows:

```
java.lang.Object
   └─ java.lang.Throwable
       ├─ java.lang.Exception
       │   ├─ java.io.IOException
       │   ├─ java.sql.SQLException
       │   └─ java.lang.RuntimeException
       │       ├─ java.lang.NullPointerException
       │       ├─ java.lang.ArrayIndexOutOfBoundsException
       │       └─ java.lang.ArithmeticException
       └─ java.lang.Error
           ├─ java.lang.OutOfMemoryError
           ├─ java.lang.StackOverflowError
           └─ java.lang.VirtualMachineError
```

Custom Exceptions

➔ You can create your own custom exceptions by extending the Exception class or any of its subclasses.
➔ Custom exceptions are useful for specific error conditions that are relevant to your application.

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

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }

    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
        System.out.println("Age is valid.");
    }
}
```

Ques) Write Singelton Design Pattern

```
16
17  private static volatile SingeltonDesignPattern instance = null;
18  private SingeltonDesignPattern() {}
19
20  /**
21   *
22   *  Singelton With Thread safety
23   */
24  public static  SingeltonDesignPattern getInstance() {
25     if (instance == null) {
26        synchronized (SingeltonDesignPattern.class) {
27           if (instance == null) {
28              instance = new SingeltonDesignPattern();
29           }
30        }
31     }
32     return instance;
33  }
34
35  public static void main(String[] args)
36  {
37     SingeltonDesignPattern instance = SingeltonDesignPattern.getInstance();
38     System.out.println(instance);
39     instance = SingeltonDesignPattern.getInstance();
```

Ques) Java 17 New Features

➔Sealed Classes and Interfaces moved to standard.
➔Add **java.time.InstantSource**
➔Context-Specific Deserialization Filters
➔ Enhanced Pseudo-Random Number Generators
➔**Preview and Incubator Features** : Pattern matching for switch, introduced as a preview feature in Java 17, allows switch statements to match patterns, providing more concise and readable code.

```
public class PatternMatchingSwitchExample {
    public static void main(String[] args) {
        Object obj = "Java 17";
        String result = switch (obj) {
            case Integer i -> "Integer: " + i;
            case String s -> "String: " + s;
            default -> "Unknown type";
        };
        System.out.println(result);
    }
}
```

Ques) Which locking mechanism we use in Singelton class and why ?

➔ Double Check Locking mechanism

Ques) Why enum is required ?

➔ Enum is a special keyword to denote group of constants.
➔ Enum brings clarity and type safety to our code.
➔ Enums, short for “enumerations,” are a **special type of class** in Java that allow you to **define a set of named constants .**
➔Enums define a set of constants for a specific type and are typically used to represent fixed sets of data, like:

Days of the week
- **Directions** (NORTH, SOUTH, EAST, WEST)
- **Seasons** (WINTER, SPRING, SUMMER, FALL)
- **Status codes** (SUCCESS, FAILURE)

➔Using enums makes your code more readable, maintainable, and less prone to errors, compared to using static final constants.

Basic Enum Example:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

```
public class EnumTest {
    public static void main(String[] args) {
        Day today = Day.MONDAY;

        if (today == Day.MONDAY) {
            System.out.println("Start of the work week!");
        }
    }
}
```

➔Enums are **type-safe**. You **cannot assign** any value **other than the defined enum constants** to a **variable of an enum type**.
➔**Enums** in Java can **contain fields, methods, and constructors**.

Enum With method

```
public enum Season {
    WINTER("Cold"),
    SPRING("Warm"),
    SUMMER("Hot"),
    FALL("Cool");

    private String description;

    // Constructor
    Season(String description) {
        this.description = description;
    }

    // Getter method
    public String getDescription() {
        return description;
    }
}
```

Here, each enum constant has an associated `description`, which is passed to the constructor. This allows for more complex enums.

```
public class EnumTest {
    public static void main(String[] args) {
        for (Season season : Season.values()) {
            System.out.println(season + " is " + season.getDescription());
        }
    }
}
```

Enum With Fields

Enums can have fields to store additional information for each constant. These fields can be initialized using constructors, as shown in the previous example.

```
public enum StatusCode {
    SUCCESS(200),
    BAD_REQUEST(400),
    NOT_FOUND(404),
    INTERNAL_SERVER_ERROR(500);

    private int code;

    StatusCode(int code) {
        this.code = code;
    }

    public int getCode() {
        return code;
    }
}
```

```
public class EnumTest {
    public static void main(String[] args) {
        StatusCode status = StatusCode.NOT_FOUND;

        System.out.println("Status Code: " + status.getCode());
    }
}
```

Enum With Abstract Method

Enums in Java can have abstract methods, which must be implemented by each enum constant. This allows you to define unique behavior for each constant.

```
public enum Operation {
    ADD {
        @Override
        public int apply(int x, int y) {
            return x + y;
        }
    },
    SUBTRACT {
        @Override
        public int apply(int x, int y) {
            return x - y;
        }
    },
    MULTIPLY {
        @Override
        public int apply(int x, int y) {
            return x * y;
        }
    },
    DIVIDE {
        @Override
        public int apply(int x, int y) {
            return x / y;
        }
    };

    // Abstract method
    public abstract int apply(int x, int y);
}
```

```
public class EnumTest {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;

        for (Operation op : Operation.values()) {
            System.out.println(a + " " + op + " " + b + " = " + op.apply(a, b))
        }
    }
}
```

Advanced Enum Features

➔ You can **override** the **`toString()`** method in an enum.
➔ **Enums** can **implement interfaces** to ensure that all constants adhere to certain behaviors.

```
interface Level {
    String getLevelInfo();
}

public enum GameLevel implements Level {
    EASY {
        @Override
        public String getLevelInfo() {
            return "Easy Level";
        }
    },
    MEDIUM {
        @Override
        public String getLevelInfo() {
            return "Medium Level";
        }
    },
    HARD {
        @Override
        public String getLevelInfo() {
            return "Hard Level";
        }
    }
}
```

Few useful methods in enum.

values() : Returns an array of all enum constants.
valuesOf(String name) : Returns the enum constant with the specified name.
ordinal() : Returns the position of the enum constant in its enum declaration.

```java
public enum Day {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

```java
public class EnumExample {
    public static void main(String[] args) {
        Day[] days = Day.values();

        System.out.println("All days of the week:");
        for (Day day : days) {
            System.out.println(day);
        }
    }
}
```

```java
public class EnumExample {
    public static void main(String[] args) {
        Day day = Day.valueOf("WEDNESDAY");

        System.out.println("The enum constant for 'WEDNESDAY' is: " + day);
    }
}
```

```java
public class EnumExample {
    public static void main(String[] args) {
        Day day = Day.FRIDAY;

        System.out.println("The ordinal of 'FRIDAY' is: " + day.ordinal());
    }
}
```

Output:

```
The ordinal of 'FRIDAY' is: 5
```

Ques) Can we extend enum ?

➔Directly It **cannot be extend** . Java enum is **implicitly final**.

Ques) Can we implement enum ?

➔Yes.

Ques) Can we overload an enum ?

➔No

Ques) Can we put enum as abstract ?

➔No

Ques) What is memory leak ?

➔A **memory leak** in Java refers to a situation where an application continuously consumes memory but fails to release it when it is no longer needed.

Common Cause

1. **Unclosed resources** : For example, not closing database connections, file streams, or network sockets.
2. **Static Reference** : Storing objects in static fields, which can prevent garbage collection because static fields live as long as the class itself.
3. **Collection objects**: Storing objects in collections (like List, Map, etc.) without removing them when they're no longer necessary.
4. **Circular references**: This happens when two objects reference each other and the reference chain is not broken, preventing garbage collection.

How To Solve this problem

1) Use Weak Reference for Caching and Listeners.
2) Close the resource properly.
3) Avoid storing objects in static variable.
4) Proper use of collection.
5) Avoid circular reference.

Ques) How many ways you can create a thread ?

➔ **2 ways : 1)** By extending the Thread class . **2)** By implementing the runnable interface.

Ques) Can we call run method directly to start a thread?

➔ No

Ques) What is the difference between Thead Start and Thread run method ?

➔ **Thread start** : It will start the new thread to our Thread pool.

➔ **Thread run** : will execute on the same thread and it will not create any new thread.

Ques) What is executor Service ? Explain concept and use .

➔ Java's ExecutorService is a powerful framework for managing and executing concurrent tasks in Java applications.
➔ ExecutorService is a subinterface of Executor that provides methods to manage the lifecycle of threads and to perform asynchronous task execution.

- `submit()`: Submits a task for execution and returns a `Future` representing the task.
- `invokeAll()`: Executes a collection of tasks and returns a list of `Future` objects.
- `invokeAny()`: Executes a collection of tasks and returns the result of one of the tasks.
- `shutdown()`: Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
- `shutdownNow()`: Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

2. Creating an ExecutorService

You can create an `ExecutorService` using the `Executors` factory methods, such as:

- `newFixedThreadPool(int nThreads)`: Creates a thread pool with a fixed number of threads.
- `newCachedThreadPool()`: Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
- `newSingleThreadExecutor()`: Creates an executor that uses a single worker thread.

```
rt java.util.concurrent.ExecutorService;
rt java.util.concurrent.Executors;

ic class ExecutorServiceExample {
public static void main(String[] args) {
    // Creating a fixed thread pool with 2 threads
    ExecutorService executorService = Executors.newFixedThreadPool(2);

    // Submit tasks to the executor service
    executorService.submit(() -> System.out.println("Task 1 executed by " + Thread.currentThread().getName()));
    executorService.submit(() -> System.out.println("Task 2 executed by " + Thread.currentThread().getName()));
    executorService.submit(() -> System.out.println("Task 3 executed by " + Thread.currentThread().getName()));

    // Shutdown the executor service
    executorService.shutdown();
}
```

```
public class SubmitTasksExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        // Submitting a Runnable task
        Future<?> future1 = executorService.submit(() -> {
            System.out.println("Runnable task executed by " + Thread.currentThread().getName());
        });

        // Submitting a Callable task
        Future<String> future2 = executorService.submit(() -> {
            System.out.println("Callable task executed by " + Thread.currentThread().getName());
            return "Result from Callable task";
        });

        try {
            // Getting the result of the Callable task
            String result = future2.get();
            System.out.println(result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        executorService.shutdown();
    }
}
```

Explanation:

- A fixed thread pool with 2 threads is created.
- Three tasks are submitted to the executor service.
- The tasks are executed by the threads in the pool.
- The executor service is shut down.

```
class Task implements Runnable {
    private final String name;

    public Task(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("Task " + name + " is being executed by " + Thread.currentThread().getName());
        try {
            Thread.sleep(2000); // Simulate long-running task
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class ExecutorServiceDemo {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 5; i++) {
            Task task = new Task("Task " + i);
            executorService.submit(task);
        }

        executorService.shutdown();

        try {
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }

        System.out.println("All tasks are finished.");
    }
}
```

Ques) What is Future Object ? What are the flaws and how to overcome it.

➔The Future interface in Java is part of **the java.util.concurrent** package.
➔ It represents the **result** of an **asynchronous computation**.
➔It provides methods to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation
➔The Future interface is commonly used with the **ExecutorService** to manage **asynchronous tasks**.
➔When you submit a task to an ExecutorService, it returns a Future object that you can use to interact with the task.

2. Key Methods of the Future Interface

The `Future` interface provides several key methods:

- `boolean cancel(boolean mayInterruptIfRunning)`: Attempts to cancel the execution of the task.
- `boolean isCancelled()`: Returns `true` if the task was cancelled before it completed normally.
- `boolean isDone()`: Returns `true` if the task completed.
- `V get() throws InterruptedException, ExecutionException`: Waits if necessary for the task to complete and then retrieves its result.
- `V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException`: Waits if necessary for at most the given time for the task to complete and then retrieves its result.

3. Example: Using Future with ExecutorService

Let's create an example to demonstrate how to use the `Future` interface with `ExecutorService`.

Example:

```
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;

class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        // Simulate long-running task
        Thread.sleep(2000);
        return 123;
    }
}
```

```
public class FutureExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(new MyCallable());

        System.out.println("Task submitted.");

        try {
            // Wait for the result
            Integer result = future.get();
            System.out.println("Task completed with result: " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Output:

```
Task submitted.
Task completed with result: 123
```

Limitation Of Future Object

1) No built in way to handle completion callbacks.
2) Blocking the operations to get the result,
3) Difficult to compose the multiple asynchronous computation.

```
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) throws ExecutionException, Interrupte
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(() -> {
            // Simulate long running task
            Thread.sleep(1000);
            return 42;
        });

        // Do other things while the task is running

        // Blocking call to get the result
        Integer result = future.get();
        System.out.println("Result: " + result);

        executor.shutdown();
    }
}
```

future.get() will block all the operation.

➔Suitable for simple asynchronous operations where we need to perform tasks in a separate thread and retrieve the results later.
➔It doesn't have a mechanism to create stages of processing that are chained together.
➔There is no mechanism to run Futures in parallel and after to combine their results together
➔The Future does not have any exception handling constructs.

How To solve this Problem

CompletableFuture:

➔ Ideal for complex asynchronous programming needs, including chaining multiple asynchronous operations, error handling, combining results, and implementing non-blocking algorithms.
➔CompletableFuture is part of the **java.util.concurrent** package.
➔It also allows you to combine multiple tasks that can run concurrently and then perform operations once all or part of the tasks are completed.
➔Unlike a regular Future, which can only represent a value that is computed in the future, CompletableFuture provides methods to complete the future explicitly and chain multiple asynchronous operations

Key Features of CompletableFuture:

1. **Asynchronous Execution:** You can run code asynchronously using methods like thenApply, thenAccept, etc.
2. **Combining Multiple Futures:** You can combine multiple futures and perform actions based on their completion using methods like thenCombine, allOf, anyOf, etc.
3. **Manual Completion:** You can manually complete a CompletableFuture using methods like complete() or completeExceptionally().

Syntax to Create a Thread Using `CompletableFuture`:

1. **Basic Creation of `CompletableFuture`:**

```java
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
    System.out.println("Task is running in a separate thread.");
});
```

- `runAsync()` is used when the task does not return a result.
- By default, `runAsync()` uses the **common ForkJoinPool** to execute tasks asynchronously.

2. **Creating a `CompletableFuture` that Returns a Result:**

```java
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    return 42;  // Example of returning a result.
});
```

- `supplyAsync()` is used when the task returns a value (`Integer` in this case).

Ques) Can we overload run() method in Thread?

➔ **Yes** . But thread management will when execute thread.start() , it will call run() method only.

Ques) what is equals and hashCode contract ?

➔ The equals() and hashCode() contract in Java dictates that if two objects are equal according to equals(), they must have the same hashCode(),

➔ But the converse isn't necessarily true (objects with the same hashCode() might not be equal).

Hash-Based Collections:

➔ This contract is essential for the proper functioning of hash-based collections like HashSet and HashMap.

Ques) Internal Implementation of HashMap ?

➔ HashMap stores data in key-value pairs.

➔ HashMap allows **one null key** and **multiple null values** in a collection.

➔ HashMap is not synchronized, which means it is not thread-safe.

➔ To make it thread safe , we have to externally provide thread safety

```
Map yourMap = new HashMap();
Map synchronizedMap = java.util.Collections.synchronizedMap(yourMap);
```

Or

We can create object of **ConcurrentHashMap**

➔ The capacity is the number of buckets in the hash table,

➔ Load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

➔The iterator of HashMap is fail-fast, meaning any **structural modification (insertion or removal)** after the creation of the iterator, will throw **ConcurrentModificationException**.

Methods

get() – get the element

put() – to store the element

remove() -remove the element

Iterate or Loop Through a HashMap

There are different ways we can loop through a HashMap:

1. Using `entrySet` and a for-each loop
2. Using `keySet` and a for-each loop
3. Using `values` and a for-each loop
4. Using an `Iterator`
5. Using Java 8's `forEach` method

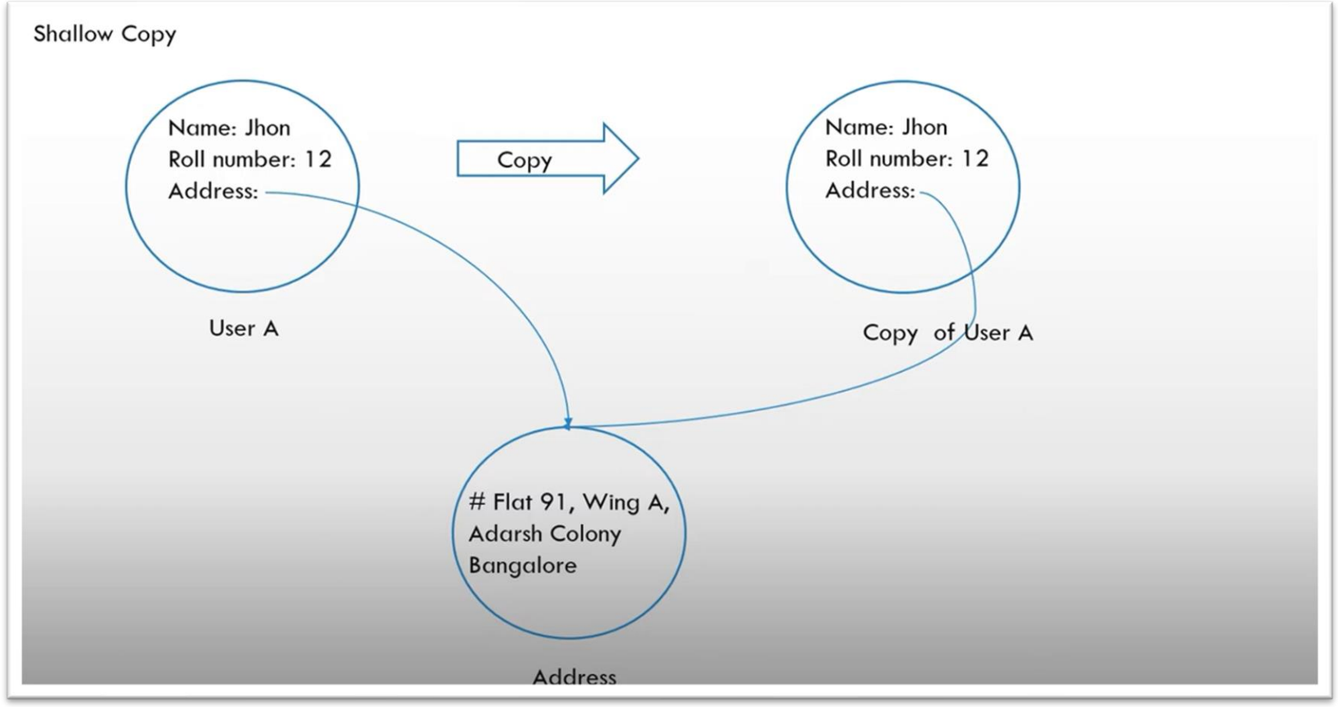
```java
public class Main {
    public static void main(String[] args) {
        // Creating a HashMap
        HashMap<String, String> map = new HashMap<>();
        map.put("Apple", "Red");
        map.put("Orange", "Orange");
        map.put("Banana", "Yellow");

        // Using `entrySet` and a `for-each` loop:
        System.out.println("Using `entrySet` and a `for-each` loop:");
        for (Map.Entry<String, String> entry : map.entrySet()) {
            System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
        }
        System.out.println();

        // Using `keySet` and a `for-each` loop:
        System.out.println("Using `keySet` and a `for-each` loop:");
        for (String key : map.keySet()) {
            System.out.println("Key = " + key + ", Value = " + map.get(key));
        }
        System.out.println();
```

Java HashMap with User-Defined Objects

If you are using user-defined objects as keys in your HashMap, you should make sure that these objects implement the `equals()` and `hashCode()` methods appropriately. If they do not, then your HashMap may not function as expected because it relies on these methods to store and retrieve objects.

Ques) What is shallow copy and deep copy ?

➔Shallow copy involves creating a new object and copying the contents of the existing object to the new one.
➔However, the references within the new object still point to the same objects as the original.
➔In other words, it copies the values of the primitive data types and the references to complex types, but not the actual objects.



➔ For user A, Address is a reference type object .
➔ When we are copying this User A to Copy of User A , we are expecting member variable which are present in User A should be copied to
Copy of User A .
➔ But In the case of shallow copy it does not happen . Member variable will be copied but the Address object will be commonly referenced for
Both the objects.
➔ It means any changes to Address object will be reflect back for both the user.
➔ In order to make copy of each and every variable present in **user A** to **copy of User A** , It **means address Object will be a separate copy** for
Copy of user A , we will have to perform **deep copy**.

Shallow copy is often achieved through the copy constructor or by manually replicating the fields of an object. Let's explore the copy constructor approach:

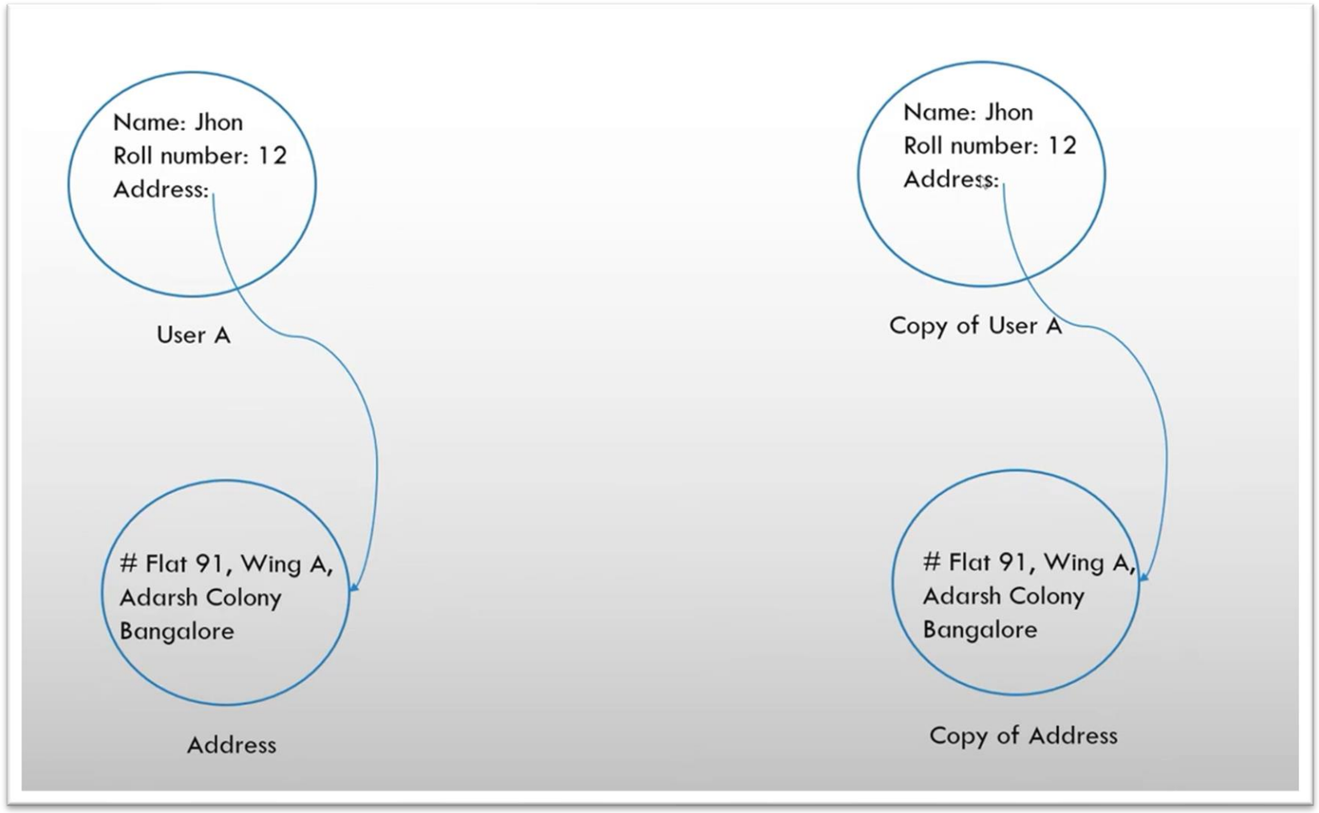
```
    public
        retu
    }

}
```

```
<terminated> ShallowCopyExample [Java Application] C:\Projects\sts-4.19.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.
shallowCopy Person : Person [name=John, address=Address [street=123 Main St]]
Shallow copy after modification : Person [name=Jane, address=Address [street=456 Side St]]
Original Person: Person [name=John, address=Address [street=456 Side St]]
Shallow Copy Person: Person [name=Jane, address=Address [street=456 Side St]]
```

```
public class ShallowCopyExample {
    public static void main(String[] args) {
        // Original object
        Person originalPerson = new Person("John", new Address("123 Main St"));

        // Shallow copy using copy constructor
        Person shallowCopyPerson = new Person(originalPerson);
        System.out.println("shallowCopy Person : "+shallowCopyPerson);
        // Modifying the shallow copy
        shallowCopyPerson.setName("Jane");
        shallowCopyPerson.getAddress().setStreet("456 Side St");
        System.out.println("Shallow copy after modification : "+shallowCopyPerson);
        // Original object remains unchanged
        System.out.println("Original Person: " + originalPerson);
        System.out.println("Shallow Copy Person: " + shallowCopyPerson);
    }
}
```

➔As per the screenshot , change in Address reference for shallowCopy object also impact Original person address object.

Deep Copy



➔It will always create separate copy of individual object in the heap memory.
➔Its advantage is that **each mutable object in the object graph is recursively copied**.
➔Since the copy isn't dependent on any mutable object that was created earlier, it won't get modified by accident like we saw with the shallow copy.
➔A **deep copy** of an object is a copy whose properties do not share the same references .

In order to achieve deep copy , in the constructor , for reference member variable assign the data with new keyword.

```java
/**
 * Deep Copy Constructor
 */

public Person1(Person1 originalPerson) {
   this.name = originalPerson.name;
   this.address = new Address1(originalPerson.address.getStreet());
}
```

```
<terminated> DeepCopyExample [Java Application] C:\Projects\sts-4.19.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v202
deepCopy Person : Person [name=John, address=Address [street=123 Main St]]
Deep copy after modification : Person [name=Jane, address=Address [street=456 Side St]]
Original Person: Person [name=John, address=Address [street=123 Main St]]
Deep Copy Person: Person [name=Jane, address=Address [street=456 Side St]]
```

```java
   @Override
   public String toString() {
      return "Address [street=" + street + "]";
   }

}

public class DeepCopyExample {
   public static void main(String args[]) {
      // Original object
      Person1 originalPerson = new Person1("John", new Address1("123 Main St"));

      // Deep copy using copy constructor
      Person1 deepCopyPerson = new Person1(originalPerson);
      System.out.println("deepCopy Person : " + deepCopyPerson);
      // Modifying the shallow copy
      deepCopyPerson.setName("Jane");
      deepCopyPerson.getAddress().setStreet("456 Side St");
      System.out.println("Deep copy after modification : " + deepCopyPerson);
      // Original object remains unchanged
      System.out.println("Original Person: " + originalPerson);
      System.out.println("Deep Copy Person: " + deepCopyPerson);
   }
```

Ques) What will have to take care when an Object will be added as a key to the hashMap object ?

```
/**
*  ************************************** When Object is used as HashMap Key ****************************************
*
*    **) hashCode : We override the hashCode() method using Objects.hash(name, age),
*                   which combines the name and age fields into a hash code.
*                   This ensures that Person objects with the same name and age will have the same hash code.

     **) equals    :   We override the equals() method to compare name and age fields of Person objects.
                       If two Person objects have the same name and age, they are considered equal.

            Using Person as a Key in the Map:

            **)  When we add person1 to the Map,
                 the hashCode() of person1 is computed and used to determine its position in the internal hash table.

            **)  When we add person3 (which has the same name and age as person1),
                 the equals() method is used to check if person1 and person3 are equal.
                 Since they are equal, the existing entry for person1 is replaced with person3 in the Map.

            **) The map contains two entries: one for Bob and one for Alice.
                Even though person1 and person3 are different objects in memory,
                they are considered equal because their name and age fields are the same.

            **) When person3 is added to the map,
                It replaces the previous entry for person1 because they are considered equal according to the equals() and hashCode() methods.

            **)  If you don't override hashCode() and equals(),
                 the HashMap will use the default hashCode() and equals() implementations from the Object class.
```

Using Person as a Key in the Map:

**) When we add person1 to the Map,
the hashCode() of person1 is computed and used to determine its position in the internal hash table.

**) When we add person3 (which has the same name and age as person1),
the equals() method is used to check if person1 and person3 are equal.
Since they are equal, the existing entry for person1 is replaced with person3 in the Map.

**) The map contains two entries: one for Bob and one for Alice.
Even though person1 and person3 are different objects in memory,
they are considered equal because their name and age fields are the same.

**) When person3 is added to the map,
It replaces the previous entry for person1 because they are considered equal according to the equals() and hashCode() methods.

**) If you don't override hashCode() and equals(),
the HashMap will use the default hashCode() and equals() implementations from the Object class.

**) The hashCode() method will return a hash code based on the memory address,
which is likely different for different instances of the same Person even if they have the same name and age.

**) The equals() method will compare object references, not their content,
which means two Person objects with the same name and age would not be considered equal.

**) When using custom objects as keys in a Map (like HashMap),
you must override both the hashCode() and equals() methods to ensure correct behavior.

```java
    // Override hashCode and equals to ensure proper functioning in a Map
    @Override
    public int hashCode() {
        return Objects.hash(name, age);  // Generate a hash based on name and age
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Check if the objects are the same
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name, person.name);  // Check equality of fields
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}
```

Problems Javadoc Declaration Console × SonarLint On-The-Fly

<terminated> Object_As_Key_In_HashMap [Java Application] C:\Projects\sts-4.19.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin

```
{Person{name='Alice', age=30}=Artist, Person{name='Bob', age=25}=Doctor}
person1 equals person3: true
```

```
}
public class Object_As_Key_In_HashMap {
    public static void main(String[] args) {
        // Create a map to store Person objects as keys
        Map<Person, String> personMap = new HashMap<>();

        // Create Person objects as keys
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);
        Person person3 = new Person("Alice", 30); // Same name and age as person1

        // Add Person objects to the map
        personMap.put(person1, "Engineer");
        personMap.put(person2, "Doctor");

        // Use the same Person object (person1 and person3 are equal)
        personMap.put(person3, "Artist"); // This should replace the entry for person1 because they are considered equal

        // Output the map
        System.out.println(personMap);
```

Ques) What If hashMap have duplicate key ? Is it will show error or what it will do

➔ If you try to put a value with a key that already exists in the HashMap, **it will not throw an error**. Instead, it will **overwrite the existing value** associated with that key.

Ques) Feature of Java8 ?

➔ Lambda Expression
➔Functional Interface
➔Stream API
➔Default Method & static method
➔Method Reference
➔New Date and Time ApI
➔Optional class
➔Javascript Engine

Ques) Can we extend One Functional Interface to another Functional Interface ?

➔Yes.

Ques) How to write Predicate functional Interface ?

```
Predicate<Integer> pr = (n) -> n % 2 == 0;
return pr.test(20);
```

Ques) How to write Function Functional Interface ?

```
16
```

```
public class FunctionJava8 {

    public static void main(String[] args) {
        Function<Integer,Integer> squareOfNumber = (number) -> number * number;
        int number =squareOfNumber.apply(4);
        System.out.println(number);
```

Ques) How to Write Supplier Functional Interface ?

```
Wed Mar 19 10:33:09 IST 2025
```

```
public class SupplierExample {

    public static void main(String[] args) {
        Supplier<Date> getTheCurrentDate = () -> new Date();
         System.out.println(getTheCurrentDate.get());
```

Ques) How to write Consumer Functional Interface ?

```
Karan


public class ConsumerExample {

   public static void main(String[] args)
   {

Consumer<String> consumer = (string) -> System.out.println(string);
consumer.accept("Karan");
   }
```

Java 8 Practice Interview Questions

1) Upper Character of String

```
upperCaseString : HELLO WORLD
import java.
import java.
import java.

blic class

public static void main(String[] args)
{
 String str = "Hello World";
 String upperCaseString=Arrays.stream(str.split("")).map(String::toUpperCase).collect(Collectors.joining());
System.out.println("upperCaseString : "+upperCaseString);
}
```

2) Reverse the String

```
String string = "Karan Raj Sinha";
  String reversedString = Stream.of(string).map(word -> new StringBuilder(word).reverse())
        .collect(Collectors.joining(""));
System.out.println(reversedString);
```

3) Find First Non-Repeating Character

```
public static Character way1(String string) {
    LinkedHashMap<Character, Integer> linkedhashMap = new LinkedHashMap<>();
    string.chars().mapToObj(c -> (char) c).forEach(word -> {
        if (linkedhashMap.containsKey(word)) {
            linkedhashMap.put(word, linkedhashMap.get(word) + 1);
        } else {
            linkedhashMap.put(word, 1);
        }
    });
    Entry<Character, Integer> entryObject = linkedhashMap.entrySet().stream().filter(entry -> entry.getValue() == 1)
            .findFirst().get();
    return entryObject.getKey();
}

private static Character way2(String string) {
    Map<Character, Long> nonRepeatingCharacter = string.chars().mapToObj(c -> (char) c)
            .collect(Collectors.groupingBy(Function.identity(), LinkedHashMap::new, Collectors.counting()));

    return nonRepeatingCharacter.entrySet().stream().filter(entry -> entry.getValue() == 1).map(Map.Entry::getKey)
            .findFirst().get();
}
```

4) Pallindrome check for String

```
package java8.string;

import java.util.stream.IntStream;

public class Pallindrom_String_Check {

  public static void main(String[] args) {
    String string = "KaraK";
    String tempString = string.replaceAll("\\s", "").toLowerCase(); //removing spaces and converting to lowercase
    System.out.println(tempString);
    int tempStringlength = tempString.length();
    boolean isPallindrome = IntStream.range(0, tempStringlength / 2)
          .allMatch(i -> tempString.charAt(i) == tempString.charAt(tempStringlength - (i + 1)));
    System.out.println(isPallindrome);
  }
```

```
karak
true
```

5) Find Duplicate characters in a string

```
HashSet<String> hashset = new HashSet<>();

Set<String> duplicateCharacterSet = Arrays.stream(string.replace(" ", "").split(""))
      .filter(word -> !hashset.add(word)).collect(Collectors.toSet());
```

6) Find Duplicate word in a string

```
String string = "This is a test. This test is easy.";
String[] stringarr = string.split("\\W");
HashSet<String> uniqueWord = new HashSet<>();
Set<String> duplicateWords = Arrays.stream(stringarr).filter(word -> !uniqueWord.add(word))
        .collect(Collectors.toSet());
System.out.println(duplicateWords);
```

7) Find all permutation of String

```
public static List<String> getPermutationsUsingJava8(String str) {
    if (str.isEmpty()) {
        return List.of("");
    }
    return IntStream.range(0, str.length()).boxed()
            .flatMap(i -> getPermutationsUsingJava8(str.substring(0, i) + str.substring(i + 1)).stream()
                    .map(s -> str.charAt(i) + s))
            .collect(Collectors.toList());
}
```

At the top level you have to call it with an empty prefix, because of course, you haven't made any first character choices at this point. Let's see what happens if we call permutation("","abc"):

- Level 1 picks a, calls permutation("a","bc")
  - Level 2 picks b, calls permutation("ab","c")
    - Level 3 picks c, calls permutation("abc", "")
      - Level 4 has an empty string, so it prints **abc**
    - Level 3 has nothing more to choose.
  - Level 2 picks c, calls permutation("ac", "b")
    - Level 3 picks b, calls permutation("acb", "")
      - Level 4 has an empty string, so it prints **acb**
    - Level 3 has nothing more to choose
  - Level 2 has nothing more to choose
- Level 1 picks b, calls permutation("b","ac")
  - Level 2 picks a, calls permutation("ba","c")
    - Level 3 picks c, calls permutation("bac", "")
      - Level 4 has an empty string, so it prints **bac**
    - Level 3 has nothing more to choose
  - Level 2 picks c, calls permutation("bc","a")
    - Level 3 picks a, calls permutation("bca", "")
      - Level 4 has an empty string, so it prints **bca**
    - Level 3 has nothing more to choose
  - Level 2 has nothing more to choose
- Level 1 picks c, calls permutation("c","ab")
  - Level 2 picks a, calls permutation("ca","b")
    - Level 3 picks b, calls permutation("cab", "")
      - Level 4 has an empty string, so it prints **cab**
    - Level 3 has nothing more to choose
  - Level 2 picks b, calls permutation("cb","a")
    - Level 3 picks a, calls permutation("cba", "")
      - Level 4 has an empty string, so it prints **cba**

Ques) Java Program to find occurrence of each word in a string.

```
String string = "This is a test.This test is easy";
Map<String, Long> elementAndItsCount = Arrays.stream(string.split("\\W"))
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
System.out.println("elementAndItsCount : "+elementAndItsCount);
```

Ques) Java Program to find occurrence of each character from a string.

```
Map<String, Long> characterAndItsCount = Arrays.stream(string.replace(" ", "").split(""))
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
System.out.println("elementAndItsCount : "+characterAndItsCount);
```

Ques) Count all the vowels and consonant from a String

```
String string = "Karan Raj Sinha";
String vowelMatcher = "[aeiouAEIOU]";
long vowelCount = string.replace(" ", "").chars().filter(ch -> vowelMatcher.contains(String.valueOf((char) ch)))
      .count();
long consonantCount = string.replace(" ", "").chars()
      .filter(ch -> !vowelMatcher.contains(String.valueOf((char) ch))).count();

System.out.println("Total vowels in string : " + vowelCount);
System.out.println("Total consonant in string : " + consonantCount);
```

Ques) Remove all the vowels and consonant from string.

```
String string = "Karan Raj Sinha";
String vowelRemoved = "";
String consonantRemoved = "";
String vowelMatcher = "[aeiouAEIOU]";
String consonantMatcher =  "[^aeiouAEIOU]";
vowelRemoved = string.replaceAll(vowelMatcher, "");
consonantRemoved = string.replaceAll(consonantMatcher, "");

System.out.println("Only consonant  :"+vowelRemoved);
System.out.println("Only vowels : "+consonantRemoved);
```

Ques) Count the number of duplicate words ?

```
String string = "This is a test. This test is easy.";
HashSet<String> duplicateWord = new HashSet<>();
/**
 *  Way 1
 */
long count =Arrays.stream(string.split("\\W")).filter(word->!duplicateWord.add(String.valueOf(word))).count();
System.out.println("Count of duplicate words way1: "+count);

/**
 * Way 2
 */
Map<String, Long> elementsAndItsCount = Arrays.stream(string.split("\\W"))
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
long duplicateCountOfWord = elementsAndItsCount.entrySet().stream().filter(entry -> entry.getValue() > 1)
      .count();
System.out.println("duplicateCountOfWord way 2 : " + duplicateCountOfWord);
```

Ques) Count the number of Occurrence of Substring in a string

```
String string = "This is a test.This test is easy";
String substring = "test";
long numberOfOccurenceOfString = Arrays.stream(string.split("\\W+")).filter(word -> word.equals(substring))
        .count();
stem.out.println("numberOfOccurenceOfString : "+numberOfOccurenceOfString);
```

Ques) Remove duplicate word from String

```
String string = "Java is great and Java is fun and Java is powerful";
Set<String> finalString =Arrays.stream(string.split(" ")).collect(Collectors.toCollection(LinkedHashSet::new));
System.out.println(finalString);
```

Ques) Reverse Each Word of a String

```
String input = "Java is great and fun";
String finalOutput = Arrays.stream(input.split(" ")).map(word -> new StringBuilder(word).reverse())
        .collect(Collectors.joining(" "));

System.out.println("finalOutput : "+finalOutput);
```

Ques) Check if a String Contains Only Digits in Java

```
String string1 = "1234";
boolean isOnlyDigit = string1.chars().allMatch(Character::isDigit);
System.out.println(isOnlyDigit);
```

Ques) Check If a String contains only Character

```
String string = "hello";
boolean isAlphabet =  string.trim().chars().allMatch(Character::isAlphabetic);
System.out.println(isAlphabet);

boolean isAlphabet2 =string.matches("[a-zA-z]+");
System.out.println(isAlphabet2);
```

Ques) Maximum Occurrence of a Character in a string .

```
/**
 * Find Maximum occurence of Character
 */

String maximumNumberOfRepeatedChaarcterInAString = characterAndItsCount.entrySet().stream()
        .sorted((entry1, entry2) -> entry2.getValue().compareTo(entry1.getValue())).map(Map.Entry::getKey)
        .findFirst().get();

System.out.println("maximumNumberOfRepeatedChaarcterInAString : " + maximumNumberOfRepeatedChaarcterInAString);
```

Arrays

Ques) Swap the array

```
int arr[] = {1,2,3,4,5};
 int max = arr.length;
 IntStream.range(0, arr.length/2).forEach(index->{
    int temp = arr[index];
    arr[index]=arr[max-1-index];
    arr[max-1-index] = temp;
 });

 System.out.println(Arrays.toString(arr));
```

Ques) Duplicate in Array

```
public class Find_Duplicate_In_Array {

  public static void main(String[] args)
  {
    int arr[] = { 1, 2, 3, 4, 3, 2, 25 };
    HashSet<Integer> hashSet = new HashSet<>();
    /**
     * Whenever we will add primitive data we should boxed()
     */
    List<Integer> duplicateData = Arrays.stream(arr).boxed().filter(number -> !hashSet.add(number))
        .collect(Collectors.toList());
    System.out.println(duplicateData);
```

Ques) Find Largest from Array

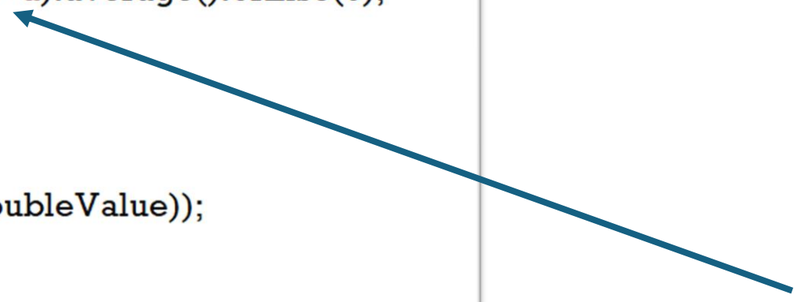
```
public class Find_Largest_Element {
    public static Integer way1(int[] arr) {
        Integer max =  Arrays.stream(arr).boxed().max(Integer::compare).orElseGet(null);
        return max;
    }
    public static Integer way2(int[] arr) {
        Integer max =  Arrays.stream(arr).boxed().sorted((I1, I2)->Integer.compare(I2, I1)).findFirst().orElse
        return max;
    }
    public static void main(String[] args)
    {
       int arr[] = { 1, 2, 3, 4, 53, 2, 25,27,51};
      Integer maxFromArrayWay1 =  way1(arr);
      System.out.println(maxFromArrayWay1);

      Integer maxFromArrayWay2 = way2(arr);
      System.out.println(maxFromArrayWay2);
    }
```

Ques) Check Both Arrays are equal or Not.

```
int arr1[] = { 1, 2, 3, 4, 3, 2, 25 };
int arr2[] = {3,2,1,4,25};

/**
 *  Equality Of Array
 */
boolean isEqual = Arrays.stream(arr1).distinct()
      .allMatch(arr1Data -> Arrays.stream(arr2).distinct().anyMatch(arr2Data -> arr2Data == arr1Data));

System.out.println("Both arrays are equal : "+isEqual);
```

Ques) Calculate Average Using Arrays

```
public static void getAverage(int[] arr) {
   double d1 = Arrays.stream(arr).average().orElse(0);
   System.out.println("Average from way1 : "+d1);
}
public static void getAverageWay2(int[] arr) {
   double d2= Arrays.stream(arr).boxed().mapToInt(a->a).average().orElse(0);
   System.out.println("Double from way2  : "+d2);
}
public static void getAverageWay3(int[] arr) {
   double d3= Arrays.stream(arr).boxed()
         .collect(Collectors.averagingDouble(Integer::doubleValue));
   System.out.println("Double from way2  : "+d3);
}
```



mapToInt(Integer::intValue) // this is also correct.

Ques) Sort according to Ascending order Array

```
public static void sortingAscendingWay1(int[] arr) {
    Object[] sortedArray = Arrays.stream(arr).boxed().sorted(Integer::compare).toArray();
    System.out.println(Arrays.toString(sortedArray));
}

public static void sortingAscendingWay2(int[] arr) {
    int[] sortedArray = Arrays.stream(arr).sorted().toArray();
    System.out.println(Arrays.toString(sortedArray));
}

public static void sortingAscendingWay3(int[] arr) {
    Arrays.sort(arr);
    System.out.println(Arrays.toString(arr));
}

public static void sortingAscendingWay4(int[] arr) {

    Arrays.stream(arr).boxed().sorted() // Sorts in ascending order
          .collect(Collectors.toList());
    System.out.println(Arrays.toString(arr));
}
```

Ques) Descending Order Sorting For Array

```
public static void sortingDescendingWay1(int[] arr) {
   Object[] sortedArrayDesc = Arrays.stream(arr).boxed().sorted(Collections.reverseOrder()).toArray();
   System.out.println(Arrays.toString(sortedArrayDesc));
}

public static void sortingDescendingWay2(int[] arr) {
   int[] sortedArrayDesc = Arrays.stream(arr).boxed().sorted(Collections.reverseOrder()).mapToInt(Integer::intValue).toArray();
   System.out.println(Arrays.toString(sortedArrayDesc));
}

public static void sortingDescendingWay3(int[] arr) {
   int[] sortedArray = Arrays.stream(arr).boxed().sorted(Collections.reverseOrder(Integer::compareTo)).mapToInt(Integer::intValue).toArray();
   System.out.println(Arrays.toString(sortedArray));
}

public static void sortingDescendingWay4(int[] arr) {
   List<Integer> arrList = Arrays.stream(arr).boxed().collect(Collectors.toList());
   Integer[] integerArray = Arrays.stream(arr).boxed().toArray(Integer[]::new);

   Arrays.sort(integerArray,Collections.reverseOrder());
   System.out.println(Arrays.toString(integerArray));
}
```

Ques) Sum Of n natural number.

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter the number till which you want sum");
int number = sc.nextInt();

int sumOfNaturalNumber = IntStream.rangeClosed(0, number).reduce((num1, num2) -> num1 + num2).orElse(0);
System.out.println("Sum Of " + number + "natural number is : " + sumOfNaturalNumber);
```

Ques) Reverse a Number

```
int number = 4529;

int reversed =Integer.parseInt(new StringBuilder(String.valueOf(number)).reverse().toString());
System.out.println(reversed);
```

Ques) Find Prime Number till given number

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number till which you want prime Number");
    int number = sc.nextInt();
    List<Integer> primeNumbers = IntStream.rangeClosed(2, 10).filter(Find_Prime_Number::isPrime).boxed()
            .collect(Collectors.toList());
    System.out.println("Prime Number till " + number + " : " + primeNumbers);
}

public static boolean isPrime(int number) {
    boolean isPrime = number > 1
            && IntStream.rangeClosed(2, (int) Math.sqrt(number)).allMatch(n -> number % n != 0);
    return isPrime;
}
```

Ques) Fibonacci Series

```
public static void Java7Way() {
        int n = 10;
       int a = 0, b = 1;
       System.out.println("Fibonacci Series:");
       for (int i = 0; i < n; i++) {
          System.out.println(a);
          int next = a + b;
          a = b;
          b = next;
       }
}
public static void Java8Way() {
    System.out.println("Java8 way");
     Stream.iterate(new int[]{0, 1}, f -> new int[]{f[1], f[0] + f[1]})
     .limit(10)
     .map(f -> f[0])
     .forEach(System.out::println);
}
```

Ques) Find the last element from the list of String.

```
public static void main(String[] args) {
    List<String> strings = Arrays.asList("Java", "Python", "C++", "JavaScript", "Ruby");
    Optional<String> lastElement = strings.stream().reduce((first,second)->second);
    lastElement.ifPresent(element->System.out.println(element));
```

Numbers

Ques) Armstrong Number

```java
/**
* Armstrong number is a number that is equal to the sum of its own digits,
* , each raised to the power of the number of digits.
*/
public class Check_Armstrong_Number {

  public static void main(String[] args)
  {
     int number = 153; // 3 digit
     String numberInString = String.valueOf(number);
     int length = numberInString.length();
     int digitSum = numberInString.chars()
                        .map(Character::getNumericValue)
                        .map(numb -> (int) Math.pow(numb, length))
                        .sum();

    if (digitSum == number) {
       System.out.println(number + " is  armstrong");
    } else {
       System.out.println(number + " is  not armstrong");
    }
```

Ques) Strong Number

```java
/**
 * --) A Number is said to be a strong number If its factorial sum is equal to the number.
 * --) 145 = 1!+4!+5! = 1+24+120 = 145
 */
public class Check_For_Strong_Number {

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int number = sc.nextInt();
      String numberAsString = String.valueOf(number);
      int sum = Arrays.stream(numberAsString.split(""))
                  .mapToInt(string -> Integer.valueOf(string))
                  .boxed()
                  .map(Check_For_Strong_Number::factorial)
                  .reduce((num1, num2) -> num1 + num2).get();

      if(sum == number) {
         System.out.println(number+ " is a Strong Number");
      }
      else {
         System.out.println(number+ " is not a Strong Number");
      }
```

Ques) Check Number is Prime or Not

```java
public class Check_Number_Is_Prime_Or_Not {

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int number = sc.nextInt();
      boolean isPrime = IntStream.rangeClosed(2, (int)Math.sqrt(number))
                                 .allMatch(n->number%n!=0);
      System.out.println(number +" is prime : "+isPrime);
```

Ques) Pallindrom Number Check

```
public class Check_Pallindrome_Number {

  public static void main(String[] args) {
     System.out.println("Enter the number to check pallindrome : ");
     Scanner sc= new Scanner(System.in);
     int number = sc.nextInt();

     String numberAsString = String.valueOf(number);
     int length = numberAsString.length();

     boolean isPallindrome =IntStream
                                .rangeClosed(0, (int)length/2)
                                .allMatch(position->numberAsString.charAt(position)==numberAsString.charAt(length-1-position));
     System.out.println(number + "  is  Pallindrome : "+isPallindrome);
  }
```

Ques) Find Prime Numbers

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number till which you want prime Number");
    int number = sc.nextInt();
    List<Integer> primeNumbers = IntStream.rangeClosed(2, 10).filter(Find_Prime_Number::isPrime).boxed()
            .collect(Collectors.toList());
    System.out.println("Prime Number till " + number + " : " + primeNumbers);
}

public static boolean isPrime(int number) {
    boolean isPrime = number > 1
            && IntStream.rangeClosed(2, (int) Math.sqrt(number)).allMatch(n -> number % n != 0);
    return isPrime;
}
```

Ques) Sum Of Natural Number

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number till which you want sum");
    int number = sc.nextInt();

    int sumOfNaturalNumber = IntStream.rangeClosed(0, number).reduce((num1, num2) -> num1 + num2).orElse(0);
    System.out.println("Sum Of " + number + "natural number is : " + sumOfNaturalNumber);
```

Employee

1) Second Highest Salary in the Organization

```
private static void secondHighestSalaryOfTheOrganisation(List<Employee> employeeList) {
    System.out.println("Second Highest Salary In the Organisation : ");
    Optional<Employee> optionalEmployeeObject = employeeList.stream()
            .sorted(Comparator.comparing(Employee::getSalary).reversed()).skip(1).findFirst();

    optionalEmployeeObject.ifPresent(employee->System.out.println("EmpName : "+employee.getEmpName() + " : "+employee.getSalary()))

    /**
     *  Another way
     */

    double secondHighestsalary =employeeList.stream().map(employee->employee.getSalary())
    .sorted(Collections.reverseOrder(Double::compareTo)).skip(1).findFirst().orElseGet(null);

    System.out.println("secondHighestsalary  :"+secondHighestsalary);
}
```

2) Older Employee Of the Organisation

```
private static void olderEmployeeInOrganisation(List<Employee> employeeList) {
    System.out.println("Older Employee In the Organisation : ");
    Optional<Employee> olderEmployee= employeeList.stream().sorted(Comparator.comparing(Employee::getAge).reversed()).findFirst();
    olderEmployee.ifPresent(employee->System.out.println("EmpName : "+employee.getEmpName() + " Age : "+employee.getAge()));
}
```

3) Average and Total Salary of Organisation

```
private static void averageAndTotalSalaryOfOrganisation(List<Employee> employeeList) {
   System.out.println("Average  Salary Of the Organisation : ");

   /**
    * Note :
    * ===========
    *
    *  --)  If the Stream is IntStream , we can directly apply sum(),average() functions.
    *  --) Thats why For this question, instead of map which will return Stream<Double> I choose MapTODouble which return DoubleStream.
    *  --) Any Place where you have to do some task like
    *
    *                              findAverage()
    *                              findSum()
    *
    *          Choose IntStream or DoubleStream to apply sum() and average() method  . Later on apply method getAsDouble/getAsInt to parse.
    */
   double averageSalaryOfOrganisation =employeeList.stream().mapToDouble(Employee::getSalary).average().getAsDouble();
   System.out.println("averageSalaryOfOrganisation : "+averageSalaryOfOrganisation);

   double totalSalaryOfOrganisation = employeeList.stream().mapToDouble(Employee::getSalary).sum();
   System.out.println("totalSalaryOfOrganisation : "+totalSalaryOfOrganisation);
   System.out.println("*****************************************************");
}
```

4) Average Salary of Male And Female

```
private static void averageSalaryOfMaleAndFemaleEmployee(List<Employee> employeeList) {
    System.out.println("AverageSalaryOfMaleAndFemale");
    double averageSalaryOfMale = employeeList.stream()
                                        .filter(employee->employee.getGender().equals("Male"))
                                        .mapToDouble(Employee::getSalary)
                                        .average()
                                        .getAsDouble();
    double averageSalaryOfFeMale = employeeList.stream()
            .filter(employee->employee.getGender().equals("Female"))
            .mapToDouble(Employee::getSalary)
            .average()
            .getAsDouble();
    System.out.println("averageSalaryOfMale : "+averageSalaryOfMale);
    System.out.println("averageSalaryOfFeMale : "+averageSalaryOfFeMale);
    System.out.println("*************************************************");
}
```

5) Most Experience Person in Organisation

```
private static void mostExperienceInOrganisation(List<Employee> employeeList) {

    System.out.println("Most Experience Person in the Organisation : ");
    Optional<Employee> employeeSortingbasedonJoining = employeeList.stream()
          .sorted(Comparator.comparing(Employee::getYearOfJoining)).findFirst();

    /** Another Way */
    Optional<Employee> employeeSortingbasedonJoining2 = employeeList.stream()
          .sorted(Comparator.comparingInt(Employee::getYearOfJoining)).findFirst();
    /** Another Way */

    Optional<Employee> employeeSortingbasedonJoining3 = employeeList.stream()
          .sorted((e1,e2)->Integer.compare(e1.getYearOfJoining(), e2.getYearOfJoining())).findFirst();

       employeeSortingbasedonJoining.ifPresent(employee -> System.out
          .println("EmployeeName : " + employee.getEmpName() + " Joining : " + employee.getYearOfJoining()));

       System.out.println("*************************************************");
 }
```

Remember one thing

```
employeeList.stream()
      .sorted(Employee::getYearOfJoining).findFirst();
```

➔ If we directly try to write like above ss . It will give error because sorted expect Comparator of Employee Object as we are streaming on empList.

➔ sorted(Employee::getYearOfJoining) will give **int** which is wrong as per implementation.

6) **Yongest Employee In the Production Department**

```
private static void youngestEmployeeInProductionDepartment(List<Employee> employeeList)
{
   System.out.println("Youngest Employee In Production  ");
   Employee youngestEmployeeInProduction =
         employeeList.stream()
         .sorted(Comparator.comparing(Employee::getYearOfJoining).reversed())
         .filter(employee->employee.getDepartment().equals("Product Development")).findFirst().get();
   System.out.println("YoungestEmployeeInProduction : "+youngestEmployeeInProduction);
   System.out.println("**************************************************");
}
```

7) **Average Salary of Each Department**

```
private static void averageSalaryOfEachDepartment(List<Employee> employeeList) {
   System.out.println("Average Salary Of Each Department In the Organisation");
   Map<String, Double> averageSalaryOfEachDepartment = employeeList.stream()
         .collect(Collectors.groupingBy(Employee::getDepartment, Collectors.averagingDouble(Employee::getSalary)));
   System.out.println(averageSalaryOfEachDepartment);
   System.out.println("**************************************************");

}
```

8) Count of Employee In Each Department

```
private static void countOfEmployeeInEachDepartment(List<Employee> employeeList) {
    System.out.println("Count Of Employee In Each Department In the Organisation");
    Map<String, Long> employeeCountFromEachDepartment = employeeList.stream()
            .collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));
    employeeCountFromEachDepartment.entrySet().forEach(entry -> System.out
            .println("Department : " + entry.getKey() + " Total EMployee : " + entry.getValue()));
    System.out.println("************************************************");

}
```

9) Employee Joined After 2015

```
private static void employeeJoineAfter2015(List<Employee> employeeList) {
    System.out.println("Employee Joined After 2015");
    List<Employee> employeeJoinedAfter2015 = employeeList.stream()
            .filter(employee -> employee.getYearOfJoining() > 2015).collect(Collectors.toList());
    System.out.println(employeeJoinedAfter2015);
    System.out.println("************************************************");

}
```

10) Highest Salary of Employee

```
private static void highestSalaryOfEmployee(List<Employee> employeeList) {
    System.out.println("Highest Salary Of Employee");
    Employee employeeWithHighestSalary = employeeList.stream()
            .sorted(Comparator.comparing(Employee::getSalary).reversed()).findFirst().get();
    System.out.println("employeeWithHighestSalary : " + employeeWithHighestSalary);
    System.out.println("************************************************");

}
```

11) Average age of Male and Female

```
private static void averageAgeOfMaleAndFemale(List<Employee> employeeList) {
    System.out.println("Average Age Of Male And Female");
    double averageAgeofMaleInTheOrganisation = employeeList.stream()
            .filter(employee -> employee.getGender().equals("Male")).mapToInt(Employee::getAge).average()
            .getAsDouble();
    double averageAgeofFemaleInTheOrganisation = employeeList.stream()
            .filter(employee -> employee.getGender().equals("Female")).mapToInt(Employee::getAge).average()
            .getAsDouble();

    System.out.println("averageAgeofFemaleInTheOrganisation : " + averageAgeofFemaleInTheOrganisation);
    System.out.println("averageAgeofMaleInTheOrganisation : " + averageAgeofMaleInTheOrganisation);

    /** Another way */
    Map<String, Double> averageAgeOfMaleAndFemale = employeeList.stream()
            .collect(Collectors.groupingBy(Employee::getGender, Collectors.averagingDouble(Employee::getAge)));

    System.out.println("averageAgeOfMaleAndFemale : " + averageAgeOfMaleAndFemale);
    System.out.println("*****************************************************");

}
```

12) Count Of Male And Female In the Organisation

```
private static void countOfMaleAndFemale(List<Employee> employeeList) {
    System.out.println("Count of Total Male And Female");
    Map<String, Long> numberOfMaleAndFemale = employeeList.stream()
            .collect(Collectors.groupingBy(Employee::getGender, Collectors.counting()));

    System.out.println("numberOfMaleAndFemale :"+numberOfMaleAndFemale);
    System.out.println("**************************************************");

}
```

13) Name Of All the Departments

```
private static void nameOfAllTheDepartment(List<Employee> employeeList) {
    System.out.println("Name Of All The Department");
    List<String> allTheDepartmentInTheOrgansiation = employeeList.stream().map(Employee::getDepartment)
            .collect(Collectors.toList());
    System.out.println("allTheDepartmentInTheOrgansiation : " + allTheDepartmentInTheOrgansiation);
    System.out.println("**************************************************");

}
```

14) Count Of Employee In Each Department

```
private static void countOfEmployeeInEachDepartment(List<Employee> employeeList) {
    System.out.println("Count Of Employee In Each Department In the Organisation");

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

    employeeCountFromEachDepartment.entrySet().forEach(entry -> System.out
            .println("Department : " + entry.getKey() + " Total EMployee : " + entry.getValue()));
    System.out.println("***********************************************");

}
```

Tricky Questions

```
2
3  public class Tricky1 {
4
5      public static void main(String[] args) {
6
7          /**
8           *  What would be the output
9           */
0            int i = Integer.MAX_VALUE;
1            System.out.println(++i);  // -ve of highest range
2
```

```
<terminated> Tricky1 [Java Application] C:\Projec
-2147483648
```

```java
class Test
{
   @Override
   public int hashCode() {
      return 1;
   }
   @Override
   public boolean equals(Object test) {
      return true;
   }
}
public class Tricky2 {

   public static void main(String[] args)
   {
   Map<Test,String> map = new HashMap<>();
    Test t1 = new Test();
    Test t2 = new Test();
    map.put(t1,"hi");
    map.put(t2,"hello");

    System.out.println( map.get(t1));
    System.out.println( map.get(t2));
```

```
<terminated> Tricky2 [Java Application] C:\Projects\sts-4.19.0.RELE
hello
hello
```

1 `HashMap.get(key)` Internal Working

`HashMap` uses an **array of buckets** (an array of `Node<K, V>` objects) and a **hashing mechanism** to store and retrieve data efficiently.

Steps in `get()` for `HashMap`:

1. **Compute Hashcode**
   - The key's `hashCode()` method is called to generate a hash value.
   - Hash value is further processed using `hash()` function to minimize collisions.
2. **Find Bucket Index**
   - The hash value is converted to an index in the underlying array:

     java Copy Edit

     ```
     index = (n - 1) & hash
     ```

     where `n` is the array size.
3. **Check for Matching Entry**
   - If no entry is found at that index, return `null`.
   - Otherwise, compare the key using `.equals()`.
   - If a match is found, return the corresponding value.