1. **Difference between Abstraction and Encapsulation**.

Abstraction and **Encapsulation** are two key concepts in object-oriented programming (OOP) that are often used together but serve different purposes.

Abstraction: Abstraction refers to the concept of hiding the complex implementation details of a system and exposing only the essential features or functionalities. The goal is to focus on what an object does rather than how it does it.

In Java, abstraction can be achieved using **abstract classes** or **interfaces**. By defining abstract methods, you can specify the behavior that must be implemented, but the actual implementation is left to the subclasses.

Real-time Example of Abstraction:

Think of a **remote control**. A remote control allows you to interact with different devices (like TV, AC, etc.), but you don't need to understand how the internals of the remote or the devices work. You only need to know the buttons (functions) on the remote and how to use them (e.g., "Power On", "Change Channel", "Adjust Volume").

- **Remote Control** is the **abstract class**.
- **TV** and **AC** are the subclasses that provide specific implementations for the buttons.

```
abstract class RemoteControl {
   abstract void powerOn();
   abstract void powerOff();
}

class TV extends RemoteControl {
   void powerOn() {
      System.out.println("Turning on the TV");
   }

   void powerOff() {
      System.out.println("Turning off the TV");
   }
}

class AC extends RemoteControl {
   void powerOn() {
      System.out.println("Turning on the AC");
   }

   void powerOff() {
      System.out.println("Turning off the AC");
   }
}
```

In this example, RemoteControl defines the actions (abstraction), and the TV and AC provide the specific implementation.

Encapsulation:

Encapsulation is the process of bundling data (variables) and methods that operate on the data into a single unit, typically a **class**. It also involves restricting direct access to some of an object's components, which can prevent unintended interference and misuse of the data. This is achieved using **access modifiers** (private, public, etc.) to restrict access to the fields and provide controlled access through **getters** and **setters**.

Real-time Example of Encapsulation:

Consider a **Bank Account**. You would not want the balance to be directly accessible or modifiable by external entities; instead, you'd want to control how the balance is accessed or updated.

- **BankAccount** is an encapsulated class with private data (balance).
- Public methods (getters and setters) provide controlled access to modify or view the balance.

```
class BankAccount {
   private double balance;  // Private data (Encapsulation)

   // Getter for balance
   public double getBalance() {
      return balance;
   }

   // Setter for balance
   public void deposit(double amount) {
      if (amount > 0) {
         balance += amount;  // Ensuring that deposit only adds a positive amount
      }
   }

   public void withdraw(double amount) {
      if (amount > 0 && amount <= balance) {
         balance -= amount;  // Ensuring that withdrawal doesn't exceed balance
      }
   }
}
```

In this case, the balance field is hidden (encapsulated) from direct access, and changes to the balance can only happen through the controlled methods deposit() and withdraw().

Key Differences:

Aspect	Abstraction	Encapsulation
Definition	Hiding implementation details and showing only essential features.	Bundling data and methods and restricting direct access to data.
Focus	Focus on **what** the object does.	Focus on **how** data is stored and accessed.
Achieved by	Abstract classes and interfaces.	Access modifiers (private, public, etc.) and methods.
Goal	To reduce complexity and improve usability.	To protect data and ensure controlled access to it.
Visibility	Hides implementation but exposes method signatures.	Hides data (variables) and provides controlled access via methods.

Real-time Comparison:

- **Abstraction**: When you use an **ATM machine**, you know how to withdraw money (the interface), but you don't know how the internal systems (like server communication, account validation, etc.) work.
- **Encapsulation**: The **ATM machine** encapsulates your personal data, like your account number and balance, behind the scenes. You can only interact with it through specific methods like **PIN entry** and **withdrawal**.

In summary:

- **Abstraction** simplifies the interaction with complex systems by focusing on essential features.
- **Encapsulation** ensures that an object's internal data is safe and accessed only in a controlled manner.

2. Can we achieve Encapsulation without Abstraction if yes, how?

Yes, **Encapsulation** can be achieved without **Abstraction** in Java, although they are often used together in object-oriented programming. Let's break down how encapsulation works on its own and how you can achieve it without explicitly using abstraction.

Encapsulation Without Abstraction:

Encapsulation simply refers to **bundling data (fields) and methods (functions) together into a single unit (class)** and controlling the access to that data. This is often done using **access modifiers** (private, public, protected, etc.), which allow you to control how the fields of an object can be accessed or modified. You can achieve encapsulation without the need for abstraction (abstract classes or interfaces).

How Encapsulation Works Independently:

1. **Private Fields**: You make the fields (variables) of a class private so they cannot be accessed directly from outside the class.
2. **Public Methods**: You provide public getter and setter methods to allow controlled access to the private fields. This ensures that the fields can only be modified or retrieved in a controlled manner.

Example of Encapsulation Without Abstraction:

Let's say you have a simple **Employee** class where you want to encapsulate the details (like name and salary), but you don't need any abstract methods or interfaces (abstraction).

```
class Employee {
    // Private fields (data encapsulation)
    private String name;
    private double salary;

    // Public constructor to initialize the employee object
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // Getter for name (encapsulation)
    public String getName() {
        return name;
    }

    // Setter for name (encapsulation)
    public void setName(String name) {
        this.name = name;
    }

    // Getter for salary (encapsulation)
    public double getSalary() {
        return salary;
    }

    // Setter for salary (encapsulation)
    public void setSalary(double salary) {
        if (salary > 0) {
            this.salary = salary;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an Employee object
        Employee emp = new Employee("John Doe", 50000);

        // Accessing and modifying fields through getter and setter (encapsulation)
        System.out.println("Employee Name: " + emp.getName());
        System.out.println("Employee Salary: " + emp.getSalary());
```

```
        // Modifying salary through setter (with validation)
        emp.setSalary(60000);
        System.out.println("Updated Salary: " + emp.getSalary());
    }
}
```

In this example:

- **Encapsulation**: The name and salary fields are private, and you access or modify them through the public methods (getName(), setName(), getSalary(), setSalary()). This encapsulates the data and ensures controlled access.
- **No Abstraction**: There are no abstract classes or interfaces here. All methods and fields are concrete and belong to the same class (Employee), and there's no abstraction happening.

Why Encapsulation Doesn't Require Abstraction:

- **Abstraction** deals with hiding the implementation details and showing only the essential functionality (usually using abstract classes or interfaces), while **Encapsulation** simply protects and controls access to data.
- You can achieve **Encapsulation** by **controlling access** to data using access modifiers and methods like getters and setters without needing to abstract the functionality or hide the details of how things work internally (like in an abstract class or interface).

Encapsulation Focuses on Data Control:

Encapsulation is mainly about the **protection and validation of data**, while abstraction is about **simplifying the interface and hiding implementation details**. Therefore, you can have encapsulation in a class that doesn't require any abstraction mechanisms such as abstract classes or interfaces.

Summary:

- **Encapsulation** can be used without **Abstraction** by simply controlling access to the class fields with private variables and public getter/setter methods.
- **Abstraction** isn't necessary for encapsulation, but they often complement each other in more complex designs.

3. Explain about String constant pool and where objects and literals are stored?

In Java, the **String Constant Pool** is a special area in memory that stores unique instances of string literals to optimize memory usage and improve performance. By storing only one copy of each distinct string value, Java ensures that identical string literals reference the same memory location, reducing redundancy.

Storage of String Literals and Objects

- **String Literals**: When you create a string using a literal (e.g., String s1 = "Hello";), the JVM checks the String Constant Pool. If the literal already exists, it returns a reference to the existing string. If not, it adds the new literal to the pool.
- **String Objects**: Using the new keyword (e.g., String s2 = new String("Hello");) creates a new String object in the heap memory, even if an identical string exists in the pool. This approach does not automatically place the string in the pool. However, invoking the intern() method on this object will add it to the pool if it's not already present.

Real-Time Example

Consider a scenario where an application processes user input for a survey:

```
String response1 = "Yes";
String response2 = "Yes";
String response3 = new String("Yes");
```

- response1 and response2 are string literals. The JVM checks the String Constant Pool and stores "Yes" once. Both variables point to this single instance.
- response3 uses the new keyword, creating a separate object in the heap. It does not reference the pool's "Yes" unless response3.intern() is called.

This mechanism ensures efficient memory usage by avoiding duplicate storage of identical strings, which is particularly beneficial in applications handling large volumes of repetitive string data.

Understanding the String Constant Pool helps developers write memory-efficient code and make informed decisions about string handling in Java applications.

4. An interface have private methods?

Yes, starting from Java 9, interfaces can have private methods. This enhancement allows developers to encapsulate common code within private methods, promoting code reusability and cleaner interface design.

Key Points:

- **Private Methods:** These methods are accessible only within the interface itself. They cannot be called or overridden by implementing classes or subinterfaces.
- **Private Static Methods:** Similar to private methods but declared with the static keyword. They can be called by other static and default methods within the interface.

Usage Example:

```
public interface ExampleInterface {

    // Default method
    default void defaultMethod() {
        commonMethod();
    }

    // Another default method
    default void anotherDefaultMethod() {
        commonMethod();
    }

    // Private method
    private void commonMethod() {
        System.out.println("This is a private method in the interface.");
    }
}
```

In this example:

- commonMethod() is a private method that contains code shared by defaultMethod() and anotherDefaultMethod(). This setup avoids code duplication and keeps the interface implementation clean.

Benefits:

- **Code Reusability:** Encapsulating shared code within private methods allows multiple default methods to reuse common functionality without exposing it to implementing classes.
- **Encapsulation:** Private methods enable hiding implementation details within the interface, ensuring that only relevant methods are exposed to implementing classes.

Considerations:

- **Access Limitations:** Private methods in interfaces are strictly for internal use within the interface. Implementing classes cannot access or override these methods.

This feature enhances the flexibility and maintainability of interfaces by allowing internal method definitions that support default and static methods without cluttering the public API.

5. Purpose of private methods inside an interface if they cannot be assessed outside the interface?

Private methods in Java interfaces, introduced in Java 9, serve the purpose of encapsulating common code within the interface itself. This allows for code reusability among default and static methods without exposing these helper methods to the implementing classes. By keeping such methods private, the interface maintains a clean and controlled API, ensuring that only the intended functionalities are accessible to external classes. cite turn0search1

Key Benefits:

- **Code Reusability:** Private methods enable the sharing of common code among multiple default or static methods within the interface, reducing redundancy.
- **Encapsulation:** They hide the internal implementation details from the implementing classes, exposing only the necessary methods.

Example:

```
public interface Vehicle {

    default void start() {
        checkEngine();
        System.out.println("Vehicle is starting.");
    }

    default void stop() {
        System.out.println("Vehicle is stopping.");
    }

    private void checkEngine() {
        System.out.println("Performing engine check.");
    }
}
```

In this example:

- The Vehicle interface has two default methods: start() and stop().
- Both default methods can utilize the private method checkEngine() to perform a common task without exposing checkEngine() to the implementing classes.

By using private methods in interfaces, developers can create more modular, readable, and maintainable code structures.

6. How static works in inheritance explain with real time example?

In Java, **static methods** are associated with the class itself rather than instances of the class. This characteristic influences how they behave in the context of inheritance. Let's explore this concept with a real-world example.

Key Characteristics of Static Methods in Inheritance:

1. **Class Association**: Static methods belong to the class, not to any specific object instance.
2. **Inheritance**: Static methods are inherited by subclasses. However, if a subclass defines a static method with the same signature as one in its superclass, it **hides** the superclass's method rather than overriding it. This means the method in the superclass is still accessible using the superclass's name.
3. **Polymorphism**: Static methods are **not** polymorphic. The method that gets called is determined by the reference type at compile time, not the actual object type at runtime.

Real-World Example:

Consider a scenario with a base class Vehicle and a derived class Car, both having a static method describe().

```
class Vehicle {
   public static void describe() {
      System.out.println("This is a vehicle.");
   }
}

class Car extends Vehicle {
   public static void describe() {
      System.out.println("This is a car.");
   }
}

public class Main {
   public static void main(String[] args) {
      Vehicle myVehicle = new Vehicle();
      Vehicle myCarAsVehicle = new Car();
      Car myCar = new Car();

      myVehicle.describe();        // Outputs: This is a vehicle.
      myCarAsVehicle.describe();   // Outputs: This is a vehicle.
      myCar.describe();            // Outputs: This is a car.
   }
}
```

Explanation:

- myVehicle.describe(); calls the describe() method of the Vehicle class, outputting "This is a vehicle."
- myCarAsVehicle.describe(); despite referencing a Car object, the reference type is Vehicle. Since static methods are resolved at compile time based on the reference type, it calls the Vehicle class's describe() method, outputting "This is a vehicle."
- myCar.describe(); calls the describe() method of the Car class, outputting "This is a car."

This example illustrates that static methods are not subject to runtime polymorphism. The method invoked is determined by the reference type at compile time, not by the actual object type at runtime.

Key Takeaways:

- Static methods are associated with the class itself and are resolved at compile time.
- They can be inherited but are hidden, not overridden, in subclasses.
- The method that gets called depends on the reference type, not the object type.

7. Explain call by value and call by reference.

In Java, **method parameters are passed by value**, meaning that a copy of the argument is passed to the method. This approach ensures that modifications within the method do not affect the original variable. Let's delve into this concept with detailed examples.

Call by Value in Java

When a method is invoked, Java creates a copy of the actual parameter's value and passes it to the method. Consequently, any changes made to the parameter inside the method do not impact the original variable.

Example with Primitive Data Types:

```
public class CallByValueExample {
   public static void main(String[] args) {
      int originalValue = 10;
      modifyValue(originalValue);
      System.out.println("After method call, originalValue = " + originalValue);
```

```
    }

    public static void modifyValue(int value) {
        value = 20;
    }
}
```

Output:

After method call, originalValue = 10

Explanation:

- originalValue is initialized to 10.
- The modifyValue method is called with originalValue as an argument. Inside the method, value is set to 20. However, this change does not affect originalValue in the main method because only a copy of its value was passed.

Call by Reference in Java

Java does not support **call by reference** in the traditional sense, as it does not allow methods to directly access and modify the original variables passed to them. However, when dealing with objects, Java passes the **reference to the object by value**. This means that while the reference itself is copied, both the original and the copied reference point to the same object. Therefore, modifications to the object's state via the copied reference are reflected in the original object.

Example with Objects:

```
class Person {
    String name;
}

public class CallByReferenceExample {
    public static void main(String[] args) {
        Person person = new Person();
        person.name = "Alice";
        modifyPerson(person);
        System.out.println("After method call, person.name = " + person.name);
    }

    public static void modifyPerson(Person p) {
        p.name = "Bob";
    }
}
```

Output:

After method call, person.name = Bob

Explanation:

- A Person object is created with the name "Alice".
- The modifyPerson method is called with the person object as an argument. Inside the method, the name field of the object is changed to "Bob". Since both the original reference and the method parameter reference point to the same object, this modification is reflected outside the method.

Important Note:

While the object's internal state can be modified through the copied reference, reassigning the reference itself within the method does not affect the original reference.

Example Demonstrating Reference Reassignment:

```
class Person {
   String name;
}

public class ReferenceReassignmentExample {
   public static void main(String[] args) {
      Person person = new Person();
      person.name = "Alice";
      reassignReference(person);
      System.out.println("After method call, person.name = " + person.name);
   }

   public static void reassignReference(Person p) {
      p = new Person();
      p.name = "Charlie";
   }
}
```

Output:

After method call, person.name = Alice

Explanation:

- A new Person object is created and assigned the name "Alice".
- The reassignReference method is called with the person object. Inside the method, p is reassigned to a new Person object with the name "Charlie". This reassignment affects only the local copy of the reference p and does not impact the original person reference in the main method. Therefore, the original object's name remains "Alice".

Summary

- **Call by Value:** Java passes a copy of the variable's value to methods. For primitive data types, this means that modifications within the method do not affect the original variable.
- **Call by Reference:** Java does not support traditional call by reference. However, when passing objects, the reference to the object is passed by value, allowing methods to modify the object's internal state. Reassigning the reference within the method does not affect the original reference.

Understanding these concepts is crucial for predicting how data will behave when passed to methods, ensuring accurate and bug-free code.

8. Can we change the scope of overridden methods? Any rules of it?

In Java, when overriding methods in a subclass, the access modifier of the overriding method can be changed, but there are specific rules to ensure compatibility and maintain encapsulation.

Rules for Changing Access Modifiers in Overridden Methods:

1. **Increase Visibility:** The overriding method can have a more permissive (wider) access level than the method in the superclass. This means you can change the access modifier to one that allows broader access.
2. **Maintain or Restrict Visibility:** The overriding method cannot have a more restrictive access level than the method in the superclass. Doing so would violate the principle of substitutability, where an instance of the subclass should be usable wherever an instance of the superclass is expected.

Access Modifier Hierarchy (from most restrictive to least restrictive):

- private
- *default* (package-private)

- protected
- public

Examples:

1. **Valid Override with Increased Visibility:**

```
class SuperClass {
   protected void display() {
      System.out.println("SuperClass display");
   }
}

class SubClass extends SuperClass {
   @Override
   public void display() {
      System.out.println("SubClass display");
   }
}
```

In this example, the display method in SuperClass has protected access, and the overriding method in SubClass has public access, which is more permissive. This is allowed.

2. **Invalid Override with Decreased Visibility:**

```
class SuperClass {
   public void show() {
      System.out.println("SuperClass show");
   }
}

class SubClass extends SuperClass {
   @Override
   protected void show() {
      System.out.println("SubClass show");
   }
}
```

Here, the show method in SuperClass has public access, but the overriding method in SubClass has protected access, which is more restrictive. This will result in a compile-time error.

Key Takeaways:

- Overriding methods can increase visibility but cannot decrease it.
- Attempting to reduce the access level of an overridden method will cause a compile-time error.
- These rules ensure that the subclass remains substitutable for the superclass, preserving the integrity of polymorphism.

Adhering to these guidelines is crucial for maintaining proper access control and ensuring that subclasses can seamlessly replace their superclasses without violating access constraints.

9. Explain thread life cycle.

In Java, a thread undergoes several states during its lifecycle, each representing a distinct phase of its execution. Understanding these states is crucial for effective multithreaded programming. Here's an overview of the thread lifecycle, accompanied by real-world analogies to illustrate each state:

1. **New (Born) State:**
   - **Definition:** A thread is in the 'New' state when it is created but not yet started.

    - **Analogy:** Consider a person hired for a job but who hasn't begun working yet.
    - **Example:** Creating a thread instance:
    - Thread thread = new Thread();
2. **Runnable (Ready-to-Run) State:**
    - **Definition:** After invoking the start() method, the thread enters the 'Runnable' state, indicating it's ready to run but awaiting CPU allocation.
    - **Analogy:** The hired person is ready and waiting for their turn to use a workstation.
    - **Example:** Starting a thread:
    - thread.start();
3. **Running State:**
    - **Definition:** When the CPU assigns time to the thread, it transitions to the 'Running' state and executes its task.
    - **Analogy:** The person is actively working at their workstation.
    - **Note:** In Java, the 'Running' state is not explicitly distinct from 'Runnable'; a thread in 'Runnable' can be running or ready to run.
4. **Blocked State:**
    - **Definition:** A thread enters the 'Blocked' state when it's waiting to acquire a monitor lock to enter or re-enter a synchronized block/method.
    - **Analogy:** The person needs a specific tool that's currently in use by someone else, so they wait until it becomes available.
    - **Example:** A thread trying to enter a synchronized block that's locked by another thread.
5. **Waiting State:**
    - **Definition:** A thread is in the 'Waiting' state when it waits indefinitely for another thread to perform a particular action.
    - **Analogy:** The person is waiting for a colleague to provide essential information before proceeding.
    - **Example:** A thread waiting indefinitely for another thread to notify it:
    - synchronized (object) {
    - object.wait();
    - }
6. **Timed Waiting State:**
    - **Definition:** A thread enters the 'Timed Waiting' state when it waits for another thread to perform an action within a specified time.
    - **Analogy:** The person waits for a colleague's response but will proceed if there's no reply within a set time.
    - **Example:** A thread sleeping for a specific duration:
    - Thread.sleep(1000); // Sleeps for 1 second
7. **Terminated (Dead) State:**
    - **Definition:** A thread reaches the 'Terminated' state once it completes its execution or is terminated abnormally.
    - **Analogy:** The person has finished their work and leaves the office.
    - **Example:** After the run() method concludes, the thread is terminated.

Real-World Example:

Consider a scenario where multiple users are downloading files concurrently using a download manager. Each download operation can be represented as a separate thread:

- **New State:** A user initiates a download, creating a new download task (thread).
- **Runnable State:** The download task is queued and ready to start once resources are available.
- **Running State:** The download task actively downloads the file.
- **Blocked State:** The download task waits for network resources if the bandwidth is currently fully utilized by other tasks.
- **Waiting State:** The download task waits for user authentication before proceeding.
- **Timed Waiting State:** The download task waits for a set time before retrying a failed connection.
- **Terminated State:** The download completes successfully or fails after retries, ending the task.

Understanding these thread states and their transitions is vital for designing efficient multithreaded applications, ensuring optimal resource utilization and responsiveness.

10. Can we write default and static method in a functional interface?, if yes will lambda expressions will be allowed to use with it?

Yes, you can write **default** and **static methods** in a **functional interface** in Java, and they won't affect the ability to use lambda expressions with it. Here's how:

1. Functional Interface:

- A **functional interface** is an interface that has exactly one **abstract method**.
- It can have multiple **default** or **static methods**, but only one abstract method is required to qualify it as a functional interface.

2. Default Methods:

- **Default methods** in interfaces were introduced in Java 8. They provide a way to add methods to interfaces without breaking the existing implementation of classes that already implement the interface.
- These methods are **not abstract** and have a body. They can provide default behavior for the implementing classes.
- **Default methods in functional interfaces** don't affect the **lambda expression** usage. The lambda expression only needs to implement the **single abstract method** (SAM).

3. Static Methods:

- **Static methods** can be defined in interfaces as well. These methods belong to the interface, not to the instance of the class implementing the interface.
- Static methods don't change the behavior of a lambda expression either since lambdas only target the **abstract method**.

Example:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b); // The single abstract method (SAM)

    // Default method
    default int add(int a, int b) {
        return a + b;
    }

    // Static method
    static int subtract(int a, int b) {
        return a - b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Using Lambda Expression to implement the single abstract method
        Calculator calculator = (a, b) -> a * b; // Implementing calculate method

        // Using the lambda expression for the abstract method
        System.out.println("Multiplication: " + calculator.calculate(5, 2)); // Outputs 10

        // Using the default method from the interface
        System.out.println("Addition: " + calculator.add(5, 2)); // Outputs 7

        // Using the static method from the interface
        System.out.println("Subtraction: " + Calculator.subtract(5, 2)); // Outputs 3
    }
}
```

Output:

```
Multiplication: 10
Addition: 7
Subtraction: 3
```

Key Points:

1. **Default Methods**: Can provide a default implementation for functionality that is optional for implementing classes.
2. **Static Methods**: Belong to the interface and are not inherited by implementing classes.
3. **Lambda Expressions**: Are used to implement only the **abstract method** in a functional interface. They will not be impacted by the default or static methods.
4. **Lambda Expression Usage**: In the above example, the lambda expression is used to implement the calculate method, which is the single abstract method, while default and static methods are simply invoked directly on the interface.

Conclusion:

- **Yes**, you can have **default** and **static methods** in a functional interface.
- **No**, the presence of default or static methods will **not prevent lambda expressions** from being used, as lambda expressions are only concerned with implementing the single abstract method (SAM).

11. If we have an integer a = 5 outside, can we use it in filter() in stream API to check the condition?

Yes, you can use the integer a = 5 (or any other variable) in the filter() method of the Stream API, but there are certain nuances regarding the variable's scope and how it is used.

Key Point:

In Java, variables used inside a lambda expression must be **effectively final**. This means that the variable should not be modified after its initial assignment if it's used in a lambda expression (including within methods like filter()).

Example:

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

public class StreamExample {
   public static void main(String[] args) {
      int a = 5;  // Declare and initialize the variable

      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

      // Use filter with a lambda expression to check the condition based on 'a'
      numbers.stream()
            .filter(num -> num > a)  // 'a' is used here
            .forEach(System.out::println);  // Prints numbers greater than 5
   }
}
```

Output:

```
6
7
8
9
```

Explanation:

- The **variable a** is used in the filter() method to compare each element of the stream against a (which is 5 in this case).
- Since a is **effectively final** (it is initialized once and never modified afterward), it is valid to use it in the lambda expression inside the filter() method.

Important Details:

1. **Effectively Final**: The variable a must not be reassigned after it is declared. If you try to change its value after its initial assignment, it will not be allowed inside the lambda expression.

Example of incorrect usage:

```
int a = 5;
a = 10;  // Error: local variables referenced from a lambda expression must be final or effectively final

numbers.stream()
    .filter(num -> num > a)
    .forEach(System.out::println);
```

2. **Why "Effectively Final"?**: Java's lambdas (and anonymous inner classes) need to access variables from the enclosing scope, and allowing modifications to such variables could lead to unpredictable behavior or threading issues. By requiring variables to be effectively final, Java ensures that these variables do not change while being used inside lambdas.

Conclusion:

- **Yes**, you can use an integer like a = 5 in the filter() method in the Stream API, as long as it is **effectively final**.
- The lambda expression will reference the variable a without any issues as long as a is not modified after its initial assignment.

12. Explain object cloning

In Java, **object cloning** refers to the process of creating a new object that is an exact copy of an existing object. This is typically done using the clone() method, which is defined in the Object class.

Key Points:

- **Shallow Copy**: The default clone() method creates a **shallow copy** of an object, meaning it copies the object's primitive fields directly but does not create copies of objects referenced by the fields (i.e., references are copied, not the objects themselves).
- **Deep Copy**: A **deep copy** involves recursively copying all objects referenced by the original object.

To use cloning in Java, the class must implement the **Cloneable** interface. This is a marker interface (it doesn't have any methods), and if a class does not implement Cloneable and the clone() method is called, it will throw a **CloneNotSupportedException**.

Steps to Clone an Object in Java:

1. The class must implement the Cloneable interface.
2. Override the clone() method to make it accessible (since it's protected in the Object class).
3. Use the clone() method to create a new instance that is a copy of the original.

Example of Object Cloning in Java:

Let's consider a real-time example: Cloning a Person object.

```
class Address {
   String street;
   String city;

   // Constructor
   public Address(String street, String city) {
      this.street = street;
      this.city = city;
   }

   // Clone method for Address (Deep Copy)
   public Address clone() {
      return new Address(this.street, this.city);
   }
}
```

```
class Person implements Cloneable {
    String name;
    int age;
    Address address;

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

    // Overriding clone() method to create a shallow copy of Person object
    @Override
    public Person clone() {
        try {
            Person cloned = (Person) super.clone();
            // Deep cloning the Address object to avoid shared references
            cloned.address = this.address.clone();
            return cloned;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Original object
        Address address = new Address("123 Main St", "Springfield");
        Person originalPerson = new Person("John Doe", 30, address);

        // Cloning the original person
        Person clonedPerson = originalPerson.clone();

        // Printing the original and cloned person details
        System.out.println("Original Person: " + originalPerson.name + ", " + originalPerson.age + ", Address: " +
originalPerson.address.street);
        System.out.println("Cloned Person: " + clonedPerson.name + ", " + clonedPerson.age + ", Address: " +
clonedPerson.address.street);

        // Modify the address in cloned object
        clonedPerson.address.street = "456 Elm St";

        // Verify that modifying cloned address does not affect the original address
        System.out.println("After modification:");
        System.out.println("Original Person Address: " + originalPerson.address.street); // Should be "123 Main St"
        System.out.println("Cloned Person Address: " + clonedPerson.address.street);   // Should be "456 Elm St"
    }
}
```

Output:

```
Original Person: John Doe, 30, Address: 123 Main St
Cloned Person: John Doe, 30, Address: 123 Main St
After modification:
Original Person Address: 123 Main St
Cloned Person Address: 456 Elm St
```

Explanation:

1. **Cloning Process**:
    - The Person class implements the Cloneable interface and overrides the clone() method.
    - The clone() method first performs a shallow copy (by calling super.clone()), and then it performs a deep copy of the Address object by calling the clone() method of the Address class.
2. **Shallow Copy**:
    - The clone() method of Person creates a shallow copy of the object (super.clone()), which means it copies the values of name and age directly.
    - However, the address field is still a reference to the original Address object. To ensure a deep copy, we manually clone the Address object inside the clone() method.
3. **Deep Copy**:
    - The Address class also has a clone() method, which ensures that when a Person object is cloned, its Address object is also cloned, creating a completely new Address instance for the cloned Person.
4. **Modifying the Cloned Object**:
    - After cloning, we modify the street field of the cloned Address.
    - As a result, the original Person's Address remains unaffected, confirming that a deep copy was made.

Shallow vs. Deep Copy:

- **Shallow Copy**: Only the references to objects are copied, not the objects themselves. So, changes made to a mutable object in the copied instance will affect the original.

  Example: If we hadn't deep cloned the Address object, changes made to the address in the cloned Person would have reflected in the original Person as well, because both would share the same Address object.

- **Deep Copy**: A completely new object is created for all mutable fields. Changes in the copied object do not affect the original object.

When to Use Object Cloning:

- **Performance**: Cloning can be useful when creating a new object that is a copy of an existing one, especially when the cost of creating an object from scratch is expensive.
- **Avoiding Side Effects**: In cases where you want to avoid shared references between the original and the cloned object (as with deep cloning).

Conclusion:

- Object cloning in Java creates a duplicate of an object, and it's often used for efficiency or when you need to create a copy of an object without affecting the original.
- Java provides a clone() method, but you must handle deep cloning manually for objects containing mutable fields.
- **Shallow cloning** is default, while **deep cloning** requires additional handling to ensure no shared references between the original and the clone.

13. Explain method-overriding rules.

Method Overriding in Java is a feature that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is a key concept in **polymorphism**, enabling the subclass to modify or extend the behavior of a superclass method.

Rules for Method Overriding:

1. **Same Method Signature**:
    - The **method name**, **return type**, and **parameter list** must be the same in both the subclass and the superclass.
    - Example:

```
class Animal {
   void makeSound() {
      System.out.println("Animal makes a sound");
   }
}
```

```
class Dog extends Animal {
   // Overriding the makeSound method
   @Override
   void makeSound() {
      System.out.println("Dog barks");
   }
}
```

2. **Return Type**:
    - The **return type** of the overridden method in the subclass must be the same as the return type of the method in the superclass.
    - From Java 5 onward, you can **covariantly return** a more specific type (i.e., the return type in the subclass can be a subclass of the return type in the superclass).
    - Example (covariant return type):

```
class Animal {
   Animal getAnimal() {
      return new Animal();
   }
}

class Dog extends Animal {
   @Override
   Dog getAnimal() {
      return new Dog();
   }
}
```

3. **Access Modifier**:
    - The **access level** of the overridden method in the subclass should be the same or more **accessible** than the method in the superclass.
    - For example:
        - If the superclass method is **public**, the overriding method in the subclass can be public but **cannot** be protected or private.
        - If the superclass method is **protected**, the overriding method can be protected or public, but **not** private.
        - If the superclass method is **private**, it **cannot** be overridden at all (private methods are not inherited by subclasses).
    - Example:

```
class Animal {
   public void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   @Override
   public void sound() {  // Same or more accessible modifier (public)
      System.out.println("Dog barks");
   }
}
```

4. **Method Parameters**:
    - The **parameter list** in the overriding method must be **exactly the same** as in the superclass method. However, you can use **varargs** in the overriding method, but the number and types of parameters should match.
    - You can also override a method with more specific types (i.e., method parameters can be **covariant**, similar to the return type).
    - Example:

```
class Animal {
```

```
    void makeSound(String sound) {
        System.out.println("Animal makes a sound: " + sound);
    }
}

class Dog extends Animal {
    @Override
    void makeSound(String sound) {
        System.out.println("Dog barks: " + sound);
    }
}
```

5. **final, static, and private Methods**:
    - **final** methods in the superclass cannot be overridden in the subclass because they are marked as unchangeable.
    - **static** methods are not considered for overriding but can be **re-declared** (which is called **method hiding**). The method in the subclass **does not override** the method in the superclass, but rather hides it.
    - **private** methods cannot be overridden because they are not inherited by the subclass.

    Example:

```
class Animal {
    final void sound() {
        System.out.println("Animal makes a sound");
    }

    static void staticMethod() {
        System.out.println("Static method in Animal");
    }

    private void privateMethod() {
        System.out.println("Private method in Animal");
    }
}

class Dog extends Animal {
    // Error: Cannot override final method from Animal
    // void sound() { ... }

    // Hiding static method (not overriding)
    static void staticMethod() {
        System.out.println("Static method in Dog");
    }

    // Error: Cannot override private method from Animal
    // void privateMethod() { ... }
}
```

6. **@Override Annotation**:
    - The @Override annotation is optional but highly recommended as it provides compile-time checking. It ensures that you are actually overriding a method and not accidentally creating a new method with a similar name.
    - If the method in the subclass does not match the method in the superclass (for example, wrong parameter types), the compiler will show an error.

    Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
```

```
class Dog extends Animal {
    @Override
    void sound() {  // Correct override
        System.out.println("Dog barks");
    }
}
```

7. **Overriding Methods Must Not Throw More Exceptions**:
    - The subclass overriding method cannot throw more **checked exceptions** than the method in the superclass. It can throw **fewer exceptions** or the same exceptions, but **not more**.
    - Unchecked exceptions (like RuntimeException and its subclasses) can be thrown in the subclass method, even if they are not declared in the superclass method.

Example:

```
class Animal {
    void makeSound() throws Exception {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() throws Exception { // OK: same exception
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() throws RuntimeException { // OK: unchecked exception
        System.out.println("Cat meows");
    }
}

class Parrot extends Animal {
    // Error: overriding method cannot throw new checked exceptions
    @Override
    void makeSound() throws IOException {
        System.out.println("Parrot chirps");
    }
}
```

Real-time Example of Method Overriding:

Let's consider a real-time scenario where a **PaymentSystem** class is extended by different payment types like **CreditCardPayment** and **PayPalPayment**.

```
// Parent class
class PaymentSystem {
    void processPayment(double amount) {
        System.out.println("Processing payment of $" + amount);
    }
}

// Child class 1: CreditCardPayment
class CreditCardPayment extends PaymentSystem {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing Credit Card payment of $" + amount);
    }
```

```
}

// Child class 2: PayPalPayment
class PayPalPayment extends PaymentSystem {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}

public class PaymentExample {
    public static void main(String[] args) {
        PaymentSystem payment = new CreditCardPayment();
        payment.processPayment(100.0);  // Calls overridden method in CreditCardPayment

        payment = new PayPalPayment();
        payment.processPayment(200.0);  // Calls overridden method in PayPalPayment
    }
}
```

Output:

```
Processing Credit Card payment of $100.0
Processing PayPal payment of $200.0
```

Conclusion:

- **Method Overriding** allows a subclass to provide its own implementation of a method defined in the superclass.
- The method signature, return type, and parameters must match between the superclass and the subclass.
- The access modifier of the overridden method should be the same or more accessible than the superclass method.
- Static, private, and final methods cannot be overridden.
- The @Override annotation ensures that the method is properly overridden and provides compile-time error checking.

14. Difference between Map and reduce?

The **Map** and **Reduce** operations are two key concepts in functional programming, and they are commonly used in stream processing, especially in Java's **Stream API**. Both of them are used to process data in parallel or sequential pipelines but serve different purposes.

Map:

The map() operation is used to **transform** each element of a stream into another element. It applies a function to each element of the stream and produces a new stream with the transformed elements.

Key Characteristics of map():

- **Transformation**: map() is used for **transforming** each element individually.
- **Function**: It takes a function as an argument that operates on each element of the stream.
- **Output**: The result of map() is a new stream with the same number of elements as the original, but with the transformed data.

Example of map():

Let's say we have a list of integers, and we want to double each element.

```
import java.util.List;
import java.util.stream.Collectors;

public class MapExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);
```

```
        List<Integer> doubled = numbers.stream()
                                .map(n -> n * 2)  // Doubling each number
                                .collect(Collectors.toList());

        System.out.println(doubled);  // Output: [2, 4, 6, 8, 10]
    }
}
```

Real-Time Example:

- In a **salary increase system**, you could use map() to apply a 10% increase to each employee's salary in a list.

Reduce:

The reduce() operation is used to **combine** all elements of a stream into a **single result**. It performs a **reduction** using an associative accumulation function, which takes two elements and combines them.

Key Characteristics of reduce():

- **Aggregation**: reduce() is used for **aggregating** elements into a single result.
- **Accumulator Function**: It takes a binary operator (a function that combines two elements of the stream) as an argument.
- **Output**: The result of reduce() is a single value that is the result of combining all elements of the stream.

Example of reduce():

Let's say we want to calculate the sum of all elements in a list of integers.

```
import java.util.List;
import java.util.Optional;

public class ReduceExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // Using reduce to sum the numbers
        Optional<Integer> sum = numbers.stream()
                                .reduce((a, b) -> a + b);

        sum.ifPresent(System.out::println);  // Output: 15
    }
}
```

Real-Time Example:

- In a **sales report**, you could use reduce() to calculate the total revenue from a list of sales transactions.

Comparison Between Map and Reduce:

Aspect	Map	Reduce
Purpose	Transforms each element in the stream.	Aggregates the elements of the stream into a single result.
Operation Type	Transformation (one-to-one mapping).	Aggregation (one-to-many or many-to-one).

Aspect	Map	Reduce
Result	Produces a new stream of the same size.	Produces a single result (e.g., sum, concatenation).
Function Type	Takes a function that operates on a single element.	Takes a binary operator (combines two elements at a time).
Use Case	Applying a transformation to each element in the stream.	Combining elements into a final result (e.g., sum, product).
Output Type	Same as input stream but transformed.	A single value (can be any type, depending on the accumulator).

Key Differences in Action:

1. **Map** transforms individual elements into another form:
    - **Before**: A list of integers [1, 2, 3]
    - **After map()**: A list of doubled integers [2, 4, 6]
2. **Reduce** aggregates the entire stream into a single result:
    - **Before**: A list of integers [1, 2, 3]
    - **After reduce()**: A single sum value 6 (1 + 2 + 3)

Real-Time Analogy:

- **Map**: Imagine a factory where each worker transforms raw material (an integer) into a new form (doubled value).
- **Reduce**: Now, at the end of the production line, all the products (doubled values) are combined (summed up) to give you the final output.

Conclusion:

- **Map** is useful for **transforming** data.
- **Reduce** is useful for **combining** data into a single value.

15. If you don't use a terminal operation in a stream pipeline with the intermediate operations be executed ? why or why not?

No, **intermediate operations** in a stream pipeline will **not be executed** unless a **terminal operation** is invoked.

In Java's Stream API, intermediate operations are **lazy**; they don't execute until a terminal operation is invoked. Intermediate operations such as map(), filter(), and sorted() define the transformations or filters that need to be applied to the stream elements, but they are not executed immediately. Instead, they are only executed when a terminal operation (like collect(), forEach(), reduce(), count(), etc.) triggers the pipeline's processing.

Why Intermediate Operations Are Lazy:

1. **Optimization**: By deferring the execution of operations, Java Streams can optimize the pipeline. For example, the stream can combine multiple operations into a single pass over the data, which can reduce the overhead of processing.
2. **Short-circuiting**: Lazy evaluation allows for **short-circuiting** behavior in some cases (like in filter() or findFirst()), where the stream can stop processing as soon as it has found what it needs.
3. **Performance**: If no terminal operation is called, no processing occurs, and the stream remains in an unprocessed state. This can save unnecessary computation, especially if the intermediate operations are complex or if only specific operations are required based on user input.

Example:

Consider this example where we have a stream with an intermediate operation but no terminal operation:

```
import java.util.List;
```

```
public class StreamLazyEvaluation {
   public static void main(String[] args) {
      List<Integer> numbers = List.of(1, 2, 3, 4, 5);

      // Intermediate operations: filter() and map()
      numbers.stream()
            .filter(n -> n % 2 == 0)  // Filtering even numbers
            .map(n -> n * 2);         // Doubling the numbers

      // No terminal operation here, so nothing happens.
   }
}
```

In this case, the stream pipeline defines intermediate operations: filter() and map(), but **no terminal operation** (like collect(), forEach(), or reduce()) is called. As a result:

- **Nothing is executed**. The intermediate operations are defined, but they are not applied to the data.
- There is **no processing** of the list numbers because the pipeline has not been "triggered" by a terminal operation.

How Terminal Operations Trigger Execution:

Now, let's add a terminal operation, such as forEach():

```
import java.util.List;

public class StreamLazyEvaluation {
   public static void main(String[] args) {
      List<Integer> numbers = List.of(1, 2, 3, 4, 5);

      // Intermediate operations: filter() and map()
      numbers.stream()
            .filter(n -> n % 2 == 0)  // Filtering even numbers
            .map(n -> n * 2)          // Doubling the numbers
            .forEach(System.out::println);  // Terminal operation triggers the stream processing
   }
}
```

Output:

```
4
8
```

In this case:

- The **forEach()** terminal operation triggers the pipeline.
- The stream processes each element by first filtering out the odd numbers (filter(n -> n % 2 == 0)), then doubling the even numbers (map(n -> n * 2)), and finally printing the result.

Summary:

- **Intermediate operations** (like map(), filter(), etc.) **do not execute** until a **terminal operation** (like collect(), forEach(), etc.) is invoked.
- **Lazy evaluation** improves performance by allowing the stream pipeline to be optimized or short-circuited.
- **No terminal operation** means the stream pipeline remains unexecuted.

16. Internal working of HashMap using real time example?

A **HashMap** in Java is a part of the java.util package and implements the Map interface. It stores key-value pairs, where each key maps to a single value, and it provides **constant time performance** for basic operations like get() and put(), assuming the hash function disperses the elements properly among the buckets.

The internal working of HashMap involves several key concepts, such as **hashing**, **buckets**, and **collisions**. Let's dive into how these work internally.

1. Hashing

When you insert a key-value pair into a HashMap, the **hash code** of the key is used to compute an index in an array called the **backing array** or **bucket array**. This is done by the hashCode() method of the key object.

- **hashCode()**: Each object in Java has a hash code, which is an integer that represents the object.
- **Index Calculation**: The hash code is then processed (e.g., using modulo or bitwise operations) to determine an index in the backing array (also known as the **bucket array**). The array holds **buckets** where key-value pairs are stored.

2. Buckets

A **bucket** is simply a container (usually a linked list or a balanced tree) in which entries (key-value pairs) are stored at a specific index. If two keys have the same hash code (which is known as a **hash collision**), they will be stored in the same bucket.

3. Collisions

A **collision** happens when two distinct keys produce the same hash code, causing them to map to the same index in the array. In such cases:

- The HashMap resolves collisions by chaining the entries at the same index. This is typically done using a **linked list** (or in some cases a **balanced tree** if the number of elements in the bucket exceeds a threshold).

4. Resizing

When the number of entries in the HashMap exceeds a certain threshold (typically 75% of the current capacity), the HashMap will resize itself. This means:

- The backing array (bucket array) is expanded, and the entries are rehashed to new positions in the larger array.
- Resizing ensures that the HashMap maintains efficient performance as more key-value pairs are added.

5. Load Factor

The **load factor** is a measure of how full the HashMap can get before it needs to resize. The default load factor is 0.75, meaning when 75% of the capacity is filled, the map resizes.

Real-Time Example of HashMap:

Let's say we have a system that stores **Employee IDs and their respective salaries**. This is a classic scenario where a HashMap would be useful.

```
import java.util.HashMap;
import java.util.Map;

public class EmployeeSalaryExample {
    public static void main(String[] args) {
        // Creating a HashMap to store Employee ID and Salary
        Map<Integer, Double> employeeSalaries = new HashMap<>();

        // Adding key-value pairs (Employee ID and Salary)
```

```
        employeeSalaries.put(101, 75000.0);  // Employee ID 101 has a salary of 75,000
        employeeSalaries.put(102, 80000.0);  // Employee ID 102 has a salary of 80,000
        employeeSalaries.put(103, 90000.0);  // Employee ID 103 has a salary of 90,000

        // Getting a salary for a specific Employee ID
        System.out.println("Salary of Employee 101: " + employeeSalaries.get(101));  // Output: 75000.0

        // Checking if an employee exists
        System.out.println("Does Employee 104 exist? " + employeeSalaries.containsKey(104));  // Output: false

        // Removing an employee record
        employeeSalaries.remove(102);  // Removes the entry for Employee ID 102

        // Printing the entire map
        System.out.println("Employee Salaries: " + employeeSalaries);
    }
}
```

How it works internally:

1. **Adding Entries**:
    - For employeeSalaries.put(101, 75000.0), the key 101 is passed through the hashCode() method to generate a hash code.
    - The hash code is then mapped to an index in the bucket array using a formula (e.g., hashCode % capacity).
    - The HashMap checks if there are any existing entries at that index. If not, it places the key-value pair in that bucket. If there is a collision, the new key-value pair is added to the existing linked list at that index.
2. **Getting Values**:
    - When you call employeeSalaries.get(101), the HashMap computes the hash code for the key 101, finds the corresponding index in the bucket array, and looks through the bucket to retrieve the salary (value) associated with Employee ID 101.
3. **Handling Collisions**:
    - If two Employee IDs like 101 and 105 have the same hash code, both key-value pairs will be stored in the same bucket. The HashMap will use a linked list or tree to store these entries. When get(101) is called, the map traverses the bucket to find the correct key-value pair.
4. **Resizing**:
    - If the HashMap grows and reaches 75% of its capacity (i.e., the number of entries is large relative to the bucket array's size), it will resize and rehash all the existing entries, distributing them across a larger array to maintain performance.

Real-Time Analogy:

Think of a **HashMap** as a **filing cabinet**:

- The **filing cabinet** has multiple **drawers** (buckets) where files (key-value pairs) are kept.
- When you put a file with an ID (key), you place it into a drawer based on the file's **hash code**.
- If two files have the same ID (collision), they are placed in the same drawer, either stacked in a **linked list** or kept in **separate folders**.
- If the cabinet (the array of buckets) starts to get too full, it gets **expanded** to accommodate more files, ensuring easy access.

Key Points to Remember:

- **Hashing** is used to determine where each key-value pair goes.
- **Buckets** store key-value pairs, and collisions are handled using a linked list or tree.
- **Resizing** occurs when the load factor exceeds a certain threshold to maintain constant time performance.
- **Load Factor** is the threshold for resizing, which is typically 0.75 by default.

Performance Characteristics:

- **Insertion**: Average time complexity is O(1), but in case of collisions or resizing, it can take O(n).
- **Access**: Average time complexity for get() is O(1), assuming a good distribution of hash codes.
- **Iteration**: Iterating over all entries in the HashMap takes O(n).

17. If two key have same HashCode, How does HashMap store them internally?

If two keys have the same **hash code**, it results in a **hash collision**. In a HashMap, the handling of hash collisions is a critical part of how it stores entries efficiently. Let's walk through the process step by step to understand **how HashMap deals with hash collisions**.

How HashMap Handles Hash Collisions:

1. **Hash Calculation**:
    - When a key-value pair is inserted, the **hash code** of the key is computed using the hashCode() method.
    - This hash code is then mapped to an **index** in the **backing array** (the bucket array), which is essentially an array of **buckets** (linked lists or trees). The index is computed using a formula like index = (hashCode & (array.length - 1)), which helps in distributing the hash values uniformly across the array.
2. **Collision Occurrence**:
    - If two or more keys produce the **same hash code** (i.e., they map to the same index in the bucket array), a **collision** occurs.
    - **Collision Handling**: In earlier versions of Java (prior to Java 8), HashMap used a **linked list** to handle collisions. In later versions of Java (Java 8 and beyond), if the number of entries in a bucket becomes too large (typically greater than 8), it may convert the linked list into a **balanced tree** (a **Red-Black Tree**) for improved performance.
3. **Bucket Storage with Collisions**:
    - **Linked List Approach** (Java 7 and earlier): In case of a hash collision, HashMap simply stores the entries in a linked list at that particular bucket index. Each entry in the linked list consists of a **key-value pair** and a reference to the next entry in the list.
    - **Tree-based Approach** (Java 8 and later): When the bucket size exceeds a threshold (usually 8), the HashMap switches from using a linked list to a **balanced tree structure** (Red-Black Tree). This helps improve the performance of the get() and put() operations from **O(n)** in a linked list to **O(log n)** in a tree-based structure.
4. **Lookup and Resolution**:
    - **After Collision**: When searching for a value using get(key), the HashMap first calculates the hash code of the key and determines the bucket index.
    - If there is only one entry in the bucket (no collision), it directly returns the associated value.
    - If there are multiple entries (due to a collision), the HashMap checks the key in each entry, one by one (in the case of a linked list) or in a more efficient manner if a Red-Black Tree is used, until it finds the matching key.

Example of HashMap with Collisions:

Let's consider the following example:

```
import java.util.HashMap;

public class HashMapCollisionExample {
   public static void main(String[] args) {
      // Creating a HashMap
      HashMap<MyKey, String> map = new HashMap<>();

      // Adding two keys with the same hash code
      map.put(new MyKey(1), "Value 1");
      map.put(new MyKey(2), "Value 2");

      // Retrieving values using keys
      System.out.println(map.get(new MyKey(1))); // Output: Value 1
      System.out.println(map.get(new MyKey(2))); // Output: Value 2
   }
}
```

```
class MyKey {
   int id;

   public MyKey(int id) {
      this.id = id;
   }

   @Override
   public int hashCode() {
      return 100;  // Same hash code for both objects
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj) return true;
      if (obj == null || getClass() != obj.getClass()) return false;
      MyKey myKey = (MyKey) obj;
      return id == myKey.id;
   }
}
```

Explanation of the Code:

1. **Key Objects**: The MyKey class has an overridden hashCode() method that always returns 100 for any MyKey object.
   - This causes **hash collisions** because both MyKey(1) and MyKey(2) will hash to the same index (100).
2. **Putting Elements**:
   - When map.put(new MyKey(1), "Value 1") is called, the HashMap computes the hash code (which is 100 for both keys) and places the entry at index 100 in the bucket array.
   - When map.put(new MyKey(2), "Value 2") is called, it also maps to index 100 (the same as MyKey(1)), and hence the entry is added to the **same bucket**.
3. **Collision Resolution**:
   - In **Java 7 and earlier**, the HashMap would store both MyKey(1) and MyKey(2) in the same **linked list** at bucket index 100.
   - In **Java 8 and later**, if the number of entries in the bucket exceeds the threshold (usually 8), the linked list would be replaced by a **Red-Black Tree** for better performance.
4. **Getting Values**:
   - When map.get(new MyKey(1)) is called, the HashMap computes the hash code (100), finds the corresponding bucket at index 100, and then looks through the entries in the linked list (or tree) to find the entry with the key MyKey(1) and returns "Value 1".

HashMap Collision Resolution in Detail:

- **Linked List (Java 7 and earlier)**:
  - When there is a hash collision, the HashMap uses a **linked list** to store all the entries at the same bucket. The linked list is essentially a chain of entries.
  - In the case of a collision, the HashMap checks each element in the linked list by **calling the equals() method** on the key, and once it finds the matching key, it retrieves the associated value.
- **Red-Black Tree (Java 8 and later)**:
  - When the number of elements in a bucket exceeds a certain threshold (typically 8), the HashMap **switches to a Red-Black Tree** (a balanced binary search tree).
  - This change reduces the lookup time for keys in that bucket from **O(n)** (in a linked list) to **O(log n)** (in a tree), improving performance when the number of collisions is high.

Why Handle Collisions Efficiently?

- **Performance**: Collisions are inevitable in any real-world scenario. Efficient collision resolution is crucial to maintain the HashMap's **O(1)** expected time complexity for operations like put(), get(), and remove().
- **Space**: Using a linked list or a tree structure ensures that space is used efficiently without requiring excessive resizing or restructuring.

Key Points to Remember:

- **Hash collision** occurs when two different keys have the same hash code.
- **Linked list** stores colliding elements in earlier versions of Java (< Java 8), and **Red-Black Tree** is used in later versions (Java 8+) for performance optimization.
- **Collision handling** ensures that the HashMap can still maintain efficient **lookup, insertion**, and **deletion** operations even in the case of multiple keys hashing to the same bucket index.

18. How many types of class loaders are there?

In Java, there are **three primary types of class loaders**:

1. Bootstrap Class Loader

- The **Bootstrap Class Loader** is the **parent** of all class loaders in Java. It is responsible for loading core Java classes (the ones in the **rt.jar** file, which contains essential Java libraries like java.lang.*, java.util.*, etc.).
- This loader is implemented in native code and is part of the Java Virtual Machine (JVM). It loads classes from the **JVM's core libraries**.

Example: Classes like String, Integer, Thread, Object, etc., are loaded by the **Bootstrap Class Loader**.

Key points:

- It loads classes from the **JVM's classpath** (usually from the rt.jar file).
- It cannot be overridden or customized by the application.

2. Extension Class Loader (or Platform Class Loader)

- The **Extension Class Loader** is responsible for loading classes from the **JRE's extension directory** (lib/ext or the location specified by the java.ext.dirs system property).
- It loads classes that are part of the Java extension libraries, such as javax.* or other optional libraries bundled with Java.

Example: Classes in the javax package, like javax.servlet.* or classes from other **extensions** or third-party libraries, can be loaded by the **Extension Class Loader**.

Key points:

- It loads classes from the **extension directories** (lib/ext or java.ext.dirs).
- It sits just below the **Bootstrap Class Loader** in the **Class Loader Hierarchy**.

3. System (or Application) Class Loader

- The **System Class Loader** (also called the **Application Class Loader**) is the **default** class loader used by Java applications. It is responsible for loading classes from the **classpath** (directories or JAR files that are part of the application).
- This loader is used to load user-defined classes (application-level classes) that are specified in the classpath when running a Java program.

Example: Any class defined in your Java project or external libraries (for example, your application classes in the src folder or external JARs like commons-lang.jar) will be loaded by the **System Class Loader**.

Key points:

- It loads classes from the **classpath** or **user-defined directories**.
- It can be customized by setting the classpath using the -cp or -classpath option.

Class Loader Hierarchy:

The class loaders work in a **parent-child hierarchy**, with each loader delegating the responsibility of loading classes to its parent (known as **delegation model**). The hierarchy is as follows:

1. **Bootstrap Class Loader** (top-most, native code)
    - Loads core Java classes (e.g., String, Thread).
2. **Extension Class Loader**
    - Loads classes from the JRE's extension directories (lib/ext).
3. **System (Application) Class Loader** (bottom-most)
    - Loads classes from the application's classpath.

The **System Class Loader** is typically used to load **user-defined classes** (i.e., your classes or classes from external libraries), and if it cannot find the class, it will delegate the task to the **Extension Class Loader**. If that fails, the request is passed to the **Bootstrap Class Loader**.

4. Custom Class Loaders

- Java allows you to create your own **custom class loaders** by extending the ClassLoader class. Custom class loaders are useful for loading classes in special ways, such as loading classes from a database or remote locations (for example, web servers, cloud storage, etc.).
- A **custom class loader** can override the findClass() method to specify how classes are loaded, and it can also define its own **parent class loader**.

Example: You might create a custom class loader to load classes from a non-standard location, such as from a database or a remote server.

Key points:

- Custom class loaders are used to implement **advanced loading strategies** (e.g., loading classes over a network or from non-standard sources).
- Custom class loaders can be **dynamic** (load classes on demand).

Summary of Class Loaders:

Class Loader	Description	Loads From	Examples
Bootstrap Class Loader	Loads core Java classes that are essential to JVM execution.	JRE's rt.jar	Classes like java.lang.String, java.util.*, etc.
Extension Class Loader	Loads extension libraries (optional Java APIs).	JRE's lib/ext or java.ext.dirs	javax.servlet.*, third-party libraries bundled with JDK.
System Class Loader	Loads user-defined classes and libraries specified in the classpath.	Classpath (user-defined directories, JARs)	Application classes, libraries like commons-lang.jar, etc.
Custom Class Loader	User-defined class loaders for special purposes.	Custom sources (e.g., databases, remote locations)	Used for special use cases, like loading classes from the web or database.

19. Optional of methods in java.

In Java, **Optional** is a container object introduced in **Java 8** in the **java.util** package that may or may not contain a value. It was introduced to avoid **NullPointerException** (NPE) by providing a better way to handle **null values**.

Purpose of Optional in Java

Optional is used to represent **optional values** or **nullable values** without using null. This can help avoid situations where NullPointerException might occur due to accessing methods or properties on a null object.

Instead of directly returning null from a method or assigning null to a variable, an Optional is used to represent the absence of a value explicitly.

How Optional Works

An Optional object is used to hold a **value** or a **lack of value** (i.e., null). It provides various utility methods that allow us to check if a value is present, get the value, or provide a default value in case of absence.

Common Methods of Optional

1. **of(T value)**:
   Returns an Optional containing the given non-null value.
2. Optional<String> opt = Optional.of("Hello");
3. **empty()**:
   Returns an empty Optional, which represents the absence of a value.
4. Optional<String> emptyOpt = Optional.empty();
5. **ofNullable(T value)**:
   Returns an Optional containing the value if it is non-null, otherwise an empty Optional.
6. Optional<String> opt = Optional.ofNullable(null); // Empty Optional
7. **isPresent()**:
   Returns true if there is a value present, otherwise false.
8. Optional<String> opt = Optional.of("Hello");
9. boolean isPresent = opt.isPresent(); // Returns true
10. **ifPresent(Consumer<? super T> action)**:
    If the value is present, it executes the provided action (which is a Consumer).
11. opt.ifPresent(val -> System.out.println(val)); // Will print "Hello" if value is present.
12. **get()**:
    Returns the value if present, otherwise throws NoSuchElementException.
13. String value = opt.get(); // Returns the value, will throw an exception if empty.
14. **orElse(T other)**:
    Returns the value if present; otherwise, returns the provided default value.
15. String result = opt.orElse("Default Value"); // Returns "Hello" if present, else "Default Value".
16. **orElseGet(Supplier<? extends T> other)**:
    Similar to orElse, but it takes a Supplier for lazy evaluation of the default value.
17. String result = opt.orElseGet(() -> "Generated Default Value"); // Lazy evaluation of default value.
18. **orElseThrow(Supplier<? extends X> exceptionSupplier)**:
    Returns the value if present, or throws an exception generated by the provided supplier.
19. String result = opt.orElseThrow(() -> new IllegalArgumentException("Value is missing"));
20. **map(Function<? super T, ? extends U> mapper)**:
    If a value is present, it applies the provided mapper function to the value and returns an Optional of the result. Otherwise, it returns an empty Optional.
21. Optional<String> transformed = opt.map(String::toUpperCase); // Transforms the value to uppercase if present.
22. **flatMap(Function<? super T, Optional<U>> mapper)**:
    Similar to map, but the function must return an Optional rather than a direct value. Useful when you want to chain operations that return Optional values.
23. Optional<String> result = opt.flatMap(s -> Optional.of(s.toUpperCase()));
24. **filter(Predicate<? super T> predicate)**:
    If the value is present and the predicate test passes, it returns an Optional containing the value; otherwise, it returns an empty Optional.
25. Optional<String> filtered = opt.filter(val -> val.length() > 3); // Filters based on a condition.

Real-Time Example:

Let's say we are writing a method that searches for a **user** in a database by userId. If the user is found, the method should return the User object, but if not, it should return an empty result.

Without using Optional, we would have to return null, and the caller would need to handle the possibility of null.

```
public User findUserById(String userId) {
   // This might return null if the user is not found
   return database.find(userId);
}
```

With Optional, we can handle the **absence of a value** more gracefully:

```
public Optional<User> findUserById(String userId) {
   // Returns Optional containing the User object if found, otherwise an empty Optional
   return Optional.ofNullable(database.find(userId));
}
```

Now, when calling the method:

```
Optional<User> userOpt = findUserById("12345");
userOpt.ifPresent(user -> System.out.println("User found: " + user.getName())); // If user is present, prints their name
```

In case the user is not found:

```
User defaultUser = userOpt.orElse(new User("default", "Default User")); // Returns default user if not found
```

Why Use Optional?

1. **Avoid NullPointerException**: Optional explicitly represents the absence of a value and forces developers to deal with the potential absence, reducing the risk of NullPointerException.
2. **Readable and Expressive Code**: Methods returning Optional can express that the return value might be absent, improving code readability and understanding.
3. **Composability**: Optional supports functional operations like map, flatMap, and filter, which enable more functional programming styles in Java.

Best Practices with Optional:

- **Return Optional from methods**: If your method can return null, consider returning Optional to indicate that the result might be absent.
- **Avoid using Optional for fields or method parameters**: Optional is meant to represent optional return values, not for general-purpose object fields or method arguments.
- **Avoid using get() unless absolutely necessary**: Use methods like orElse, orElseGet, or ifPresent instead of calling get() directly to avoid NoSuchElementException.

Summary of Common Optional Methods:

Method	Description
of(value)	Returns an Optional containing the value (throws NPE if value is null).
ofNullable(value)	Returns an Optional containing the value or an empty Optional if value is null.
empty()	Returns an empty Optional.
isPresent()	Returns true if the value is present.

Method	Description
ifPresent(Consumer)	Executes the provided action if the value is present.
get()	Returns the value if present (throws exception if absent).
orElse(defaultValue)	Returns the value if present, or the provided default value.
orElseGet(Supplier)	Returns the value if present, or evaluates the provided Supplier.
map(Function)	Applies a function to the value if present and returns an Optional of the result.
flatMap(Function)	Similar to map, but the function must return an Optional.
filter(Predicate)	Returns an empty Optional if the value doesn't match the predicate.

20. What is lazy loading in stream?

Lazy loading in the context of **Streams** in Java refers to the concept where the elements in the stream are processed only when a **terminal operation** is invoked. In other words, intermediate operations on the stream (such as map, filter, sorted, etc.) are not executed immediately when they are applied, but instead, they are **lazy** — they only get executed when a terminal operation (like collect, forEach, reduce, etc.) is called on the stream.

This lazy behavior allows for more **efficient processing** of streams, as the operations are applied **on-demand** rather than eagerly processing all the elements at once.

How Lazy Loading Works in Streams

- **Intermediate operations** in the stream (e.g., filter, map, distinct, etc.) return a new stream and do **not** immediately process the data. They are simply **staged** and **set up** for execution when the terminal operation is triggered.
- **Terminal operations** (e.g., collect, forEach, reduce, count, etc.) **trigger** the processing of the stream, and the elements are processed only at that time.
- **Short-circuiting operations** such as findFirst, anyMatch, and limit are examples of lazy-loaded operations that can stop the processing early, based on conditions met during the stream traversal.

Example of Lazy Loading in Streams:

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

public class LazyLoadingExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

      // Creating a stream pipeline with intermediate operations
      numbers.stream()
            .filter(n -> {
               System.out.println("Filtering: " + n);
               return n % 2 == 0;
            })
            .map(n -> {
               System.out.println("Mapping: " + n);
               return n * 2;
            })
            .forEach(System.out::println);  // Terminal operation
   }
}
```

Output:

```
Filtering: 1
Filtering: 2
Mapping: 2
4
Filtering: 3
Filtering: 4
Mapping: 4
8
Filtering: 5
Filtering: 6
Mapping: 6
12
Filtering: 7
Filtering: 8
Mapping: 8
16
Filtering: 9
Filtering: 10
Mapping: 10
20
```

Explanation of Output:

- As shown in the example, the filter and map operations are **lazy**. They only process the stream when the terminal operation forEach is called.
- The **filter** operation is applied on each element as it is streamed, but **only the elements that pass the filter condition** are further processed.
- The **map** operation is also applied lazily — it only transforms the filtered elements, and its processing is triggered when the terminal operation (forEach) is executed.

Benefits of Lazy Loading in Streams:

1. **Performance Optimization:**
   - Since intermediate operations are **not executed until necessary**, unnecessary processing is avoided.
   - For example, if a findFirst operation is used in the stream pipeline, once the first matching element is found, the stream will stop processing further elements, improving performance.
2. **Short-Circuiting:**
   - **Lazy loading** allows short-circuiting, meaning that operations like findFirst, anyMatch, and limit will stop processing as soon as the result is found.
   - This is useful in cases where we don't need to process the entire stream (e.g., finding the first matching element).
3. **Better Memory Efficiency:**
   - Streams can process data in a **pipelined** manner without needing to load all elements into memory at once. This is particularly beneficial for large datasets or infinite streams.
   - The elements are processed one at a time as needed, reducing memory overhead.

Example of Lazy Short-Circuiting:

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

public class LazyShortCircuitExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

      // Using a short-circuiting operation `findFirst`
      numbers.stream()
           .filter(n -> n % 2 == 0)
           .findFirst()
```

```
            .ifPresent(System.out::println);
    }
}
```

Output:

2

Explanation:

- The findFirst operation is a **short-circuiting** terminal operation, meaning it will stop processing the stream as soon as the first element that satisfies the condition (n % 2 == 0) is found.
- The stream processing stops after finding the first even number, which is 2, and no further elements are processed.

Key Points About Lazy Loading in Java Streams:

- **Intermediate operations** (like filter, map, etc.) are **lazy** and are executed only when a terminal operation is invoked.
- **Terminal operations** (like collect, forEach, reduce) **trigger** the processing of the stream.
- **Short-circuiting operations** can stop the stream pipeline early if certain conditions are met.
- Lazy loading allows for more **efficient** and **flexible** stream processing, as it avoids unnecessary computations and allows for early termination of operations.

21. Explain terminal operations.

Terminal operations in Java Streams are operations that trigger the processing of the stream and produce a **result** or a **side-effect**. Unlike intermediate operations (such as filter, map, sorted, etc.), which are **lazy** and only set up a stream pipeline, terminal operations are **eager** and actually initiate the stream's computation.

When a terminal operation is invoked on a stream, the stream is **consumed**, and no further operations can be performed on that stream. Terminal operations typically return a result or modify the state in some way (like printing data or collecting it into a collection).

Characteristics of Terminal Operations:

1. **Consumes the Stream**: Once a terminal operation is invoked, the stream is considered **consumed** and cannot be used again.
2. **Triggers Processing**: It starts the actual processing of all intermediate operations in the stream pipeline.
3. **Results in a Final Output**: Terminal operations return a specific result (such as a value or a collection) or perform an action.

Types of Terminal Operations:

There are several types of terminal operations in Java Streams, categorized based on their purpose:

1. Reduction Operations

Reduction operations are used to **combine** elements of the stream into a single result.

- **reduce()**: Performs a reduction on the elements of the stream using an associative accumulation function and returns an Optional.
- Optional<Integer> sum = numbers.stream()
- .reduce((a, b) -> a + b); // Sum of all numbers
- **count()**: Returns the number of elements in the stream.
- long count = numbers.stream().count(); // Count the number of elements
- **min() / max()**: Returns the minimum or maximum element according to a comparator.
- Optional<Integer> min = numbers.stream().min(Integer::compareTo);

2. Collection Operations

Collection operations accumulate the stream's elements into a collection (like List, Set, etc.) or another type.

- **collect()**: A very versatile terminal operation that transforms the elements of the stream into another form, typically a collection.
- List<Integer> list = numbers.stream().collect(Collectors.toList()); // Collect into a List

  You can also use collectors to perform other tasks like grouping, partitioning, or collecting statistics:

  Map<Integer, List<Integer>> grouped = numbers.stream()
  .collect(Collectors.groupingBy(n -> n % 2));

3. For-Each Operations

These operations are used for **side-effects** (such as printing or modifying data).

- **forEach()**: Performs an action on each element of the stream.
- numbers.stream().forEach(System.out::println); // Prints each number

  Note: forEach() is a **terminal operation**, so after it's called, the stream is consumed and can't be used again.

4. Matching Operations

These operations test whether elements in the stream satisfy certain conditions.

- **allMatch()**: Checks if **all** elements of the stream match a given predicate.
- boolean allEven = numbers.stream().allMatch(n -> n % 2 == 0); // Checks if all numbers are even
- **anyMatch()**: Checks if **any** element of the stream matches a given predicate.
- boolean anyEven = numbers.stream().anyMatch(n -> n % 2 == 0); // Checks if any number is even
- **noneMatch()**: Checks if **none** of the elements match a given predicate.
- boolean noneNegative = numbers.stream().noneMatch(n -> n < 0); // Checks if no number is negative

5. Find Operations

These operations are used to find specific elements based on a given condition.

- **findFirst()**: Returns the **first** element of the stream, or an empty Optional if the stream is empty.
- Optional<Integer> first = numbers.stream().findFirst(); // Finds the first element
- **findAny()**: Returns any element from the stream. It's often used in parallel streams where the order of processing is not guaranteed.
- Optional<Integer> any = numbers.stream().findAny(); // Finds any element (first for sequential stream)

6. Side-Effect Operations

These operations perform a **side-effect** on the elements of the stream (e.g., printing elements or modifying external state).

- **forEachOrdered()**: Similar to forEach, but guarantees the order of processing in **sequential streams**. For parallel streams, it maintains the encounter order.
- numbers.stream().forEachOrdered(System.out::println); // Guarantees order in sequential streams

7. Short-Circuiting Operations

Short-circuiting operations allow the stream to stop processing early when a result is found, making them more efficient.

- **findFirst()**: Finds the **first** element in the stream and stops further processing once it's found.
- Optional<Integer> firstEven = numbers.stream().filter(n -> n % 2 == 0).findFirst(); // Stops after first even number is found

- **anyMatch()**: Checks if **any** element matches the predicate and stops further processing if the condition is met.
- boolean anyEven = numbers.stream().anyMatch(n -> n % 2 == 0); // Stops once any even number is found
- **limit()**: Limits the number of elements to process in the stream.
- numbers.stream().limit(3).forEach(System.out::println); // Process only the first 3 elements

Example Using Different Terminal Operations:

```
import java.util.*;
import java.util.stream.*;

public class TerminalOperationsExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

      // 1. count()
      long count = numbers.stream().count();
      System.out.println("Count: " + count);

      // 2. collect()
      List<Integer> evenNumbers = numbers.stream()
                                  .filter(n -> n % 2 == 0)
                                  .collect(Collectors.toList());
      System.out.println("Even Numbers: " + evenNumbers);

      // 3. forEach()
      numbers.stream().forEach(n -> System.out.print(n + " "));

      // 4. anyMatch()
      boolean hasEven = numbers.stream().anyMatch(n -> n % 2 == 0);
      System.out.println("\nContains even number: " + hasEven);

      // 5. findFirst()
      Optional<Integer> firstEven = numbers.stream().filter(n -> n % 2 == 0).findFirst();
      firstEven.ifPresent(n -> System.out.println("First Even Number: " + n));

      // 6. reduce()
      Optional<Integer> sum = numbers.stream().reduce((a, b) -> a + b);
      sum.ifPresent(s -> System.out.println("Sum: " + s));
   }
}
```

Output:

```
Count: 10
Even Numbers: [2, 4, 6, 8, 10]
1 2 3 4 5 6 7 8 9 10
Contains even number: true
First Even Number: 2
Sum: 55
```

Summary of Terminal Operations:

Operation	Description
collect()	Collects the elements of the stream into a collection or another form.
forEach()	Performs an action for each element in the stream.
count()	Returns the number of elements in the stream.

Operation	Description
reduce()	Performs a reduction on the elements using an associative accumulation function.
min() / max()	Returns the minimum or maximum element according to a comparator.
anyMatch()	Checks if any element matches the given predicate.
allMatch()	Checks if all elements match the given predicate.
noneMatch()	Checks if no element matches the given predicate.
findFirst()	Returns the first element in the stream.
findAny()	Returns any element from the stream (for parallel streams, any element).

Conclusion:

Terminal operations in Java Streams trigger the actual processing of the stream. Once invoked, the stream is consumed, and no further operations can be performed. These operations return a result or produce a side-effect, like collecting data into a collection, printing to the console, or performing reductions on the data.

22. Give a scenario where we can use Flatmap?

The flatMap operation in Java Streams is used when you have a collection of collections (or nested structures) and you want to **flatten** them into a single stream of elements. It is often used to **transform and flatten** data structures, such as lists of lists, or collections of collections, into a single stream that can be further processed.

Scenario Where flatMap Can Be Used:

A real-world example could be processing a list of **students**, where each student has a **list of courses** they are enrolled in, and you want to create a **flat list of all courses** that students are enrolled in, rather than keeping the nested structure.

Example Scenario:

Imagine you have a Student class that contains a list of courses, and you want to extract a list of all courses across all students. In this case, flatMap can be used to **flatten** the list of courses for each student into a single stream.

Code Example:

```
import java.util.*;
import java.util.stream.*;

class Student {
   private String name;
   private List<String> courses;

   public Student(String name, List<String> courses) {
      this.name = name;
      this.courses = courses;
   }

   public List<String> getCourses() {
      return courses;
   }
}

public class FlatMapExample {
```

```
    public static void main(String[] args) {
        // List of students with their courses
        List<Student> students = Arrays.asList(
            new Student("John", Arrays.asList("Math", "Physics", "Chemistry")),
            new Student("Alice", Arrays.asList("Biology", "History")),
            new Student("Bob", Arrays.asList("Math", "Literature"))
        );

        // Using flatMap to flatten the list of courses for each student
        List<String> allCourses = students.stream()
                                  .flatMap(student -> student.getCourses().stream())  // Flatten courses list of each student
                                  .distinct()  // Remove duplicates
                                  .collect(Collectors.toList());

        System.out.println("All Courses: " + allCourses);
    }
}
```

Explanation:

- The flatMap operation is used to convert each Student object (which contains a list of courses) into a stream of courses.
- For each student, student.getCourses().stream() converts the list of courses into a stream of strings (courses).
- The flatMap then **flattens** these streams of courses into a single stream of courses.
- After flattening, we can apply other operations like distinct() to remove duplicate courses.

Output:

All Courses: [Math, Physics, Chemistry, Biology, History, Literature]

Why flatMap is Useful Here:

- Without flatMap, you'd end up with a stream of lists (Stream<List<String>>), which isn't very useful for further operations (like filtering or counting). flatMap **flattens** the structure, allowing you to treat the collection of lists as a single, continuous stream of elements.
- This is very helpful when working with data that involves **nested collections**, and you want to manipulate the elements at a lower level.

Use Case Scenarios for flatMap:

- **Handling collections of collections**: Flattening lists of lists, or arrays of arrays into a single stream.
- **Transforming objects that contain collections**: When objects contain collections or nested data structures (like List<List<T>>), and you want to extract or process all the elements within those collections.
- **Combining multiple streams into a single stream**: When you have multiple streams of data that need to be combined into one.

Real-World Example in a Business Scenario:

Imagine you are working with **e-commerce orders**. Each order can have a list of products. If you want to generate a flat list of all products across all orders (e.g., to analyze product popularity), flatMap can be used:

```
class Order {
    private List<Product> products;

    public Order(List<Product> products) {
        this.products = products;
    }

    public List<Product> getProducts() {
        return products;
```

```
    }
}

class Product {
    private String name;

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

    public String getName() {
        return name;
    }
}

public class ECommerceExample {
    public static void main(String[] args) {
        List<Order> orders = Arrays.asList(
            new Order(Arrays.asList(new Product("Laptop"), new Product("Mouse"))),
            new Order(Arrays.asList(new Product("Phone"), new Product("Headphones"))),
            new Order(Arrays.asList(new Product("Tablet"), new Product("Keyboard")))
        );

        // Using flatMap to get a flat list of all products
        List<String> allProductNames = orders.stream()
                                        .flatMap(order -> order.getProducts().stream())  // Flatten products in each order
                                        .map(Product::getName)  // Extract product names
                                        .collect(Collectors.toList());

        System.out.println("All Products: " + allProductNames);
    }
}
```

Output:

All Products: [Laptop, Mouse, Phone, Headphones, Tablet, Keyboard]

Here, flatMap helps in flattening the nested structure of orders and products into a single list of products, allowing you to easily process or analyze them.

23. Explain about singleton and factory design pattern using real time example.

Singleton Design Pattern

The **Singleton design pattern** is a **creational pattern** that ensures a class has only one instance and provides a global point of access to that instance. This pattern is used when exactly one object is needed to coordinate actions across the system.

Real-Time Example of Singleton Design Pattern:

Let's consider a **Logger** class in a system. In most systems, logging should be done through a single, shared instance of a logger to avoid unnecessary resource consumption, like creating multiple file handles or database connections. Therefore, we can use the Singleton pattern to ensure only one instance of the logger is used across the entire application.

Code Example:

```
public class Logger {
    // Step 1: Create a private static variable of the same class
    private static Logger instance;

    // Step 2: Private constructor to prevent instantiation
```

```
    private Logger() {}

    // Step 3: Public method to provide access to the instance
    public static Logger getInstance() {
        // If no instance exists, create one, otherwise return the existing one
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    // Method to log messages
    public void log(String message) {
        System.out.println("Logging: " + message);
    }
}

public class SingletonExample {
    public static void main(String[] args) {
        // Accessing the Logger instance
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        // Both logger1 and logger2 will refer to the same instance
        logger1.log("System started");
        logger2.log("System running");

        // Check if both logger instances are the same
        System.out.println("Are both instances same? " + (logger1 == logger2)); // Output: true
    }
}
```

Explanation:

- **Private static instance**: The instance variable holds the only instance of the Logger class.
- **Private constructor**: The constructor is private to prevent instantiation from outside the class.
- **Public method getInstance()**: This is the method that returns the single instance of the Logger class. If the instance is null, it creates a new one; otherwise, it returns the existing one.

Key Points:

- The Singleton ensures that only one instance of the Logger class exists.
- The Logger instance is created lazily (only when it is first requested).
- logger1 and logger2 will always refer to the same instance, as demonstrated by the == check.

Factory Design Pattern

The **Factory design pattern** is another **creational pattern** that provides a way to create objects without specifying the exact class of object that will be created. It defines an interface for creating an object, but lets subclasses decide which class to instantiate. The Factory pattern is often used when the exact type of object needs to be determined at runtime.

Real-Time Example of Factory Design Pattern:

Imagine a scenario where you're working on a **payment gateway** system that supports multiple payment methods (e.g., credit card, PayPal, bank transfer). Depending on the user's choice, a different payment processor needs to be instantiated. The Factory pattern can help by providing an interface for creating payment processors without specifying the exact class of object to create.

Code Example:

```
// Step 1: Create a PaymentProcessor interface
interface PaymentProcessor {
    void processPayment(double amount);
}

// Step 2: Concrete classes implementing the PaymentProcessor interface
class CreditCardPayment implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing Credit Card payment of $" + amount);
    }
}

class PayPalPayment implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}

class BankTransferPayment implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing Bank Transfer payment of $" + amount);
    }
}

// Step 3: Create a PaymentProcessorFactory that will return the appropriate payment processor
class PaymentProcessorFactory {
    public static PaymentProcessor getPaymentProcessor(String type) {
        if ("credit_card".equalsIgnoreCase(type)) {
            return new CreditCardPayment();
        } else if ("paypal".equalsIgnoreCase(type)) {
            return new PayPalPayment();
        } else if ("bank_transfer".equalsIgnoreCase(type)) {
            return new BankTransferPayment();
        }
        throw new IllegalArgumentException("Invalid payment type");
    }
}

// Step 4: Client code that uses the factory to get the correct payment processor
public class FactoryPatternExample {
    public static void main(String[] args) {
        // Get the payment processor based on user input
        PaymentProcessor processor = PaymentProcessorFactory.getPaymentProcessor("paypal");

        // Use the processor to process a payment
        processor.processPayment(100.50);
    }
}
```

Explanation:

- **PaymentProcessor Interface**: Defines a processPayment method that all concrete payment classes must implement.
- **Concrete Classes**: CreditCardPayment, PayPalPayment, and BankTransferPayment implement the PaymentProcessor interface.
- **Factory Class**: The PaymentProcessorFactory class contains a static method getPaymentProcessor that decides which payment processor to return based on the input type. This allows the client code to remain decoupled from the actual class that will be used.
- **Client Code**: The client simply calls PaymentProcessorFactory.getPaymentProcessor() to get the appropriate payment processor and uses it.

Key Points:

- The client doesn't need to know which class (e.g., CreditCardPayment, PayPalPayment) is being instantiated. It only interacts with the PaymentProcessor interface.
- The factory encapsulates the logic of choosing which payment processor to create, allowing easy extension if new payment methods need to be added later.

Comparison of Singleton and Factory Patterns

Feature	Singleton Pattern	Factory Pattern
Purpose	Ensures only one instance of a class is created	Provides an interface for creating objects, but allows subclasses to decide which class to instantiate
Usage	When you need a single global instance (e.g., Logger, Database Connection)	When the type of object to create is determined at runtime (e.g., Payment Processing)
Instance Management	Guarantees a single instance of the class	Creates different instances based on input or conditions
Example	Logger class with a single instance	PaymentProcessorFactory that creates different payment processors

Conclusion:

- **Singleton Pattern** is useful when you need to ensure that only one instance of a class exists across the application, such as for logging or database connections.
- **Factory Pattern** is ideal when you need to create objects without knowing their exact type at compile-time, and you want to delegate the responsibility of object creation to a separate factory class.

Both patterns help in organizing object creation logic and provide flexibility, but they serve different purposes.

24. Can violating the open-closed principle (OCP) still be benficiak in certain situations?

Yes, while the **Open-Closed Principle (OCP)** is a fundamental design principle in object-oriented programming (OOP) that suggests classes should be **open for extension, but closed for modification**, there can be certain situations where violating it might still be beneficial. Let’s explore these scenarios:

Understanding the Open-Closed Principle (OCP):

The **Open-Closed Principle** states that a class should be:

- **Open for extension**: You should be able to add new behavior to a class.
- **Closed for modification**: You should not have to modify the existing class or code when adding new behavior.

OCP encourages you to write code that is **extensible** without modifying the existing codebase, which leads to fewer chances of introducing bugs and higher maintainability. However, there are situations where violating this principle might be more practical.

Situations Where Violating OCP Might Be Beneficial:

1. Short-Term Projects or Prototyping:

In some cases, especially in **rapid prototyping** or **short-term projects**, adhering to OCP may introduce unnecessary complexity. The design might need to be **iterated frequently**, and applying OCP rigorously could make the code harder to change in a fast-paced environment.

- **Example**: If you're building a quick prototype of a new feature and know that the design will change drastically soon, it may be more efficient to directly modify the existing code, rather than spending time abstracting the system to make it extensible. This could save time in the short run while still achieving the necessary functionality.

  Drawback: This might introduce technical debt, and as the project grows, maintaining and refactoring the code might become challenging.

2. Complexity and Over-Engineering:

Applying OCP could result in **over-engineering** if used unnecessarily. Over-engineering happens when you try to **anticipate future changes** or **design for scalability** in a small, non-complex system. In such cases, introducing **abstraction layers** and extra classes could make the system unnecessarily **complex** and harder to understand or modify.

- **Example**: Consider a system where a simple function does the job, but you try to apply patterns like **Strategy**, **Abstract Factory**, or **Visitor** to make it more extensible, even though no changes are expected in the near future. This could make the code harder to maintain and understand without providing tangible benefits.

  Drawback: It can make code unnecessarily complex, which contradicts the principle of writing clean, understandable code.

3. Performance Considerations:

Sometimes, violating OCP might be necessary for **performance optimization**. If a class or method is highly optimized for specific tasks and introducing flexibility (by making it extensible) through abstraction layers would introduce unnecessary overhead (e.g., **virtual method calls**, **additional indirection**, or **object creation**), it might be better to violate OCP to keep things simple and efficient.

- **Example**: In **high-performance systems** such as gaming engines, network servers, or financial systems, excessive abstraction can lead to performance penalties due to **extra method invocations** or **additional layers of abstraction**. In such cases, it might be better to directly modify the class and make the necessary changes to optimize performance, even if it violates OCP.

  Drawback: This might affect long-term maintainability and scalability, especially if the system becomes more complex over time.

4. Tightly Coupled Systems (Legacy Code):

In the case of **legacy code** that was not designed with OCP in mind, sometimes the cost of refactoring the entire system to make it more extensible could outweigh the benefits. In these cases, it might make more sense to **violate OCP** temporarily, especially if the changes are urgent or necessary for **backward compatibility**.

- **Example**: If you have a legacy codebase that doesn't follow OCP, and you need to add new features quickly, you might end up modifying existing classes instead of extending them. Refactoring everything to adhere to OCP could take a long time and require significant resources. In such cases, a more **pragmatic approach** might be better in the short term.

  Drawback: Long-term maintainability could suffer, and making changes to the legacy code might introduce bugs or regressions.

5. Small Teams or Solo Developers:

In small teams or when working solo, you might prioritize **speed of development** and **simplicity** over long-term extensibility. Trying to enforce OCP might slow you down and add unnecessary complexity, especially when you're the only one working on the project and are aware of all the current requirements and future plans.

- **Example**: If you're building a small web app as a solo developer, it might be more efficient to directly modify the code to add features rather than abstracting everything into extendable classes. If you're confident that the app will not require many changes or won't scale significantly, violating OCP in favor of rapid development might be acceptable.

Drawback: As the project grows or new team members are added, the lack of extensibility could become a problem, leading to refactoring overhead.

Conclusion:

While violating the **Open-Closed Principle (OCP)** is generally discouraged in favor of writing maintainable, extensible code, there are **specific scenarios** where it might make sense to do so:

- Rapid prototyping or short-term projects.
- Avoiding unnecessary complexity or over-engineering in simple systems.
- Performance optimizations where abstraction introduces overhead.
- When working with legacy code or tight deadlines in small teams.
- For quick, iterative changes where the future requirements are uncertain.

However, **violating OCP** should be seen as a **trade-off**—it can provide short-term benefits, but it may lead to **higher maintenance costs** in the future. Therefore, it's essential to evaluate the situation carefully and balance **simplicity**, **maintainability**, and **performance** when deciding whether or not to violate the principle.

25. Why does java separate checked and unchecked exceptions and can we create our own custom checked exception?

Why does Java separate Checked and Unchecked Exceptions?

Java separates exceptions into **checked** and **unchecked** exceptions to provide a clear distinction between **recoverable** and **non-recoverable** errors. This separation helps developers handle different types of errors appropriately and ensures that they deal with potential problems in a predictable and structured way.

1. Checked Exceptions:

- **Definition**: Checked exceptions are exceptions that the Java compiler forces the programmer to handle. These exceptions are typically used for situations where recovery is possible, or you can anticipate the exception and handle it accordingly.
- **Purpose**: The primary goal of checked exceptions is to ensure that the developer is **aware of potential errors** and deals with them explicitly, usually by **catching** the exception or **declaring** it in the method signature using throws.

Examples of Checked Exceptions:

- IOException (e.g., when reading from or writing to files)
- SQLException (e.g., when interacting with a database)
- ClassNotFoundException (e.g., when a class cannot be found at runtime)

Why use Checked Exceptions?

- They force developers to think about how to handle recoverable conditions, like file I/O issues, database connectivity issues, etc.
- Checked exceptions represent situations that are **likely to happen**, so the program can respond to these issues (e.g., retry the operation, provide a fallback mechanism).

2. Unchecked Exceptions:

- **Definition**: Unchecked exceptions (also known as **runtime exceptions**) are exceptions that do not require explicit handling. The compiler doesn't force you to handle or declare these exceptions, which means the programmer is **not obliged** to catch them or declare them in the method signature.
- **Purpose**: Unchecked exceptions typically represent **programming errors** or **unexpected situations** that cannot be easily recovered from. These are usually logical errors, like **null pointer access** or **array index out-of-bounds**.

Examples of Unchecked Exceptions:

- NullPointerException

- ArrayIndexOutOfBoundsException
- IllegalArgumentException

Why use Unchecked Exceptions?

- Unchecked exceptions represent **unexpected conditions** that typically arise due to bugs, like accessing null references or invalid arguments.
- Since these errors often represent **programming mistakes**, there's usually no reasonable way to recover from them. Hence, they are not explicitly handled.

Can We Create Our Own Custom Checked Exception?

Yes, Java allows developers to create **custom checked exceptions**. A custom checked exception is a class that extends Exception (not RuntimeException). Once you create a custom checked exception, you must either catch it using a try-catch block or declare it in the method signature using the throws keyword.

How to Create a Custom Checked Exception:

Here's an example of how you can define and use a custom checked exception in Java:

```
// Step 1: Define the custom checked exception class by extending Exception
class CustomCheckedException extends Exception {
    public CustomCheckedException(String message) {
        super(message);
    }
}

// Step 2: Throw and handle the custom checked exception

public class CustomCheckedExceptionDemo {
    // Step 3: Method that throws the custom checked exception
    public static void riskyMethod() throws CustomCheckedException {
        // Simulating an error
        boolean errorOccurred = true;

        if (errorOccurred) {
            throw new CustomCheckedException("Something went wrong in riskyMethod");
        }
    }

    public static void main(String[] args) {
        try {
            // Call method that might throw the custom checked exception
            riskyMethod();
        } catch (CustomCheckedException e) {
            // Handle the custom checked exception
            System.out.println("Caught the custom checked exception: " + e.getMessage());
        }
    }
}
```

Explanation:

1. **CustomCheckedException**: A custom checked exception class that extends Exception. This makes it a checked exception, meaning the compiler will require you to handle it (either by catching it or declaring it in the method signature).
2. **riskyMethod()**: This method throws the custom checked exception. The throws keyword in the method signature signifies that this method can throw CustomCheckedException.
3. **main()**: The main method handles the exception with a try-catch block. If riskyMethod() throws the exception, the catch block will catch and process it.

Key Points:

- When you create a **custom checked exception**, you extend Exception, not RuntimeException.
- Java requires you to either handle or declare this exception explicitly (using throws).
- Custom checked exceptions are often used when you anticipate errors that can be reasonably recovered from, such as problems with file handling, network communication, etc.

Comparison: Checked vs Unchecked Exceptions

Feature	Checked Exceptions	Unchecked Exceptions
Inheritance	Inherits from Exception	Inherits from RuntimeException
Compiler Requirement	Must be either caught or declared using throws	Not required to be handled or declared
Use Case	Recoverable or anticipated errors (e.g., file I/O)	Programming errors (e.g., null pointer, index out of bounds)
Examples	IOException, SQLException	NullPointerException, IllegalArgumentException

Conclusion:

- Java separates **checked** and **unchecked exceptions** to differentiate between recoverable and non-recoverable errors.
- You can create your own **custom checked exception** by extending the Exception class. However, once you create it, you must either catch or declare it (using the throws keyword) in the methods that throw it.

26. Why doesn't the collection interface extend the Map interface?

The **Collection** interface in Java does **not extend the Map interface** due to fundamental differences in their design and purpose. Let's dive into the reasons:

1. Different Concepts:

- **Collection**: The Collection interface is the root interface in the Java Collections Framework that represents a group of individual elements. It is the parent interface of all **single-object** collections like List, Set, and Queue.

  The key characteristic of Collection is that it is a **group of objects** (typically in the form of **sets** or **lists**) and each element in the collection is treated uniformly without any key-value association.

- **Map**: The Map interface represents a **key-value pair** mapping. It stores data in a way that each element has a **unique key** associated with a corresponding **value**. A Map is not a collection of individual objects, but rather a collection of key-value pairs.
  - Examples: HashMap, TreeMap, LinkedHashMap.

2. Hierarchical Structure and Purpose:

- The Collection interface is the base for **one-to-one mapping** of elements, while the Map interface is designed for **one-to-many** mappings (i.e., each key maps to one or more values). Therefore, **they represent different types of data structures**.
  - **Collection** (like List and Set) represents collections of objects.
  - **Map** represents a collection of **key-value pairs**.

If Map were to extend Collection, it would imply that a **map is just a collection of elements**, but **a map works differently**. A Map doesn't treat elements as individual objects but as key-value pairs, so it doesn't conform to the same structural expectations of a Collection.

3. Logical Separation:

- **Separation of concerns**: The core concept behind Java Collections is **logical separation** between structures that represent just **groups of individual elements** (like List, Set, etc.) and **structures that map keys to values** (like Map).
- **Incompatible Operations**: The operations that apply to Collection are generally about manipulating a single element, such as adding, removing, and querying elements. On the other hand, the operations for Map deal with key-value pairs, like putting a key-value pair, getting a value by key, etc.
    - For instance, Collection operations like add(E) and remove(Object) are simple and only work with the objects themselves. But Map operations like put(K key, V value) and get(Object key) are concerned with handling pairs of objects—one representing the key and the other representing the value.

4. API Design and Usability:

- **Consistency**: In the Java Collections Framework, having Map as a separate interface allows for a more consistent and clear distinction between **collections** and **maps**.
    - Collection focuses on groups of individual elements, while Map is focused on key-value pairs.
- By keeping them separate, Java can ensure that operations designed for a collection (like add(), remove()) don't inadvertently get mixed up with operations designed for maps (like put(), get()).

5. Practical Reasons:

- If Map were to extend Collection, a Map could be treated as a collection of values, which would **break** the logic of handling keys separately from values. A map is fundamentally a different concept from a collection of elements.
    - For example, consider the Map operation put(K, V)—how would this be represented as a collection operation? Should the map's keys be treated as individual elements in the collection? Or should the values be part of the collection?

  This conflict would create confusion in both the API and usage, so separating Map from Collection ensures clearer, more intuitive semantics.

6. Examples of Java Collection and Map Interfaces:

- **Collection Interface**:
    - List (e.g., ArrayList, LinkedList)
    - Set (e.g., HashSet, TreeSet)
    - Queue (e.g., LinkedList, PriorityQueue)
- **Map Interface**:
    - HashMap
    - TreeMap
    - LinkedHashMap
    - Hashtable

Conclusion:

- The **Collection interface** represents a **group of individual elements** (sets, lists, queues), while the **Map interface** represents a **collection of key-value pairs**. Since their purposes, structures, and operations are fundamentally different, it makes sense that Map is not a subclass of Collection.
- By keeping them separate, Java provides a more **logical and clear distinction** between collections and mappings, ensuring that operations designed for each type are distinct and well-defined.

27. Difference between HashMap and TreeMap with order of time complexity of retrieval and search?

Difference Between HashMap and TreeMap

Both HashMap and TreeMap are popular implementations of the Map interface in Java, but they differ significantly in terms of their **underlying data structure**, **ordering of elements**, and **time complexities** for various operations.

Here's a breakdown of their key differences:

1. Underlying Data Structure

- **HashMap**:
  - Uses a **hash table** for storing elements. The keys are hashed using their hashCode() method, and this hash code is used to compute the index of the bucket where the key-value pair is stored.
- **TreeMap**:
  - Uses a **Red-Black Tree** (a type of self-balancing binary search tree) for storing elements. This tree structure ensures that the keys are always stored in a **sorted order** according to their natural ordering or a custom comparator.

2. Ordering of Elements

- **HashMap**:
  - Does **not maintain any order** of its elements. The order of keys and values is unpredictable and can vary between different runs of the program.
  - Elements are arranged based on the hash values of the keys.
- **TreeMap**:
  - Maintains elements in a **sorted order**. By default, it sorts the keys in **ascending order** according to their natural ordering (i.e., the compareTo() method of the key objects).
  - If a custom order is needed, you can pass a Comparator when creating the TreeMap.

3. Performance (Time Complexity)

Insertion and Retrieval:

- **HashMap**:
  - **Insertion**: O(1) on average (constant time). This is because the key is hashed, and the value is inserted directly into the corresponding bucket. However, in the worst case (if many keys collide), it can take O(n) time, where n is the number of elements.
  - **Retrieval**: O(1) on average (constant time). Like insertion, this operation is typically constant time, as it uses the key's hash code to find the appropriate bucket.
- **TreeMap**:
  - **Insertion**: O(log n) where n is the number of elements in the map. In a TreeMap, elements are inserted into the Red-Black Tree, which requires balancing the tree to maintain the sorted order.
  - **Retrieval**: O(log n) because it requires searching the tree, which takes logarithmic time.

Search/Contains Check:

- **HashMap**:
  - **Search** (containsKey()): O(1) on average.
  - **Search** (containsValue()): O(n) because it requires scanning through all the values in the map.
- **TreeMap**:
  - **Search** (containsKey()): O(log n) because it uses the Red-Black Tree to perform a search.
  - **Search** (containsValue()): O(n) because, like HashMap, it has to traverse the entire map to check for the presence of a value.

4. Null Keys and Values

- **HashMap**:
  - Allows **one null key** and **multiple null values**. This is because a null key does not conflict with any valid hash code.
- **TreeMap**:
  - Does **not allow null keys**. This is because the natural ordering or a custom comparator is used for sorting, and null cannot be compared.
  - It **allows null values**, though.

5. Usage Scenarios

- **HashMap**:
  - Use HashMap when you **don't care about the order** of elements and want **fast lookup** and **insertion** operations.

    - Ideal for scenarios where the main concern is efficient access by key.
- **TreeMap**:
    - Use TreeMap when you need the **elements sorted** based on the keys (either natural order or custom comparator).
    - Ideal for scenarios where you need **sorted data** or need to retrieve elements in a range or order, such as getting the first or last entry, or iterating in sorted order.

6. Examples

Example of HashMap:

```
import java.util.HashMap;

public class HashMapExample {
   public static void main(String[] args) {
      HashMap<String, Integer> map = new HashMap<>();
      map.put("Apple", 3);
      map.put("Banana", 1);
      map.put("Cherry", 2);

      System.out.println(map); // Output order is unpredictable
   }
}
```

Example of TreeMap:

```
import java.util.TreeMap;

public class TreeMapExample {
   public static void main(String[] args) {
      TreeMap<String, Integer> map = new TreeMap<>();
      map.put("Apple", 3);
      map.put("Banana", 1);
      map.put("Cherry", 2);

      System.out.println(map); // Output will be in sorted order: {Apple=3, Banana=1, Cherry=2}
   }
}
```

Summary Table:

Feature	HashMap	TreeMap
Data Structure	Hash table	Red-Black Tree
Ordering	No ordering (unordered)	Sorted according to key's natural order or custom comparator
Insertion/Lookup Time	O(1) on average, O(n) in worst case	O(log n)
Allows Null Keys/Values	Allows 1 null key, multiple null values	No null keys, allows null values
Search Time	O(1) for keys, O(n) for values	O(log n)
Common Use Cases	Fast lookups, when order is not important	Sorted data, range queries, ordered iteration

Conclusion:

- **Use HashMap** when you need **fast access** and **don't need the data to be ordered**.
- **Use TreeMap** when you need to maintain **sorted order** of keys or need operations like **range queries** or **sorted iteration**.

28. Can we store as a key in TreeMap?, why or Why not?

No, you **cannot store null as a key in a TreeMap**. The reason lies in the way TreeMap maintains its elements in sorted order using the **natural ordering** of the keys or a **custom comparator**.

Here's why:

1. **Natural Ordering**:
    - TreeMap relies on the compareTo() method (which is part of the Comparable interface) to order its keys. When a key is inserted, TreeMap compares the new key with the existing keys using this method.
    - However, the compareTo() method cannot handle null values because comparing null to any other object would result in a **NullPointerException**.
    - Since null can't be compared, the TreeMap throws an exception when you try to insert a null key.
2. **Custom Comparator**:
    - If you're using a **custom comparator** (via TreeMap(Comparator)), the comparator must also be capable of handling null keys.
    - If the comparator doesn’t explicitly handle null (by, for example, defining a rule for comparing null with other values), trying to insert null as a key will result in an exception (NullPointerException).

Example:

```
import java.util.TreeMap;

public class TreeMapExample {
   public static void main(String[] args) {
      try {
         TreeMap<String, Integer> map = new TreeMap<>();
         map.put(null, 1);  // Throws NullPointerException
      } catch (NullPointerException e) {
         System.out.println("Cannot store null as a key in TreeMap");
      }
   }
}
```

Output:

```
Cannot store null as a key in TreeMap
```

Why does HashMap allow null keys?

In contrast, a HashMap allows **one null key** because it uses a **hash table** internally, and the insertion of null is handled differently. The hashCode() method of a null key doesn't cause a NullPointerException, so it can be safely stored as a key in a HashMap.

Conclusion:

You cannot store null as a key in a TreeMap because the sorting mechanism of TreeMap requires comparing keys, and null cannot be compared to other keys. If you need to store null values, consider using a HashMap instead.