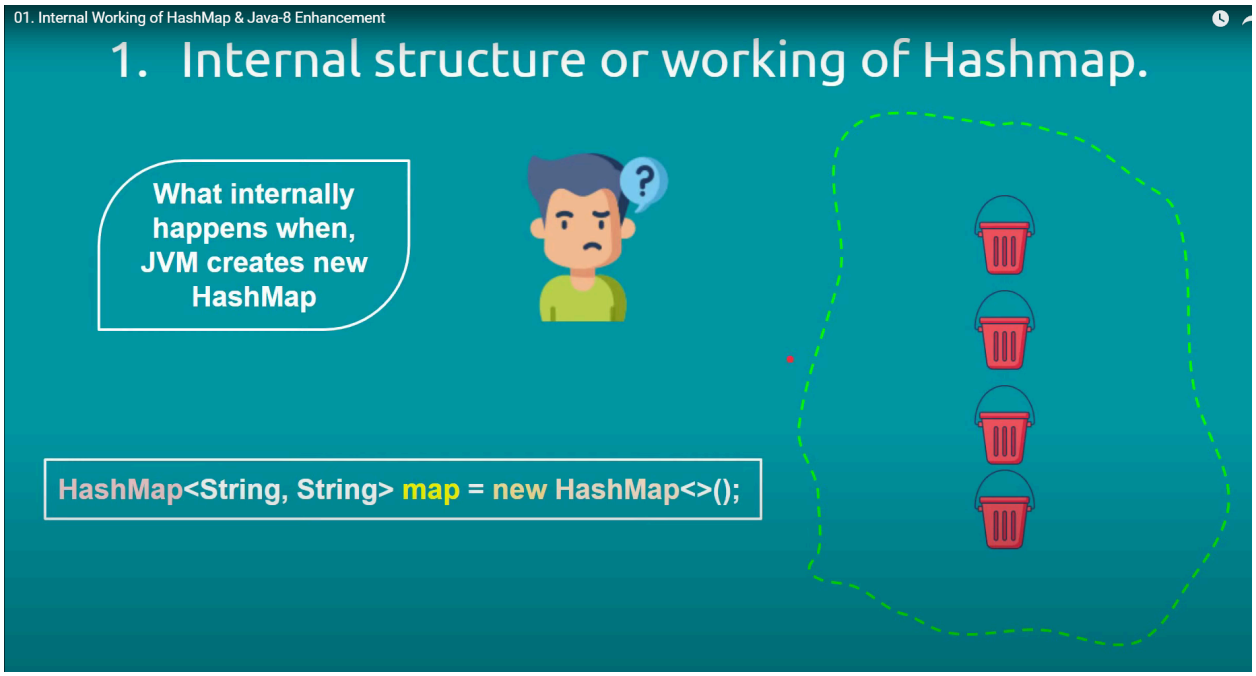
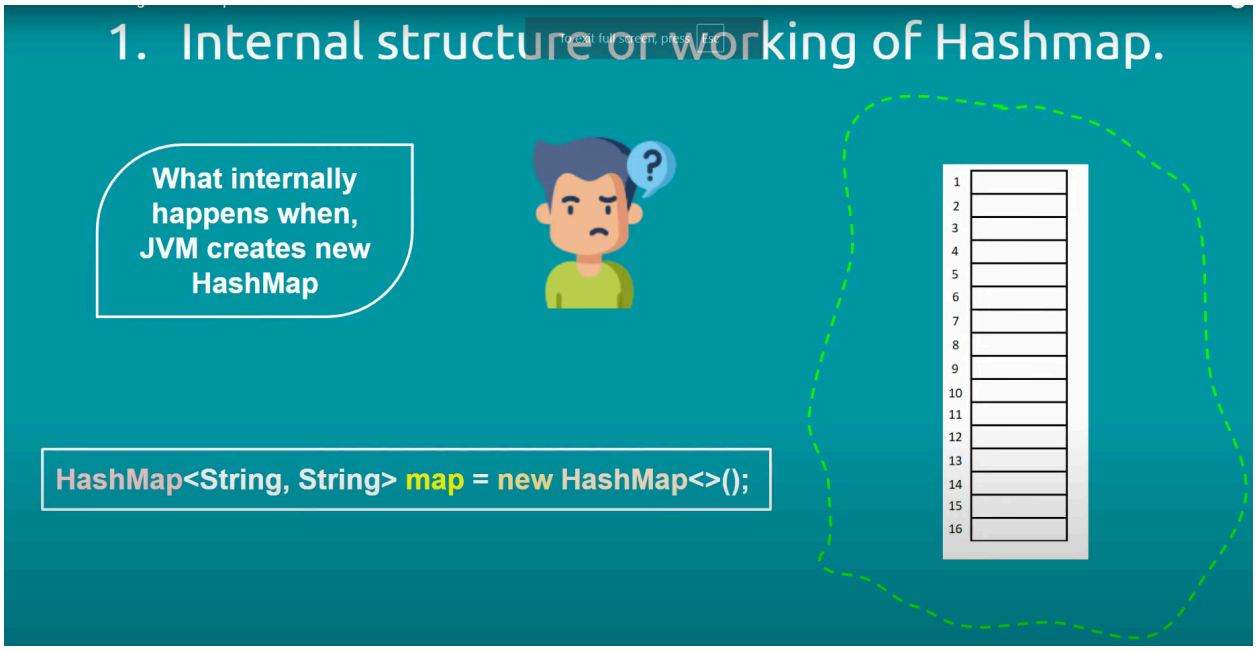
Experienced Interview Question

1) Internal Working of HashMap in Java

Analysis diagram



Actual Diagram



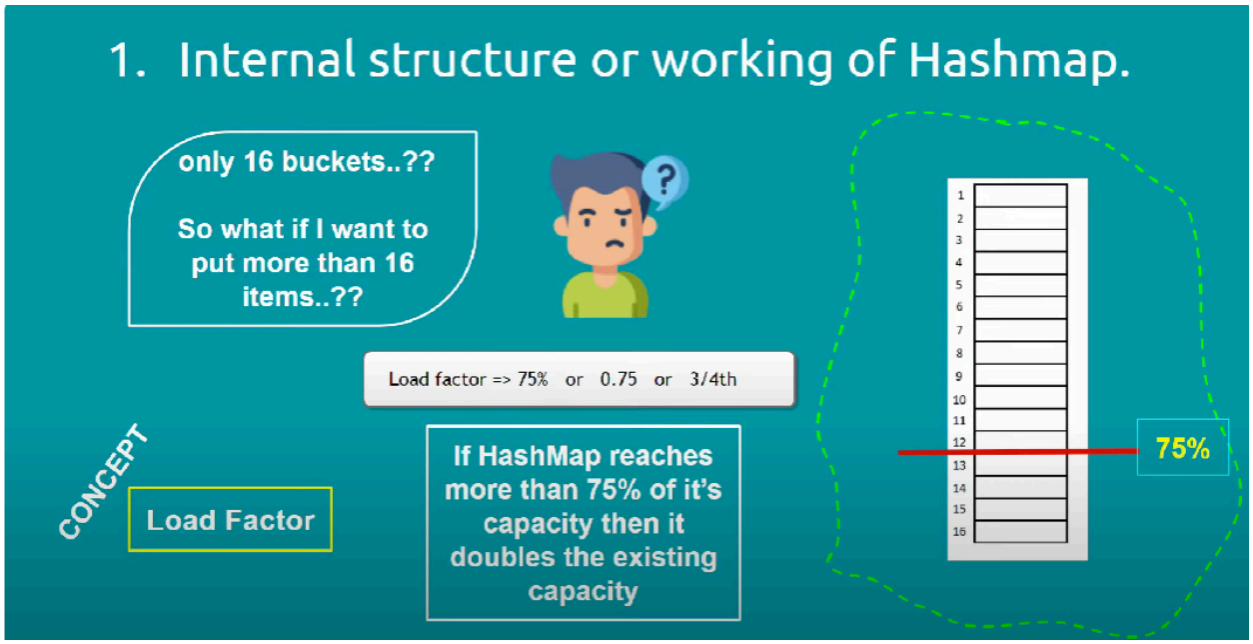
When we run the above line code JVM will create 16 (default initial capacity/size of hashmap) buckets in heap memory.

As we already know The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

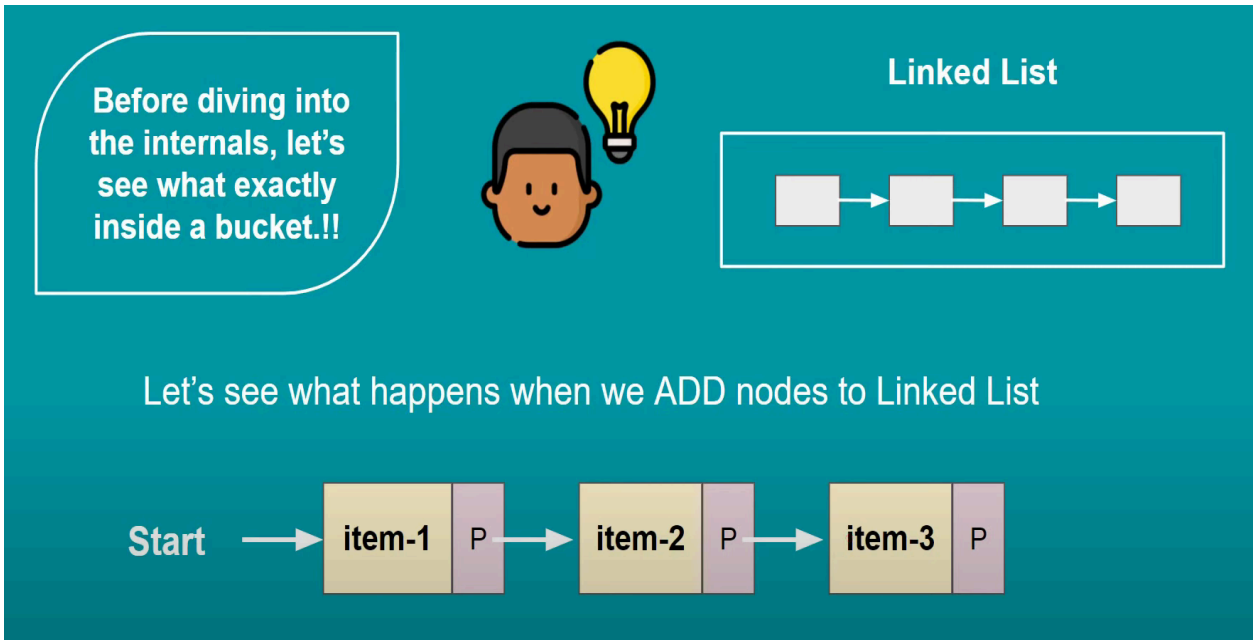
This means **when the number of elements exceeds 16 * 0.75 = 12**, the HashMap will **resize**.
When the number of elements (entries) exceeds the **threshold** (12 in this case), **rehashing** happens.
The capacity is **doubled** (from 16 to 32).
The threshold is also updated: 32 * 0.75 = 24.

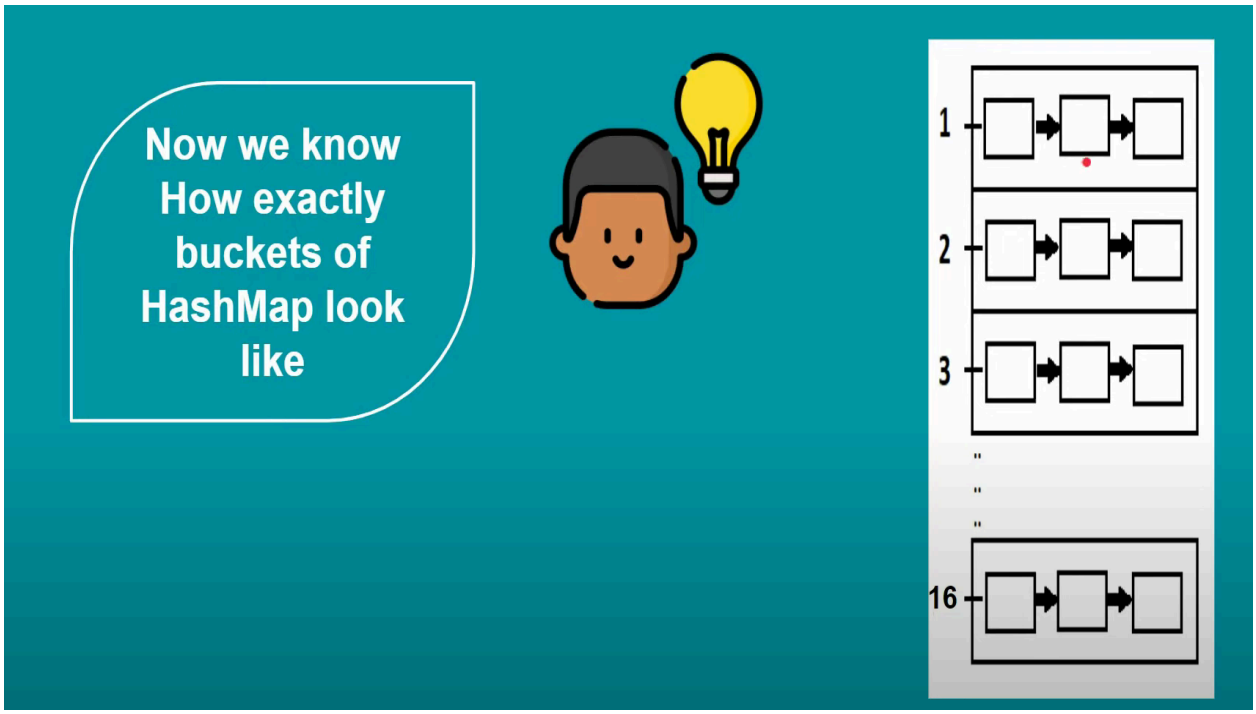
Example Scenario

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();

        // Adding 12 elements (75% of 16)
        for (int i = 1; i <= 12; i++) {
            map.put(i, "Value" + i);
        }

        // Before rehashing
        System.out.println("Size before threshold breach: " + map.size());

        // Adding the 13th element, triggering rehashing
        map.put(13, "Value13");

        // After rehashing
        System.out.println("Size after threshold breach: " + map.size());
    }
}
```



Bucket is nothing but a linked list here.
Before Java 8, **a bucket was essentially a linked list**. But after Java 8, **it can be either a linked list or a Red-Black Tree**, depending on the number of elements in a bucket.

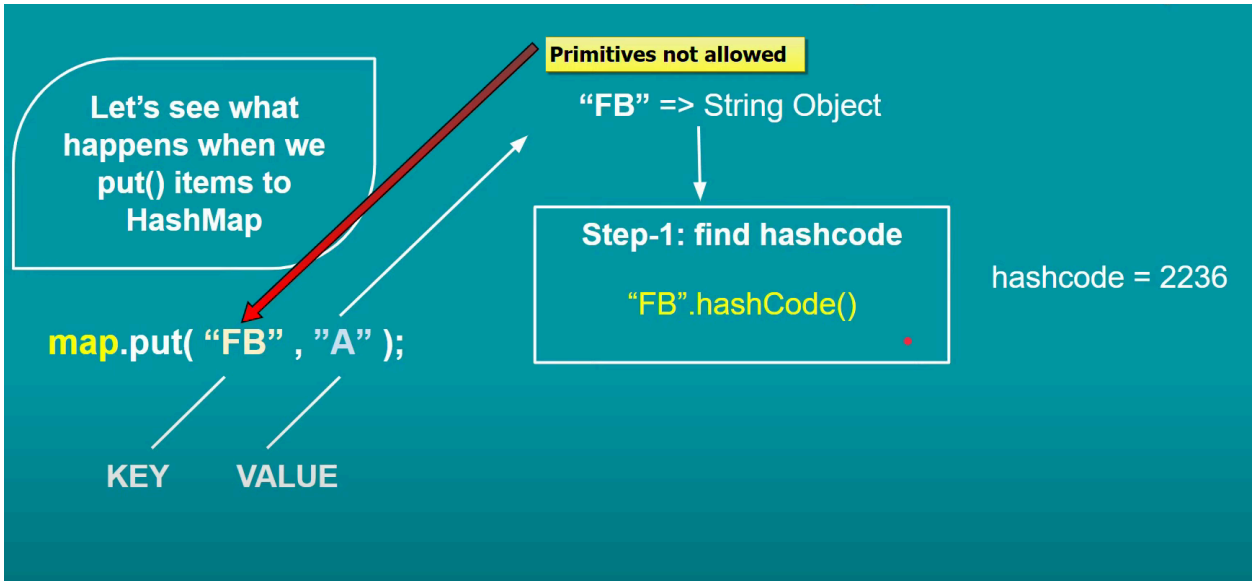


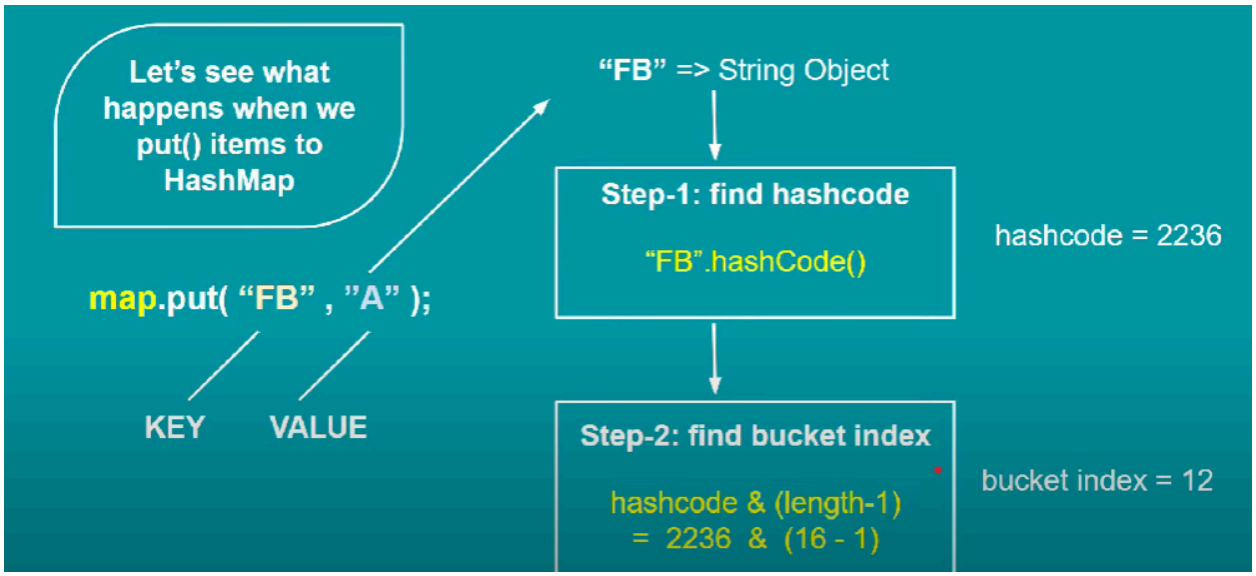


primitive types (e.g., int, double, char, etc.) are **not allowed** as keys in a HashMap. However, their **wrapper classes** (Integer, Double, Character, etc.) **can be used** as keys because they are objects.

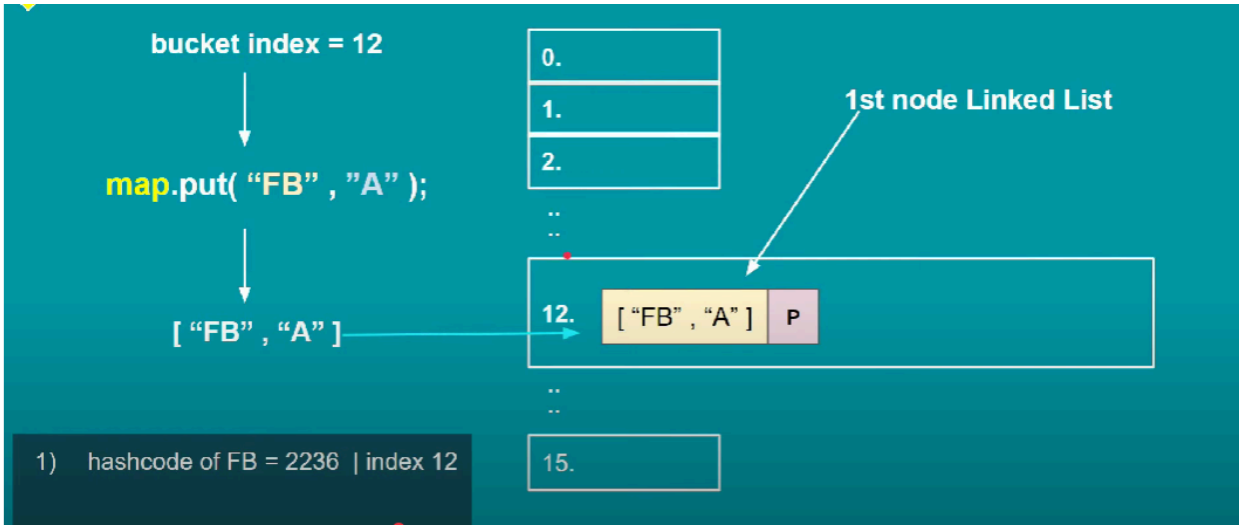
Put Method of HashMap

First item

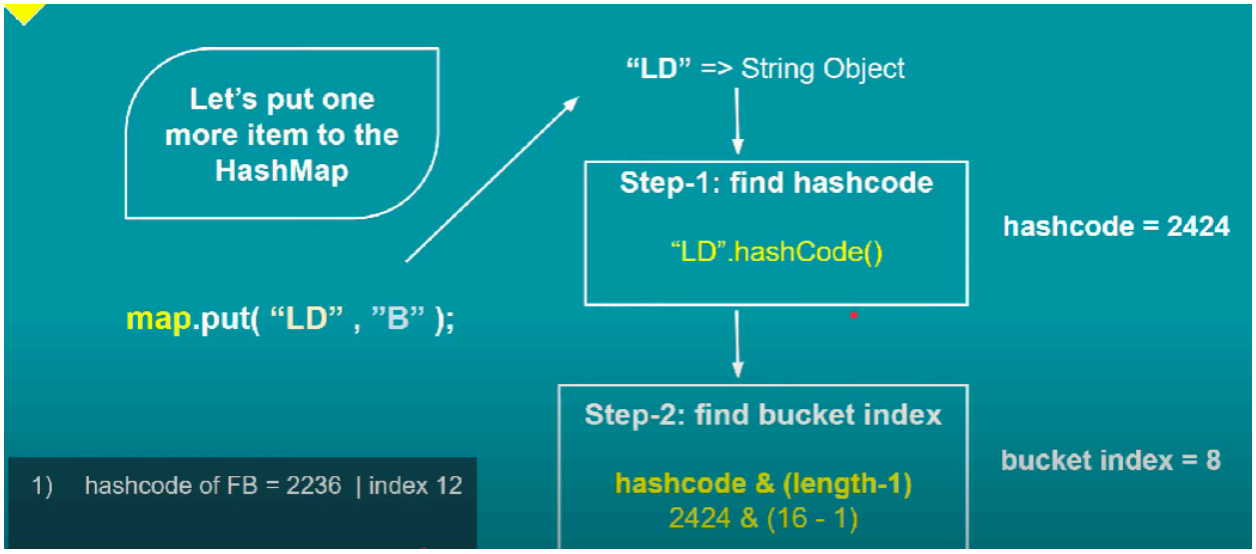

Let's see what happens when we put() items to HashMap
Primitives not allowed
"FB" => String Object
Step-1: find hashcode
"FB".hashCode()
hashcode = 2236
map.put("FB" , "A");
KEY
VALUE

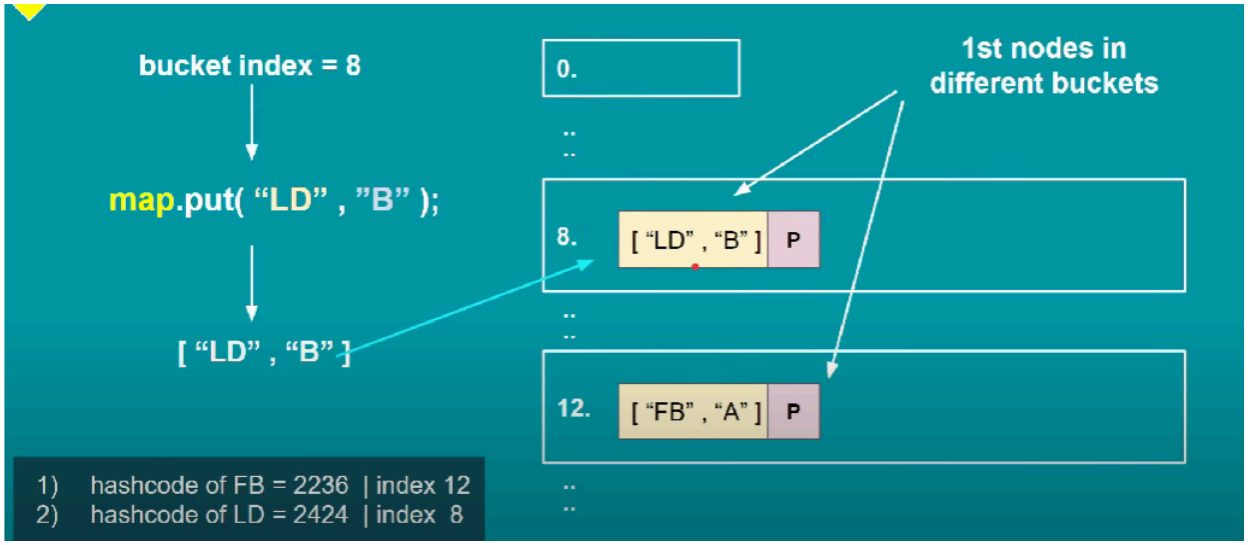

Let's see what happens when we put() items to HashMap
"FB" => String Object
Step-1: find hashcode
"FB".hashCode()
hashcode = 2236
map.put("FB" , "A");
KEY
VALUE
Step-2: find bucket index
hashcode & (length-1)
= 2236 & (16 - 1)
bucket index = 12

So the date first key & value will be stored in bucket index=12

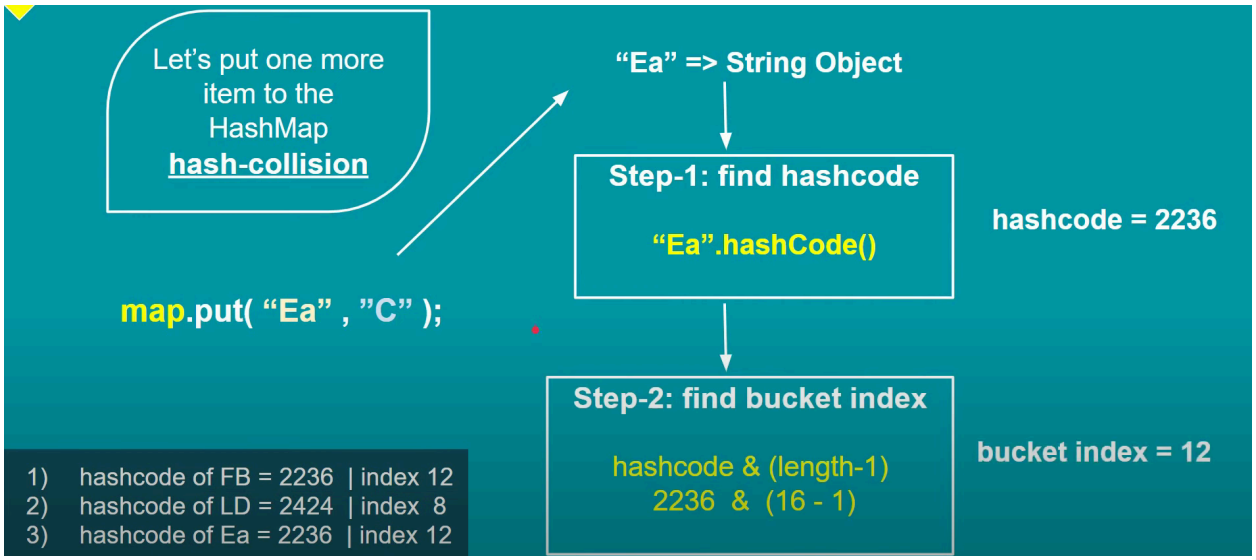


Second item





Third item

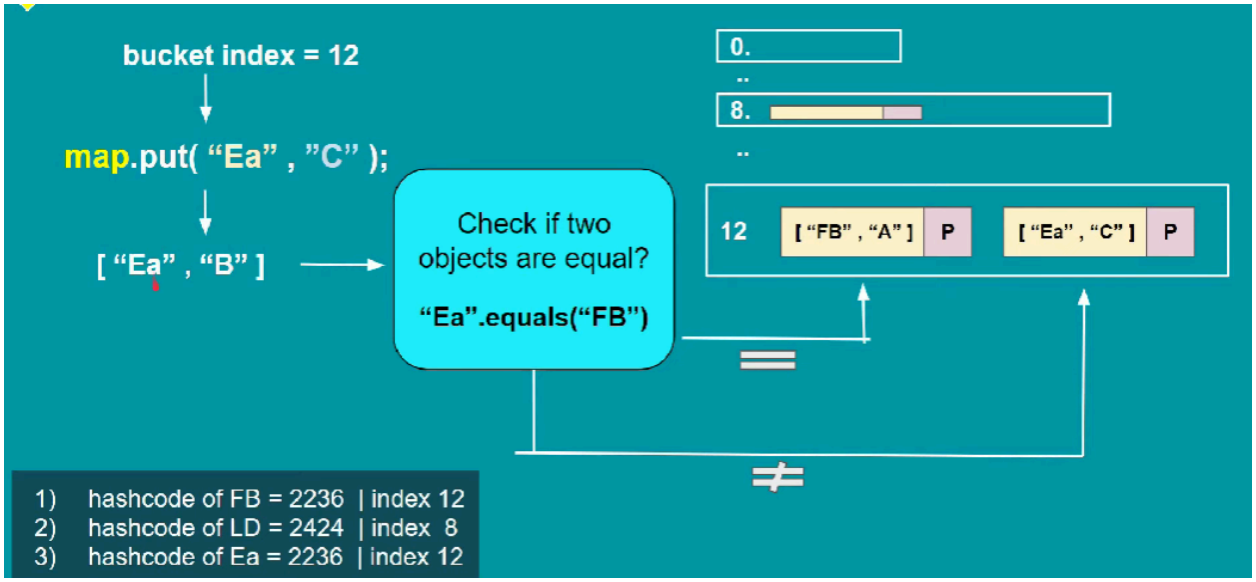


Here is hash collision , as the hashcode of the third(2236) item is the same as the first(2236) item .

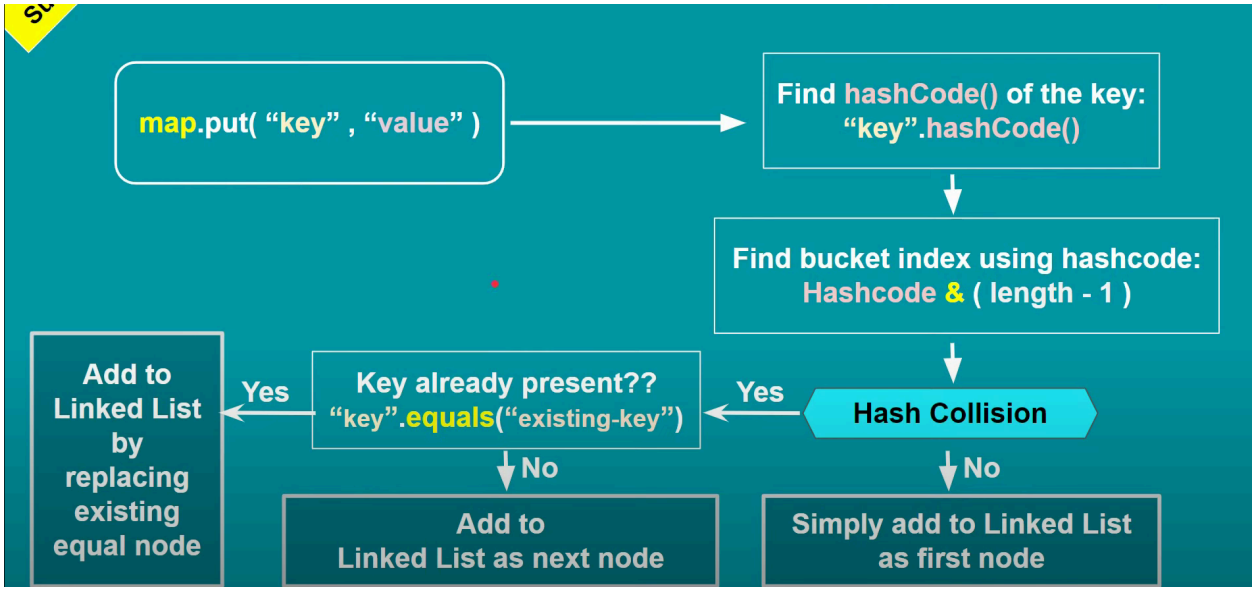
So in this case the item has to store in bucket 12 right

So JVM will check whether the same key is present in the bucket or not , if it is present the existing item details will be replaced , so that’s why hashmap does not support duplicate values.

If the keys are different then items will be stored in a bucket.

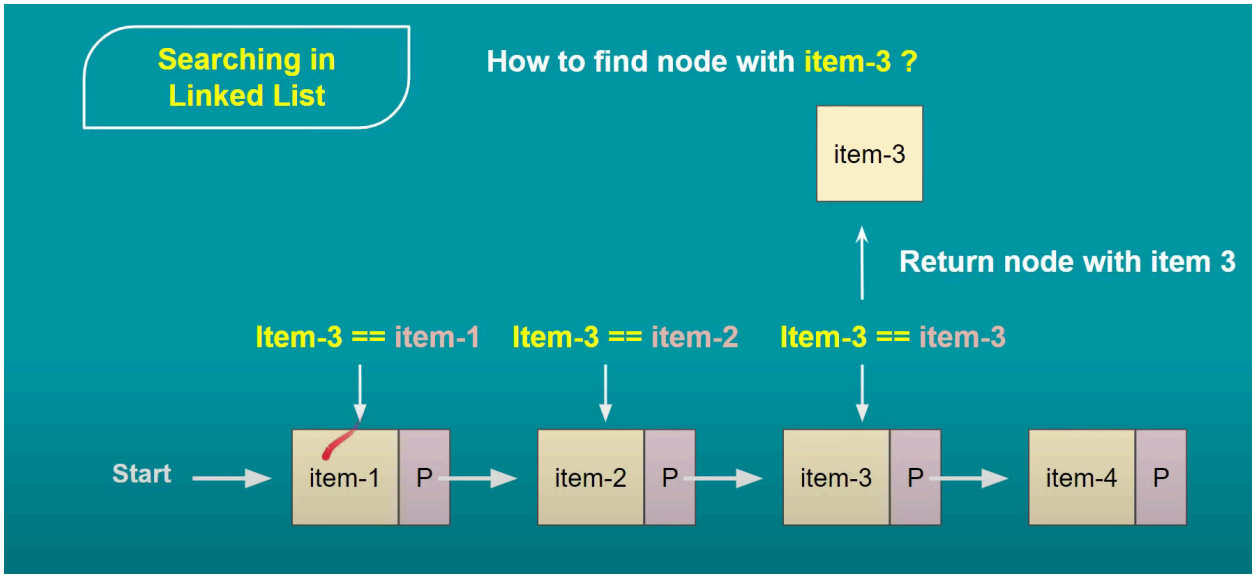


This the overview of **put** method in Hashmap



Get Method of HashMap

For getting particular item it uses searching concept
In a **linked list**, searching for a particular item involves **traversing each node** one by one until the desired value is found (or the end of the list is reached).

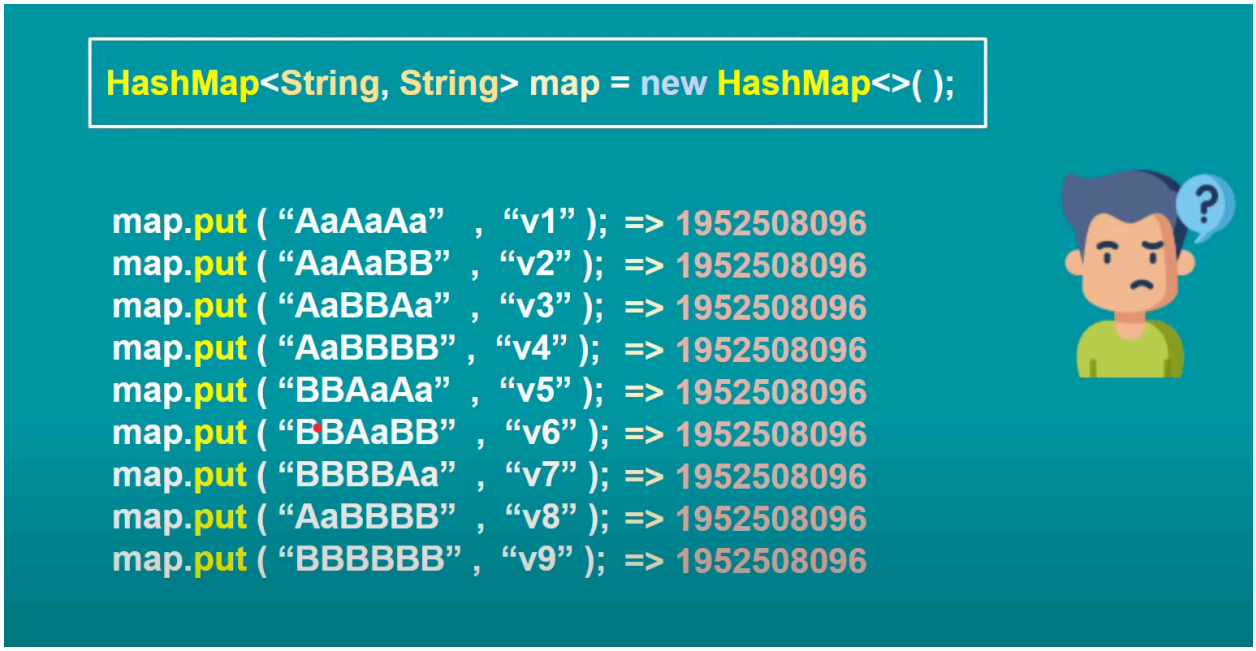


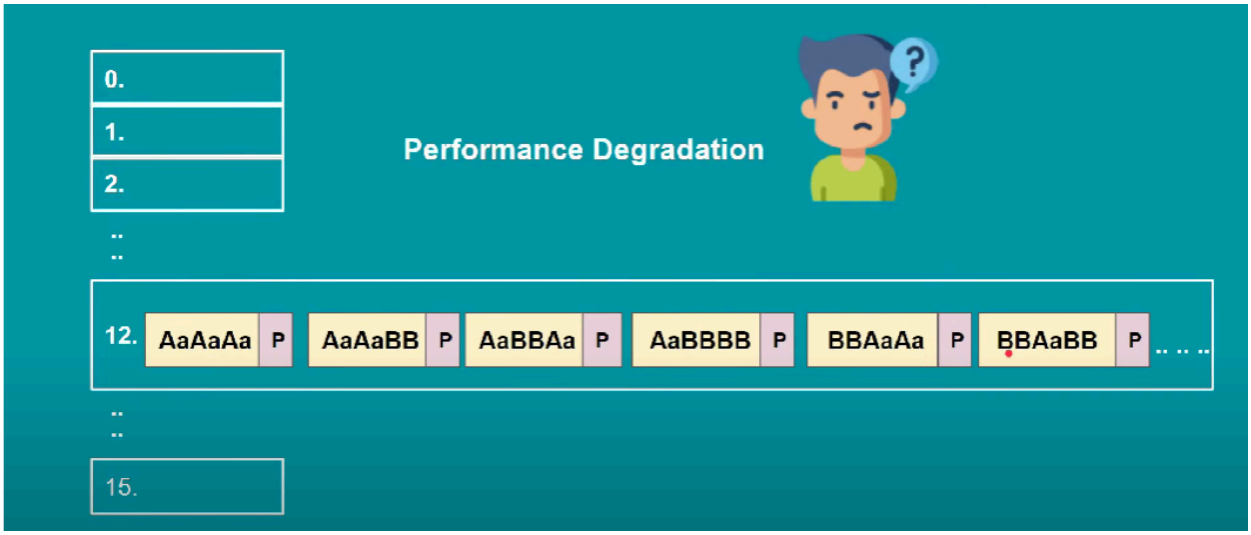
Retrieving an item from a **HashMap** is very fast, if the bucket has more nodes then the time complexity will vary .

Handling Collisions: If multiple elements (key-value pairs) are in the same bucket, HashMap uses:

- **Linked List (before Java 8)** → Search time becomes **O(n)** in the worst case.
- **Balanced Tree (after Java 8, when too many nodes exist in a bucket)** → Search time improves to **O(log n)**.

Ex:
If multiple keys **have the same hashcode**, they will be placed in the **same bucket**, which leads to **hash collisions**. This increases retrieval time during the get() operation because instead of directly accessing the value, Java's **HashMap** has to search through multiple entries within the bucket.



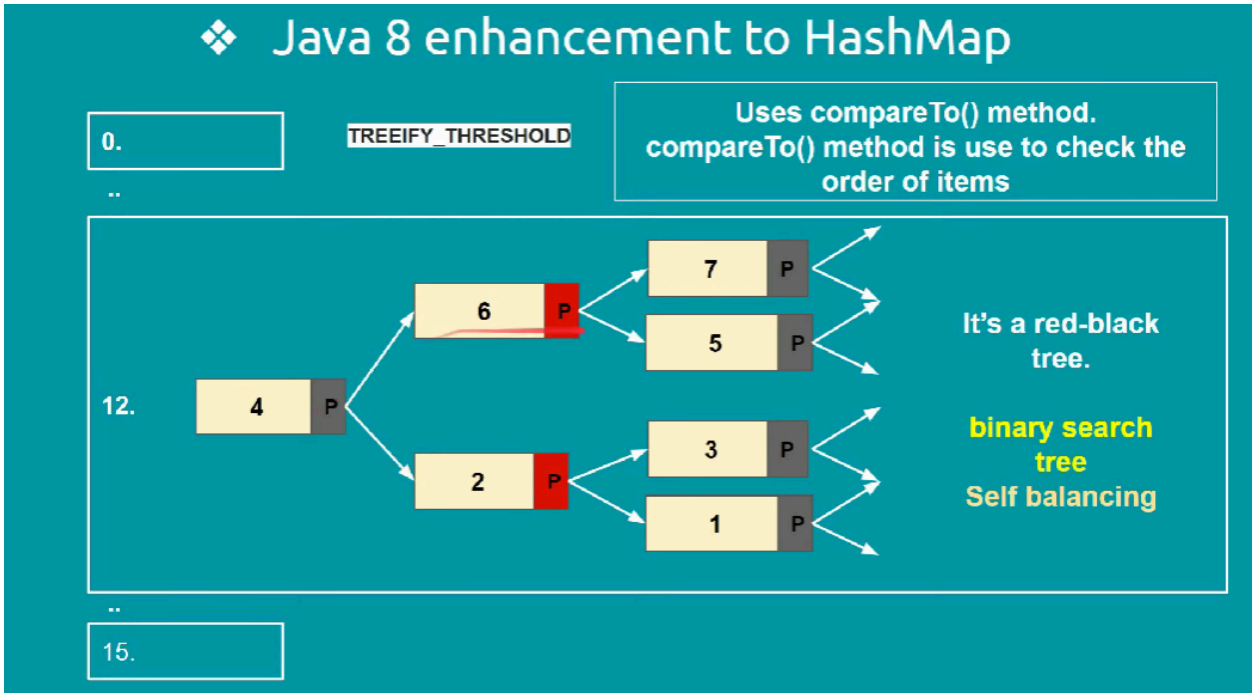


Yes! Because of the **performance degradation** caused by **hash collisions**, developers introduced the **treeify threshold** concept in **Java 8 HashMap** to optimize retrieval time.

What is the Treeify Threshold?

The **treeify threshold** is a predefined limit (TREEIFY_THRESHOLD = 8) in **Java 8 HashMap**. When the number of nodes in a single bucket exceeds this threshold, the **linked list is converted**

into a balanced Red-Black Tree. This improves lookup time from **O(n) (linked list traversal) to O(log n) (tree search).**



Ex:

A tree data structure, specifically a Binary Search Tree (BST) or Red-Black Tree (used in Java 8+ HashMap), stores data in a hierarchical way based on comparisons.

How Data is Stored in a Binary Search Tree (BST)

1. Each node contains a value.
2. The left child holds values smaller than the parent.
3. The right child holds values greater than the parent.
4. This rule applies recursively to all nodes.

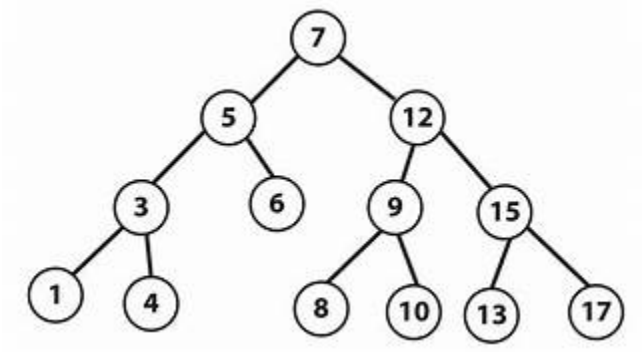
Example: Inserting Numbers into a BST

Let's insert the numbers 50, 30, 70, 20, 40, 60, 80 into a BST.

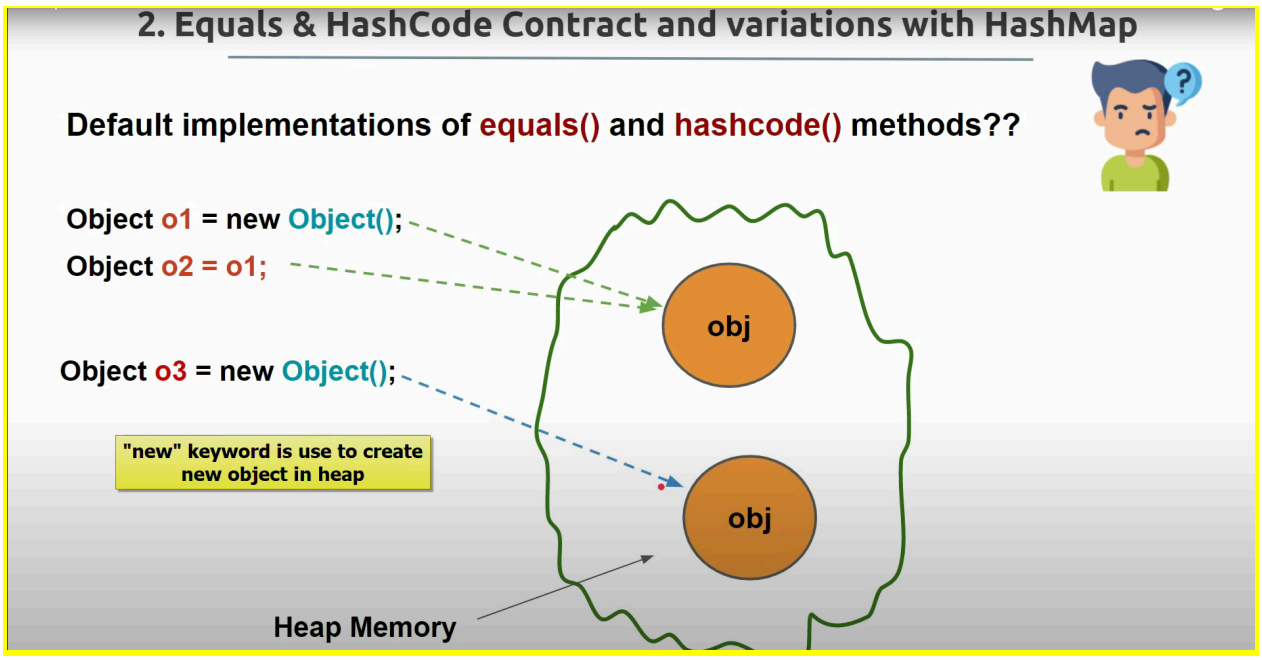
```
      50
     / \
   30   70
  / \   / \
 20  40 60  80
```

- **50 is the root.**
- **30 is less than 50, so it goes to the left.**
- **70 is greater than 50, so it goes to the right.**
- **20 is less than 30, so it goes to the left of 30.**
- **40 is greater than 30, so it goes to the right of 30.**
- **60 is less than 70, so it goes to the left of 70.**
- **80 is greater than 70, so it goes to the right of 70.**

Ex2

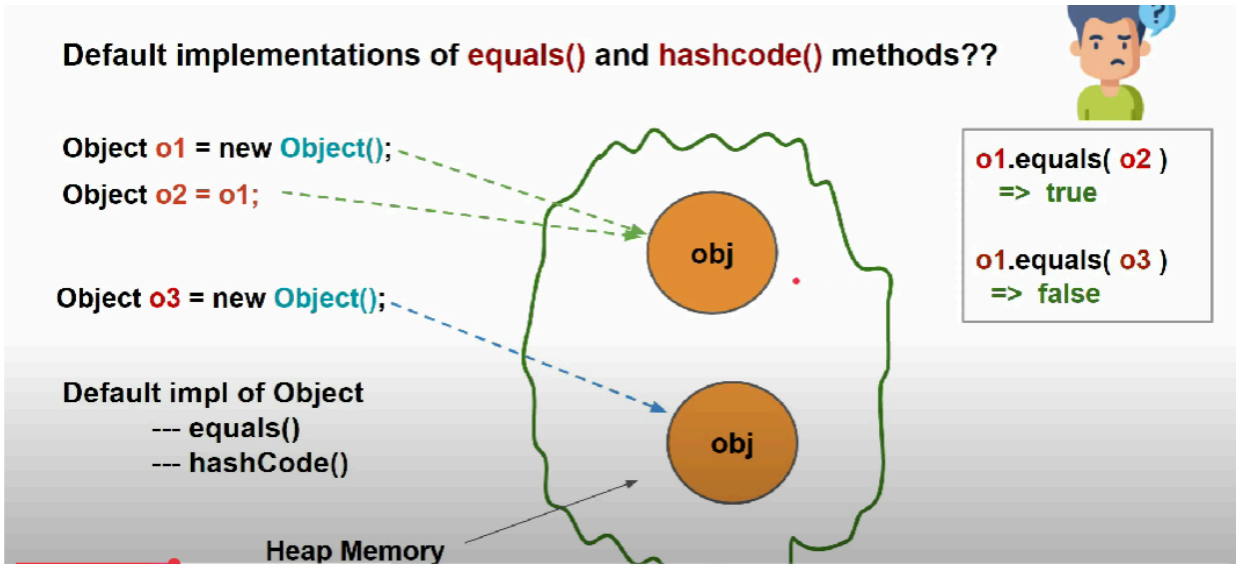


2) What is the contract between equals() and hashCode() methods in Java?



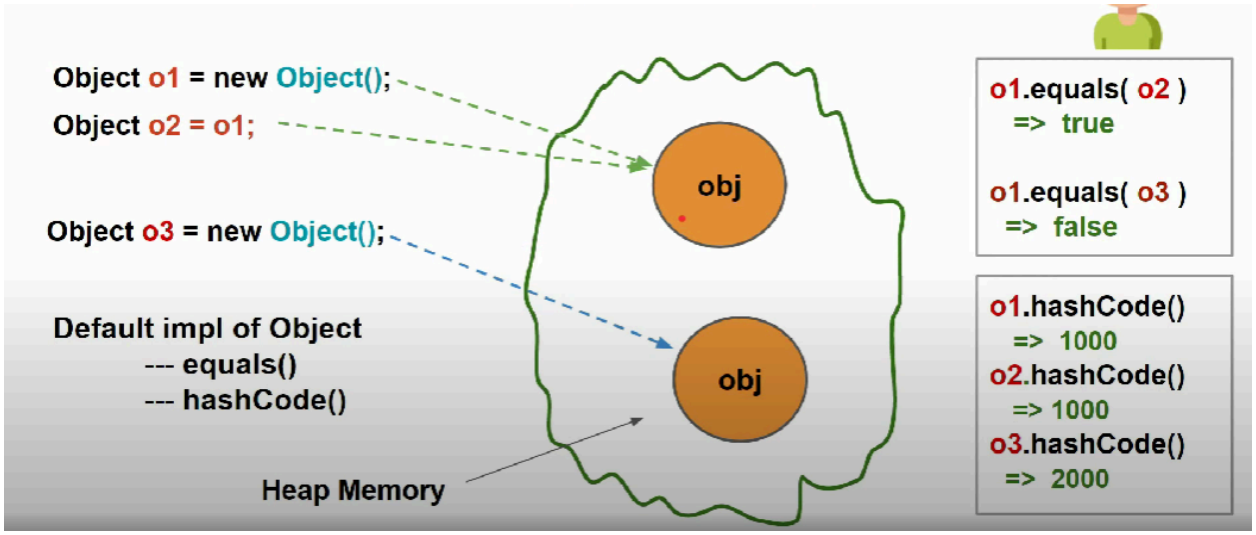
If a create an object using a new keyword an object will be created in heap memory and reference will be pointed. In the second we are just assigning references of o1 to o2 , both will point to the same object.

Now will understand **equals** method using above example



equals method will return true if references will be pointed to the same object .

Based on the memory address of object , JVM will create Hash Codes



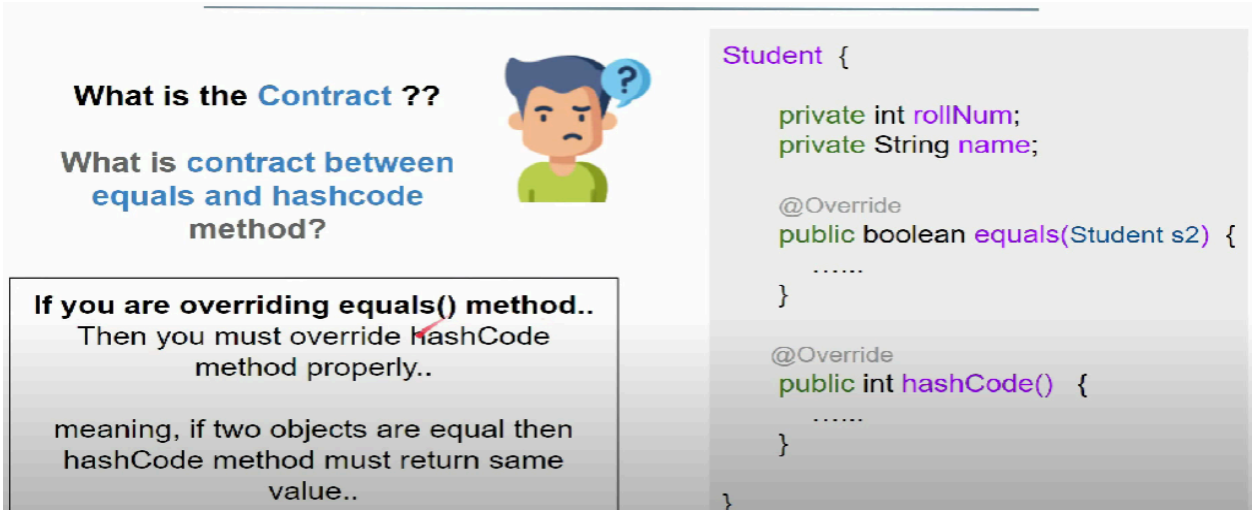
Here object o1,o2 are pointing to the same object so they will probably have the same hashcode. o3 will have different hashcodes.

We can override the default equals and hashcode methods.

The contract between equals() and hashCode() :

Case1: If we are overriding **equals()** method. Then we should override the **hashCode()** method properly.
If two objects are equal (equals() returns true), they must have the same hashCode().

If two objects have the same hashCode(), they may or may not be equal.



equals() Method

```
public boolean equals(Object obj)
```

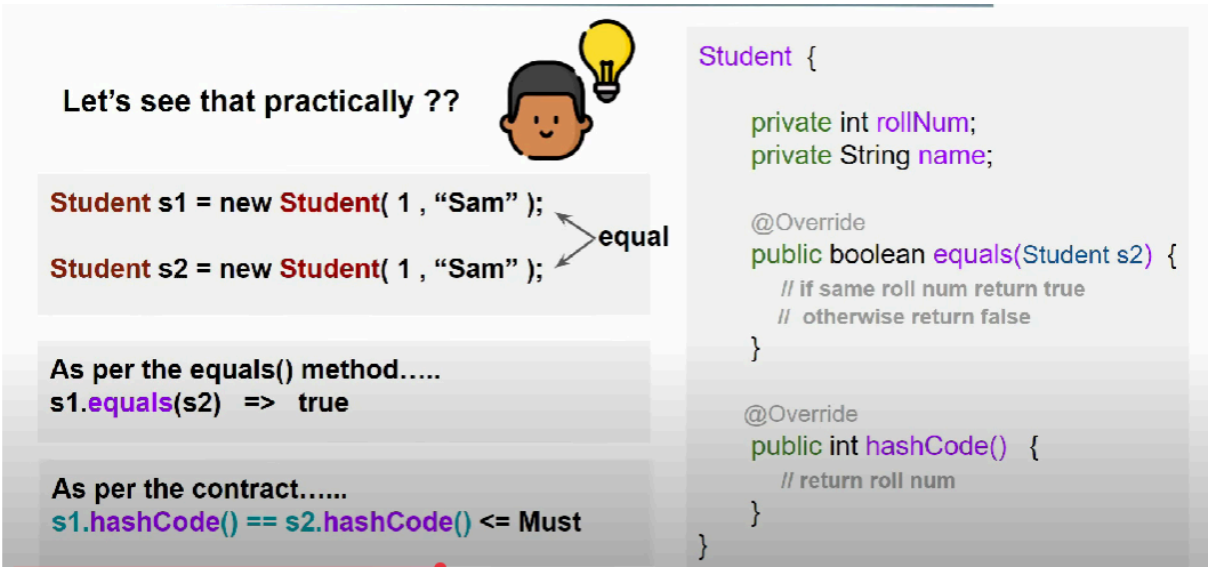
The **equals()** method of Object class checks the equality of the objects and accordingly it **returns true** or **false**. The default implementation, as provided by Object class, checks the equality of the objects on the basis if both references refer to the same object. It does not check the value or state of the objects. But we can override this method to provide our own implementation to compare the state or value of the objects.

Ex:

Here we are using student roll as keys, and we overriding default equals(it compares references) to our own equals method logic . So here the reference doesn't matter but the values are equal are not matter.

We compare only the roll number, ignoring the name.

s1.equals(s2) → true because they have the same roll number.



Every Java object has two very important methods **equals()** and **hashCode()** and these methods are designed to be overridden according to their specific **general contract**. An **Object** class is the parent class of every class, the default implementation of these two methods is already present in each class. However, we can override these methods based on the requirement.

hashCode() Method

`public int hashCode()`

This method returns an **integer** value, which is referred to as the hash code value of an object. Every Object, at the time of creation assigned with a unique 32-bit, signed int value. This value is the hash code value of that object.

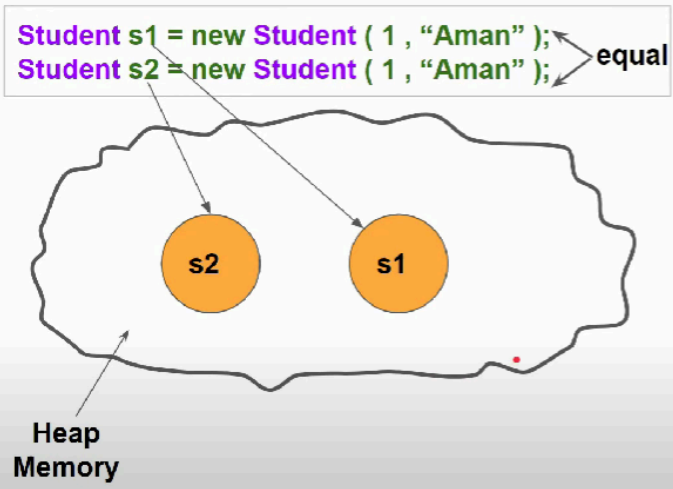
What Happens If We Override Equals() But Not hashCode()?

I will not follow contract !!
What will happen ??

Then it will create problems when using Hashing structures in Java..

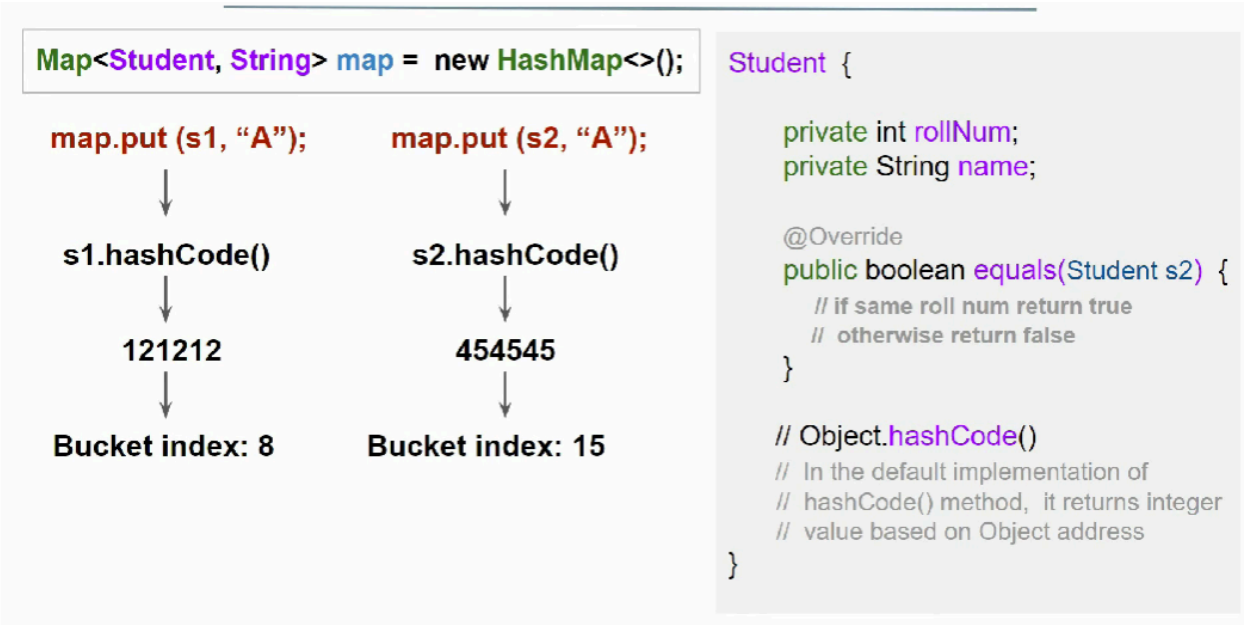
So, let's see that in action for HashMap!

```
Student {

    private int rollNum;
    private String name;

    @Override
    public boolean equals(Student s2) {
        // if same roll num return true
        // otherwise return false
    }

    @Override
    public int hashCode() {
        // return roll num
    }
}
```




```
Student {

    private int rollNum;
    private String name;

    @Override
    public boolean equals(Student s2) {
        // if same roll num return true
        // otherwise return false
    }

    // Object.hashCode()
    // In the default implementation of
    // hashCode() method, it returns integer
    // value based on Object address
}
```

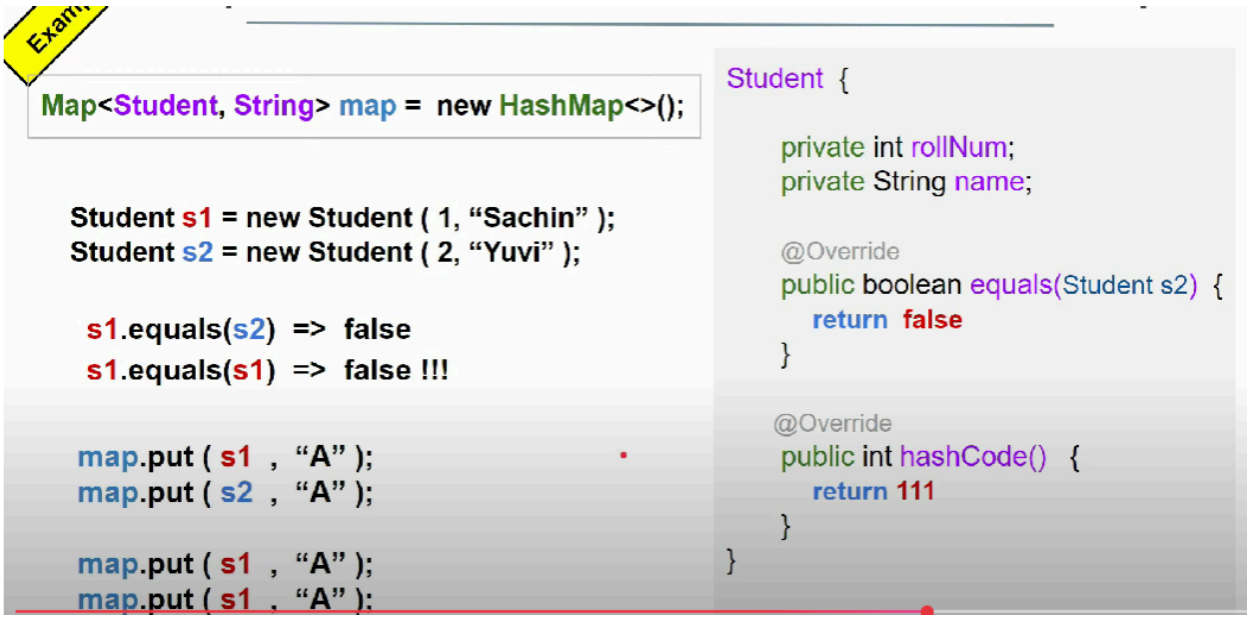
Then it will use the default hashCode method (default Object.hashCode() is used). So JVM will generate different hash codes

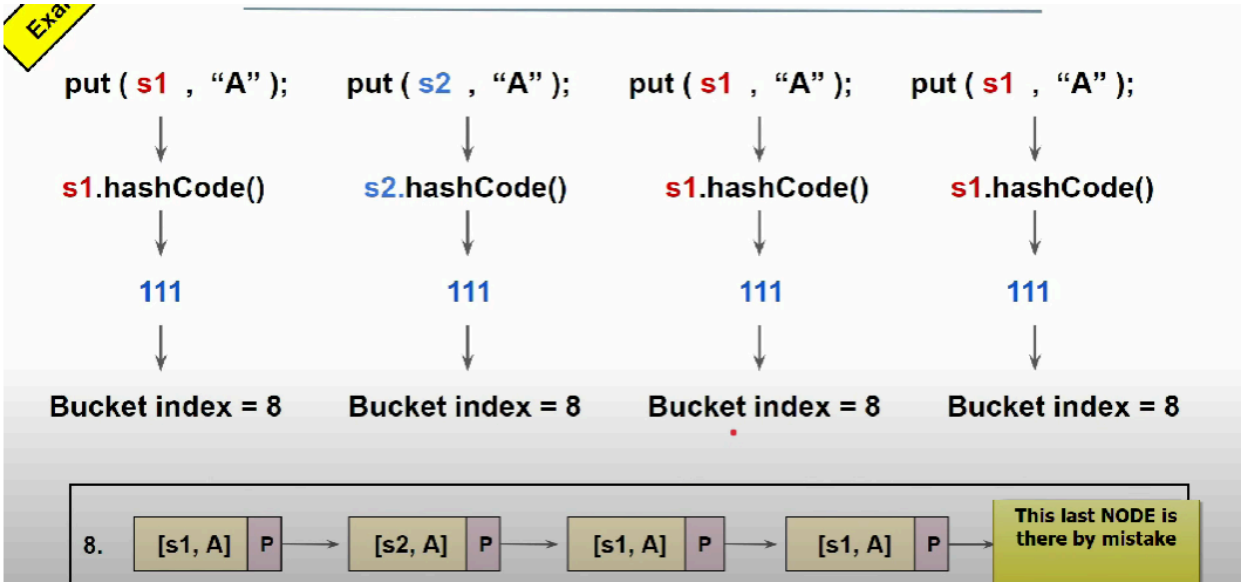


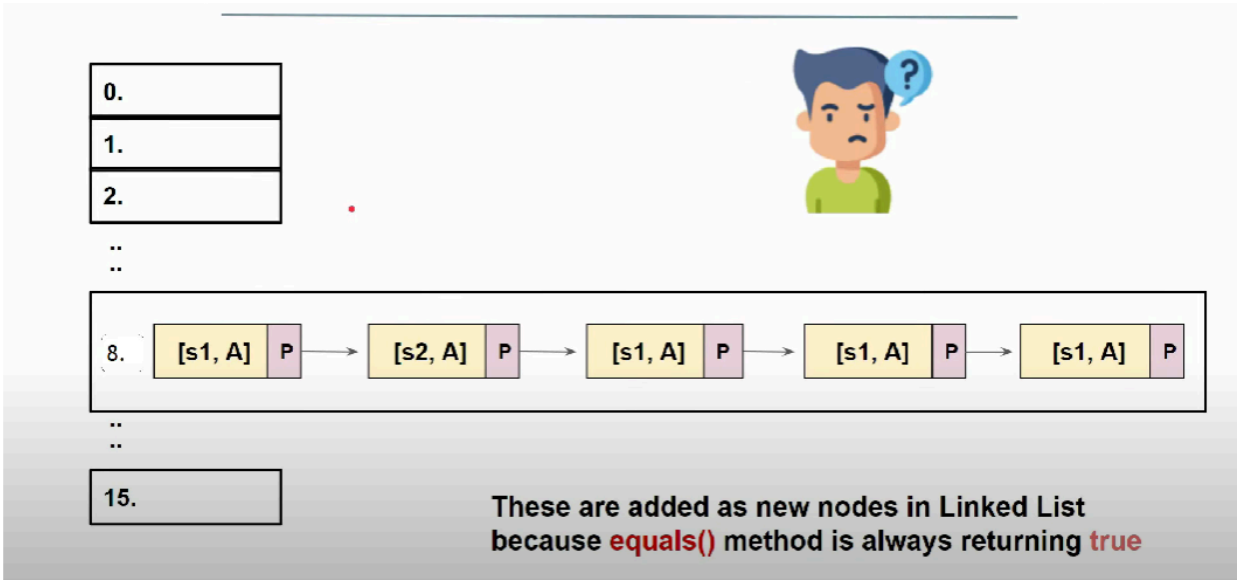
Interview question **(If two objects have the same hashCode(), they may or may not be equal)**

What Happens When hashCode() is Same but equals() is False?

1. **All objects go to the same bucket** because they have the same hash code.
2. **But they are stored as separate entries** in the bucket because equals() determines uniqueness.







For any non-null reference variables a, b and c

a.equals(a) should always return **true**.

a.equals(b) should return **true** if and only if **b.equals(a)** returns true.

If **a.equals(b)** returns **true** and **b.equals(c)** returns **true** then **a.equals(c)** should return **true**.

Multiple calling of **a.equals(b)** should consistently return **true** or consistently return **false** If the value of the object is not modified for either object.

a.equals(null) should return **false**.

So it is necessary to override the **hashCode()** method of Object class if we are overriding the **equals()** method.

---

3) If two interfaces have the same default method, and a third interface or class implements both, you must explicitly specify which interface's default method to call.

Example:

```
interface First {
    default void show() {
        System.out.println("First interface show()");
    }
}

interface Second {
    default void show() {
        System.out.println("Second interface show()");
    }
}

interface Third extends First, Second {
    @Override
    default void show() {
        // Calling First interface's default method
        First.super.show();
    }
}
```

Explanation:

1. **Both First and Second interfaces** define the same default method show().
2. **Third interface extends both First and Second**:
    - If it does not override show(), there will be a compilation error due to ambiguity.
    - To resolve this, Third must override show() and explicitly call First.super.show().

if the Third **interface extends only First** and **not Second**, then there is no ambiguity, and we can directly call the show() method from First.

Example:

```
interface First {
    default void show() {
        System.out.println("First interface show()");
    }
}

interface Second {
    default void show() {
        System.out.println("Second interface show()");
    }
}

// Third interface extends only First
interface Third extends First {
    // No need to override 'show()' because it inherits from First
}
```

4) How to create immutable class

Creating an **immutable class** in Java means designing a class whose instances (objects) cannot be modified after creation. Immutable objects provide thread safety, simplify debugging, and are commonly used in functional programming.

Steps to Create an Immutable Class in Java

1. Declare the class **as final** → Prevent subclassing.
2. Make **all fields private** and **final** → Prevent direct access and modification.
3. Provide **only getters, no setters** → Prevent modifying field values.
4. Initialize fields **using a constructor** → Assign values only once.

simple example of creating an immutable class in Java

```
package com.ram;
final class Employee {
  private final String name; // Immutable field
  private final int age;     // Immutable field
  // Constructor to initialize fields
```

```
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Getter methods (No setters)
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
public class ImmutableClass {
    public static void main(String[] args) {
            Employee emp = new Employee("John", 25);
        System.out.println(emp.getName()); // Output: John
        System.out.println(emp.getAge());  // Output: 25
        // Cannot modify name or age because there are no setters
        // student.name = "Mike";  ❌ Not Allowed
        // student.age = 30;       ❌ Not Allowed
    }
}
```

Output
John
25

Explanation 🔑

1. **final class** → No one can extend the Student class.
2. **private final fields** → Once initialized, they cannot be modified.
3. **Only getter methods** → No setters are provided, so the fields are **read-only**.

5) Java Comparable vs Comparator

Both Comparable and Comparator are used to **compare and sort objects** in Java, but they have key differences in how they are implemented and used.

- **Comparable:** It is used to define the **natural ordering of the objects** within the class.

- **Comparator**: It is used to define **custom sorting logic** externally.

1. Comparable Interface (Natural Ordering)

- Used when **a class itself** wants to define its **default sorting order**.
- Implements **compareTo(T obj)** method.
- Modifies the **class itself** to define the sorting logic.
- **Sorting Example**: Sorting by name, ID, or age in a **natural order**.
- It defines natural ordering within the class. It is implemented in the class.

```
package com.comparableandcomparator;
import java.util.*;
class Student implements Comparable<Student> {
	private String name;
	private int age;
	public Student(String name, int age) {
		this.name = name;
		this.age = age;
	}
	// Implementing compareTo() to sort by age (ascending order)
	@Override
	public int compareTo(Student other) {
		return this.age - other.age; // Ascending order
	}
	@Override
	public String toString() {
		return name + " (" + age + ")";
	}
}
public class ComparableExample {
	public static void main(String[] args) {
```

```
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 25));
        students.add(new Student("Bob", 22));
        students.add(new Student("Charlie", 30));
        Collections.sort(students); // Sorts using Comparable (by age)
        System.out.println(students);
    }
}
```

[Bob (22), Alice (25), Charlie (30)]

2. Comparator Interface (Custom Sorting)

- Used when you **want multiple sorting options** (e.g., sorting by name, age, etc.).
- Implements **compare(T obj1, T obj2)** method.
- Does **not modify** the original class.
- More flexible as **multiple comparators** can be created.
- We use **Comparator** to define **custom sorting logic** without modifying the original class.

```
package com.comparableandcomparator;
import java.util.*;
class Worker {
    private String name;
    private int age;
    public Worker(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
```

```
        }
        public int getAge() {
                return age;
        }
        @Override
        public String toString() {
                return name + " (" + age + ")";
        }
}
// Comparator to sort by Name
class NameComparator implements Comparator<Worker> {
        @Override
        public int compare(Worker s1, Worker s2) {
                return s1.getName().compareTo(s2.getName());
        }
}
// Comparator to sort by Age (Descending order)
class AgeComparator implements Comparator<Worker> {
        @Override
        public int compare(Worker s1, Worker s2) {
                return s2.getAge() - s1.getAge(); // Descending order
        }
}
public class ComparatorExample {
        public static void main(String[] args) {
                List<Worker> students = new ArrayList<>();
                students.add(new Worker("Alice", 25));
                students.add(new Worker("Bob", 22));
                students.add(new Worker("Charlie", 30));
                // Sorting by Name
```

```
        Collections.sort(students, new NameComparator());
        System.out.println("Sorted by Name: " + students);
        // Sorting by Age (Descending)
        Collections.sort(students, new AgeComparator());
        System.out.println("Sorted by Age (Descending): " + students);
    }
}
```

Sorted by Name: [Alice (25), Bob (22), Charlie (30)]

Sorted by Age (Descending): [Charlie (30), Alice (25), Bob (22)]

Difference Between Comparable and Comparator

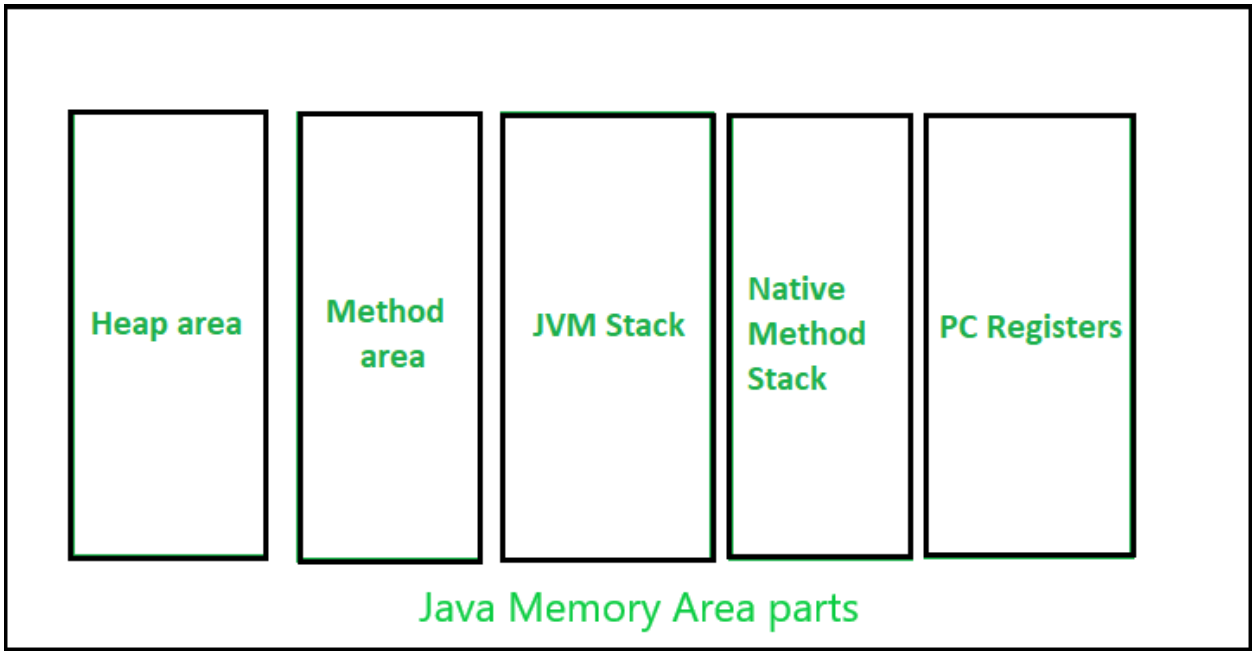
Features	Comparable	Comparator
Definition	It defines natural ordering within the class.	It defines external sorting logic.
Method	compareTo()	compare()
Implementation	It is implemented in the class.	It is implemented in a separate class.
Sorting Criteria	Natural order sorting	Custom sorting
Usage	It is used for single sorting order.	It is used for multiple sorting orders.

6) Metaspace vs. PermGen in Java

Before understanding the metaspace, let's first understand the JVM memory structure.

JVM Memory Structure:
JVM defines various run-time data areas which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of the thread are created during instantiation and destroyed when the thread exits.

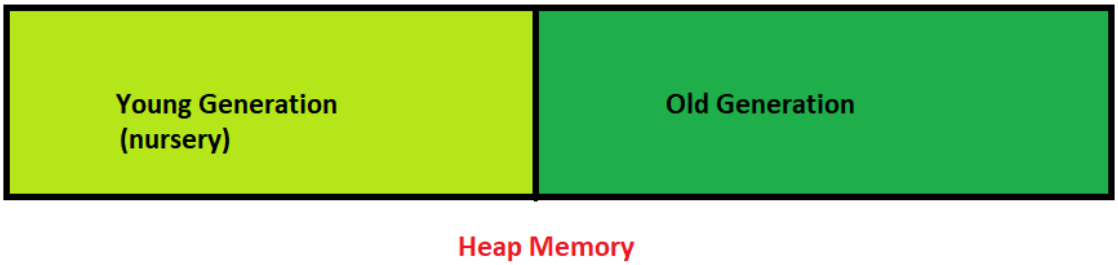
JVM Memory Structure is divided into multiple memory areas like heap area, stack area, method area, PC Registers etc. The following image illustrates the different memory areas in Java:



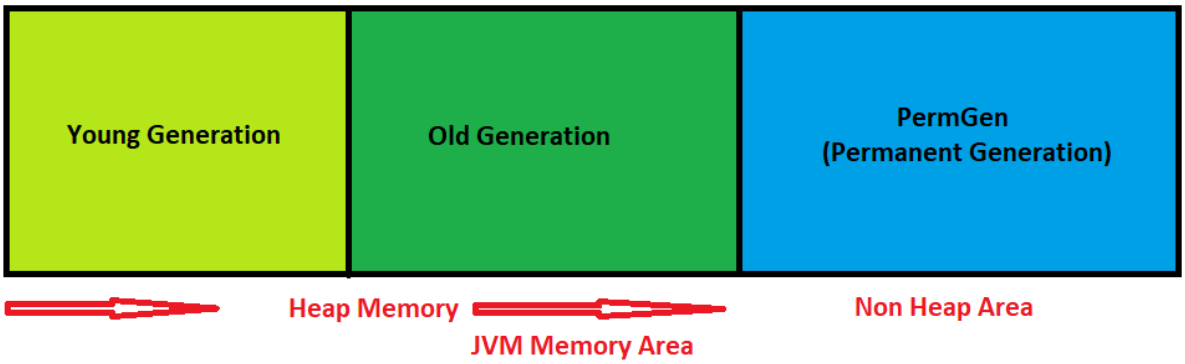
JVM Memory area parts

Here, the heap area is one of the most important memory areas of JVM. Here, all the java objects are stored. The heap is created when the JVM starts. The heap is generally divided into two parts. That is:

1. Young Generation(Nursery): All the new objects are allocated in this memory. Whenever this memory gets filled, the garbage collection is performed. This is called the Minor *Garbage Collection*.
2. Old Generation: All the long lived objects which have survived many rounds of minor garbage collection are stored in this area. Whenever this memory gets filled, the garbage collection is performed. This is called the Major *Garbage Collection*.



Apart from the heap memory, JVM also contains another type of memory which is called as Permanent Generation or “PermGen”.



In Java, class metadata (information about loaded classes, methods, and runtime constant pools) was historically stored in the **PermGen (Permanent Generation) space before Java 8**. However, **Java 8 replaced PermGen with Metaspace** for better memory management and performance.

PermGen Memory in Java

- **Definition**:
    - A special space in the Java heap, separate from the main memory.
    - Stores static content and application metadata required by the JVM.
    - A fixed-size memory area inside the JVM **heap** used to store class metadata, method metadata, interned Strings, and static variables.
    - Introduced in **Java 1.2** and used until **Java 7**.
- **Key Features**:
    - Contains metadata, which is data describing other data.
    - Garbage collection occurs, similar to other memory areas.

    - **String pool** was part of PermGen before Java 7.
    - **Method Area** (within PermGen) stores class structures and method/constructor code.
- **Limitations**:
    - Fixed size leads to **OutOfMemoryError**.
    - Default size:
        - **32-bit JVM → 64 MB**
        - **64-bit JVM → 82 MB**
    - JVM frequently performs garbage collection to free space (costly operation).
    - Manual resizing is allowed, but auto-scaling is **not possible**.
    - Garbage collector is **not efficient** in cleaning up PermGen memory.

Replacement of PermGen in Java 8

Metaspace (Introduced in Java 8)

What is Metaspace?

- A **new memory space introduced in Java 8** that replaces PermGen.
- Unlike PermGen, **Metaspace is part of native memory (outside the heap)**.
- **Automatically resizes** itself based on requirements (limited by system memory).
- Reduces **OutOfMemoryErrors** related to class metadata.

Key Features of Metaspace

✅ **No fixed size (Dynamically grows & shrinks)** → Unlike PermGen, Metaspace can expand as needed, reducing memory management overhead.
✅ **Better GC Performance** → Since Metaspace is in **native memory**, it does not interfere with the heap, improving Garbage Collection efficiency.
✅ **Stores only class metadata** → Unlike PermGen, which stored interned Strings, Metaspace only holds class-related information.
✅ **Controlled via JVM options** → Even though it auto-expands, developers can set limits if needed..

Aspect	PermGen (Java 7 and below)	Metaspace (Java 8 and later)
Stored in	JVM Heap	Native Memory (outside heap)
String Pool	Stored in PermGen	Moved to Heap (since Java 7)
Size	Fixed, needs manual tuning	Dynamic, no fixed limit
Performance	GC issues, manual tuning required	More efficient, automatic resizing
OutOfMemoryError	`OutOfMemoryError: PermGen space`	`OutOfMemoryError: Metaspace` (only if limited)

7. can we call start method twice on thread in java

No, you **cannot** call the start() method twice on a thread in Java. Doing so will result in a **java.lang.IllegalThreadStateException**.

Explanation

- The start() method is used to **start a new thread** and call its run() method.
- Once a thread has been started, it goes from **New → Runnable → Running → Terminated**.
- If you try to call start() again on a **terminated thread**, Java will throw an exception.

Example: Calling start() Twice

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

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

        t.start(); // ✅ First start - Works fine
        t.start(); // ❌ Second start - Throws IllegalThreadStateException
    }
}
```

Output

```
Thread is running...
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Thread.java:744)
    at ThreadStartExample.main(ThreadStartExample.java:12)
```

8) Serialization and Deserialization in Java

What is Serialization?

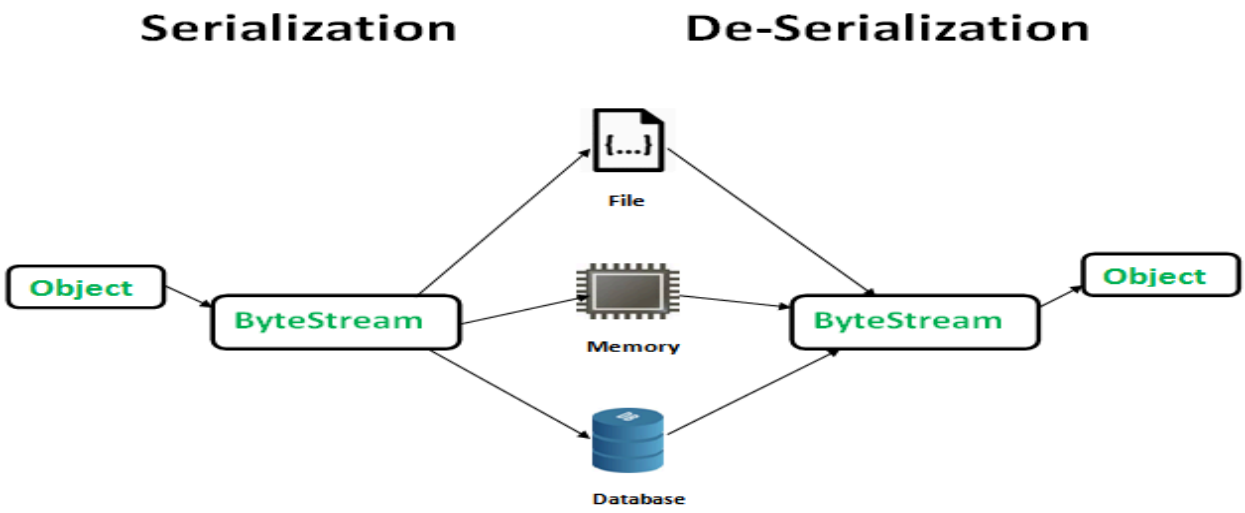
Serialization in Java is the process of converting an object into a byte stream so that it can be:

- Saved to a file
- Sent over a network
- Stored in a database

This allows objects to be **persisted** and later **recreated** using **deserialization**.

What is Deserialization?

Deserialization is the reverse process of **serialization**. It converts the byte stream back into an object.



How to Serialize and Deserialize an Object in Java?

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface.

Step 1: Implement Serializable Interface

Java provides the Serializable interface (from java.io package), which marks a class as **serializable**.

Step 2: Use ObjectOutputStream for Serialization

To serialize an object, use ObjectOutputStream along with FileOutputStream.

The ObjectOutputStream class contains a **writeObject()** method for serializing an Object.

Step 3: Use ObjectInputStream for Deserialization

To deserialize an object, use ObjectInputStream along with FileInputStream.

The ObjectInputStream class contains a **readObject()** method for deserializing an object.

Use Cases of Serialization

1. Storing objects in files/databases
2. Sending objects over a network (RMI, Socket Programming)

Real-World Applications of Serialization

✅ **1. Saving User Preferences**

- When you save settings in an application, the configuration objects are serialized and stored.
- Example: Android Shared Preferences, Game Settings

```
package com.serializationanddeserialization;
import java.io.*;
//Step 1: Implement Serializable interface
class Person implements Serializable {
        private static final long serialVersionUID = 1L; // Version control (Optional)
        String name;
        int age;
        // Constructor
        public Person(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }
    // Display object data
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
public class SerializationExample {
    public static void main(String[] args) {
        // Create an object of Person
        Person person = new Person("John Doe", 30);
        // Serialization (Saving object to file)
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("person.ser"))) {
            oos.writeObject(person);
            System.out.println("Serialization Successful!");
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Deserialization (Reading object from file)
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.ser"))) {
            Person deserializedPerson = (Person) ois.readObject();
            System.out.println("Deserialization Successful!");
            deserializedPerson.display();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```
Serialization Successful!
Deserialization Successful!
Name: John Doe, Age: 30
```

Transient Keyword (transient)

- Fields marked as transient are **not serialized**.

Example:

```
class Employee implements Serializable {
   String name;
   transient double salary;  // This field won't be serialized

        }
```

What is serialVersionUID?

serialVersionUID is a unique identifier assigned to a Serializable class. It is used during deserialization to verify that the sender and receiver of a serialized object have compatible class definitions. If the class definition changes after an object has been serialized (e.g., adding or removing fields), deserialization may fail unless serialVersionUID is explicitly defined.

Understanding serialVersionUID in Simple Words

1. **Serialization Process:**
    - When you **make a class Serializable**, Java assigns a unique number to it called serialVersionUID.
    - This serialVersionUID is **generated based on the class structure** (fields, methods, etc.).
    - The object is then **converted into a byte stream** and saved to a file or sent over a network.
2. **Deserialization Process:**
    - When you **deserialize the object**, Java checks the serialVersionUID of the class **at the time of serialization** with the current serialVersionUID of the class.
    - **If they match** → The object is successfully deserialized.
    - **If they don't match** → Java throws an InvalidClassException, meaning the class has changed in a way that is incompatible with the saved object.

serialVersionUID

- Used to **maintain version control** of a serialized class.
- If the class structure changes (e.g., adding/removing fields), deserialization may fail.
- Declaring serialVersionUID ensures compatibility.

9. Transaction Management in Spring

A **transaction** in Spring ensures that a series of operations execute as a single unit, meaning either all operations succeed (**commit**) or all fail (**rollback**) in case of an error. This helps maintain **data consistency** in applications.

Why is Transaction Management Needed?

1. **Ensures Data Integrity** – Prevents partial updates in case of failure.
2. **Manages Multiple Database Operations** – Ensures all operations succeed or fail together.
3. **Handles Concurrency** – Prevents inconsistent data updates in multi-threaded environments.
4. **Supports Rollback Mechanism** – Restores database state if something goes wrong.

Ways to Manage Transactions in Spring

Spring provides different ways to handle transactions:

1️⃣ Declarative Transaction Management (Recommended)

- Uses **@Transactional** annotation.
- Spring manages transactions automatically.
- No need to write explicit transaction handling code.

✅ Example: Using @Transactional in a Service Class

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class PaymentService {

    @Transactional
    public void processPayment(String userId, double amount) {
        debitAccount(userId, amount);
        creditMerchant(amount);
    }

    private void debitAccount(String userId, double amount) {
        // Deduct amount from user's account
    }

    private void creditMerchant(double amount) {
```

```
        // Add amount to merchant's account
    }
}
```

🔹 **By default, if an exception occurs, the transaction is rolled back.**
🔹 **Spring manages commit & rollback automatically.**

🔹 **Rollback Behavior**

- By default, @Transactional **rolls back only for unchecked exceptions** (RuntimeException).
- To roll back for checked exceptions, use: @Transactional(rollbackFor = Exception.class)

2️⃣ Programmatic Transaction Management

- Uses **TransactionTemplate** or **PlatformTransactionManager**.
- Gives manual control over transactions.
- Useful when transaction boundaries are dynamic.

✅ **Example: Using TransactionTemplate**

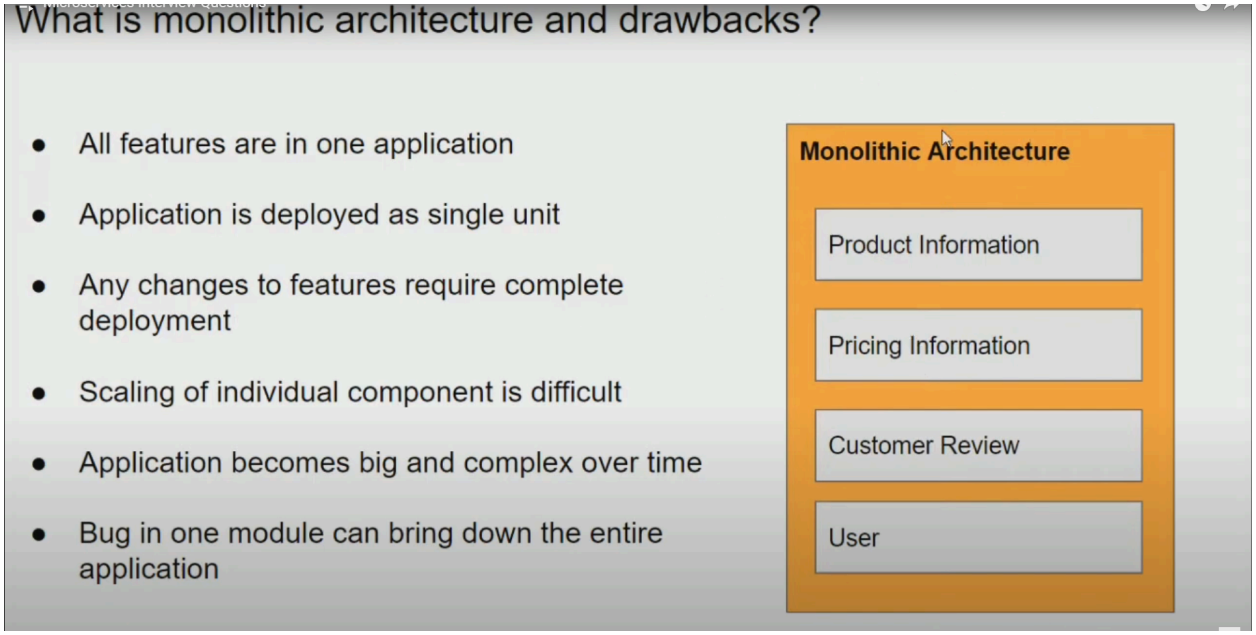
```
import org.springframework.transaction.support.TransactionTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PaymentService {

    @Autowired
    private TransactionTemplate transactionTemplate;

    public void processPayment(String userId, double amount) {
        transactionTemplate.execute(status -> {
            debitAccount(userId, amount);
            creditMerchant(amount);
            return null; // Returning null means commit, an exception will trigger rollback });
        });
    }

    private void debitAccount(String userId, double amount) {
        // Deduct amount from user's account
```

```
    }

    private void creditMerchant(double amount) {
        // Add amount to merchant's account
    }
}
```

- **Explicit control over transactions, but requires more boilerplate code.**

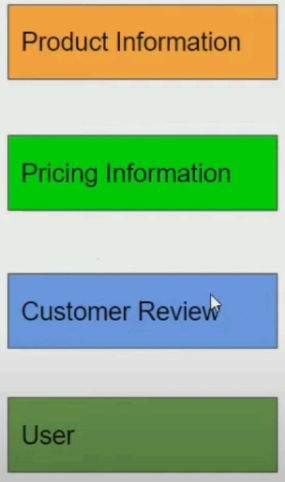
Use Programmatic Transactions 🔧 → Only when needed
When you need **manual control** over commit/rollback

10.Microservices Questions



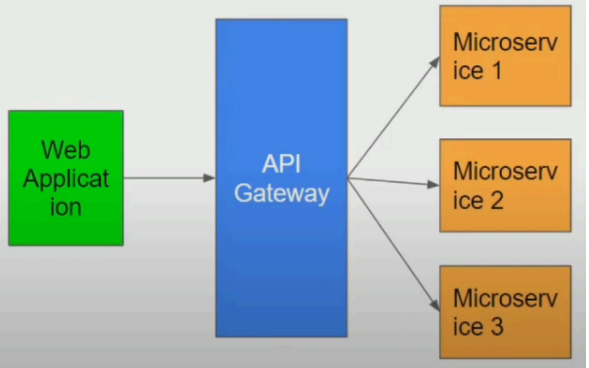
What is microservices?

- Microservices architecture is a collection of services or small projects
- Each microservice represents a business functionality
- Deployed independently
- Single team own it
- Highly maintainable and testable
- Different Technical Stack



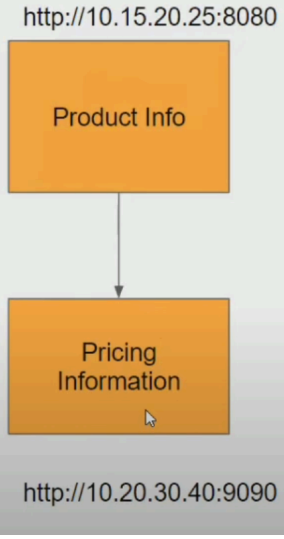
What is API Gateway?

- API Gateway sits between client and microservices
- It acts as an entry point to the microservices
- All HTTP calls shall go through API gateway
- Common cross cutting concerns like Authentication, Authorization, Rate limiting can be handled
- For high availability you can deploy multiple instances of API gateway behind load balancer
- Ex. Zuul, Spring Cloud gateway



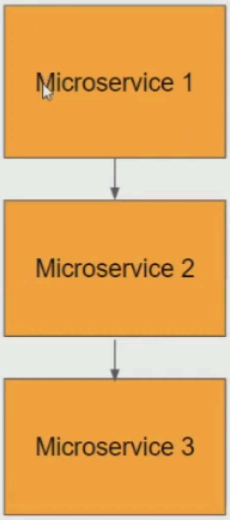
What is service discovery in microservices?

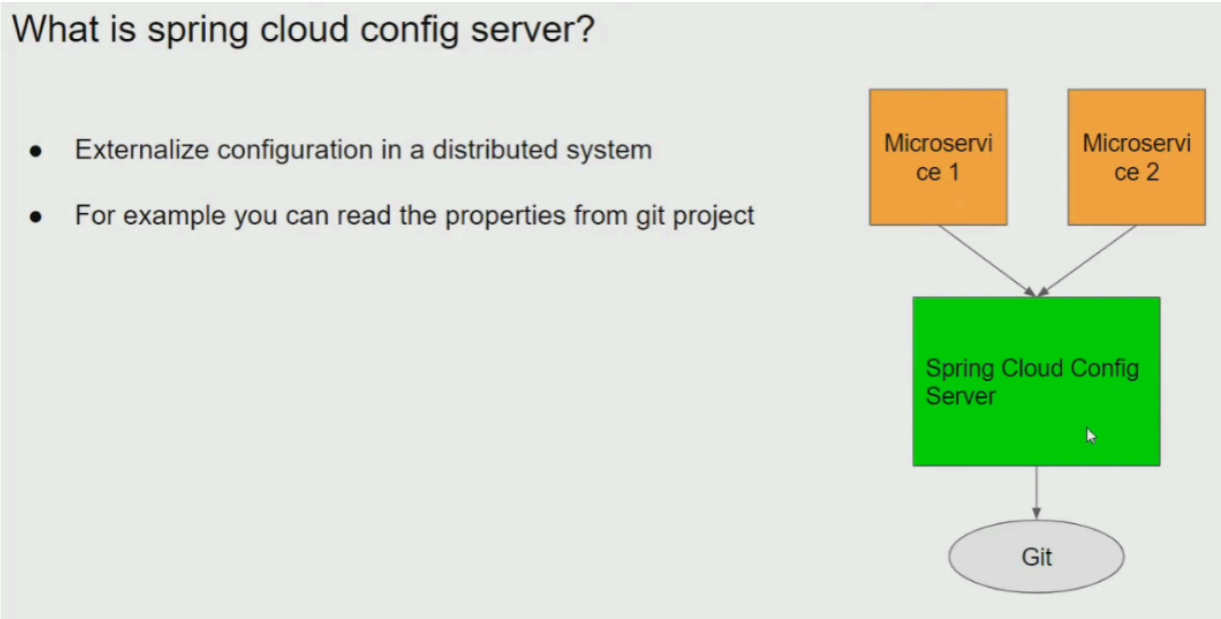
- One microservice can discover URL address of other microservice
- Each microservice register its address with discovery server
- Example Product info service can discover the pricing information service URL
- There are two types of service discovery
  a.) Client side
  b.) Server side discovery
- Eureka is a netflix library to perform Service discovery
- Consul by HashiCorp

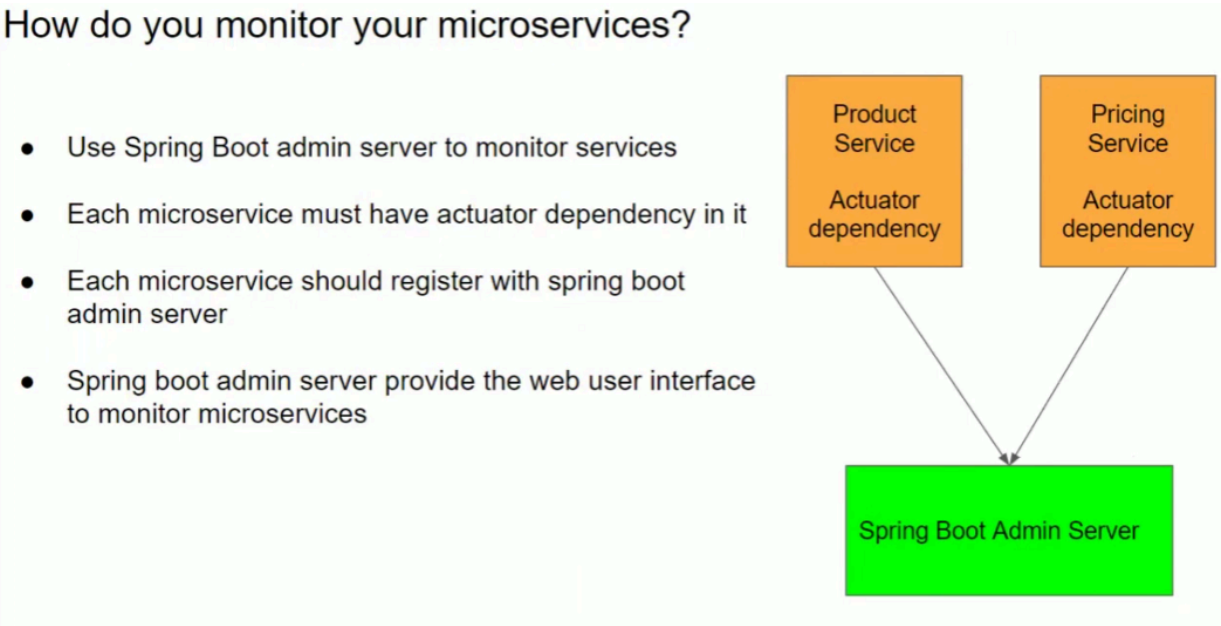


How do you handle fault tolerance in microservices?

- Fault tolerance is used to avoid cascading failures
- Microservice should continue to operate even if some other microservices fail in the system
- In case of failure, return default response from fallback method
- Hystrix is used for circuit breaker and return fallback response
- Resilience4j can also be used for fault tolerance







11. Sort Values from HashMap;

```
package com.ram;
import java.util.Comparator;
import java.util.HashMap;
public class HashmapValueSorting {
	public static void main(String[] args) {
		HashMap<String,Integer> map = new HashMap<>();
		map.put("a", 22);
```

```
                map.put("b", 4);
                map.put("c", 12);
                map.put("d", 9);


//map.entrySet().stream().sorted((j,k)->j.getValue()-k.getValue()).forEach(fin->System.out.printl
n(fin.getKey()+" "+fin.getValue()));
                map.entrySet()
                  .stream()
                  .sorted(Comparator.comparing(entry -> entry.getValue())) // Using lambda
                  .forEach(entry -> System.out.println(entry.getKey() + " " + entry.getValue()));
        }
}
```

12. Find Palindrome words from sentence

```
package com.ram;
public class FindPalindromeWordsFromSentence {
        public static void main(String[] args) {
                String str = "My name is nitin and I can speak malayalam";
                String[] splitArray = str.split(" ");
                for (String s : splitArray) {
                        if (isPalindrome(s)) {
                                System.out.println(s + " is palindrome");
                        }
                        /*
                         * if (isPalindrome(s.toLowerCase())) { // Convert to lowercase for
                         * case-insensitive check System.out.println(s + " is palindrome"); }
                         */
                }
        }
        // name
        public static boolean isPalindrome(String input) {
                int i = 0;
                int j = input.length() - 1;
                while (i < j) {
                        if (input.charAt(i) != input.charAt(j)) {
                                return false;
                        }
                        i++;
```

```
                j--;
            }
            return true;
        }
}
```

nitin is palindrome
I is palindrome
malayalam is palindrome

How This Works

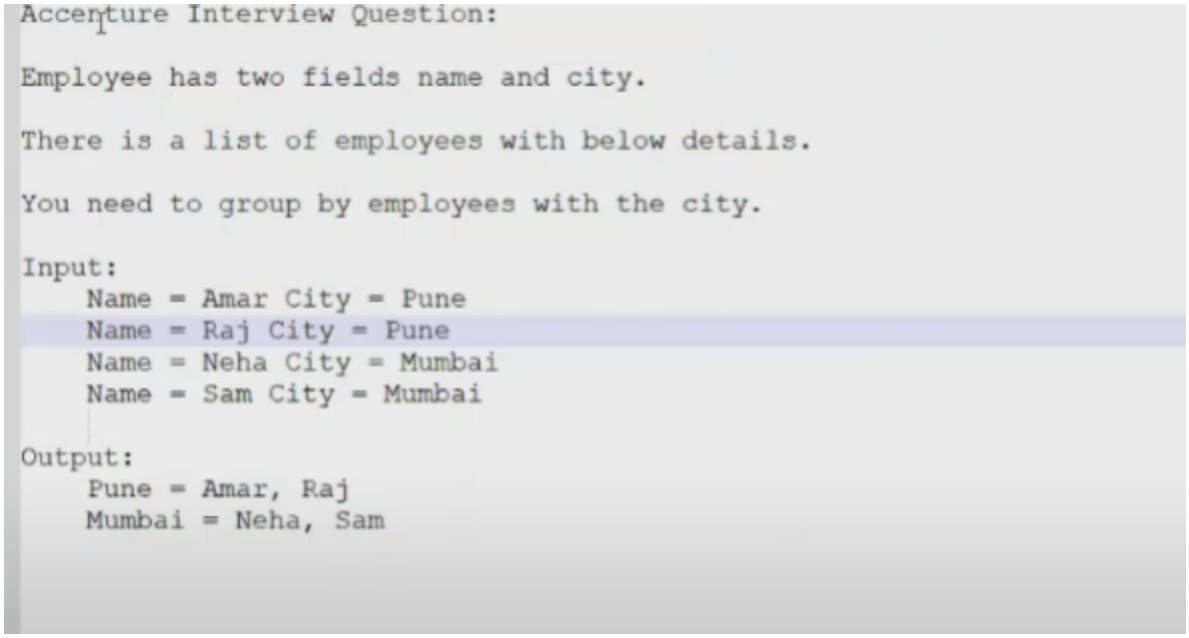
1. **Splits the sentence** into words using split(" ").
2. **Converts each word to lowercase** to make the palindrome check case-insensitive.
3. **Checks if each word is a palindrome** using the isPalindrome() method.
4. **Prints the palindromic words**.

13. Palindrome

```
package com.ram;
public class PlaindromeString {
    public static void main(String[] args) {
        String str = "madam";
        // Way 1: Using StringBuilder reverse()
        StringBuilder sb = new StringBuilder(str);
        sb.reverse();
        System.out.println(str);
        System.out.println(sb);
        if (str.equals(sb.toString()))
            System.out.println("Plaindrome");
        else
            System.out.println(" not Plaindrome");
        // wayb2:
        String str1 = "mom";
        String str2 = "";
        for (int i = str1.length() - 1; i >= 0; i--) {
            str2 = str2 + str1.charAt(i);
```

```
                }
                System.out.println(str2);
                if (str1.equals(str2))
                        System.out.println("Plaindrome");
                else
                        System.out.println(" not Plaindrome");
                // Way 3: Using Two Pointers (Fixing the Logical Issue)
                String str3 = "mom";
                int i = 0, j = str3.length() - 1;
                boolean isPalindrome = true; // Flag variable to track palindrome status
                while (i < j) {
                        if (str3.charAt(i) != str3.charAt(j)) {
                                isPalindrome = false;
                                break; // Exit the loop immediately if characters do not match
                        }
                        i++;
                        j--;
                }
                if (isPalindrome) {
                        System.out.println("It is a palindrome");
                } else {
                        System.out.println("It is not a palindrome");
                }
        }
}
```

14. GroupBy




```java
package com.ram;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
class Employeess {
	private String name;
	private String city;
	public Employeess(String name, String city) {
		super();
		this.name = name;
		this.city = city;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
		this.city = city;
```

```
	}
	@Override
	public String toString() {
		return "Employee [name=" + name + ", city=" + city + "]";
	}
}
public class GroupBy {
	public static void main(String[] args) {
		List<Employeess> list = new ArrayList<Employeess>();
		list.add(new Employeess("Amar", "Pune"));
		list.add(new Employeess("Raj", "Pune"));
		list.add(new Employeess("Neha", "Mumbai"));
		list.add(new Employeess("Sam", "Mumbai"));
		System.out.println("Before grouping");
		for (Object o : list) {
			System.out.println(o);
		}
		// After
		System.out.println("After grouping");
		Map<String, List<Employeess>> groupByCity =
list.stream().collect(Collectors.groupingBy(e -> e.getCity()));
		System.out.println(groupByCity);
	}
}
```

```
Before grouping
Employee [name=Amar, city=Pune]
Employee [name=Raj, city=Pune]
Employee [name=Neha, city=Mumbai]
Employee [name=Sam, city=Mumbai]
After grouping
{Pune=[Employee [name=Amar, city=Pune], Employee [name=Raj, city=Pune]],
Mumbai=[Employee [name=Neha, city=Mumbai], Employee [name=Sam, city=Mumbai]]}
```

15.Armstrong number

```
Problem statement
How to check if a given number is armstrong number or not?

Input
153
1^3 + 5^3 + 3^3 = 1+125+27 = 153

Output
153 is Armstrong number


Input
125

Output
125 is not Armstrong number
```

```java
package com.ram;
import java.util.Scanner;
public class ArmstrongNum {
	public static void main(String[] args) {
		int num = 153;
		int temp = num;
		int sum = 0;
		// Scanner
		/*
		 * Scanner sc = new Scanner(System.in); System.out.print("enter input value:");
		 * int input = sc.nextInt(); System.out.println("input value:"+ input);
		 */
		/*
		 * Quotient = 153 / 10 = 15 (integer division) Remainder = 153 % 10 = 3
		 * Dividend=(Divisor×Quotient)+Remainder 153=(10×15)+3
		 */
		while (num > 0) {
			int rem = num % 10;
			sum = sum + (rem * rem * rem);
			int quotient = num / 10;
			num = quotient;
		}
		if (temp == sum) {
			System.out.println("given num is armstrong");
		}else {
			System.out.println("given num is not armstrong");
		}
```

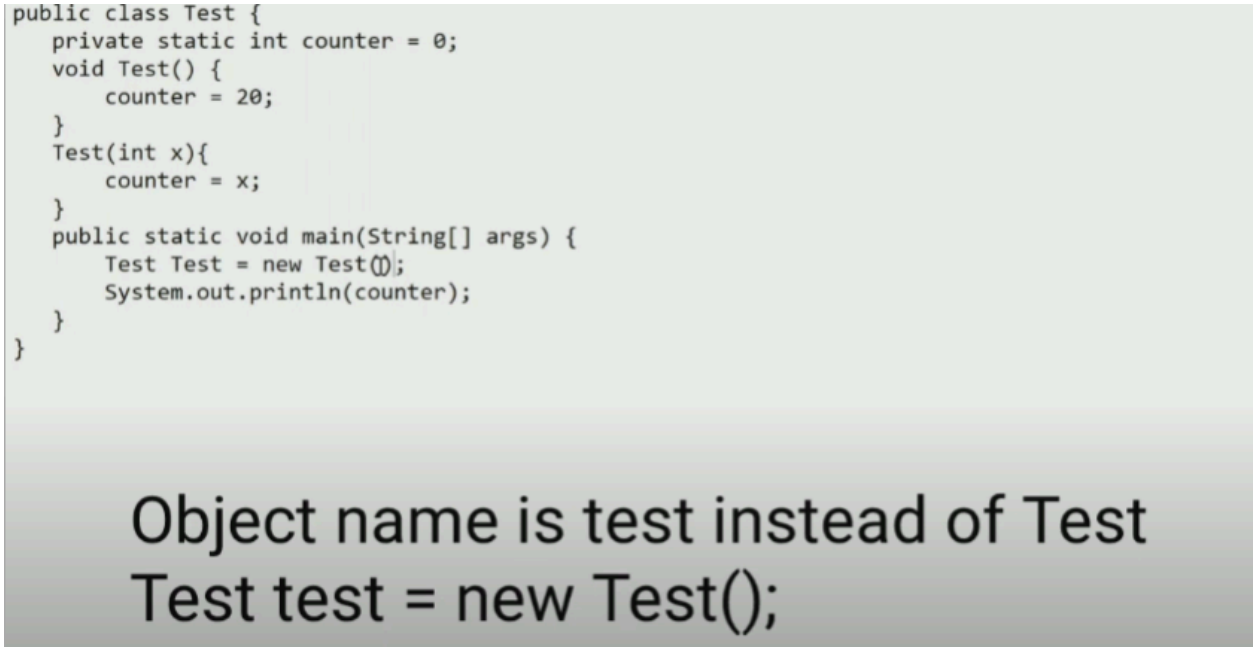
```
        }
}
```

given num is armstrong

Ex2: 125

given num is not armstrong

16. Guess output




```
public class Test {
    private static int counter = 0;

    void Test() {  // This is NOT a constructor, it's a method (wrong naming)
        counter = 20;
    }

    Test(int x) {  // This is the actual constructor
        counter = x;
    }

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

```
        Test Test = new Test();  // Compilation Error: No default constructor
        System.out.println(counter);
    }
}
```

Mistaken Constructor (void Test())

- This is NOT a constructor because it has a return type (void).
- In Java, constructors **do not have a return type**.
- So, this method will never be called automatically when an object is created.

No Default Constructor (Test())

- There is only a parameterized constructor Test(int x), but the code tries to create an object using new Test(), which does not exist.
- This results in a **compilation error**:
  error: constructor Test in class Test cannot be applied to given types;

17. Printing numbers from **1 to 100 without using numbers**

Approach 1: Using Character Arithmetic

Since 'A' has an ASCII value of 65, we can use it to generate numbers:

Since 'A' has an ASCII value of **65**, we can use it to generate numbers:

```
public class PrintNumbers {
    public static void main(String[] args) {
        for (char ch = 'A' / 'A'; ch <= ('d'); ch++) {  // 'd' is ASCII 100
            System.out.println((int) ch);
        }
    }
}
```

✔ **Explanation:**

- 'A' / 'A' → 65 / 65 → 1
- 'd' has ASCII value 100, so the loop runs till 100.

Approach 2: Using Array Length

```
public class PrintNumbers {
```

```
    public static void main(String[] args) {
        String s = ".....";  // Length = 10
        for (int i = s.length() / s.length(); i <= s.length() * s.length(); i++) {
            System.out.println(i);
        }
    }
}
```

✔ **Explanation:**

- s.length() = 10, so s.length() * s.length() = 100.
- s.length() / s.length() gives 1, so the loop runs from 1 to 100.

```
package com.ram;
public class PrintOneToHundredWithoutNumber {
        public static void main(String[] args) {
    //Way1
                for (char ch = 'A' / 'A'; ch <= ('d'); ch++) { // 'd' is ASCII 100
                        System.out.println((int) ch);
                }
                // way2
                String s = "."; // Length = 10
                for (int i = s.length() / s.length(); i <= s.length() * s.length(); i++) {
                        System.out.println(i);
                }
        }
}
```

Here are some common **ASCII values** for characters:

Uppercase Letters (A-Z)

Character	ASCII Value
A	65
B	66
C	67
D	68

E	69
...	...
Z	90

Lowercase Letters (a-z)

Character	ASCII Value
a	97
b	98
c	99
d	100
e	101
...	...
z	122

18. Find Duplicate Numbers Using Java7 And Java8

```
package com.ram;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
public class DuplicateNumbersusingJava7ndJav8 {
	public static void main(String[] args) {
		List<Integer> numbers = Arrays.asList(10, 20, 30, 30, 40, 50, 50);
		// Java7
		Set<Integer> uniqueNumbers = new HashSet<>();
		for (Integer num : numbers) {
			if (!uniqueNumbers.add(num)) {
				System.out.println(num);
			}
		}
```

```
	}
}
```

Using Java8

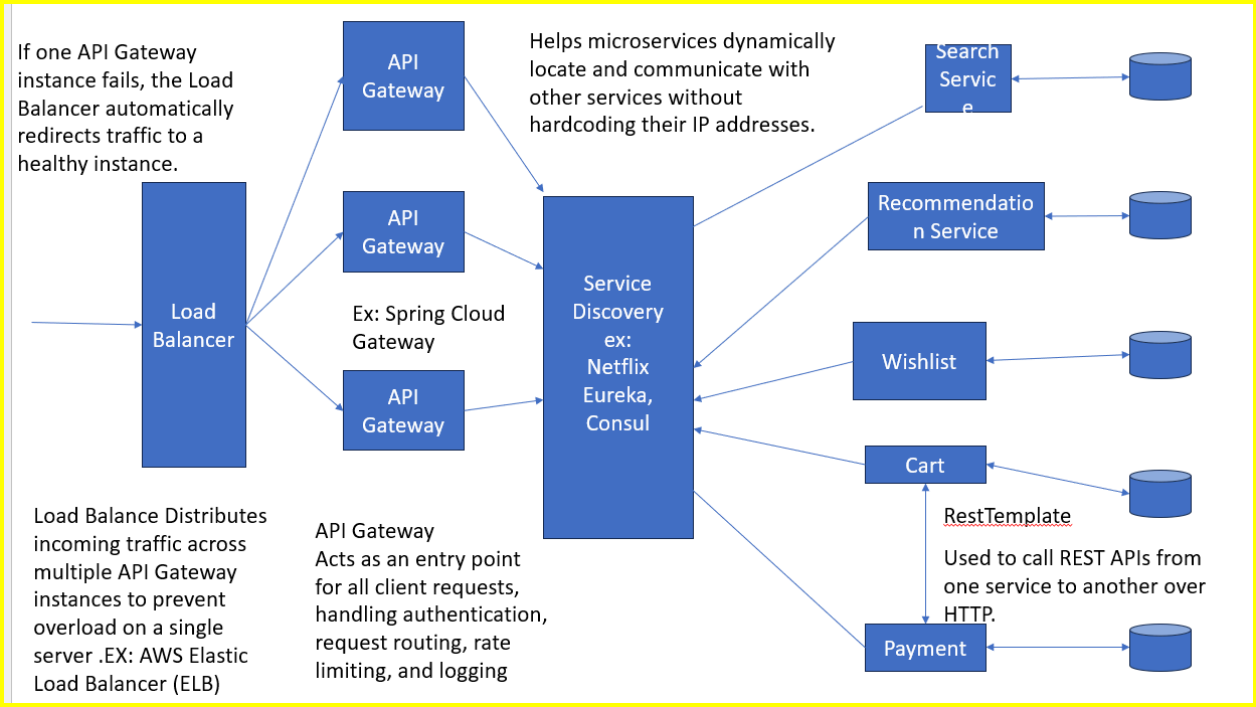
```
package com.ram;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
public class DuplicateNumbersusingJava7ndJav8 {
	public static void main(String[] args) {
		List<Integer> numbers = Arrays.asList(10, 20, 30, 30, 40, 50, 50);
		// Java7
		/*
		 * Set<Integer> uniqueNumbers = new HashSet<>(); for (Integer num : numbers)
{
		 * if (!uniqueNumbers.add(num)) { System.out.println(num); } }
		 */
		// Java8
		Set<Integer> uniqueNumbers = new HashSet<>();
		Set<Integer> duplicates = numbers.stream().filter(n -> !uniqueNumbers.add(n)) //
If add() returns false, it's a

                                                                    // duplicate
				.collect(Collectors.toSet());
		System.out.println("Duplicates using Java 8: " + duplicates);
	}
}
```

```
Duplicates using Java 8: [50, 30]
```

```java
public class Program1 {
    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(10, 20, 20, 25, 25, 27, 27, 30);

        Set<Integer> hs = new HashSet<Integer>();

        numbers.stream().filter(n -> !hs.add(n)).forEach(System.out::println);

    }
```

```
Markers Properties Servers Data Source Explorer Snippets Console Progress Debug
<terminated> Program1 (1) [Java Application] C:\Users\taufi\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_18.0.1.v20220515-1614\jre\bin\javaw.exe (23-Aug-20
20
25
27
```

19. Design E-Commerce Application



1. High-Level Architecture Flow

📌 Step-by-Step Process from User Request to Response:

1. **Client Request:** A user initiates a request (e.g., searching for a product, adding an item to a cart, or making a payment).

2. **Load Balancer:** Distributes incoming requests across multiple instances of API Gateways to ensure even load distribution.
3. **API Gateway:** Acts as a single entry point for all services, handling authentication, request routing, and response aggregation.
4. **Service Discovery:** Helps in dynamically locating the instances of microservices (e.g., Search, Recommendation, Cart, Payment).
5. **Microservices Processing:** Based on the request, a relevant microservice (e.g., Search, Recommendation, Cart, Payment) processes the request and interacts with the database.
6. **Inter-Service Communication:** Services communicate with each other using REST APIs, gRPC, or an event-driven mechanism (e.g., message queues).
7. **Database Interaction:** Each microservice interacts with its dedicated database for scalability and data consistency.
8. **Response Generation:** The processed data is sent back through the API Gateway to the client.

---

2. Breakdown of Each Component in Detail

1️⃣ Load Balancer

🔹 **What is it?**

A Load Balancer is a service that distributes incoming traffic across multiple API Gateway instances to prevent overload on a single server.

🔹 **Use Case:**

- Ensures high availability and reliability.
- Routes requests efficiently to reduce latency.

🔹 **Types:**

- **Hardware-based:** F5, Citrix NetScaler
- **Software-based:** Nginx, HAProxy
- **Cloud-based:** AWS Elastic Load Balancer (ELB), Google Cloud Load Balancer

🔹 **Failure Handling:**

- If one API Gateway instance fails, the Load Balancer automatically redirects traffic to a healthy instance.

2️⃣ API Gateway

🔹 **What is it?**

Acts as an entry point for all client requests, handling authentication, request routing, rate limiting, and logging.

🔹 **Use Case:**

- Security (JWT, OAuth2)
- Request Aggregation (Combining responses from multiple microservices)
- Load Balancing & Routing

🔹 **Popular Tools:**

- Spring Cloud Gateway
- Kong API Gateway
- Nginx API Gateway

🔹 **Failure Handling:**

- Uses Circuit Breaker (Hystrix, Resilience4j) to prevent overload.
- Implements retry logic for failed requests.

3️⃣ Service Discovery

🔹 **What is it?**

Helps microservices dynamically locate and communicate with other services without hardcoding their IP addresses.

🔹 **Use Case:**

- Automatically registers and deregisters microservices.
- Enables auto-scaling of services.

🔹 **Popular Tools:**

- Netflix Eureka
- Consul
- Kubernetes Service Discovery

🔹 **Failure Handling:**

- Uses **heartbeat checks** to monitor service health.
- If a service is down, it is removed from the registry.

4️⃣ Search Service

- 🔹 **What is it?**

  Fetches products from the database.

- Returns results based on user queries.

5️⃣ Recommendation Service

- 🔹 **What is it?**

Suggests products to users based on browsing history, purchase history, and trending products.

6️⃣ Wishlist Service

- 🔹 **What is it?**

Allows users to save products for future purchases.

7️⃣ Cart Service

- 🔹 **What is it?**

Handles shopping cart operations like adding, updating, or removing products.

8️⃣ Payment Service

- 🔹 **What is it?**

Handles online payments via credit cards, UPI, PayPal, etc.

Handling Failures in Microservices

- 🔹 **What Happens If One Service Goes Down?**

  - Service Discovery removes the failed service from the registry.
  - API Gateway reroutes requests to a **backup instance**.
  - Circuit Breaker prevents repeated failures from affecting the entire system.
  - Asynchronous Messaging (Kafka) ensures that events are not lost.

20. Java try-with-resources

In Java, **try-with-resources** is a feature introduced in **Java 7** that simplifies resource management, such as closing files, sockets, database connections, etc. It ensures that resources implementing the AutoCloseable or Closeable interface are closed automatically at the end of the block, even if an exception occurs.

Syntax

```
try (ResourceType resource = new ResourceType()) {
    // Use the resource
} catch (Exception e) {
    e.printStackTrace();
}
```

Try-with-resources with JDBC (Database)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class TryWithResourcesJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/testdb";
        String user = "root";
        String password = "password";

        String query = "INSERT INTO users (name, email) VALUES (?, ?)";

        try (Connection conn = DriverManager.getConnection(url, user, password);
             PreparedStatement pstmt = conn.prepareStatement(query)) {

            pstmt.setString(1, "John Doe");
            pstmt.setString(2, "john@example.com");
            pstmt.executeUpdate();

            System.out.println("Record inserted successfully!");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

✔ Here, both Connection and PreparedStatement are automatically closed at the end of the try block.

Advantages of Try-with-Resources

- **Automatic resource closing**: No need for explicit close() calls.
- **Less boilerplate code**: Eliminates finally blocks for resource cleanup.
- **Exception safety**: Even if an exception occurs, the resources are closed properly.

Without Try-with-Resources (Manual Resource Management)

Here, we **manually** close the Connection and PreparedStatement inside a finally block.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class WithoutTryWithResourcesJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/testdb";
        String user = "root";
        String password = "password";

        String query = "INSERT INTO users (name, email) VALUES (?, ?)";

        Connection conn = null;
        PreparedStatement pstmt = null;

        try {
            conn = DriverManager.getConnection(url, user, password);
            pstmt = conn.prepareStatement(query);

            pstmt.setString(1, "Bob");
            pstmt.setString(2, "bob@example.com");
            pstmt.executeUpdate();

            System.out.println("Record inserted successfully!");
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (pstmt != null) pstmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

❌ Disadvantages of Manual Resource Management

- **More code**: You have to explicitly close each resource.
- **Risk of resource leaks**: If you forget to close resources, it can cause memory leaks.
- **Nested try-catch blocks**: Handling multiple resources manually makes code harder to read.

Custom AutoCloseable Resource

If you have a custom resource that needs proper closing, implement AutoCloseable:

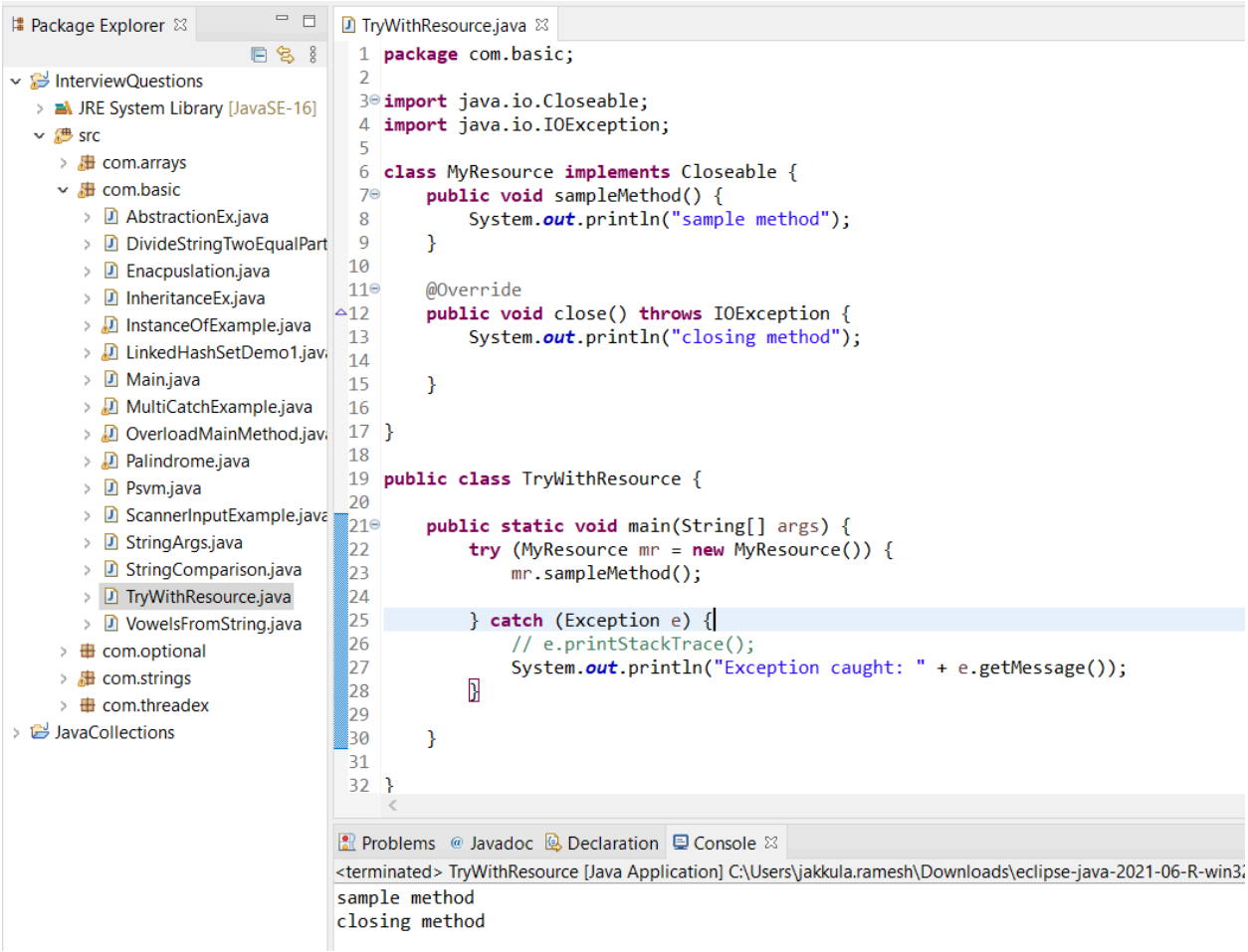
The MyResource class **overrides** the close() method from the AutoCloseable (or Closeable) interface.

```
class MyResource implements AutoCloseable / Closeable {
    public void doSomething() {
        System.out.println("Resource is in use...");
    }

    @Override
    public void close() {
        System.out.println("Resource closed.");
    }
}

public class CustomResourceExample {
    public static void main(String[] args) {
        try (MyResource resource = new MyResource()) {
```

```
            resource.doSomething();
        }
    }
}
```

✔️ Output:

```
Resource is in use...
Resource closed.
```



Key Points:

1. Only classes implementing AutoCloseable or Closeable can be used.

Multiple resources can be declared in a single try block.

```
try (Resource1 r1 = new Resource1(); Resource2 r2 = new Resource2()) {
    // Use resources
}
```

2.
3. Resources are closed in reverse order of declaration.
4. Avoid explicit close() calls inside try-with-resources since it's handled automatically.

21. SOLID Principles in Java

The **SOLID** principles are five key design principles that help in building **scalable, maintainable, and flexible** software. They are widely used in **Object-Oriented Programming (OOP)** and are fundamental for writing **clean, modular, and reusable** Java code.

🔷 SOLID Principles in Java

Principle	Description	Benefit
S - Single Responsibility Principle (SRP)	A class should have only **one reason to change** (one responsibility).	Increases maintainability and reduces complexity.
O - Open/Closed Principle (OCP)	Classes should be **open for extension, but closed for modification**.	Allows adding new functionality without modifying existing code.
L - Liskov Substitution Principle (LSP)	Subtypes should be **replaceable** for their base types **without breaking functionality**.	Ensures proper inheritance and avoids unexpected behaviors.
I - Interface Segregation Principle (ISP)	Clients should **not be forced to depend** on interfaces they **do not use**.	Promotes smaller, more specific interfaces.
D - Dependency Inversion Principle (DIP)	High-level modules should **not depend on low-level modules**. Both should depend on **abstractions**.	Improves flexibility and makes code loosely coupled.

1. Single Responsibility Principle (SRP)

👉 **A class should have only one reason to change** (only one responsibility).

❌ Bad Example (Violating SRP)

```
class Report {
    void generateReport() {
        System.out.println("Generating Report...");
    }

    void printReport() {
        System.out.println("Printing Report...");
    }
```

```
    void saveToDatabase() {
        System.out.println("Saving report to database...");
    }
}
```

🔴 **Problem:**

- This class **handles multiple responsibilities**: **generating, printing, and saving reports**.
- If we need to change the **database logic**, we have to modify this class.

✅ **Good Example (Following SRP)**

```
class ReportGenerator {
    void generateReport() {
        System.out.println("Generating Report...");
    }
}

class ReportPrinter {
    void printReport() {
        System.out.println("Printing Report...");
    }
}

class ReportRepository {
    void saveToDatabase() {
        System.out.println("Saving report to database...");
    }
}
```

✔ **Each class now has a single responsibility!**
✔ **More maintainable and scalable.**

2. Open/Closed Principle (OCP)

- **Open for extension** (you can add new functionality).
- **Closed for modification** (existing code should not be changed).

❌ **Bad Example (Violating OCP)**

```
class PaymentProcessor {
    void processPayment(String type) {
        if (type.equals("CreditCard")) {
            // Process credit card payment
        } else if (type.equals("PayPal")) {
            // Process PayPal payment
        }
    }
}
```

- If a **new payment type (e.g., Bitcoin)** is added, the class **must be modified**.
- This **violates OCP** because modifying existing code may introduce bugs.

✅ **Good Example (Following OCP)**

```
interface Payment {
    void pay();
}

class CreditCardPayment implements Payment {
    public void pay() { System.out.println("Paid via Credit Card"); }
}

class PayPalPayment implements Payment {
    public void pay() { System.out.println("Paid via PayPal"); }
}

class PaymentProcessor {
    void processPayment(Payment payment) {
        payment.pay();
    }
}
```

✔ **New payment methods can be added without modifying PaymentProcessor!**

3. Liskov Substitution Principle (LSP)

- It applies to inheritance in such a way that the derived(child) classes must be completely substitutable for their base(parent) classes. In other words, if class A is a subtype of class

B, then we should be able to replace B with A without interrupting the behavior of the program.

- **Subclasses should behave like their parent class without breaking functionality**.

This principle is a bit trickier than others, so let's go directly to some code examples.

```
package com.smartwatch;

public abstract class SmartWatch {

    public abstract void countSteps();

    public abstract void keepTrackOfHeartRate();

    public abstract void receiveNotification();

    public abstract void sendNotification();
}
```

Here we have an abstract class SmartWatch and a list of methods which represent what a smart watch can do. And let's imagine that we have two smartwatches – Apple Watch and Garmin. Let's start with an Apple Watch.

```
package com.smartwatch;

public class AppleWatch extends SmartWatch {

    @Override
    public void countSteps() {
        // business logic
    }

    @Override
    public void keepTrackOfHeartRate() {
        // business logic
    }

    @Override
    public void receiveNotification() {
        // business logic
    }

    @Override
    public void sendNotification() {
        // business logic
    }
}
```

An Apple Watch can do all the things that are represented by the methods in the abstract SmartWatch class, so we can say that both classes can replace each other without any issue for our program.

And now let's have a look at a Garmin Watch. A Garmin Watch can receive a notification, but can't send a notification (at least mine can't). So having this type of a program design violates The Liskov Substitution Principle. What is the solution here? The solution lies in having instead of one abstract class with a list of all possible features several interfaces with separate features.

```java
package com.smartwatch;

public interface StepCounter {

    void countSteps();
}
```

```java
package com.smartwatch;

public interface HeartRateTracker {

    void keepTrackOfHeartRate()
}
```

```java
package com.smartwatch;

public interface NotificationReceiver {

    void receiveNotification();
}
```

```java
package com.smartwatch;

public interface NotificationSender{

    void sendNotification();
}
```

And now our Apple Watch can simply implement all the interfaces and our Garmin watch can take only those interfaces which have feature available for it.

```java
package com.smartwatch;

public class Garmin implements StepCounter, HeartRateTracker,
NotificationReceiver {

    @Override
    public void countSteps() {
        // business logic
    }

    @Override
    public void keepTrackOfHeartRate() {
        // business logic
    }

    @Override
    public void receiveNotification() {
        // business logic
    }

}
```

Ex2:

Bad Example (Violating LSP)

In this example, we have a **Vehicle** base class with the methods fly(), move().
But **not all vehicles can fly (e.g., Car, Ship)**, so Car.fly() doesn't make sense!

Problem:

- If we replace a Vehicle with a Car, the program **breaks** because Car cannot fly.
- **Violates Liskov Substitution Principle!** ❌

```
// Parent class Vehicle
abstract class Vehicle {
    abstract void move();
    abstract void fly(); // 🚨 Not all vehicles can fly!
}

// Airplane can move and fly ✅
class Airplane extends Vehicle {
    @Override
    void move() {
        System.out.println("Airplane is moving on the runway...");
    }
```

```
    @Override
    void fly() {
        System.out.println("Airplane is flying...");
    }
}

// Car can move, but it CANNOT fly ❌
class Car extends Vehicle {
    @Override
    void move() {
        System.out.println("Car is driving on the road...");
    }

    @Override
    void fly() {
        throw new UnsupportedOperationException("Cars cannot fly!");
    }
}
// Testing the vehicles
public class LiskovViolation {
    public static void main(String[] args) {
        Vehicle airplane = new Airplane();
        Vehicle car = new Car();
        Vehicle ship = new Ship();

        airplane.move();
        airplane.fly(); // ✅ Works fine

        car.move();
        car.fly(); // ❌ Throws an exception! Violates LSP

        }
}
```

Good Example (Following LSP)

Solution:

✔ Instead of a single Vehicle class, we create separate interfaces for Flying and Non-Flying vehicles.
✔ Now, only the Airplane implements Flyable, while Car and Ship only have Movable.

```
// Interface for all vehicles that can move
interface Movable {
    void move();
}

// Interface for vehicles that can fly
interface Flyable {
    void fly();
}

// Airplane can move and fly ✅
class Airplane implements Movable, Flyable {
    @Override
    public void move() {
        System.out.println("Airplane is moving on the runway...");
    }

    @Override
    public void fly() {
        System.out.println("Airplane is flying...");
    }
}

// Car can only move ✅
class Car implements Movable {
    @Override
    public void move() {
        System.out.println("Car is driving on the road...");
    }
}

// Testing the vehicles
public class LiskovCorrect {
    public static void main(String[] args) {
        Movable airplane = new Airplane();
        Movable car = new Car();
        Movable ship = new Ship();
```

```
        Flyable flyingAirplane = new Airplane(); // Only airplanes can fly ✅

        airplane.move();
        flyingAirplane.fly(); // ✅ Works correctly

        car.move(); // ✅ Works correctly, no unnecessary fly() method
    }
}
```

✔ Now, Car and Ship don't have fly() at all, so they follow Liskov Substitution Principle! ✅
✔ Replacing Movable with Car, Ship, or Airplane does not break the code. ✅

4. Interface Segregation Principle (ISP)

A class should not be forced to implement interfaces it does not use.

Instead of **one big interface**, create **multiple smaller interfaces** to ensure that classes only implement what they actually need.

Bad Example (Violating ISP)

Worker interface forces **both developers and robots** to implement eat(),
but robots **don't eat**!

```
interface Worker {
    void work();
    void eat(); // ❌ Robots don't eat!
}


class HumanWorker implements Worker {
    public void work() {
        System.out.println("Human is working...");
    }
    public void eat() {
```

```
        System.out.println("Human is eating...");
    }
}
class RobotWorker implements Worker {
    public void work() {
        System.out.println("Robot is working...");
    }

    public void eat() { // ❌ Robots don't eat!
        throw new UnsupportedOperationException("Robots don't eat!");
    }
}
```

✅ **Good Example (Following ISP)**

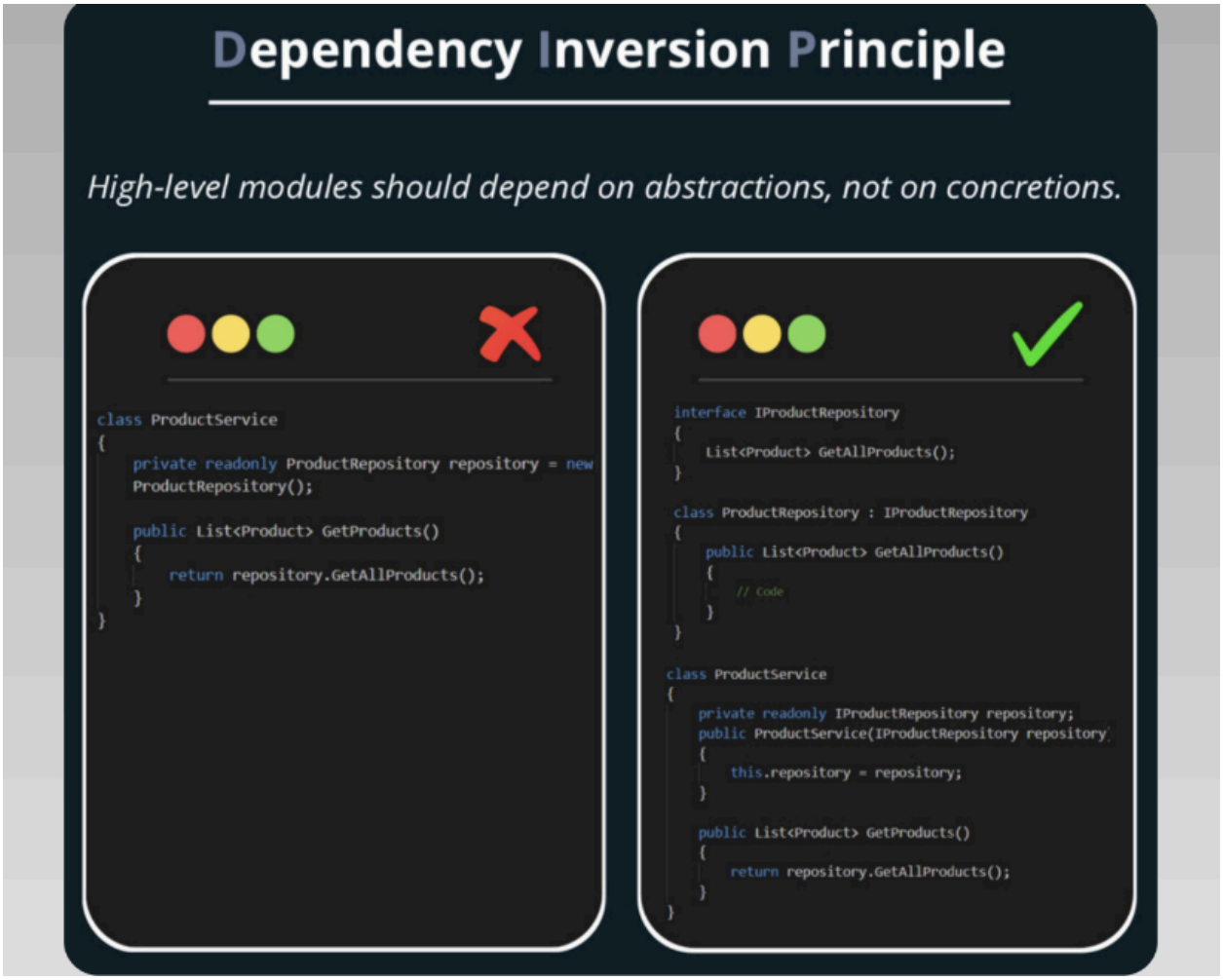
✔ We create **separate interfaces** for eating and working.
✔ **Robots only implement Workable** (no eat()).

```
interface Workable {
    void work();
}
interface Eatable {
    void eat();
}
class HumanWorker implements Workable, Eatable {
    public void work() {
```

```
        System.out.println("Human is working...");
    }

    public void eat() {
        System.out.println("Human is eating...");
    }
}
class RobotWorker implements Workable {
    public void work() {
        System.out.println("Robot is working...");
    }
}
```

5. Dependency Inversion Principle (DIP)

👉 **High-level modules should not depend on low-level modules.**
👉 We should depend on abstractions (interfaces and abstract classes) instead of concrete implementations (classes).



Payment Processing System

🔴 **Bad Example (Violating DIP)**
The ShoppingCart class **depends directly** on CreditCardPayment.
If we want to **add PayPal or UPI payment methods**, we must **modify ShoppingCart**.

```
class CreditCardPayment {
    void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card");
    }
}

class ShoppingCart {
    private CreditCardPayment payment; // ❌ Direct dependency on a concrete class

    ShoppingCart() {
```

```
        this.payment = new CreditCardPayment();
    }

    void checkout(double amount) {
        payment.pay(amount);
    }
}
```

🔵 **Good Example (Following DIP)**
✔ ShoppingCart depends on an **abstraction (Payment)**, not a concrete class.
✔ We can **easily add PayPalPayment, UPIPayment, etc., without modifying ShoppingCart**.

```
// Abstraction
interface Payment {
    void pay(double amount);
}

// Concrete implementations
class CreditCardPayment implements Payment {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card");
    }
}

class PayPalPayment implements Payment {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal");
    }
}

class ShoppingCart {
    private Payment payment; // ✅ Depends on abstraction, not a concrete class

    ShoppingCart(Payment payment) {
        this.payment = payment;
    }

    void checkout(double amount) {
        payment.pay(amount);
    }
```

```
}

// Usage
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart1 = new ShoppingCart(new CreditCardPayment());
        cart1.checkout(100);

        ShoppingCart cart2 = new ShoppingCart(new PayPalPayment());
        cart2.checkout(200);
    }
}
```

22.How micro services communicate to each other

Microservices communicate with each other primarily through inter-service communication mechanisms. The two main types of communication are:

- **REST (HTTP) APIs**: Services communicate using HTTP requests (GET, POST, PUT, DELETE) over RESTful endpoints.

REST API Communication Between Microservices

1. Exposing a REST API (Service A)

A Spring Boot microservice exposing a REST endpoint:

```
@RestController
@RequestMapping("/orders")
public class OrderController {
    @GetMapping("/{orderId}")
    public ResponseEntity<Order> getOrder(@PathVariable String orderId) {
        Order order = new Order(orderId, "Product XYZ", 2);
        return ResponseEntity.ok(order);
    }
```

```
}
```

- URL: http://order-service/orders/123
- Returns order details for orderId=123.

---

2. Consuming the API in Another Microservice (Service B)

Service B calls the REST API of Service A using **RestTemplate**:

```
@Service
public class OrderService {

    private final RestTemplate restTemplate = new RestTemplate();

    public Order fetchOrderDetails(String orderId) {
        String url = "http://order-service/orders/" + orderId;
        return restTemplate.getForObject(url, Order.class);
    }
}
```

If a microservice is taking too long to communicate with other microservices, it can lead to performance bottlenecks and degraded user experience. Here's how you can handle such situations:

Implement Timeouts

- Set **reasonable timeouts** for inter-service calls to prevent long waiting periods.

Example (Spring Boot + REST Template):

```
HttpComponentsClientHttpRequestFactory factory = new
HttpComponentsClientHttpRequestFactory();
```

```
factory.setConnectTimeout(5000); // 5 seconds

factory.setReadTimeout(5000); // 5 seconds

RestTemplate restTemplate = new RestTemplate(factory);
```

Use Circuit Breaker Pattern

- Prevent cascading failures when a service is slow or unavailable.

Example using **Resilience4j**:

```
@CircuitBreaker(name = "myService", fallbackMethod = "fallbackResponse")

public String callAnotherService() {

   return restTemplate.getForObject("http://slow-service/api/data", String.class);

}


public String fallbackResponse(Exception e) {

   return "Fallback response due to service delay!";

}
```

- Popular Circuit Breaker Libraries:
    - **Resilience4j** (Java)
    - **Netflix Hystrix** (Deprecated but still used)
    - **Istio Service Mesh** (For Kubernetes)

For above rest template example we can achieve **Timeouts and Resilience 4j** as below

Here's how you can **add timeouts and Resilience4j Circuit Breaker** to your REST API communication in a microservices setup.

1. Setting Timeout in RestTemplate

You should configure a timeout to avoid long waits if the dependent service is slow or unresponsive.

Using RestTemplate with Timeout

Modify OrderService to set timeouts:

```
@Bean
public RestTemplate restTemplate() {
 HttpComponentsClientHttpRequestFactory factory = new
HttpComponentsClientHttpRequestFactory();
  factory.setConnectTimeout(5000); // 5 seconds connection timeout
  factory.setReadTimeout(5000); // 5 seconds read timeout
   return new RestTemplate(factory);
}
```

Then use it in the service:

```
@Service
public class OrderService {
   private final RestTemplate restTemplate;
   @Autowired
   public OrderService(RestTemplate restTemplate) {
      this.restTemplate = restTemplate;
   }
   public Order fetchOrderDetails(String orderId) {
      String url = "http://order-service/orders/" + orderId;
      return restTemplate.getForObject(url, Order.class);
   }
}
```

Implementing Resilience4j Circuit Breaker

Resilience4j helps prevent cascading failures when a service is slow or unavailable.

Add Dependencies (if using Spring Boot)

In pom.xml, add:

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>1.7.1</version>
</dependency>
```

Enable Circuit Breaker in OrderService

Modify fetchOrderDetails() to use Resilience4j:

```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
@Service
public class OrderService {
    private final RestTemplate restTemplate;
    @Autowired
    public OrderService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }
    @CircuitBreaker(name = "orderService", fallbackMethod = "fallbackOrderDetails")
    public Order fetchOrderDetails(String orderId) {
```

```
        String url = "http://order-service/orders/" + orderId;
        return restTemplate.getForObject(url, Order.class);
    }
    public Order fallbackOrderDetails(String orderId, Throwable throwable) {
        System.out.println("Fallback called due to: " + throwable.getMessage());
        return new Order(orderId, "Fallback Product", 0);
    }
}
```

23.HoriZontal and Vertical Scaling

What is Scaling in Microservices?

Scaling refers to the ability of a system to **handle increased load** by adjusting its resources. When a microservice experiences high traffic or demand, it must scale efficiently to maintain performance and availability.

There are **two primary types of scaling**:

1. **Vertical Scaling (Scaling Up)**
2. **Horizontal Scaling (Scaling Out)**

Vertical Scaling (Scaling Up)

- **Definition**: Increasing the computing power (CPU, RAM, storage) of a **single server or instance** to handle more load.
- **How It Works**:
  - Upgrade the hardware of the existing server.
  - Increase memory, processing power, or disk space.
- **Example**:
  - A database server is running slowly, so we **upgrade** from 8GB RAM to 32GB RAM.
- **Applications**:
  - Useful for **monolithic applications** and databases where moving to multiple instances is complex.
- **Limitations**:

    - There is a **hardware limit** (e.g., a server can only have so much RAM).
    - **Downtime** may be required for upgrades.

Example of Vertical Scaling in a Database

Before Scaling (Small Instance)

- **DB Server**: 2 CPU, 4GB RAM
- **Issue**: High query load, slow performance

After Scaling (Larger Instance)

- **DB Server**: 8 CPU, 32GB RAM
- **Result**: Faster query execution, improved performance

Horizontal Scaling (Scaling Out)

- **Definition**: Adding **multiple instances** (or nodes) of a service to distribute the load.
- **How It Works**:
    - Instead of making one server bigger, you add **more servers**.
    - A **load balancer** distributes requests among the instances.
- **Example**:
    - A web application experiencing high traffic deploys **more instances** behind a load balancer.
- **Applications**:
    - **Microservices architectures** where each service runs independently.
    - **Cloud-based applications** that need to scale dynamically.
- **Advantages**:
    - **No hardware limits** (can keep adding instances).
    - **Fault tolerance** (if one instance fails, others handle traffic).
    - **Better cost efficiency** (only add instances when needed).
- **Limitations**:
    - Requires **load balancing**.
    - **More complex** than vertical scaling (needs service discovery, session management).

Example of Horizontal Scaling in a Web Application

Before Scaling (Single Instance)

- A web application is running on **one server**.
- **Issue**: High traffic slows it down.

After Scaling (Multiple Instances)

- The app is deployed on **five instances** behind a **load balancer**.
- **Result**: Requests are distributed evenly, improving speed.

24.HIghest Number from list using Java8 various ways

```
public class HighestNumber {

	public static void main(String[] args) {

		List<Integer> list = Arrays.asList(3,2,2,3,7,9,5);

		Optional<Integer> high =
list.stream().sorted(Comparator.reverseOrder()).findFirst();

		System.out.println(high.get());

		//or

		Integer mx = list.stream().max((i,j)->i.compareTo(j)).get();

		System.out.println(mx);

	}

}
```

9

9

25.How to convert Stream to array

We can convert a **Stream** to an **array** using the .toArray() method

```
List<Integer> li = List.of(12, 13, 1, 2, 6, 8, 9, 9, 1, 4);

		//converting stream to array

		Object[] array = li.stream().toArray();

		  // Print the array

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

26.what is fail-fast and fail-safe in java

In Java, **fail-fast** and **fail-safe** are terms used to describe the behavior of collections when they are modified during iteration. Specifically, they determine how an iterator responds when the underlying collection is changed (e.g., adding, removing elements) while iterating over it.

1. Fail-Fast

A **fail-fast** iterator immediately throws a **ConcurrentModificationException** if the underlying collection is modified during iteration, except through the operator's own methods (iterator.remove();).

Behavior:

- **Fail-fast** iterators detect structural changes made to the collection during iteration and throw an exception to avoid unpredictable behavior.
- They usually work by checking for **modification counts** during iteration. If the modification count has changed since the creation of the iterator, an exception is thrown.

Common Collections with Fail-Fast Iterators:

- **ArrayList**
- **HashMap**
- **HashSet**
- **LinkedList**

Example of Fail-Fast:

```
import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class FailFastExample {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<>();

        list.add(1);

        list.add(2);
```

```
        list.add(3);

        Iterator<Integer> iterator = list.iterator();

        while (iterator.hasNext()) {
            Integer value = iterator.next();
            System.out.println(value);
            // Modifying the collection during iteration
            list.remove(1);  // This will cause a ConcurrentModificationException
        }
    }
}
```

Output:

```
1
2
Exception in thread "main" java.util.ConcurrentModificationException
```

In this example, when the collection (list) is modified (list.remove(1)) during iteration, the iterator detects the modification and throws a ConcurrentModificationException.

---

2. Fail-Safe

A **fail-safe** iterator allows modifications to the underlying collection during iteration without throwing an exception. Instead, it works on a **copy** of the collection, ensuring that changes to the original collection do not affect the iteration.

Behavior:

- **Fail-safe** iterators allow structural changes to the collection without throwing an exception, because they operate on a **clone** or a **snapshot** of the collection.
- **Fail-safe** is typically slower than **fail-fast** because it involves copying or snapshotting the collection.
- These iterators do not throw ConcurrentModificationException.

Common Collections with Fail-Safe Iterators:

- **CopyOnWriteArrayList**
- **CopyOnWriteArraySet**
- **ConcurrentHashMap**

Example of Fail-Safe:

```
import java.util.concurrent.CopyOnWriteArrayList;

public class FailSafeExample {

    public static void main(String[] args) {

        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();

        list.add(1);

        list.add(2);

        list.add(3);


        for (Integer value : list) {

            System.out.println(value);

            // Modifying the collection during iteration

            list.remove(1);  // No exception will be thrown

        }

    }

}
```

Output:

1

2

3

In this example, the collection (CopyOnWriteArrayList) can be modified (list.remove(1)) during iteration without throwing a ConcurrentModificationException, because the iterator works on a **snapshot** or **copy** of the collection.

27. Generate Random numbers using Java 8

Random in Java 8

The Random class in Java is part of the java.util package and is used to generate random numbers of different data types like int, double, long, float, and boolean. Java 8 introduced new methods to generate streams of random numbers.

Syntax

Random random = new Random();

By default, Random() uses the current time as the seed.

You can also provide a seed for predictable random numbers:
Random random = new Random(12345); // Seeded for reproducibility

Parameters Accepted

The Random class provides several methods that accept different parameters:

1. **nextInt()** - Returns a random integer.
    - nextInt(int bound) → Generates a random integer between 0 (inclusive) and bound (exclusive).
2. **nextDouble()** - Returns a random double between 0.0 and 1.0 (exclusive).
3. **nextLong()** - Returns a random long value.

4. **nextBoolean()** - Returns true or false.

5. **Stream Methods (Java 8)**

    - ints() → Generates an IntStream of random integers.
    - longs() → Generates a LongStream of random long values.
    - doubles() → Generates a DoubleStream of random double values.

Examples

1. Generating Random Number

```java
package com.random;

import java.util.Random;

public class RandomExample2 {

    public static void main(String[] args) {

        Random random = new Random();
        System.out.println("Random Integer: " + random.nextInt(10,100));
        System.out.println("Random Integer (0 to 99): " + random.nextInt(100));
        System.out.println("Random Double: " + random.nextDouble());
        System.out.println("Random Boolean: " + random.nextBoolean());
    }

}
```

Javadoc Declaration Console

```
<terminated> RandomExample2 [Java Application] C:\ProgramData\chocolatey\lib\springtoolsuite\tools\sts-4.23.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.wi
Random Integer: 25
Random Integer (0 to 99): 24
Random Double: 0.7489410691072029
Random Boolean: false
```

2. Generating a Stream of Random Numbers (Java 8)

Generating Random Streams (Java 8 Feature): Java 8 introduced methods to generate streams of random numbers:

a) Generating a Stream of Random Integers

```
random.ints(5).forEach(System.out::println); // Generates 5 random int values
```

b) Generating Random Numbers in a Specific Range

```
random.ints(5, 10, 20).forEach(System.out::println); // Generates 5 numbers between 10 and 19
```

- The ints(count, origin, bound) method generates count random numbers between origin (inclusive) and bound (exclusive).

RandomExample.java

```java
package com.random;

import java.util.Random;

public class RandomExample {

    public static void main(String[] args) {
        Random rand = new Random();
        rand.ints(5,10,100).forEach(System.out::println);
        //if you want sort them
        //rand.ints(5,10,100).sorted().forEach(System.out::println);
    }
}
```

Javadoc Declaration Console

```
<terminated> RandomExample [Java Application] C:\ProgramData\chocolatey\lib\springtoolsuite\tools\sts-4.23.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_2
12
61
44
98
18
```

Or

RandomExample.java RandomExample2.java

```java
package com.random;

import java.util.Random;
import java.util.stream.IntStream;

public class RandomExample {

    public static void main(String[] args) {
        Random rand = new Random();

        IntStream randomInts = rand.ints(5, 0, 100);
        randomInts.forEach(System.out::println);
    }
}
```

Javadoc Declaration Console

```
<terminated> RandomExample [Java Application] C:\ProgramData\chocolatey\lib\springtoolsuite\tools\sts-4.23.1.RELEASE\plugins\org.eclipse.justj.openjdk
0
49
24
71
99
```

Conclusion

- The Random class in Java 8 provides methods to generate random numbers of different types.
- Java 8 introduced stream-based methods (ints(), longs(), doubles()) for efficient random number generation.
- The class can be initialized with or without a seed, where using a seed ensures reproducible result

28.What is a Default Method in Java?

A **default method** in Java is a method inside an **interface** that has a concrete implementation. It was introduced in **Java 8** to allow adding new methods to existing interfaces **without breaking** the classes that implement them. We can add as many as default methods in the interface.

```
interface Vehicle {
```

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

Classes implementing Vehicle will inherit the start() method without needing to override it.

Why Are Default Methods Required?

Before Java 8, interfaces could only contain **abstract methods**, meaning every class implementing the interface had to define all its methods. However, this made it hard to **add new methods** to existing interfaces without breaking backward compatibility.

Default methods solve this problem by allowing interfaces to have new methods with a default implementation, ensuring old classes still work **without modification**.

29.Java's Concurrency Package (java.util.concurrent)

Java's **java.util.concurrent** package provides a set of **thread-safe** classes and utilities to simplify concurrent programming. It was introduced in **Java 5** to improve performance and avoid low-level synchronization issues with synchronized, wait(), and notify().

Key Components of java.util.concurrent

1 Concurrent Collections

Thread-safe alternatives to traditional collections (`ArrayList`, `HashMap`, etc.).

Class	Description
`ConcurrentHashMap<K, V>`	High-performance thread-safe `HashMap` with bucket-level locking.
`CopyOnWriteArrayList<E>`	A thread-safe list where writes create a new copy, useful for infrequent updates.
`CopyOnWriteArraySet<E>`	Similar to `CopyOnWriteArrayList`, but for sets.
`ConcurrentLinkedQueue<E>`	A non-blocking queue based on linked nodes.
`LinkedBlockingQueue<E>`	A blocking queue with a linked list implementation.
`PriorityBlockingQueue<E>`	A thread-safe priority queue.

✅ Example: Using ConcurrentHashMap

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

```
public class ConcurrentMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        map.put("A", 1);
        map.put("B", 2);
        System.out.println(map.get("A")); // Output: 1
    }
}
```

Both ConcurrentHashMap and HashMap are used to store key-value pairs, but they differ in **thread safety**, **performance**, and **use cases**.

What is HashMap?

A **HashMap<K, V>** is a non-thread-safe key-value data structure that allows **one null key and multiple null values**.

✅ **Key Features of HashMap**

- 🚀 **Fast & Efficient** for single-threaded operations.
- ❌ **Not Thread-Safe** – If multiple threads modify it, ConcurrentModificationException can occur.
- ✅ **Allows null keys and values** (null keys: 1, null values: multiple).
- 🐢 **Performance degrades** in multi-threaded environments due to race conditions.

```
import java.util.*;
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(1, "A");
        map.put(2, "B");
        map.put(3, "C");
```

```
        System.out.println(map.get(2)); // Output: B
    }
}
```

Problem with HashMap in Multithreading

```
import java.util.*;
public class HashMapConcurrencyIssue {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();

        // Thread 1
        new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                map.put(i, "Value" + i);
            }
        }).start();

        // Thread 2
        new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println(map.get(i));
            }
        }).start();
    }
```

}

❌ Problem

- Race Condition: Multiple threads reading & writing simultaneously cause data corruption.
- ConcurrentModificationException can occur if one thread iterates while another modifies.

What is ConcurrentHashMap?

A **ConcurrentHashMap<K, V>** is a thread-safe version of HashMap from java.util.concurrent. It **allows multiple threads** to read and write **without blocking the entire map**.

✅ Key Features of ConcurrentHashMap

- ✅ Thread-Safe – Handles multiple threads efficiently.
- 🚀 High Performance – Uses bucket-level locking (Java 8).
- ❌ No null keys or null values.
- ✅ Fail-Safe Iterator – No ConcurrentModificationException.

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

public class ConcurrentHashMapExample {
   public static void main(String[] args) {
      ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
      map.put("A", 1);
      map.put("B", 2);

      System.out.println(map.get("A")); // Output: 1
   }
}
```

30.computeIfAbsent Method in ConcurrentHashMap

Syntax of computeIfAbsent() Method

public void computeIfAbsent(K key, Function mappingFunction)

Parameters:

- **Key**: The key whose associated value is to be calculated.
- **mappingFunction**: It is a function that computes the value if the key is not present.

Return Type:

- If the key is present, then it returns the existing value.
- If the key is not present, it calculates the value using mappingFunction and then returns the calculated value.

Basic Usage

```
import java.util.concurrent.ConcurrentHashMap;

public class ComputeIfAbsentExample {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

        // Compute and insert only if key 1 is absent
        String value1 = map.computeIfAbsent(1, key -> "GeneratedValue-" + key);
        System.out.println("Key 1: " + value1);  // Output: GeneratedValue-1

        // Since key 1 now exists, it will return the existing value
        String value2 = map.computeIfAbsent(1, key -> "NewValue-" + key);
        System.out.println("Key 1 after second compute: " + value2);  // Output: GeneratedValue-1
    }
}
```

✅ **Explanation**

- Key 1 is **absent**, so it computes and inserts "GeneratedValue-1".
- When computeIfAbsent is called again with key 1, the value **remains unchanged**.

31.Java 8 stream code filters numbers that start with "1"

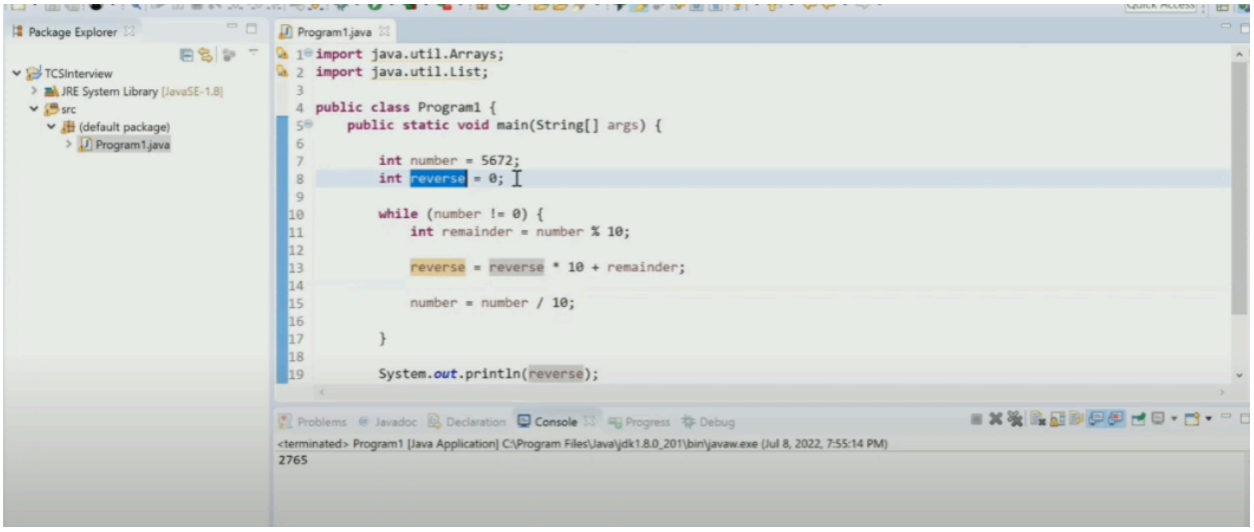
```
package com.basic;

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

```
public class Program1 {
	public static void main(String[] args) {

		List<Integer> numbers = Arrays.asList(10, 12, 50, 30, 18, 19);

		numbers.stream().map(s -> s + "") // Convert numbers to strings
				.filter(s -> s.startsWith("1")) // Keep numbers starting with "1"
				.forEach(System.out::println); // Print the results
	}
}
```

10
12
18
19

map(s -> s + "") → Converts each number to a String.
filter(s -> s.startsWith("1")) → Filters numbers that start with "1".
forEach(System.out::println) → Prints the filtered numbers.

Reverse of number



32. GroupBy and highest-paid employee from each department

Collectors.groupingBy method to group a list of Employee objects by department. Your code successfully organizes employees into a Map<String, List<Employee>>, where the key is the department name, and the value is a list of employees in that department.

To get the highest-paid employee from each department, you can use Collectors.groupingBy along with Collectors.maxBy and Comparator.comparingInt

```
package com.interviewquestions;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;
public class GrouByExample {
        public static void main(String[] args) {
     List<Employeee> employees = Arrays.asList(
        new Employeee("Emp1", "CS", 10000),
        new Employeee("Emp2", "CS", 15000),
        new Employeee("Emp3", "IT", 20000),
        new Employeee("Emp4", "IT", 25000)
     );
     Map<Object, Optional<Employeee>> byDepartment = employees.stream()
        .collect(Collectors.groupingBy(e ->
e.getDepartment(),Collectors.maxBy(Comparator.comparingInt(a->a.getSalary()))));
     System.out.println(byDepartment);
  }
}
```

{CS=Optional[Employee [name=Emp2, department=CS, salary=15000]],
IT=Optional[Employee [name=Emp4, department=IT, salary=25000]]}

```
groupingBy(e ->
e.getDepartment(),Collectors.maxBy(Comparator.comparingInt(a->a.getSalary())))
```

- Groups employees by department.
- Uses Collectors.maxBy to get the employee with the highest salary in each department.

Optional<Employee> Handling

- Since maxBy returns an Optional<Employee>, we use .orElse(null) while printing to avoid Optional wrapping in output.

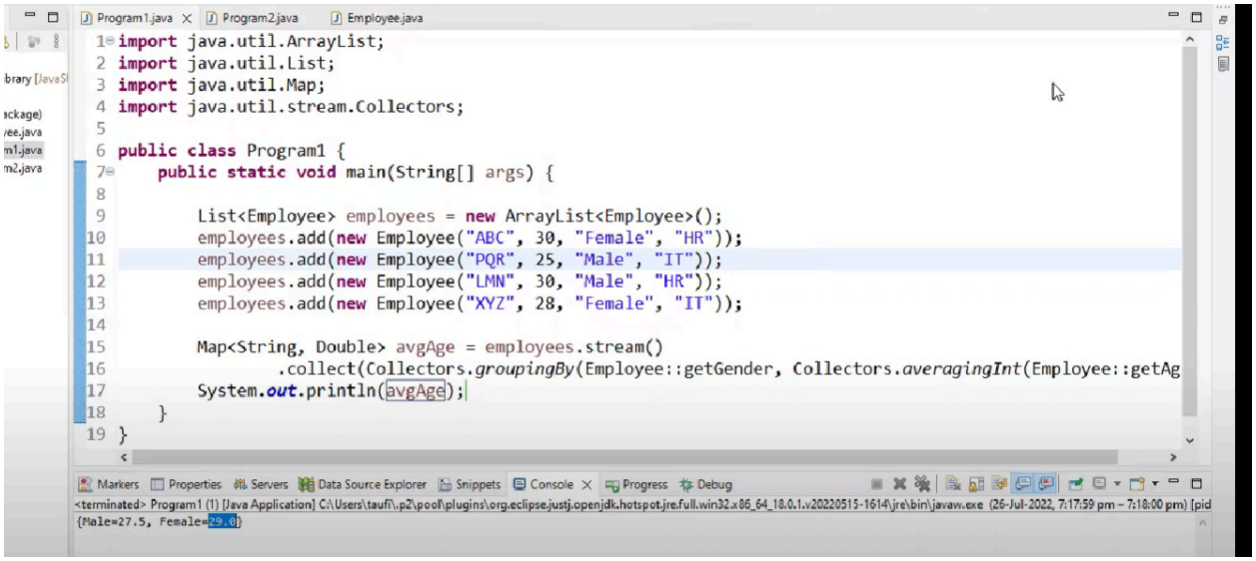
33.Counting Employees in List and In the respective Department.

```
package com.interviewquestions;
```

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;
public class GrouByExample {
        public static void main(String[] args) {
      List<Employeee> employees = Arrays.asList(
         new Employeee("Emp1", "CS", 10000),
         new Employeee("Emp2", "CS", 15000),
         new Employeee("Emp3", "IT", 20000),
         new Employeee("Emp4", "IT", 25000)
      );
      //print only the various  department name

      employees.stream().map(e->e.getDepartment()).distinct().forEach(s->System.out.println(s));
//CS
IT


      //count of total employees in List

      long count= employees.stream().count();
      System.out.println(count); //4

      //counting the employees in each department

     Map<String,Long> countofEmployeeinDepartment =
employees.stream().collect(Collectors.groupingBy(e->e.getDepartment(),Collectors.counting()));
      System.out.println(countofEmployeeinDepartment); //{CS=2, IT=2}

   }
}



CS
IT
4
```

{CS=2, IT=2}

And also Average of ages of genders



GroupBy and **AveragingInt** Method used here

34.Demonstrating the difference between Stream and Parallel Stream

Difference Between `Stream` and `ParallelStream`

In Java, both `Stream` and `ParallelStream` are used for processing collections, but they differ in how they execute tasks:

Feature	`stream()` (Sequential Stream)	`parallelStream()` (Parallel Stream)
Execution	Processes elements sequentially, one by one	Splits the task into multiple threads for parallel execution
Performance	Suitable for small datasets or operations requiring order	Better for large datasets, CPU-intensive tasks
Threading	Uses a single thread (main thread)	Uses the **ForkJoinPool** (common thread pool)
Ordering	Preserves order of elements	Order is not guaranteed
Best Use Case	When ordering matters or operations are lightweight	When working with large collections and expensive computations

EX:

Demonstrating the Difference

Let's process a list of numbers using **both sequential and parallel streams** and observe the difference.

```
package parallelstream;
import java.util.Arrays;
import java.util.List;
public class StreamExample {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    // Sequential Stream
    System.out.println("Sequential Stream Output:");
    numbers.stream()
      .forEach(num -> System.out.println(Thread.currentThread().getName() + " -> " + num));
    System.out.println("\nParallel Stream Output:");
    // Parallel Stream
    numbers.parallelStream()
      .forEach(num -> System.out.println(Thread.currentThread().getName() + " -> " + num));
  }
}
```

Sequential Stream Output:

```
main -> 1
main -> 2
main -> 3
main -> 4
main -> 5
main -> 6
main -> 7
main -> 8
main -> 9
main -> 10
```

Parallel Stream Output:

```
main -> 7
main -> 6
main -> 9
main -> 10
ForkJoinPool.commonPool-worker-3 -> 8
main -> 1
```

ForkJoinPool.commonPool-worker-1 -> 3
main -> 4
ForkJoinPool.commonPool-worker-3 -> 5

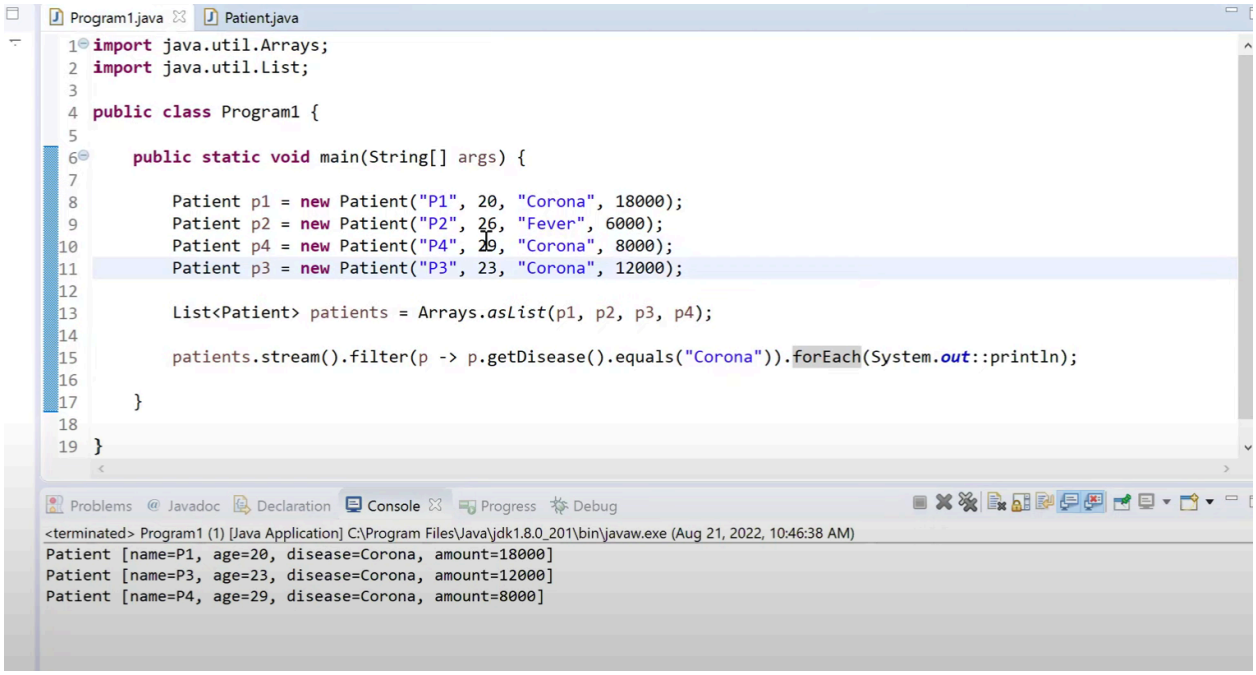
ForkJoinPool.commonPool-worker-2 -> 2

Sequential Stream executes in the **main thread**.

Parallel Stream splits tasks across **multiple worker threads** (e.g., main thread ,ForkJoinPool.commonPool-worker-1, ForkJoinPool.commonPool-worker-2, ForkJoinPool.commonPool-worker-3 ans etc..).

35. Java8 Filter method example

One filter conditions



Average

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

public class Program1 {

    public static void main(String[] args) {

        Patient p1 = new Patient("P1", 20, "Corona", 18000);
        Patient p2 = new Patient("P2", 26, "Fever", 6000);
        Patient p4 = new Patient("P4", 29, "Corona", 8000);
        Patient p3 = new Patient("P3", 23, "Corona", 12000);

        List<Patient> patients = Arrays.asList(p1, p2, p3, p4);

        Double averageBillPaid = patients.stream().filter(p -> p.getDisease().equals("Corona"))
                .collect(Collectors.averagingDouble(Patient::getAmount));

        System.out.println("Average Bill Paid " + averageBillPaid);
```

Problems Javadoc Declaration Console Progress Debug

<terminated> Program1 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_201\bin\javaw.exe (Aug 21, 2022, 10:50:04 AM)

```
Average Bill Paid 12666.666666666666
```

More than one

Program1.java Patient.java

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

public class Program1 {

    public static void main(String[] args) {

        Patient p1 = new Patient("P1", 20, "Corona", 18000);
        Patient p2 = new Patient("P2", 26, "Fever", 6000);
        Patient p4 = new Patient("P4", 29, "Corona", 8000);
        Patient p3 = new Patient("P3", 23, "Corona", 12000);

        List<Patient> patients = Arrays.asList(p1, p2, p3, p4);

        patients.stream().filter(p -> p.getDisease().equals("Corona") && p.getAge() < 25).forEach(System.out::println

    }
}
```

Problems Javadoc Declaration Console Progress Debug

<terminated> Program1 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_201\bin\javaw.exe (Aug 21, 2022, 10:47:38 AM)

```
Patient [name=P1, age=20, disease=Corona, amount=18000]
Patient [name=P3, age=23, disease=Corona, amount=12000]
```

36.How to Call a Default Method in an Interface?

Java 8 introduced **default methods** in interfaces, which provide method implementation.

Example:

```java
package com.interview;
interface Vehicle {
  default void start() {
    System.out.println("Vehicle is starting...");
  }
}
class Car implements Vehicle {}
public class Main {
  public static void main(String[] args) {
    Car car = new Car();
    car.start();  // Calls the default method
  }
}
```

The default method **start()** is inherited by the Car class.

37.ResponseEntity in Spring Boot

ResponseEntity<T> represents an HTTP response including **status code, headers, and body**.

Example:

```java
package com.interview;
@GetMapping("/employee/{id}")
public ResponseEntity<Employee> getEmployee(@PathVariable Long id) {
  Employee emp = employeeService.findById(id);
  if (emp == null) {
    return ResponseEntity.notFound().build();
  }
  return ResponseEntity.ok(emp);
}
```

✅ If the employee exists, it returns **200 OK** with the object.
✅ If not, it returns **404 Not Found**.

Agile Workflow

1. **Sprint Planning** - Define goals
2. **Daily Standups** - Short updates

3. **Development & Testing** - Implement features
4. **Sprint Review** - Demo
5. **Retrospective** - Improve process

38.JPA - Entity Relationships

Mapping Entity to Database Columns

@JoinColumn and @MappedBy Usage

Entity mapping is at the core of the Java Persistence API (JPA) and allows developers to map Java objects (entities) to database tables.

@ManyToOne Relation
@OneToMany Relation
@OneToOne Relation
@ManyToMany Relation

@ManyToOne Relationship

Definition:

- Many records of **one entity** are related to **one record** of another entity.
- This is a **foreign key relationship**.

Example:

- Many **employees** belong to **one department**.
  (*An employee can belong to only one department, but a department can have many employees.*)

Implementation:

```
package com.interview;
@Entity
class Department {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
```

```
}
@Entity
class Employee {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  @ManyToOne
  @JoinColumn(name = "department_id")  // Foreign Key
  private Department department;
}
```

🔹 **Key Points:**

- The @ManyToOne annotation in Employee establishes the relationship.
- The @JoinColumn(name = "department_id") makes it a **foreign key** in the Employee table.

@OneToMany Relationship (Inverse of @ManyToOne)

Definition:

- **One record** of an entity is related to **many records** of another entity.
- It is the **inverse side** of @ManyToOne.

Example:

- A **department** has many **employees**.

Implementation:

```
package com.interview;
@Entity
class Department {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
  private List<Employee> employees;
}
```

🔹 **Key Points:**

- mappedBy = "department" tells JPA that **this is the inverse side** of @ManyToOne.
- cascade = CascadeType.ALL ensures that when a department is deleted, its employees are deleted too.

@OneToOne Relationship (Unique Pairing)

Definition:

- **Each record** in one entity is related to **exactly one record** in another entity.

Example:

- One **employee** has **one employee ID card**.

Implementation:

```
package com.interview;
@Entity
class Employee {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  @OneToOne
  @JoinColumn(name = "card_id")  // Foreign Key
  private EmployeeCard employeeCard;
}
@Entity
class EmployeeCard {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String cardNumber;
}
```

🔹 **Key Points:**

- @OneToOne is used for **unique** relationships.
- The @JoinColumn(name = "card_id") makes EmployeeCard the **foreign key** in the Employee table

Relationship	Meaning	Example	Foreign Key In
@ManyToOne	Many → One	Many Employees → One Department	Employee Table (`department_id`)
@OneToMany	One → Many	One Department → Many Employees	Inverse of `@ManyToOne`
@OneToOne	One ↔ One	One Employee ↔ One ID Card	Employee Table (`card_id`)
@ManyToMany	Many ↔ Many	Many Students ↔ Many Courses	**New Join Table** (`student_course`)

1️⃣ Student Entity (@ManyToOne, @OneToOne, @ManyToMany)

```
package com.interview;
import jakarta.persistence.*;
import java.util.List;
@Entity
public class Student {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  // Many students belong to one department
  @ManyToOne
  @JoinColumn(name = "department_id")
  private Department department;
  // One-to-One relationship with Student ID Card
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "id_card_id")
  private StudentIdCard studentIdCard;
  // Many-to-Many relationship with Courses
  @ManyToMany
  @JoinTable(
    name = "student_course",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id")
  )
  private List<Course> courses;
  // Constructors, Getters, and Setters
}
```

2️⃣ Department Entity (@OneToMany)

```
package com.interview;
import jakarta.persistence.*;
import java.util.List;
@Entity
public class Department {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  // One department has many students
  @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
  private List<Student> students;
  // Constructors, Getters, and Setters
}
```

3️⃣ Student ID Card Entity (@OneToOne)

```
package com.interview;
import jakarta.persistence.*;
@Entity
public class StudentIdCard {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String cardNumber;
  @OneToOne(mappedBy = "studentIdCard")
  private Student student;
  // Constructors, Getters, and Setters
}
```

4️⃣ Course Entity (@ManyToMany)

```
package com.interview;
import jakarta.persistence.*;
import java.util.List;
@Entity
public class Course {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
```

```
    private String title;
    // Many courses have many students
    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
    // Constructors, Getters, and Setters
}
```

Spring @Value Annotation

The @Value annotation in Spring is **used to inject values into fields, methods, or constructor parameters from property files, environment variables, or Spring Expression Language (SpEL)**.

Key Features

- Injects values from application.properties or application.yml.
- Supports SpEL for dynamic value assignment.
- Converts values to required data types automatically.

Traditional XML-Based Configuration

Student.java (Without @Value)

```
public class Student {
    private int rollNo;
    private String name;
    private int age;
    public void setRollNo(int rollNo) { this.rollNo = rollNo; }
    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
    public void display() {
        System.out.println("Roll No: " + rollNo);
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

Student-info.properties

```
student.rollNo=101
```

student.name=Sagar

student.age=20

beans.xml

```
<context:property-placeholder location="classpath:student-info.properties"/>
<bean id="student" class="Student">
    <property name="rollNo" value="${student.rollNo}"/>
    <property name="name" value="${student.name}"/>
    <property name="age" value="${student.age}"/>
</bean>
```

Running the Application

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {
  public static void main(String[] args) {
     ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
     Student student = context.getBean("student", Student.class);
     student.display();
  }
}
```

Output

Roll No: 101

Name: Sagar

Age: 20

Using @Value Annotation (Setter Injection)

If you are using Spring Boot or Java-based configuration, beans.xml is not needed.

Instead, Spring will read values from application.properties (or student-info.properties) and inject them into the fields.

When you use @Value("${student.rollNo}"), Spring will inject values from an external properties file (e.g., student-info.properties) instead of hardcoding them in the Java class or beans.xml.

Example Without `beans.xml` (Spring Boot Style)

1 Create a `student-info.properties` File

(Place it in `src/main/resources/`)

ini

```
student.rollNo=101
student.name=John Doe
student.age=22
```

```
import org.springframework.beans.factory.annotation.Value;
public class Student {
  private int rollNo;
  private String name;
  private int age;
  @Value("${student.rollNo}")
  public void setRollNo(int rollNo) { this.rollNo = rollNo; }
  @Value("${student.name}")
  public void setName(String name) { this.name = name; }
  @Value("${student.age}")
  public void setAge(int age) { this.age = age; }
  public void display() {
     System.out.println("Roll No: " + rollNo);
     System.out.println("Name: " + name);
     System.out.println("Age: " + age);
  }
}
```

Spring Configuration (beans.xml)

```
<context:annotation-config/>
<context:property-placeholder location="classpath:student-info.properties"/>
<bean id="student" class="Student"/>
```

Running the Application

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    Student student = context.getBean("student", Student.class);
    student.display();
  }
}
```

Output

Roll No: 101

Name: Sagar

Age: 20

Using @Value Directly on Fields (No Setters Required)

The last example uses @Value directly on fields, but it doesn't provide direct values—it references property file values (${student.rollNo}, ${student.name}, ${student.age}).

In this case, **no external properties file is required** because the values are **hardcoded directly** in the @Value annotation.

```
// Importing required classes
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
// Marking the class as a Spring Bean
@Component
public class Student {
  // Assigning direct values using @Value annotation
  @Value("101") private int rollNo;
  @Value("Anshul") private String name;
```

```
    @Value("25") private int age;
    // Method to display student details
    public void display() {
        System.out.println("Roll No: " + rollNo);
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

beans.xml (Updated Configuration)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.example"/>

</beans>
```

Running the Application

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
        Student student = context.getBean("student", Student.class);
        student.display();
    }
}
```

Output

```
Roll No: 101
```

Name: Anshul

Age: 25

If we use spring boot applications so no beans.xml required and also will set values to fields in application.yaml or application.properties

Scenario	Uses `beans.xml` ?	Uses `application.properties` ?	When to Use?
Direct `@Value("value")`	❌ No	❌ No	When values are constant.
`@Value("${property}")`	❌ No	✅ Yes	When values need to be configurable.
SpEL with `@Value("${property:default}")`	❌ No	✅ Yes (Optional)	When values may be missing, need defaults, or depend on conditions.

What are Spring Profiles and how do they work?

Spring Profiles allow you to define and activate different configurations for different environments (such as development, testing, and production) in a Spring application. This helps manage environment-specific settings without modifying the application code.

Below is an example of how you can configure **Spring Profiles** using application.properties for different environments.

1. Default application.properties (Common for all environments)

This file contains settings that apply to all profiles unless overridden.

server.port=8080

spring.application.name=MySpringApp

---

2. application-dev.properties (Development Environment)

spring.datasource.url=jdbc:mysql://localhost:3306/devdb

spring.datasource.username=devuser

spring.datasource.password=devpassword

logging.level.root=DEBUG

---

3. application-prod.properties (Production Environment)

spring.datasource.url=jdbc:mysql://prod-db-server:3306/proddb

spring.datasource.username=produser

spring.datasource.password=prodpassword

logging.level.root=ERROR

Activating a Profile in various ways

You can activate a profile using:

A. In application.properties

spring.profiles.active=dev

This will load application-dev.properties.

```
# Default application.properties
server.port=8080
spring.application.name=MySpringApp
spring.profiles.active=dev
```

B. Using Command-Line Arguments

java -jar myapp.jar --spring.profiles.active=prod

This will load application-prod.properties.

C. Using an Environment Variable

export SPRING_PROFILES_ACTIVE=prod

spring.datasource.primary.jpa.hibernate.ddl-auto=update

The spring.datasource.primary.jpa.hibernate.ddl-auto property is equivalent to Hibernate's **hibernate.hbm2ddl.auto** setting

Common Values for `ddl-auto`

Value	Behavior	Use Case
none	Disables schema generation	Use in production when schema is managed manually
update	Updates schema (adds missing columns but doesn't remove existing data)	Use in development & testing
create	Drops and recreates tables on every startup	Good for testing but **destroys data**
create-drop	Same as `create`, but also drops tables when the app stops	Useful for temporary in-memory DBs
validate	Just checks if the schema matches entities, but **does not modify** it	Use in production for safety

What is @Qualifier in Spring?

@Qualifier is an annotation used in **Spring Dependency Injection** to specify which bean should be injected **when multiple beans of the same type exist**. **@Qualifier("beanName")** helps **resolve conflicts** when multiple beans of the same type exist.

Ex:

We have an EmployeeService interface with **two implementations**:

1. **FullTimeEmployeeService** (for full-time employees).
2. **PartTimeEmployeeService** (for part-time employees).

Defining the Interface

```
public interface EmployeeService {
  void getEmployeeDetails();
}
```

Two Implementations Classes

1)Full-Time Employee Service

```
@Service("fullTimeEmployeeService")  // Custom bean name
public class FullTimeEmployeeService implements EmployeeService {
  @Override
  public void getEmployeeDetails() {
    System.out.println("Full-Time Employee Details");
  }
}
```

2) **Part-Time Employee Service**

```
@Service("partTimeEmployeeService")  // Custom bean name
public class PartTimeEmployeeService implements EmployeeService {
  @Override
  public void getEmployeeDetails() {
    System.out.println("Part-Time Employee Details");
  }
}
```

A Company class depends on EmployeeService

If we try to autowire EmployeeService without specifying which one, Spring throws an error:

```
@Service
public class Company {
  @Autowired
  private EmployeeService employeeService;  // ❌ ERROR: Spring doesn't know which one to use
  public void showEmployeeDetails() {
    employeeService.getEmployeeDetails();
  }
}
```

Error

No qualifying bean of type 'EmployeeService' available:

expected single matching bean but found 2: fullTimeEmployeeService, partTimeEmployeeService

Fixing the Issue WITH @Qualifier

We explicitly tell Spring **which bean to use**.

Injecting the Full-Time Employee Service/ Part-Time Employee Service

```
@Service
public class Company {
  @Autowired
  @Qualifier("fullTimeEmployeeService")  // ✅ Now Spring knows to use
FullTimeEmployeeService
  private EmployeeService employeeService;
  public void showEmployeeDetails() {
    employeeService.getEmployeeDetails();
  }
}
```

Output:

Full-Time Employee Details

Spring Boot's Support for Messaging

Spring Boot provides built-in support for various **messaging systems** to enable **asynchronous communication** between different parts of an application or between microservices.

Messaging Protocols & Systems Supported by Spring Boot

1. JMS (Java Message Service)
2. AMQP (Advanced Message Queuing Protocol)
3. Apache Kafka
4. RabbitMQ

What is JMS?

- JMS is a **Java API** for sending, receiving, and processing **messages asynchronously**.
- It is commonly used in **enterprise applications** for **reliable messaging**.
- Works with **message brokers** like **ActiveMQ, IBM MQ, HornetQ**

How JMS Works (with a Message Broker Queue)

1️⃣ Producer sends a message to a Queue (or a Topic in Pub/Sub).
2️⃣ The Message Broker (JMS Provider) stores the message in the queue.
3️⃣ Consumer retrieves the message from the Queue (or subscribes to a Topic).

This means the message broker (such as ActiveMQ, RabbitMQ with JMS, or IBM MQ) plays a crucial role in storing and delivering messages.

So the correct flow in JMS is:
👉 **Producer → Message Broker (Queue) → Consumer**

Spring Boot JMS with ActiveMQ (Hands-on Example)

1) Download ActiveMQ from Apache ActiveMQ.
2) Extract it and start ActiveMQ using:

```
bin\activemq start    # (For Windows)
```

3) ActiveMQ Web Console:
Go to http://localhost:8161/admin/ (Username/Password: admin/admin)

📌 Simple Example: Spring Boot with ActiveMQ

Step 1: Add Dependency

xml Copy Edit

```xml
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-spring-boot-starter</artifactId>
    <version>2.0.1</version>
</dependency>
```

Step 2: Configure `application.properties`

properties Copy Edit

```properties
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

Producer (Sender)

Sends a message to the test-queue using JmsTemplate:

```
@Service
public class JmsProducer {
  @Autowired
```

```
    private JmsTemplate jmsTemplate;
    public void sendMessage(String message) {
        jmsTemplate.convertAndSend("test-queue", message);
        System.out.println("Message Sent: " + message);
    }
}
```

jmsTemplate.convertAndSend("test-queue", message);

jmsTemplate → An instance of JmsTemplate, provided by Spring Boot for interacting with JMS brokers like ActiveMQ.

convertAndSend(...) → Converts the given message into a JMS message format and sends it to the queue.

"test-queue" → The name of the destination queue where the message will be sent.

message → The actual message content (String, JSON, Object, etc.).

Consumer (Receiver)

Receives messages from test-queue using @JmsListener:

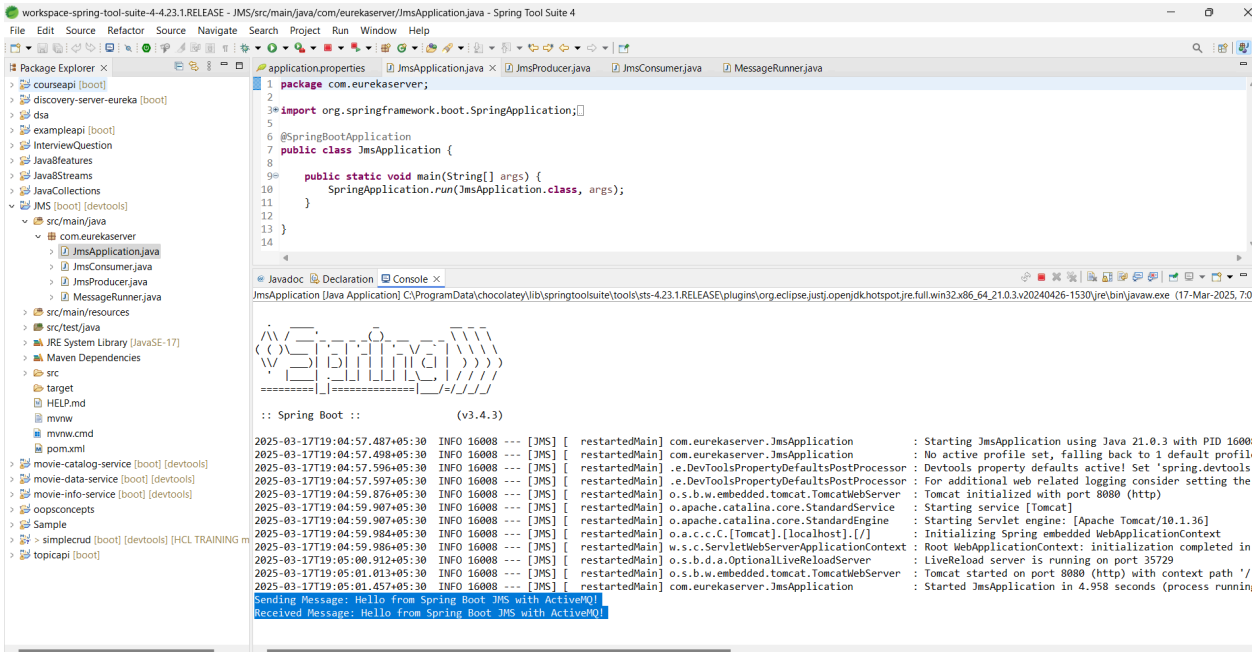
```
@Component
public class JmsConsumer {
    @JmsListener(destination = "test-queue")
    public void receiveMessage(String message) {
        System.out.println("Received Message: " + message);
    }
}
```

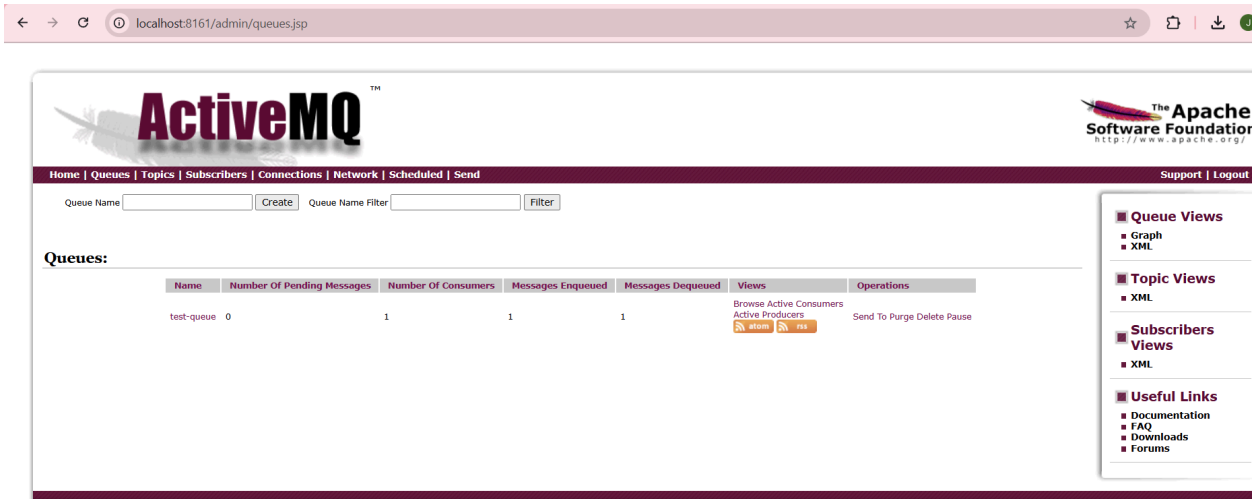
Automatically Send a Message When App Starts

We will use CommandLineRunner to trigger the message sending.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
@Component
```

```java
public class MessageRunner implements CommandLineRunner {

    @Autowired
    private JmsProducer producer;
    @Override
    public void run(String... args) throws Exception {
        producer.sendMessage("Hello from Spring Boot JMS with ActiveMQ!");
    }
}
```





Ex2:

JMS requires that any object sent as a message **must be serializable**.

```
import java.io.Serializable;
public class Order implements Serializable {
  private static final long serialVersionUID = 1L;

  private String orderId;
  private String product;
  private double price;
  // Constructors
  public Order() {}

  public Order(String orderId, String product, double price) {
    this.orderId = orderId;
    this.product = product;
    this.price = price;
  }
  // Getters and Setters
  public String getOrderId() { return orderId; }
  public void setOrderId(String orderId) { this.orderId = orderId; }
  public String getProduct() { return product; }
  public void setProduct(String product) { this.product = product; }
  public double getPrice() { return price; }
  public void setPrice(double price) { this.price = price; }
  @Override
  public String toString() {
    return "Order{orderId='" + orderId + "', product='" + product + "', price=" + price + "}";
  }
}
```

Producer (Sending an Object)

Use convertAndSend() to send the Order object.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Service;
@Service
public class JmsOrderProducer {
```

```
  @Autowired
  private JmsTemplate jmsTemplate;
  public void sendOrder(Order order) {
    jmsTemplate.convertAndSend("order-queue", order);
    System.out.println("Order Sent: " + order);
  }
}
```

Consumer (Receiving an Object)

Use @JmsListener to receive and deserialize the object.

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
@Component
public class JmsOrderConsumer {
  @JmsListener(destination = "order-queue")
  public void receiveOrder(Order order) {
    System.out.println("Received Order: " + order);
  }
}
```

Trigger the Messaging System

To test, you can create a simple CommandLineRunner to send an order when the app starts:

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
@Component
public class OrderRunner implements CommandLineRunner {
  @Autowired
  private JmsOrderProducer producer;
  @Override
  public void run(String... args) throws Exception {
    Order order = new Order("123", "Laptop", 1200.00);
    producer.sendOrder(order);
  }
}
```

Output

Order Sent: Order{orderId='123', product='Laptop', price=1200.0}

Received Order: Order{orderId='123', product='Laptop', price=1200.0}

Real-time applications

Airline Ticket Booking System

Problem:

- When a user books a flight ticket, the system needs to:
  - Reserve a seat
  - Process payment
  - Send a confirmation email

Solution with JMS:

- The booking system sends a message to ticket-reservation-queue.
- The **Reservation Service** books the seat and sends a message to payment-processing-queue.
- The **Payment Service** processes the payment and sends a message to email-queue to notify the user.

Example Flow:

```
jmsTemplate.convertAndSend("ticket-reservation-queue", new Ticket("John Doe",
"Flight123"));
```

✅ **Benefits:**
✔ Ensures reliability even if one service fails.
✔️ Handles high traffic without slowing down.

Banking Transactions (Fraud Detection)

Problem:

- Banks need to monitor transactions in real-time to detect fraud.
- Instead of directly calling fraud detection APIs synchronously, they need **asynchronous** processing.

Solution with JMS:

- Whenever a transaction happens, it is sent to a fraud-check-queue.
- A **Fraud Detection Service** listens to this queue and analyzes transactions.
- If fraud is detected, an alert message is sent to security-alert-queue.

Example Flow:

jmsTemplate.convertAndSend("fraud-check-queue", new Transaction("TXN1001", 5000.00));

✅ **Benefits:**
✔️ Allows non-blocking fraud checks.
✔️ Decouples the transaction system from fraud analysis.

E-Commerce Order Processing

Problem:

- When a customer places an order, multiple systems need to process it:
  - Payment service
  - Inventory management
  - Order confirmation email

Solution with JMS:

- The **Order Service** sends a message to a orders-queue.
- The **Payment Service** picks up the message and processes payment.
- If successful, another message is sent to the **Inventory Service** to update stock.
- Once inventory is updated, the **Email Service** sends an order confirmation.

Example Flow:

jmsTemplate.convertAndSend("orders-queue", new Order("123", "Laptop", 1200.00));

✅ **Benefits:**
✔️ Ensures reliability even if one service is down.
✔️ Decouples order, payment, and inventory systems.

🚀 Summary: Where is JMS Used?

Application Type	Use Case
E-Commerce	Order processing, inventory management
Banking & Finance	Fraud detection, real-time transaction alerts
Airline & Travel	Flight booking, seat reservations
Stock Market	Live stock price updates
Retail & Logistics	Order fulfillment, delivery tracking
Healthcare	Patient records, lab test notifications
Government	Secure document processing

AMQP (Advanced Message Queuing Protocol)

📌 What is AMQP?

- AMQP is a messaging protocol that supports reliable, scalable, and flexible messaging.
- It is most commonly used with **RabbitMQ**, a widely adopted message broker.
- Unlike JMS, AMQP is not Java-specific and supports multiple programming languages.

Real-World Applications of AMQP (RabbitMQ)

✅ Chat Applications → Sending messages between users in real time.
✅ Order Management Systems → Processing orders in e-commerce platforms.
✅ Microservices Communication → Sending data between distributed microservices.

📌 How AMQP Works?

1️⃣ A **producer** sends messages to an **exchange**.
2️⃣ The **exchange** routes messages to appropriate **queues**.
3️⃣ A **consumer** picks up messages from the **queue**.

Producer → Exchange → Queue → Consumer

Key Difference:

- JMS: Producer directly sends to Queue/Topic.
- AMQP: Producer sends to Exchange, which then routes to Queues

Real-World Use Case:

- In an **e-commerce platform**, when a customer places an order, the order must be **processed asynchronously**.
- The system should handle multiple order requests and ensure reliable **order processing**.

How AMQP (RabbitMQ) Works

✅ Step 1: The Order Service (Producer) sends an order request.
✅ Step 2: The message is sent to a RabbitMQ Exchange, which determines where to route the message.
✅ Step 3: The Exchange routes the message to the appropriate Queue.
✅ Step 4: The Order Processing Service (Consumer) picks the message from the queue and processes it.

Spring Boot with RabbitMQ – Implementation

1️⃣ Add Dependencies

```
52              <scope>test</scope>
53          </dependency>
54          <dependency>
55              <groupId>org.springframework.boot</groupId>
56              <artifactId>spring-boot-starter-amqp</artifactId>
57          </dependency>
58
```

Create RabbitMQ Config

```
package com.eurekaserver;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class RabbitMQConfig {
  @Bean
  public Queue orderQueue() {
     return new Queue("order-queue", false);
  }
}
```

Producer (Order Service)

```
package com.eurekaserver;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
@Service
public class OrderProducer {
  @Autowired
  private AmqpTemplate amqpTemplate;
  public void sendOrder(String order) {
     amqpTemplate.convertAndSend("order-queue", order);
     System.out.println("Order Sent: " + order);
  }
}
```

Consumer (Order Processing Service)

```
package com.amqp;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;
@Component
public class OrderConsumer {
  @RabbitListener(queues = "order-queue")
  public void processOrder(String order) {
     System.out.println("Processing Order: " + order);
  }
}
```

To test, you can create a simple CommandLineRunner to send an order when the app starts:

```
package com.amqp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class AmqpApplication  implements CommandLineRunner  {
        @Autowired
          private OrderProducer orderProducer;
```

```
        public static void main(String[] args) {
                SpringApplication.run(AmqpApplication.class, args);
        }
         @Override
           public void run(String... args) throws Exception {
              orderProducer.sendOrder("Order#101 - MacBook Pro");
           }
}
```

Install RabbitMQ

To use RabbitMQ first we need to install **erlang** https://www.erlang.org/downloads

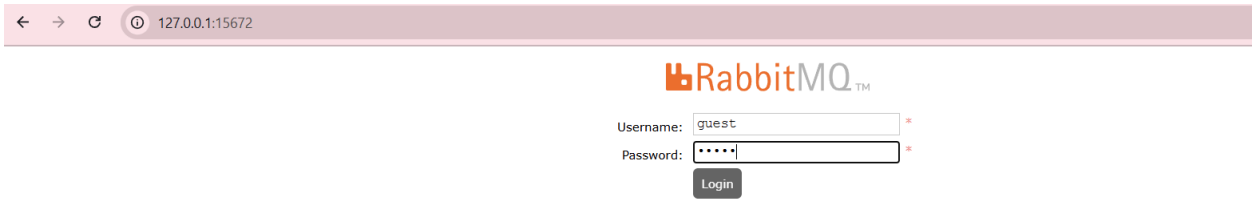
RabbitMQ : https://www.rabbitmq.com/docs/install-windows

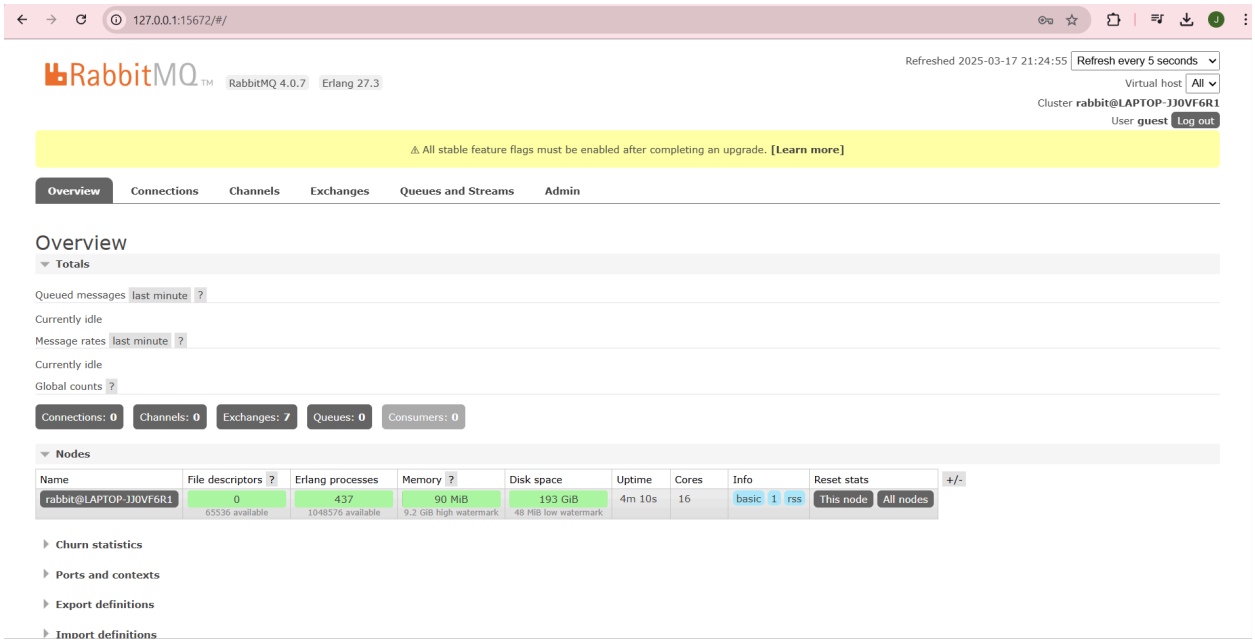
Then start RabbitMQ

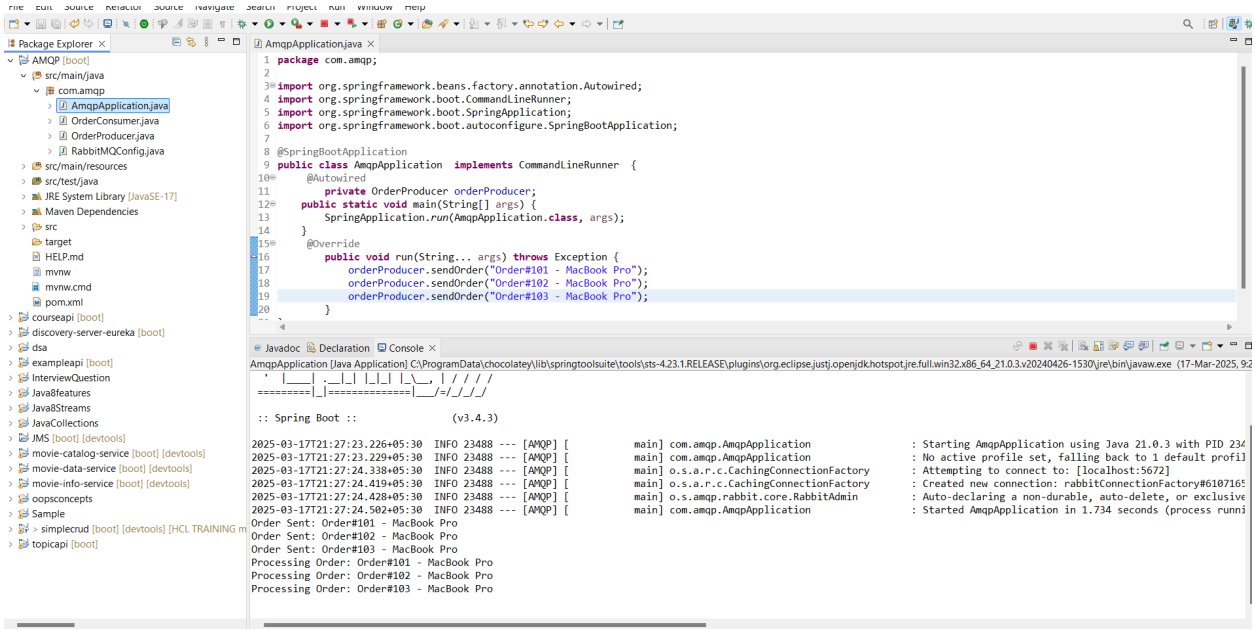
```
C:\Program Files\RabbitMQ Server\rabbitmq_server-4.0.7\sbin>rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@LAPTOP-JJ0VF6R1:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@LAPTOP-JJ0VF6R1...
Plugin configuration unchanged.

C:\Program Files\RabbitMQ Server\rabbitmq_server-4.0.7\sbin>rabbitmq-plugins

Usage

rabbitmq-plugins [--node <node>] [--timeout <timeout>] [--longnames] [--quiet] <command> [<command options>]

Available commands:

Help:

   autocomplete  Provides command name autocomplete variants
   help          Displays usage information for a command
   version       Displays CLI tools version

Monitoring, observability and health checks:

   directories   Displays plugin directory and enabled plugin file paths
   is_enabled    Health check that exits with a non-zero code if provided plugins are not enabled on target node

Plugin Management:

   disable       Disables one or more plugins
   enable        Enables one or more plugins
   list          Lists plugins and their state
   set           Enables one or more plugins, disables the rest

Use 'rabbitmq-plugins help <command>' to learn more about a specific command

C:\Program Files\RabbitMQ Server\rabbitmq_server-4.0.7\sbin>rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@LAPTOP-JJ0VF6R1:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@LAPTOP-JJ0VF6R1...
Plugin configuration unchanged.
```

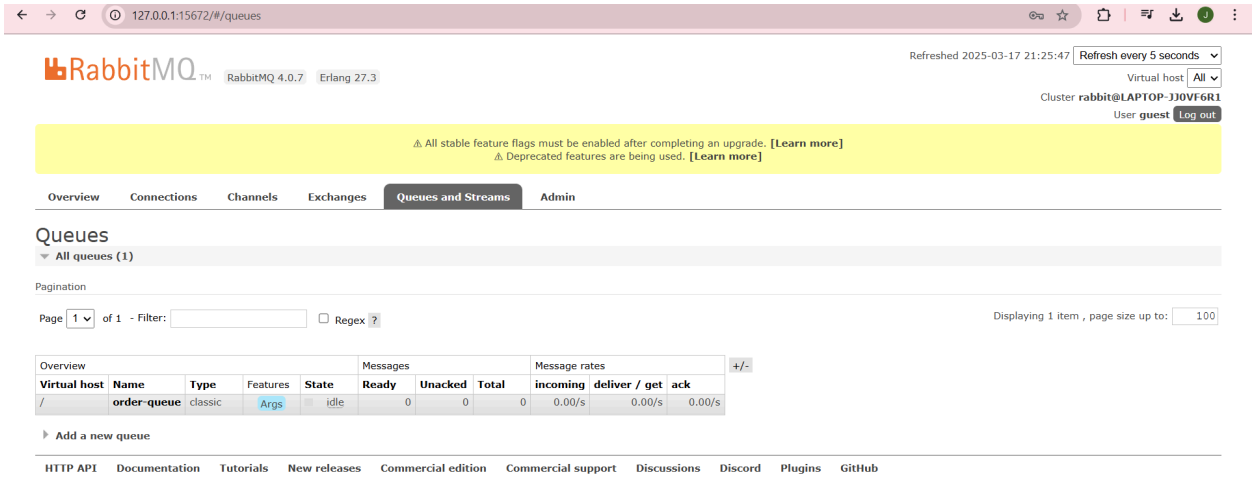
Open your browser and go to:
👉 http://localhost:15672/
Default login credentials:

- Username: guest
- Password: guest


127.0.0.1:15672
RabbitMQ
Username:
guest
Password:
Login


127.0.0.1:15672/#/
RabbitMQ
RabbitMQ 4.0.7
Erlang 27.3
Refreshed 2025-03-17 21:24:55
Refresh every 5 seconds
Virtual host
All
Cluster rabbit@LAPTOP-JJ0VF6R1
User guest
Log out
All stable feature flags must be enabled after completing an upgrade. [Learn more]
Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Overview
Totals
Queued messages last minute
Currently idle
Message rates last minute
Currently idle
Global counts
Connections: 0
Channels: 0
Exchanges: 7
Queues: 0
Consumers: 0
Nodes
Name
File descriptors
Erlang processes
Memory
Disk space
Uptime
Cores
Info
Reset stats
rabbit@LAPTOP-JJ0VF6R1
0
65536 available
437
1048576 available
90 MiB
9.2 GiB high watermark
193 GiB
48 MiB low watermark
4m 10s
16
basic 1 rss
This node
All nodes
Churn statistics
Ports and contexts
Export definitions
Import definitions


Package Explorer
AMQP [boot]
src/main/java
com.amqp
AmqpApplication.java
OrderConsumer.java
OrderProducer.java
RabbitMQConfig.java
src/main/resources
src/test/java
JRE System Library [JavaSE-17]
Maven Dependencies
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml
courseapi [boot]
discovery-server-eureka [boot]
dsa
exampleapi [boot]
InterviewQuestion
Java8features
Java8Streams
JavaCollections
JMS [boot] [devtools]
movie-catalog-service [boot] [devtools]
movie-data-service [boot] [devtools]
movie-info-service [boot] [devtools]
oopsconcepts
Sample
simplecrud [boot] [devtools] [HCL TRAINING m
topicapi [boot]
AmqpApplication.java
package com.amqp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class AmqpApplication implements CommandLineRunner {
@Autowired
private OrderProducer orderProducer;
public static void main(String[] args) {
SpringApplication.run(AmqpApplication.class, args);
}
@Override
public void run(String... args) throws Exception {
orderProducer.sendOrder("Order#101 - MacBook Pro");
orderProducer.sendOrder("Order#102 - MacBook Pro");
orderProducer.sendOrder("Order#103 - MacBook Pro");
}
Javadoc
Declaration
Console
AmqpApplication [Java Application] C:\ProgramData\chocolatey\lib\springtoolsuite\tools\sts-4.23.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.3.v20240426-1530\jre\bin\javaw.exe (17-Mar-2025, 9:2
:: Spring Boot :: (v3.4.3)
2025-03-17T21:27:23.226+05:30 INFO 23488 --- [AMQP] [main] com.amqp.AmqpApplication : Starting AmqpApplication using Java 21.0.3 with PID 234
2025-03-17T21:27:23.229+05:30 INFO 23488 --- [AMQP] [main] com.amqp.AmqpApplication : No active profile set, falling back to 1 default profil
2025-03-17T21:27:24.338+05:30 INFO 23488 --- [AMQP] [main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2025-03-17T21:27:24.419+05:30 INFO 23488 --- [AMQP] [main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#6107165
2025-03-17T21:27:24.428+05:30 INFO 23488 --- [AMQP] [main] o.s.amqp.rabbit.core.RabbitAdmin : Auto-declaring a non-durable, auto-delete, or exclusive
2025-03-17T21:27:24.502+05:30 INFO 23488 --- [AMQP] [main] com.amqp.AmqpApplication : Started AmqpApplication in 1.734 seconds (process runni
Order Sent: Order#101 - MacBook Pro
Order Sent: Order#102 - MacBook Pro
Order Sent: Order#103 - MacBook Pro
Processing Order: Order#101 - MacBook Pro
Processing Order: Order#102 - MacBook Pro
Processing Order: Order#103 - MacBook Pro



Apache Kafka

What is Kafka?

- Kafka is a distributed event streaming platform.
- Used for real-time processing of large volumes of dat**a.**
- Works as a publish-subscribe messaging system.

Apache Kafka is a distributed event streaming platform used to handle real-time data feeds. It enables systems to send, receive, store, and process events in a scalable and fault-tolerant way.

It acts as a message broker similar to JMS (Java Message Service) and RabbitMQ, but it's designed for high-throughput, low-latency streaming.

How Kafka Works?

Kafka follows a distributed publish-subscribe model with the following architecture:

1️⃣ A producer sends messages to a Kafka topic.
2️⃣ Kafka stores messages in partitions inside the topic.
3️⃣ Consumers read messages from partitions within a topic.

🛠️ Producer → Kafka Topic (Partitions) → Consumer

📌 Simple Example: Spring Boot with Kafka

Step 1: Add Dependency

xml Copy Edit

```xml
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Step 2: Configure `application.properties`

properties Copy Edit

```properties
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=my-group
```

1. JMS (Java Message Service) – Best for Enterprise Applications

Example: Banking Transaction Processing System

🔹 A bank processes loan applications and must ensure **each request is processed exactly once** with **high reliability**.

🔹 Since most banking applications are **Java-based**, JMS is the best choice for handling **transactional messaging**.

🏛️ Why JMS?

✅ **Java-native support** – Works seamlessly with Java EE applications.
✅ **Guaranteed delivery** – Ensures messages aren't lost in transactional workflows.
✅ **Point-to-Point messaging** – Loan processing is handled one message at a time.

🚀 Example Use Case:

- Loan applications are placed in a **JMS queue**.
- A backend processor picks up each application, verifies it, and updates the database.
- Once processed, the message is removed from the queue.

✅ 2. RabbitMQ (AMQP) – Best for Microservices Communication

Example: Food Delivery App (Swiggy, Uber Eats)

🔹 A food ordering app has multiple services:

- **Order Service** → Receives new orders
- **Payment Service** → Processes payments
- **Restaurant Service** → Notifies restaurants of new orders
- **Delivery Service** → Assigns a delivery person

📡 **Why RabbitMQ?**

✅ **Reliable messaging** – Ensures messages reach the right services.
✅ **Message routing (Exchange → Queue)** – Routes messages to different microservices.
✅ **Supports Acknowledgments** – If a service fails, messages can be requeued.

🚀 **Example Use Case:**
1️⃣ The **Order Service** sends a message to RabbitMQ:

Order #1234 placed by User A

2️⃣ The **Payment Service** listens for payment-related messages.
3️⃣ The **Restaurant Service** receives order notifications.
4️⃣ The **Delivery Service** assigns a delivery person once food is ready.

📌 **RabbitMQ ensures each service receives only relevant messages.**

✅ **3. Kafka – Best for Real-Time Streaming & Big Data**

Example: Ride-Sharing App (Uber, Ola, Lyft)

🔹 A ride-sharing app needs to **continuously process millions of location updates from drivers and passengers**.

⚡ **Why Kafka?**

✅ **High throughput** – Handles millions of real-time events.
✅ **Message retention & replay** – Store messages and replay them later.
✅ **Event-driven architecture** – Works well with stream processing (e.g., Apache Flink, Spark).

🚀 **Example Use Case:**
1️⃣ **Drivers** send their GPS locations every few seconds.

Driver #5678 → Latitude: 12.9716, Longitude: 77.5946

2️⃣ Kafka partitions these location updates for **load balancing**.
3️⃣ **Ride Matching Service** reads Kafka topics to find the nearest driver.
4️⃣ **Pricing Service** calculates surge pricing based on real-time demand.
5️⃣ **Analytics Dashboard** shows city-wide ride demand in real-time.

📌 **Kafka ensures the system can scale and handle massive real-time data efficiently.**

If you need Java-based transactional messaging → Use JMS.

If you need reliable messaging for microservices → Use RabbitMQ.

If you need high-throughput, real-time event processing → Use Kafka.

What is Spring WebFlux?

Spring WebFlux is a reactive-stack web framework in Spring, introduced as an alternative to Spring MVC for handling asynchronous processing and backpressure.

Mono represents a single asynchronous value or empty value, while Flux represents a stream of multiple values. Exam**ple:**

Spring WebFlux Example (REST API)

Creating a Reactive REST API using WebFlux

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {
  private final EmployeeService employeeService;
  public EmployeeController(EmployeeService employeeService) {
    this.employeeService = employeeService;
  }
  // Returns a single Employee (Mono)
  @GetMapping("/{id}")
  public Mono<Employee> getEmployeeById(@PathVariable String id) {
```

```
        return employeeService.getEmployeeById(id);
    }
    // Returns multiple Employees (Flux)
    @GetMapping
    public Flux<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }
}
```

---

simple example of using **SLF4J with Logback** for logging in a Spring Boot application:

🔧 Step 1: Add SLF4J (optional)

If you're using **Spring Boot**, SLF4J + Logback comes **pre-configured**, so no need to add anything to pom.xml. But if you want explicitly:

If you're using Maven, add this to your `pom.xml`:

xml Copy Edit

```xml
<dependencies>
    <!-- SLF4J API -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>2.0.9</version>
    </dependency>

    <!-- Logback as SLF4J implementation -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.4.11</version>
    </dependency>
</dependencies>
```

📄 Step 2: Use Logger in your class

```
package com.basic;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloController {
  private static final Logger logger = LoggerFactory.getLogger(HelloController.class);
  @GetMapping("/hello")
  public String sayHello() {
    logger.info("INFO: Hello endpoint was called");
    logger.debug("DEBUG: This is a debug log");
    logger.warn("WARN: This is a warning");
    logger.error("ERROR: Something went wrong");
    return "Hello, World!";
  }
}
```

Output in console (if running in dev profile)

INFO com.example.demo.HelloController - INFO: Hello endpoint was called
DEBUG com.example.demo.HelloController - DEBUG: This is a debug log
WARN com.example.demo.HelloController - WARN: This is a warning
ERROR com.example.demo.HelloController - ERROR: Something went wrong

Marker Interface

A **Marker Interface** in Java is a **special type of interface** that does **not contain any methods or fields**. It is used **only to "mark" or "tag" a class** as having a certain capability, often so that the JVM or a framework can treat the class differently based on the presence of this marker.

Definition

```
public interface Serializable {

   // No methods here - it's a marker interface

}
```

Purpose

Marker interfaces act as **metadata** for classes. They convey to the compiler or JVM that the class has a certain property or should be handled in a specific way.

Common Marker Interfaces in Java

Interface	Package	Purpose
`Serializable`	`java.io`	Marks a class whose objects can be serialized
`Cloneable`	`java.lang`	Marks a class that allows field-for-field copy using `Object.clone()`
`Remote`	`java.rmi`	Marks a class whose objects can be invoked remotely
`ThreadSafe`	(custom)	Can be created to mark thread-safe classes in large projects

🔍 Real-World Purpose Examples:

Marker Interface	Purpose
`Serializable`	Tell the JVM: "You can save and load this object to/from a file or network."
`Cloneable`	Tell the JVM: "It's safe to create a copy of this object using `.clone()`."
`Remote`	Tell Java RMI: "You can call methods on this object from another computer."
Custom (e.g., `Auditable`)	Tell your app: "This object needs to be tracked or logged during actions."

Example Usage

```
import java.io.Serializable;

public class Employee implements Serializable {
    private String name;
    private int id;
    // constructors, getters, setters
}
```

Now, because Employee implements Serializable, you can serialize it using Java's I/O streams:

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.ser"));

out.writeObject(new Employee("John", 101));

out.close();
```

Serializable interface is present in java.io package. It is used to make an object eligible for saving its state into a file. This is called Serialization. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable.

- The Java **serialization mechanism** uses **instanceof Serializable** to check if the class can be serialized.
- If the object doesn't implement Serializable, a NotSerializableException is thrown.

🔍 Key Difference

Feature	Marker Interface	Regular Interface
Contains Methods?	❌ No methods or fields	✅ Has one or more methods
Main Purpose	Just to **mark/tag** a class	To define a **contract** of methods the class must implement
Behavior	Used to give classes **special treatment** (often by JVM or frameworks)	Forces implementing classes to **implement behaviors** (methods)
Example	`Serializable`, `Cloneable`	`Runnable`, `Comparable`, `List`
Usage	Checked using `instanceof` or reflection	Methods are called from the interface
Analogy	A **label** or **flag**	A **rulebook** that must be followed

Explain the use of the final keyword in variable, method and class.

The final keyword in Java is a **non-access modifier** used for **variables, methods, and classes**. Its purpose is to impose **restrictions** that help ensure the integrity of your code by **preventing changes**. Here's how final behaves in each context:

✅ 1. final Variable

A final variable can be **assigned only once**. Once assigned, its value cannot be changed — making it a **constant**.

💡 Use Case:

- To define **constants** (e.g., PI, MAX_SIZE)
- To prevent accidental reassignment
- Useful in **immutability**

🔍 Example:

```
public class Example {
   final int MAX_USERS = 100;
   void test() {
      // MAX_USERS = 200; // ❌ Compilation error: can't reassign final variable
   }
}
```

🔁 For objects:

```
final List<String> names = new ArrayList<>();
names.add("John");      // ✅ allowed
// names = new ArrayList<>(); // ❌ Not allowed (reference can't be changed)
```

You **can** modify the contents of the object, but you **cannot reassign** the object reference.

✅ 2. final Method

A final method **cannot be overridden** by subclasses.

💡 **Use Case:**

- To prevent modification of critical behavior in subclasses
- Enforce standard implementation
- Useful in **security-sensitive** or **framework** base classes

🔍 **Example:**

```
class Parent {
    final void show() {
        System.out.println("This is a final method");
    }
}
class Child extends Parent {
    // void show() { } // ❌ Compilation error: cannot override final method
}
```

✅ 3. final Class

A final class **cannot be extended** (inherited). No class can be a subclass of a final class.

💡 **Use Case:**

- To create **immutable** or **utility** classes (e.g., java.lang.String, java.lang.Math)
- Prevent misuse through subclassing

Example:

```
final class Animal {
    void speak() {
        System.out.println("Animal sound");
    }
}
// class Dog extends Animal { } // ❌ Compilation error: cannot inherit from final class
```

What is a Garbage collector in JAVA?

Heap Memory Management: Java manages memory in the **heap**, which is divided into regions like:

- **Young Generation**: Stores new objects.
- **Old Generation**: Stores objects that have been around for a while and survived garbage collection in the Young Generation.
- **Permanent Generation(before java8) (or Metaspace((bob-heap area)after java 8))**: Stores class definitions, metadata, etc. (in newer versions of Java, this is Metaspace).

Garbage Collection in Java:

- Java uses **automatic memory management** through the Garbage Collector (GC), which works in the background to reclaim memory occupied by objects that are no longer in use.
- **How the Garbage Collector Works**:

  The **Garbage Collector** identifies objects that are no longer reachable (i.e., no part of the program can reference them). These unused objects are then eligible to be deleted, freeing up memory for new objects.

- **Dynamic Memory Allocation**:

  In Java, objects are created dynamically using the new keyword. The memory for these objects is allocated on the heap, and they persist as long as there are references to them.

- **Memory Leaks**:

  There is no explicit need to destroy an object as Java handles the de-allocation automatically. The technique that accomplishes this is known as **Garbage Collection**. Programs that do not de-allocate memory can eventually crash when there is no memory left in the system to allocate. These programs are said to have memory leaks.

  Garbage collection in Java happens automatically during the lifetime of the program, eliminating the need to de-allocate memory and thereby avoid memory leaks.

  In **C language**, it is the programmer's responsibility to de-allocate memory allocated dynamically using free() function.This is where Java memory management leads.

```
package com.basic;
public class GarbageCollectionExample {
	// This is the main method where the program starts
	public static void main(String[] args) {
		// Creating an object of GarbageCollectionExample
		GarbageCollectionExample obj1 = new GarbageCollectionExample();
		GarbageCollectionExample obj2 = new GarbageCollectionExample();
		// obj1 is now no longer referenced, so it is eligible for garbage collection
		obj1 = null;
		// obj2 is still referenced, so it is not eligible for garbage collection
		System.out.println("obj2 is still in use.");
		// Suggesting garbage collection (the JVM might or might not perform GC
		// immediately)
		System.gc();
	}
	// Overriding the finalize method to show when an object is garbage collected
	@Override
	protected void finalize() throws Throwable {
		System.out.println("Garbage Collection is being done on this object.");
```

```
            super.finalize();
        }
}
```

How to enable debugging log in the spring boot application?

Debugging logs can be enabled in three ways -

1) We can start the application with **--debug switch**.

2) We can set the **logging.level.root=debug property** in the **application.property file**.

3) We can set the logging level of the root logger to debug in the **supplied logging configuration file**.

If you're using a custom logging configuration (e.g., logback-spring.xml or log4j2-spring.xml), you can set the logging level for the root logger in the configuration file.

For example, in logback-spring.xml, you can set the root logger level as follows:

```
<configuration>
  <logger name="ROOT" level="DEBUG"/>
</configuration>
```

For Log4j2, in log4j2-spring.xml, the root logger can be set like this:

```
<Configuration>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

What is dependency Injection?

IoC (Inversion of Control): A design principle where control of object creation and management is transferred to the Spring container. It is the process in which Object defines its dependencies without creating them. It uses DI to achieve it.

DI (Dependency Injection): A type of IoC where the Spring container injects the necessary dependencies into an object.

Types in Spring:

- Constructor Injection
- Setter Injection
- Field Injection

1. Setter Injection

In **Setter Injection**, the Spring IoC (Inversion of Control) container injects the dependent beans into the target bean via setter methods. This allows for **optional** dependencies or dependencies that can be changed at runtime.

- How it works: **Spring calls the setter method of the target bean to inject the dependency**.
- **Example:**

```
@Component
public class CarService {
   private Engine engine;
   // Setter method for injection
   @Autowired
```

```
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
    public void startCar() {
        engine.start();
    }
}
```

In this example, Spring will inject the Engine bean into the CarService bean via the setter method setEngine().

- **Advantages:**
    - Useful when the dependency is optional or can be changed after the object is created.
    - The class can be instantiated without all dependencies being provided.
- **Disadvantages:**
    - The dependency is mutable and can be modified later, which might not be ideal in all scenarios.
    - Setter injection can make it harder to guarantee that the bean has been fully initialized when used (i.e., the dependencies may not be set properly).

---

2. Constructor Injection

In **Constructor Injection**, dependencies are injected into the target bean through its constructor. This is the most **recommended** method of DI because it makes dependencies **immutable** and ensures that all required dependencies are provided when the object is created.

- **How it works:** Spring calls the constructor of the target bean, passing the dependent beans as arguments.

- **Example:**

```
@Component
public class CarService {
    private final Engine engine;
    // Constructor-based injection
    @Autowired
    public CarService(Engine engine) {
        this.engine = engine;
    }
    public void startCar() {
        engine.start();
    }
}
```

Here, Spring will automatically inject the Engine bean into the CarService bean via the constructor.

- **Advantages:**
    - Ensures that all required dependencies are provided at the time of bean creation.
    - Makes the dependency **immutable**, ensuring better integrity and reducing the chances of misuse.
    - Encourages good design, as dependencies are explicit.
- **Disadvantages:**

- If there are many dependencies, the constructor can become long and cumbersome.
- Requires more careful design to ensure no circular dependencies (where two or more beans depend on each other).

---

3. Field Injection

In **Field Injection**, Spring injects dependencies directly into the fields of the target bean class using the **Reflection API**. This is the most **convenient** but also the **least recommended** method, as it tightly couples the class to the Spring framework.

- **How it works:** Spring uses reflection to inject the dependencies into the fields of the target bean at runtime.
- **Example:**

```
@Component
public class CarService {
   @Autowired
   private Engine engine;
   public void startCar() {
      engine.start();
   }
}
```

In this case, the Engine bean is injected directly into the CarService bean via the engine field.

- **Advantages:**
  - Very convenient and easy to implement, especially for small applications.

    - The class does not need explicit constructors or setters for dependencies.

- **Disadvantages:**

    - It tightly couples the class to the Spring framework and makes it harder to test the class without the Spring context.

    - It violates the principle of **encapsulation** because dependencies are set directly on the fields.

    - Makes it harder to see which dependencies a class needs, as they are not explicitly provided via the constructor or setters.

What are the different properties of MVC routes?

Routing is handled using annotations like @RequestMapping, @GetMapping, @PostMapping, etc., on **controller methods**.

Key Route Properties in Spring MVC

1) **HTTP Method**

Specifies which HTTP method the route responds to.

@RequestMapping(value = "/users", method = RequestMethod.GET)

Shortcut annotations:

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping
- @PatchMapping

Path Variables

Dynamic parts of the URL captured via {} and accessed with @PathVariable.

@GetMapping("/users/{id}")

public String getUser(@PathVariable("id") int userId) { ... }

Request Parameters

Parameters in the query string, handled with @RequestParam.

```
@GetMapping("/search")
public String search(@RequestParam("q") String query) { ... }
// URL: /search?q=java
```

2) **URL Pattern**

This includes:

- **Literal segments** (/users, /products)
- **Variable placeholders** (/users/{id})

Used in annotations like:

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable int id) { ... }
```

3) **Defaults**

You can simulate **default values** in Spring using method parameters with default settings or @RequestParam with defaultValue.

```
@GetMapping("/search")
public List<Product> search(
  @RequestParam(defaultValue = "all") String category
) { ... }
```

4) **Constraints**

Constraints are typically applied via:

- **Regex in @RequestMapping path**: (Advanced use case)
- **Type-safe constraints** using Java types (int, long)

- Or custom logic using @Validated, @Pattern, or @PathVariable conversion.

Example using regex constraint in @RequestMapping:

```
@GetMapping("/users/{id:[0-9]+}")
public User getUser(@PathVariable int id) { ... }
```

Or using validation annotations:

```
@GetMapping("/users/{username}")
public User getByUsername(
 @PathVariable @Pattern(regexp = "^[a-zA-Z]+$") String username
) { ... }
```

How is the routing carried out in MVC?

What is Routing in MVC?

Routing is a mechanism that maps incoming browser requests (URLs) to a specific controller and action method in an MVC application.

Internal Workflow of Routing in Spring MVC

1. **Client makes a request** (e.g., GET /users/5)
2. **DispatcherServlet** intercepts the request.
3. **HandlerMapping** searches for the controller method matching the pattern.
4. **HandlerAdapter** invokes the controller method.
5. **Response** is generated (View, JSON, etc.) and returned to the client.

How Routing Works in MVC (Step-by-step)

1.User Sends a Request: Example:

https://example.com/Product/Details/5

2.URL Breakdown:

/Product → Controller

/Details → Action Method

/5 → Parameter (like productId = 5)

3.Routing Engine Matches URL:

The routing system checks the route configuration to determine which controller and method should handle this request.

4.Controller and Action Executed:

The ProductController class's Details(int id) method is executed with id = 5.

Difference b/w View and Partial View

View : A View represents the entire page (or full HTML response) rendered and returned to the client. It typically includes the full layout (header, footer, body, etc.).

1) The view is not as lightweight as that of Partial view.
2) The view has its own layout page.
3) The Viewstart page is rendered just before rendering any view.
4) The view can have markup tags of HTML such as HTML, head, body, title, meta, etc.

Partial View : A Partial View is a reusable UI component or fragment that represents only a part of a page, such as a sidebar, navbar, footer, form, or table.

1) Partial view, as the name suggests, is lighter than View.
2) The partial view does not have its own layout page.
3) Partial view is designed particularly for rendering within the view.
4) The partial view does not contain any markup.

How does MVC work in Spring?

1. **Client sends a request** → received by **DispatcherServlet**.

2. **DispatcherServlet** : Front controller that receives all incoming requests and coordinates the flow. **HandlerMapping** Maps incoming request URLs to appropriate controller methods..

3. **Controller** processes the request → calls **Service** layer → returns **ModelAndView** (model + view name). – Handles business logic and returns model data along with the view name.

4. **DispatcherServlet** sends the view name to **ViewResolver**.

5. **ViewResolver** : Maps the logical view name to an actual view resource (like JSP or HTML).

6. **View** uses model data → renders the output → response goes back to the **client**.

   Renders the final output using model data and sends it to the client.

What are some standard Java pre-defined functional interfaces?

Some of the famous pre-defined functional interfaces from previous Java versions(Legacy Interfaces (Before Java 8)) are **Runnable, Callable, Comparator, and Comparable**. While Java 8 introduces functional interfaces like Supplier, Consumer, Predicate, etc.

Runnable: used to execute the instances of a class over another thread with no arguments and no return value.

Callable: used to execute the instances of a class over another thread with no arguments and it either returns a value or throws an exception.

Comparator: use to sort different objects in a user-defined order

Comparable: use to sort objects in the natural sort order

What is Hibernate Caching?

Hibernate Caching is a powerful mechanism provided by Hibernate ORM to enhance application performance by minimizing the number of database hits. It does so by storing objects in memory for repeated access.

When the same query or object is requested multiple times, Hibernate can serve it from the cache instead of fetching it again from the database—thus reducing latency and improving throughput.

1. First Level Cache (Session-level)

- **Enabled by default.**
- Lives within the **Hibernate Session object**.
- **Scope**: Limited to a single session.
- When you retrieve an object for the first time in a session, Hibernate stores it in the first-level cache. Any subsequent call for the same object (within the same session) hits the cache, not the database.
- **Cleared**: When the session is closed or explicitly cleared.

1) This level is enabled by default.

2) The first level cache resides in the hibernate session object.

3) Since it belongs to the session object, the scope of the data stored here will not be available to the entire application as an application can make use of multiple session objects.

✅ Example:

```
Session session = sessionFactory.openSession();

Employee emp1 = session.get(Employee.class, 1); // DB hit


Employee emp2 = session.get(Employee.class, 1); // No DB hit, fetched from first-level cache


session.close();
```

2. Second Level Cache (SessionFactory-level)

- **Needs to be explicitly enabled**.
- Lives in the **SessionFactory** scope.
- **Scope**: Shared across multiple sessions of the same application.
- Used for caching **entity objects**, **collections**, and **queries**.

1) Second level cache resides in the SessionFactory object and due to this, the data is accessible by the entire application.

2) This is not available by default. It has to be enabled explicitly.

3) EH (Easy Hibernate) Cache, Swarm Cache, OS Cache, JBoss Cache are some example cache providers.

Difference between save and saveorupdate

In **Hibernate**, both save() and saveOrUpdate() are methods provided by the Session interface to persist objects

save(Object entity)

- **Purpose:** Persists a new (transient) entity into the database.
- **Returns:** The generated identifier (e.g., primary key).
- **Behavior:**
    - Inserts a new record.
    - If the object is already persistent or has an ID that exists in DB, it may cause a duplicate key exception.
    - Does **not check** if the entity already exists in the database.

✅ Use when:

- You **know** the entity is **new**, and you want to insert it.

```
Employee emp = new Employee();

emp.setName("John");

session.save(emp);  // Will INSERT new record
```

saveOrUpdate(Object entity)

- **Purpose:** Either saves a new entity or updates an existing one.
  Behavior:
    - Checks if the entity has an ID:
        - If **no ID (or ID is not found in DB)** → it performs an **INSERT**.
        - If **ID exists** → it performs an **UPDATE**.
- **Smart decision-making:** Avoids duplicate records by checking identity.

✅ Use when:

- You're unsure whether the object is new or already in the DB.
- You want to insert **or** update based on the presence of an ID.

Employee emp = new Employee();

emp.setId(101); // Existing ID

emp.setName("Alice");

session.saveOrUpdate(emp); // Will UPDATE the existing record if ID 101 exists, otherwise INSERT

Spring Data JPA

Method	Purpose	DB Operation Type	Returns
`find()`	Fetch by ID (JPA-compliant)	SELECT	Entity / null
`findAll()`	Fetch all records	SELECT	List<Entity>
`findById()`	Fetch by ID with `Optional` return (JPA)	SELECT	Optional<Entity>
`loadAll()`	Lazy fetch all entities	SELECT	List<Entity>
`list()`	Criteria API list query	SELECT	List<Entity>
`count()`	Get count of records	SELECT	Long

Differentiate between .ear, .jar and .war files.

The .ear, .jar, and .war files are all used in Java-based applications, but they serve different purposes in the software development and deployment process. Here's a detailed differentiation:

1. .jar Files (Java ARchive)

.jar files are used to package Java class files, associated metadata, and resources like images or configuration files into a single archive.

Contents: Typically, a .jar file contains:

- Java classes
- Libraries or dependencies (e.g., .class files)
- Property files, configuration files, and other resources

Typical Structure of a `.jar` File:

vbnet Copy Edit

```
MyLibrary.jar
|
├── META-INF/
|   ├── MANIFEST.MF  (contains metadata, such as entry point for executable JARs)
|
├── com/             (package structure for your Java classes)
|   └── example/
|       └── MyClass.class
|
├── resources/       (optional, contains resources like property files, XML files)
|   └── config.properties
```

2. .war Files (Web Application ARchive)

.war files are used for packaging web applications, specifically for deployment to a servlet container (e.g., Apache Tomcat).These files are with the .war extension. The .war file contains JSP, HTML, javascript and other files necessary for the development of web applications.

Contents: A .war file contains:

- Web components such as JSP (Java Server Pages), Servlets, HTML, JavaScript, CSS files
- WEB-INF/ directory containing the web.xml configuration file and classes (Servlets, Filters, etc.)
- Static content like images, styles, and scripts.

Typical Structure of a `.war` File:

```vbnet
MyWebApp.war
|
├── META-INF/                  (optional, contains metadata about the web application)
|   └── MANIFEST.MF
|
├── WEB-INF/
|   ├── web.xml                (deployment descriptor for the web application)
|   ├── classes/               (compiled Java classes like Servlets)
|   |   └── com/
|   |       └── example/
|   |           └── MyServlet.class
|   ├── lib/                   (libraries, typically .jar files used in the web app)
|   |   └── some-library.jar
|   └── tlds/                   (optional, for tag libraries used in JSPs)
|
├── index.html                 (web resources like HTML, images, JS, CSS)
├── styles.css
└── script.js
```

3. .ear Files (Enterprise ARchive)

.ear files are used for packaging entire enterprise-level Java applications, which may consist of multiple modules like EJB (Enterprise JavaBeans), web modules, and other types of resources.

Contents: A .ear file contains:

- .jar files for Java classes and libraries
- .war files for web components (web application modules)
- EJB modules, which are responsible for handling business logic in enterprise applications
- META-INF/ directory for the deployment descriptors and configuration files.

An `.ear` file might look like this:

```sql
MyEnterpriseApp.ear
|
├── META-INF/
|   └── application.xml
|
├── webapp.war          (contains web resources like JSP, HTML, etc.)
├── businesslogic.jar   (contains Java classes for business logic)
└── utility.jar         (contains utility classes used across the app)
```

OS Question

Explain any 5 essential UNIX commands .

1) ls -> Lists files in current directory

2) cd -> Change directory to tempdir

3) mkdir -> Make a directory called graphics

4) rmdir -> Remove directory (must be empty)

5) cp -> Copy file into directory

How do you create an immutable class in hibernate?

Immutable class in hibernate creation could be in the following way. If we are using the **XML form of configuration**, then a class can be made immutable by **markingmutable=false**. The default value is true there which indicates that the class was not created by default.

In the case of **using annotations**, immutable classes in hibernate can also be created by using **@Immutable annotation.**

What is Self-Join and Cross-Join ?

Self-Join

A **Self-Join** is a type of join where a table is joined with itself. It is often used when we need to compare rows within the same table. In a self-join, you give the same table different aliases to treat it as if it were two separate tables, allowing you to compare or combine data from different rows.

Cross-Join

A **Cross-Join** is a type of join that returns the Cartesian product of two tables. This means that each row from the first table is paired with every row from the second table, resulting in a combination of all possible pairs of rows.

Let's say we have two tables: one with colors and the other with sizes, and we want to generate all possible combinations of colors and sizes.

Table: colors

color
Red
Blue
Green

Table: sizes

size
S
M
L

Cross-Join Query:

```sql
SELECT c.color, s.size
FROM colors c
CROSS JOIN sizes s;
```

Result:

color	size
Red	S
Red	M
Red	L
Blue	S
Blue	M
Blue	L
Green	S
Green	M
Green	L