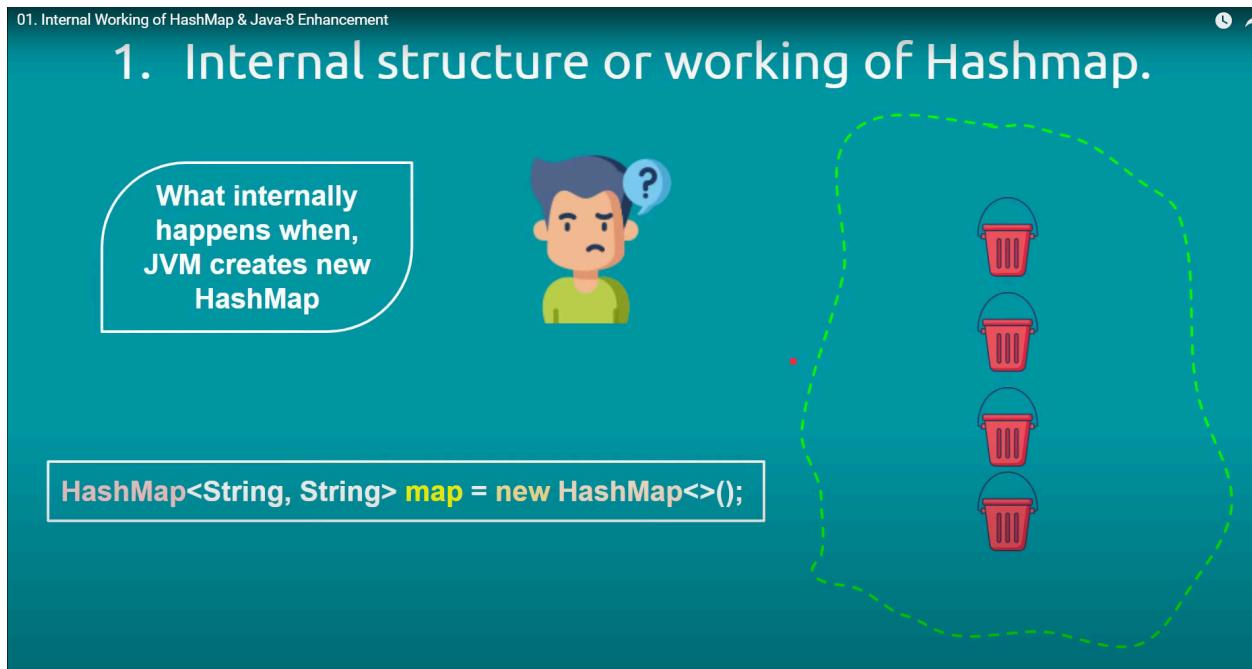


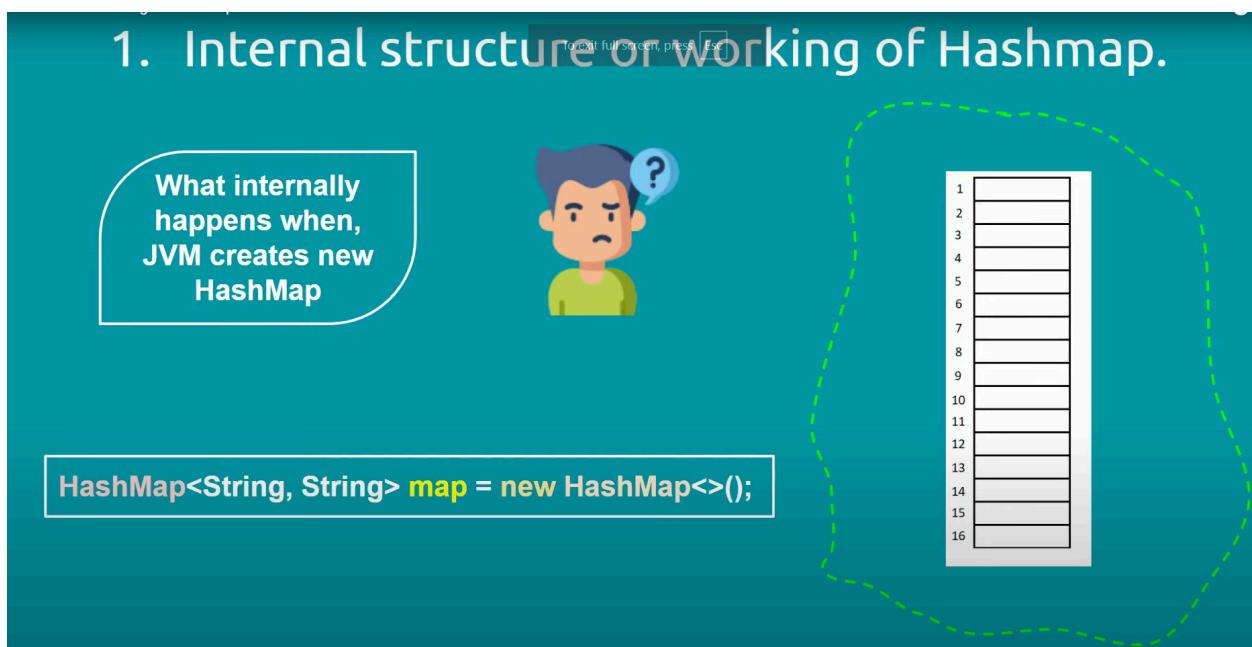
## Experienced Interview Question

### 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<>();

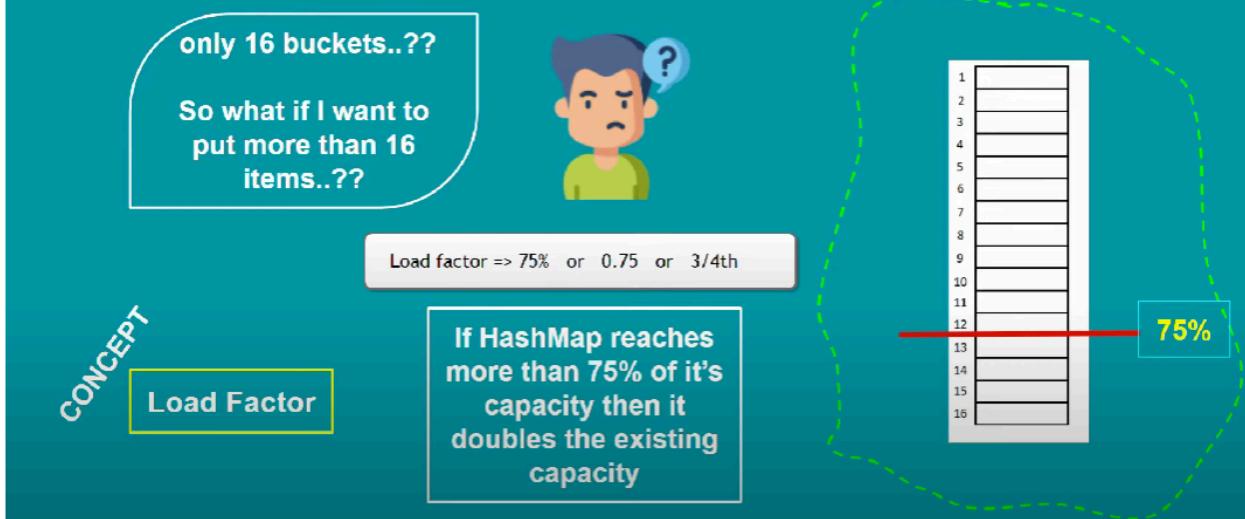
        // Adding 12 elements (75% of 16)
        for (int i = 1; i <= 12; i++) {
            map.put(i, "Value" + i);
        }

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

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

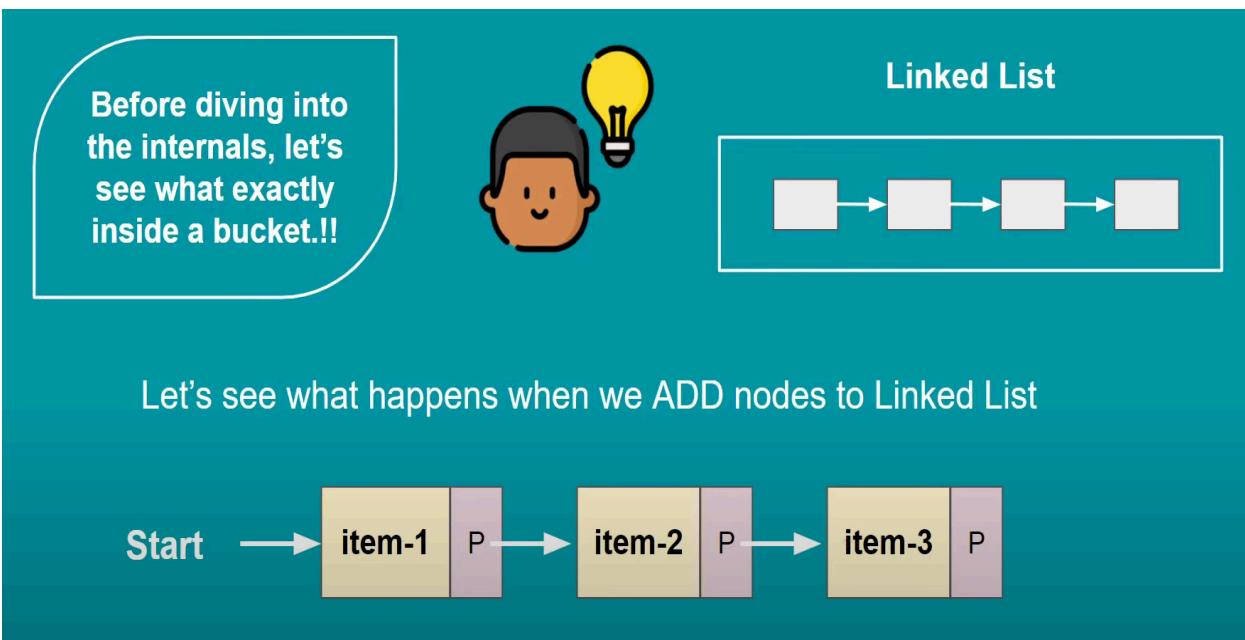
        // After rehashing
        System.out.println("Size after threshold breach: " + map.size());
    }
}
```

## 1. Internal structure or working of Hashmap.

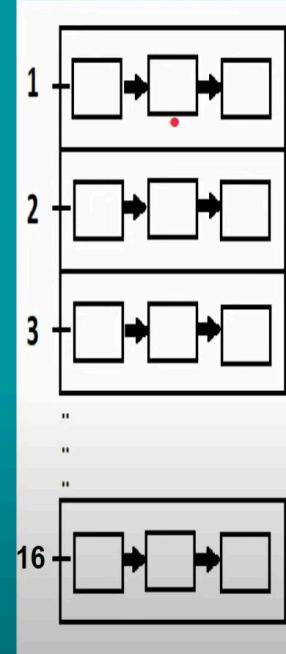


Bucket is nothing but a linked list here.

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.



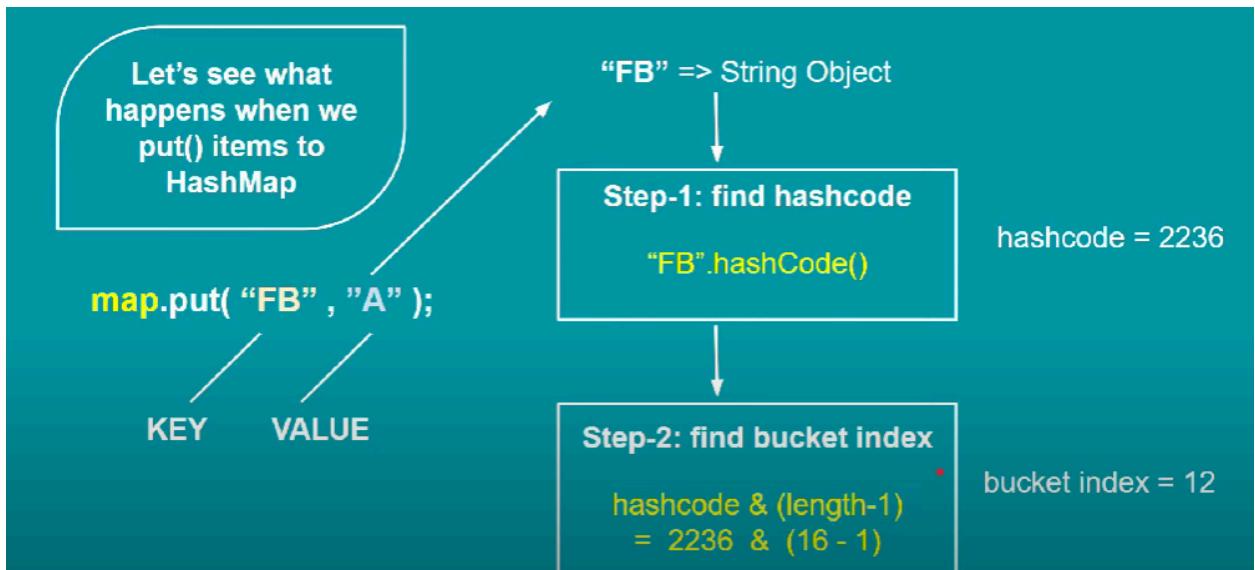
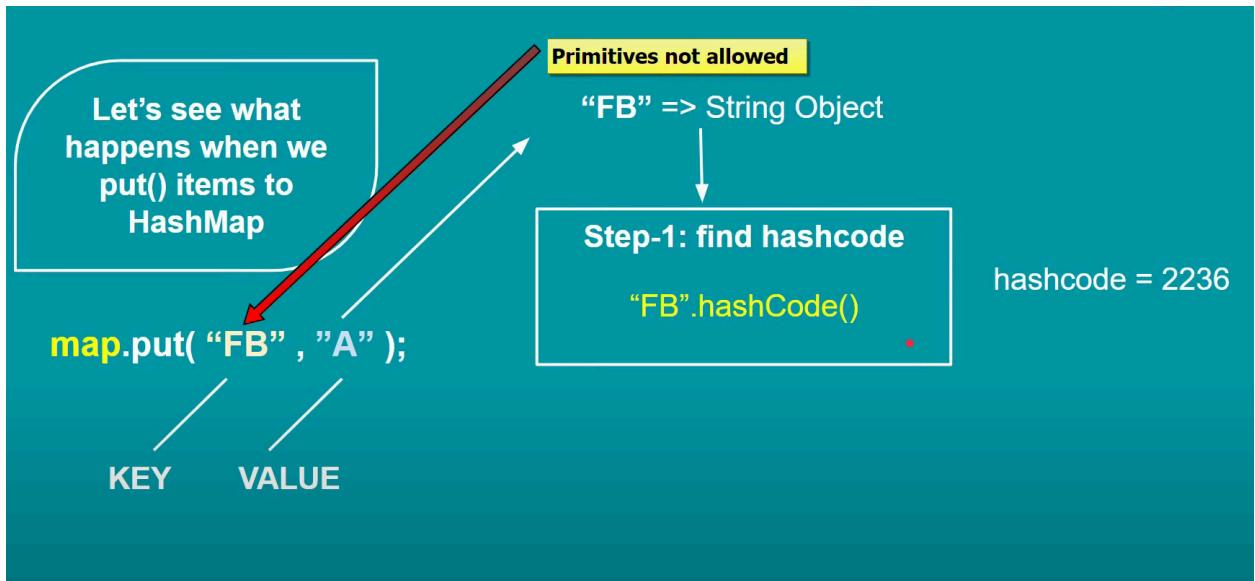
**Now we know  
How exactly  
buckets of  
HashMap look  
like**



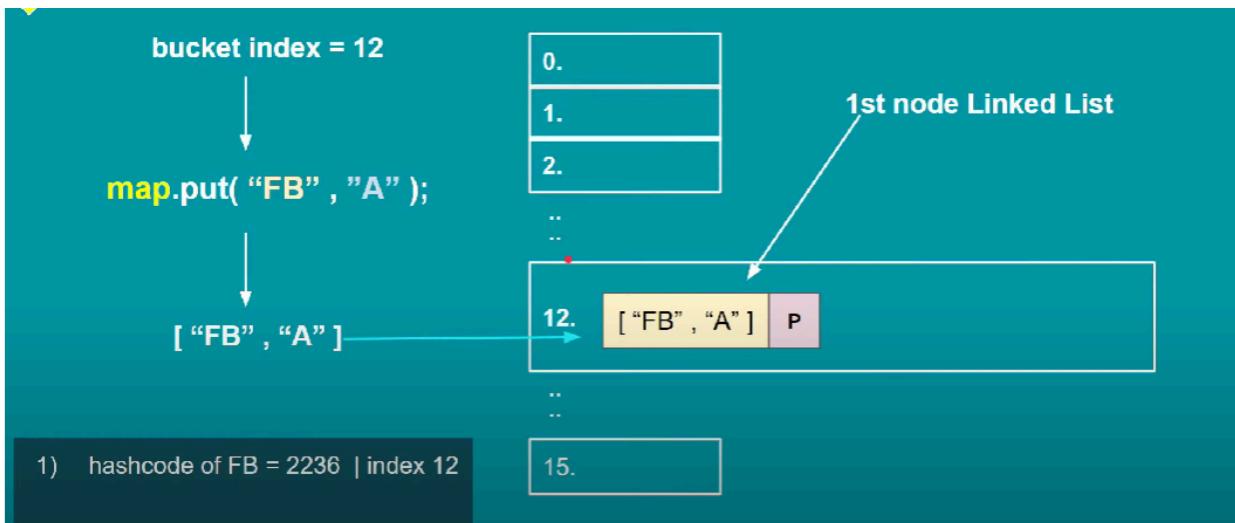
**primitive types** (e.g., int, double, char, etc.) are **not allowed** as keys in a **HashMap**. However, their **wrapper classes** (Integer, Double, Character, etc.) **can be used** as keys because they are objects.

**Put Method of HashMap**

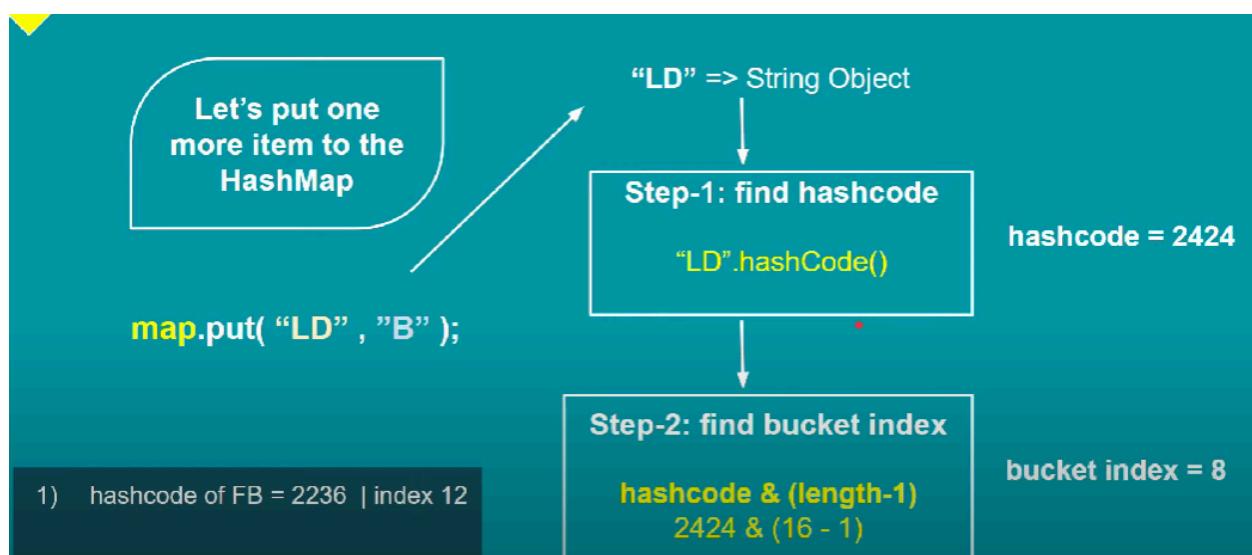
**First item**

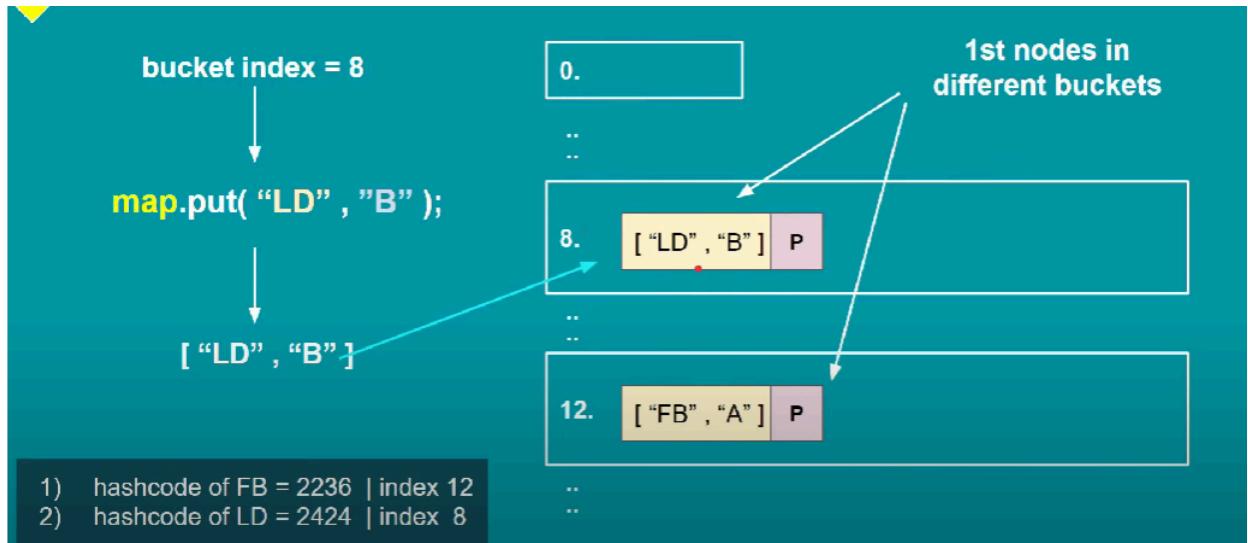


So the date first key & value will be stored in bucket index=12

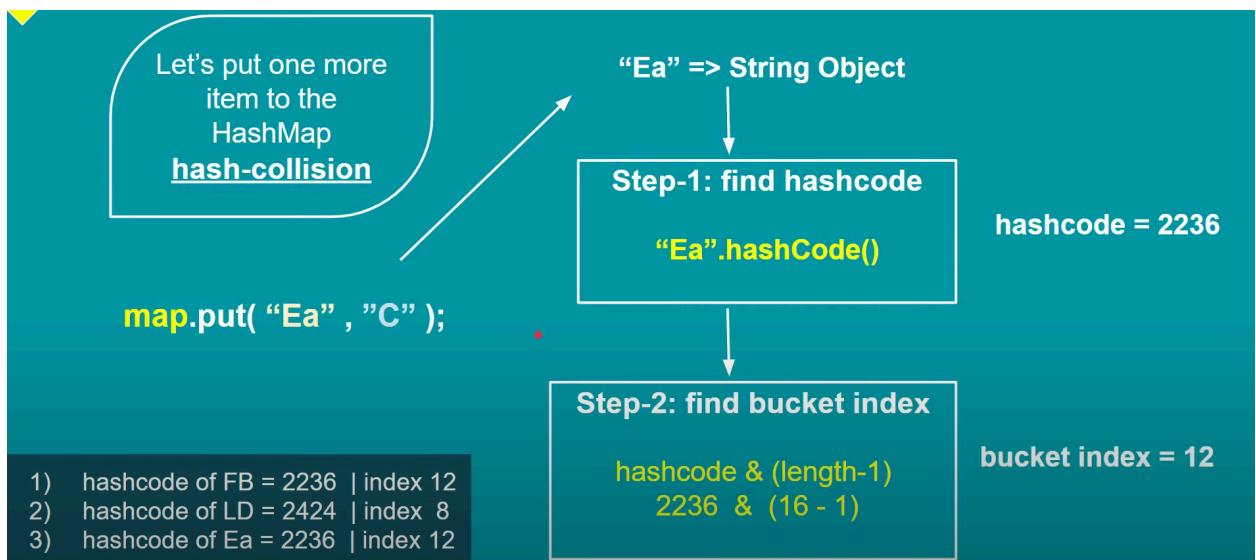


## Second item





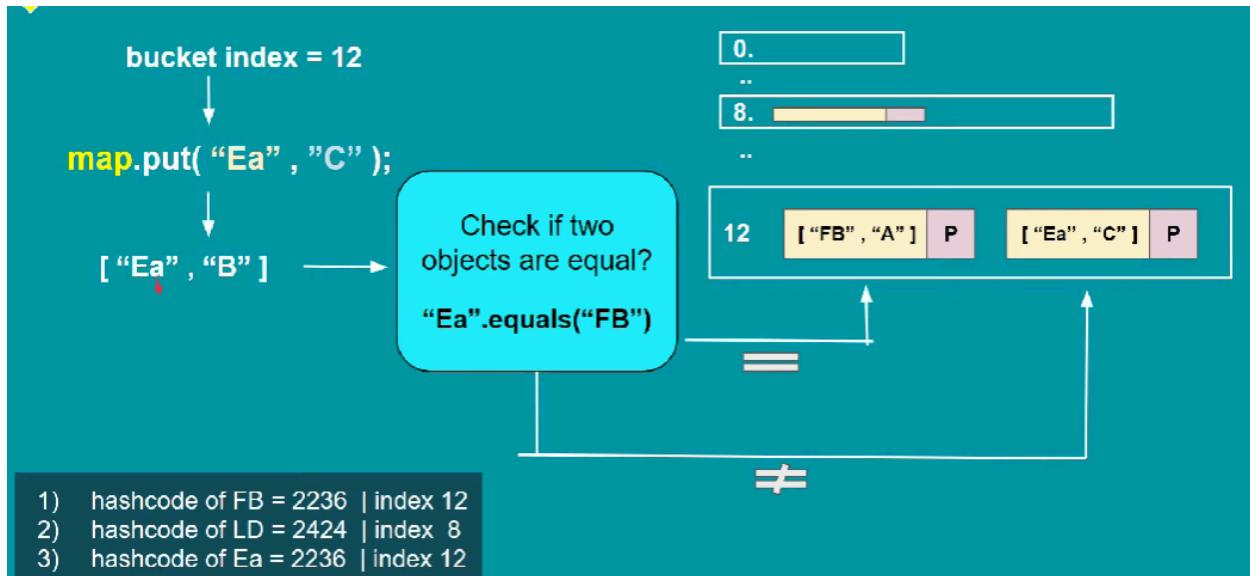
### Third item



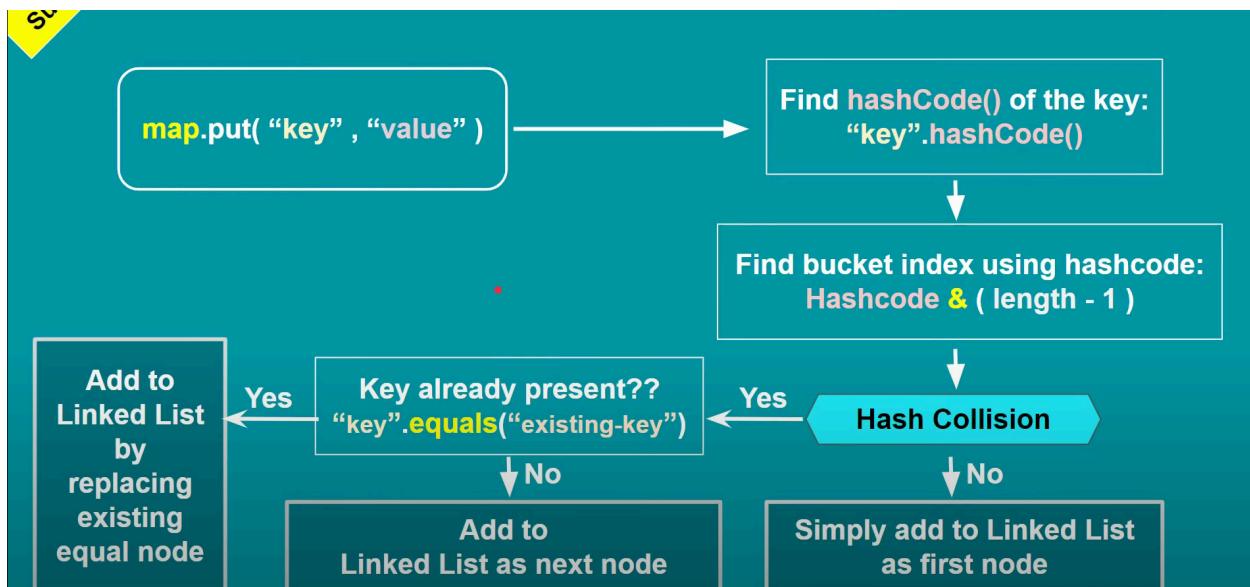
Here is hash collision , as the hashCode of the third(2236) item is the same as the first(2236) item .

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



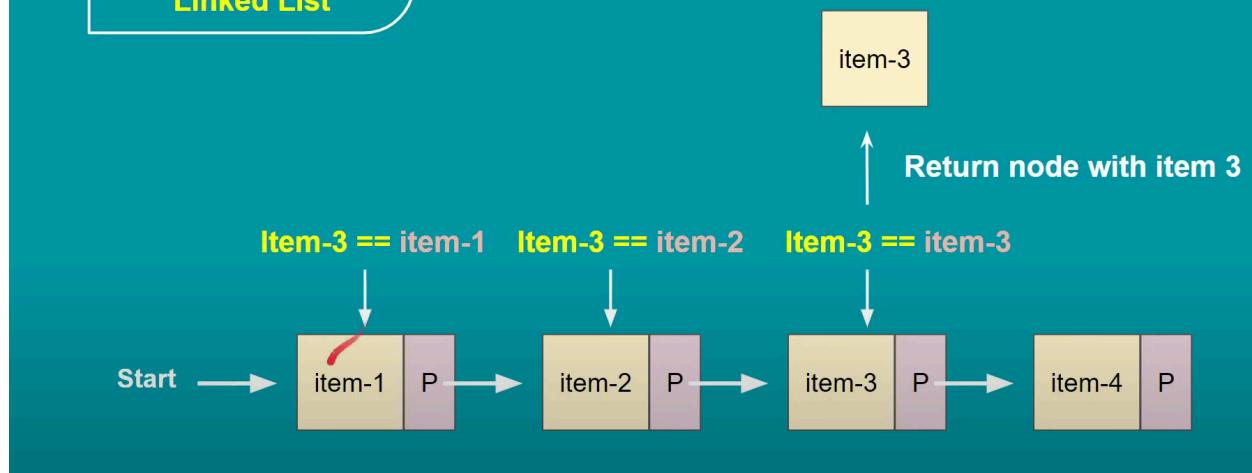
### Get Method of HashMap

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).

## Searching in Linked List

How to find node with item-3 ?



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.

```
HashMap<String, String> map = new HashMap<>();
```

```
map.put ("AaAaAa" , "v1"); => 1952508096  
map.put ("AaAaBB" , "v2"); => 1952508096  
map.put ("AaBBAa" , "v3"); => 1952508096  
map.put ("AaBBBB" , "v4"); => 1952508096  
map.put ("BBAaAa" , "v5"); => 1952508096  
map.put ("BBAaBB" , "v6"); => 1952508096  
map.put ("BBBBAa" , "v7"); => 1952508096  
map.put ("AaBBBB" , "v8"); => 1952508096  
map.put ("BBBBBB" , "v9"); => 1952508096
```



0.
1.
2.

### Performance Degradation



..

..

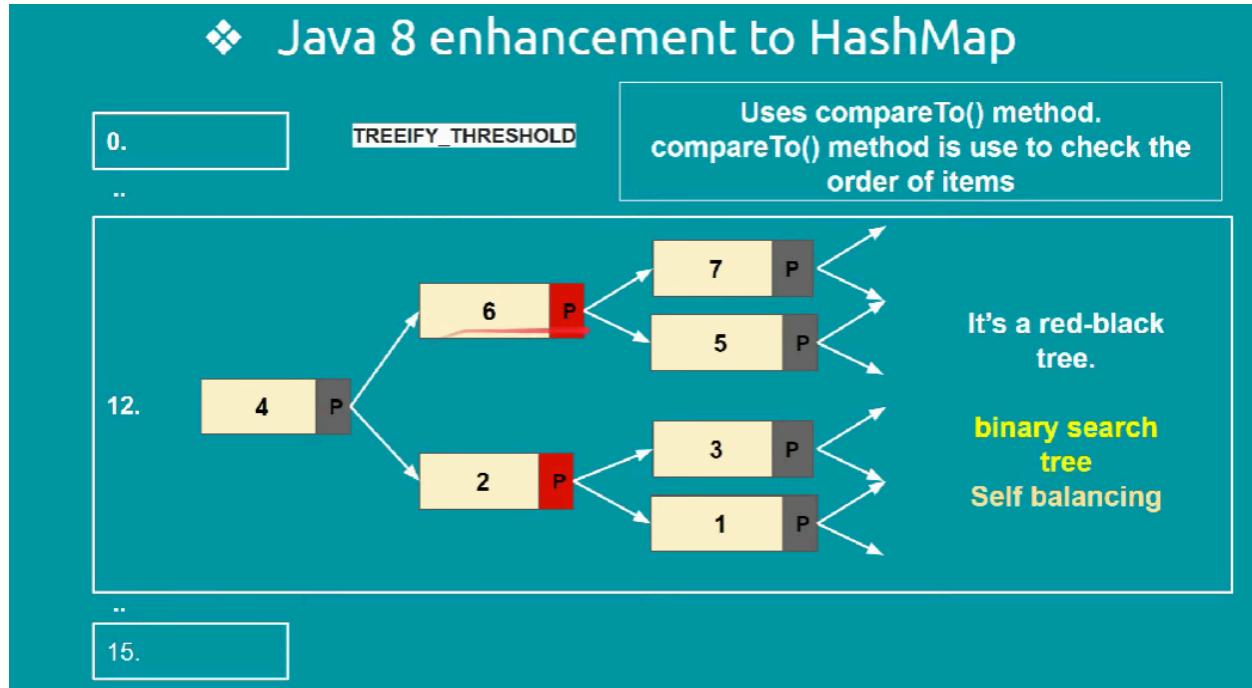
15.

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:

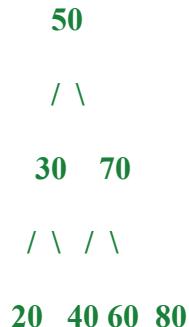
A tree data structure, specifically a Binary Search Tree (BST) or Red-Black Tree (used in Java 8+ HashMap), stores data in a hierarchical way based on comparisons.

#### 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.

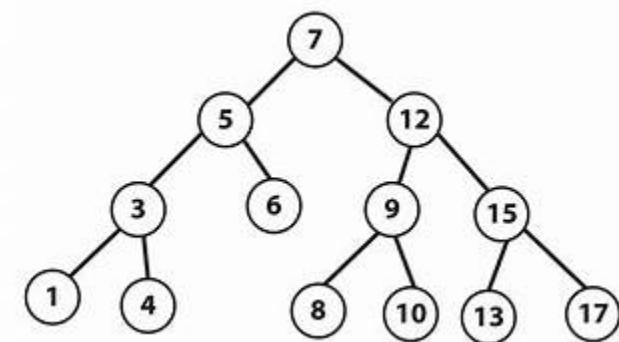
#### Example: Inserting Numbers into a BST

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



- 50 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?

## 2. Equals & HashCode Contract and variations with HashMap

Default implementations of **equals()** and **hashcode()** methods??



```
Object o1 = new Object();
Object o2 = o1;
```

```
Object o3 = new Object();
```

"new" keyword is use to create  
new object in heap

Heap Memory

If we 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 we will understand **equals** method using above example

Default implementations of **equals()** and **hashcode()** methods??



```
Object o1 = new Object();
Object o2 = o1;
```

```
Object o3 = new Object();
```

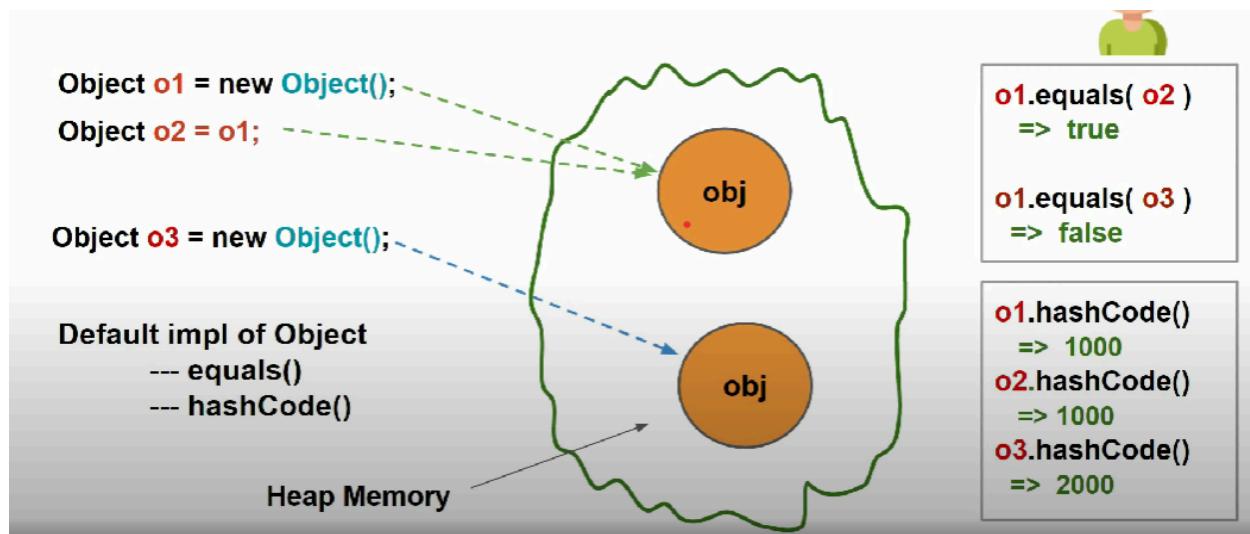
Default impl of Object  
--- equals()  
--- hashCode()

Heap Memory

**o1.equals( o2 )**  
=> true  
  
**o1.equals( o3 )**  
=> false

**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.

**What is the Contract ??**

**What is contract between equals and hashCode method?**



**If you are overriding equals() method..**

Then you must override hashCode method properly..

meaning, if two objects are equal then hashCode method must return same value..

```
Student {
```

```
    private int rollNum;  
    private String name;
```

```
    @Override  
    public boolean equals(Student s2) {  
        ....  
    }
```

```
    @Override  
    public int hashCode() {  
        ....  
    }  
}
```

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.

Let's see that practically ??



```
Student s1 = new Student( 1 , "Sam" );
Student s2 = new Student( 1 , "Sam" );
```

As per the equals() method.....

```
s1.equals(s2) => true
```

As per the contract.....

```
s1.hashCode() == s2.hashCode() <= Must
```

```
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
}
```

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..

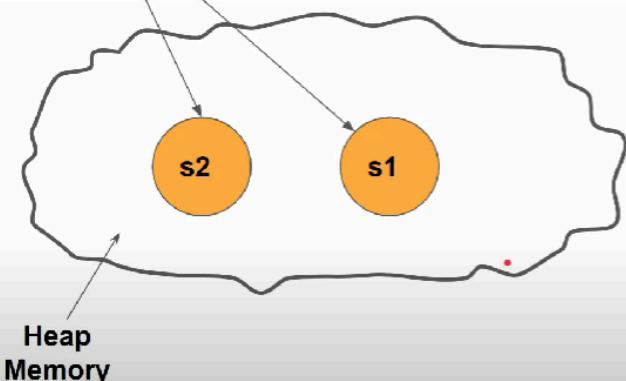
So, let's see that in action for  
HashMap!

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 s1 = new Student ( 1 , "Aman" );  
Student s2 = new Student ( 1 , "Aman" );  
equal



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

```
Map<Student, String> map = new HashMap<>();
```

```
map.put (s1, "A");
```



```
s1.hashCode()
```

```
121212
```

```
Bucket index: 8
```

```
map.put (s2, "A");
```



```
s2.hashCode()
```

```
454545
```

```
Bucket index: 15
```

```
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
```

```
}
```

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.

Example

```
Map<Student, String> map = new HashMap<>();
```

```
Student s1 = new Student ( 1, "Sachin" );  
Student s2 = new Student ( 2, "Yuvি" );
```

```
s1.equals(s2) => false
```

```
s1.equals(s1) => false !!!
```

```
map.put ( s1 , "A" );  
map.put ( s2 , "A" );
```

```
map.put ( s1 , "A" );  
map.put ( s1 , "A" );
```

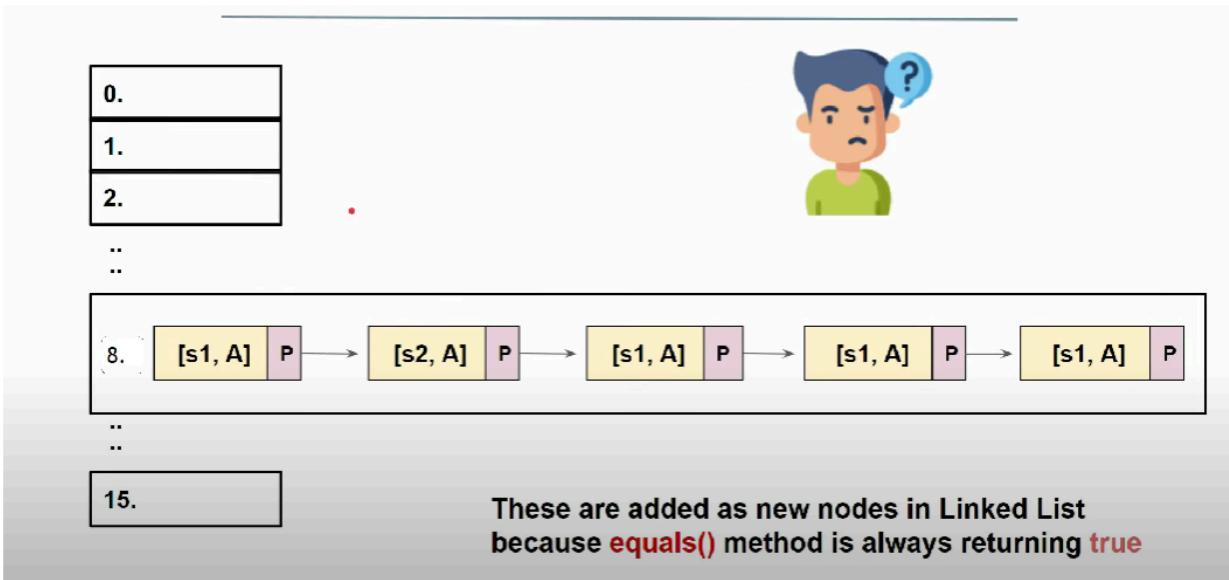
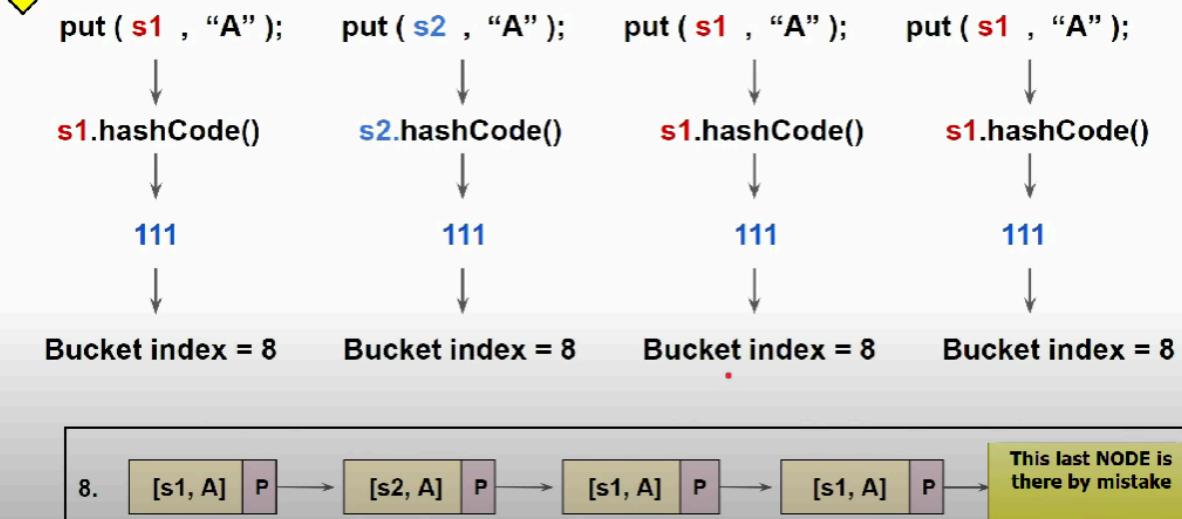
```
Student {
```

```
private int rollNum;  
private String name;
```

```
@Override  
public boolean equals(Student s2) {  
    return false  
}
```

```
@Override  
public int hashCode() {  
    return 111  
}
```

Exa



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:**
  - o If it does not override **show()**, there will be a compilation error due to ambiguity.
  - o 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"; X Not Allowed
        // student.age = 30; X 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
<b>Definition</b>	It defines natural ordering within the class.	It defines external sorting logic.
<b>Method</b>	compareTo()	compare()
<b>Implementation</b>	It is implemented in the class.	It is implemented in a separate class.
<b>Sorting Criteria</b>	Natural order sorting	Custom sorting
<b>Usage</b>	It is used for single sorting order.	It is used for multiple sorting orders.

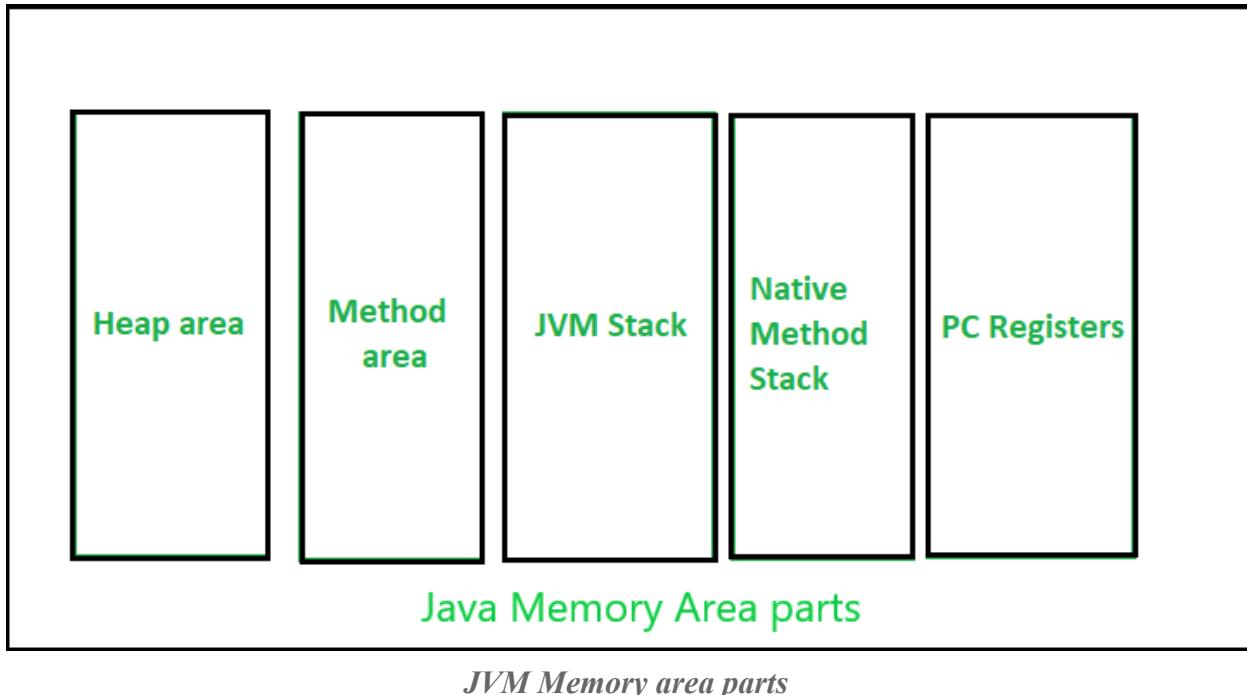
## 6) Metaspace vs. PermGen in Java

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

### JVM Memory Structure:

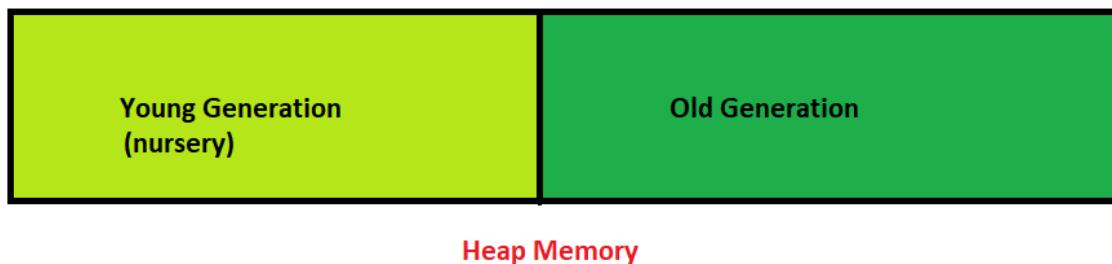
JVM defines various run-time data areas which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of the thread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

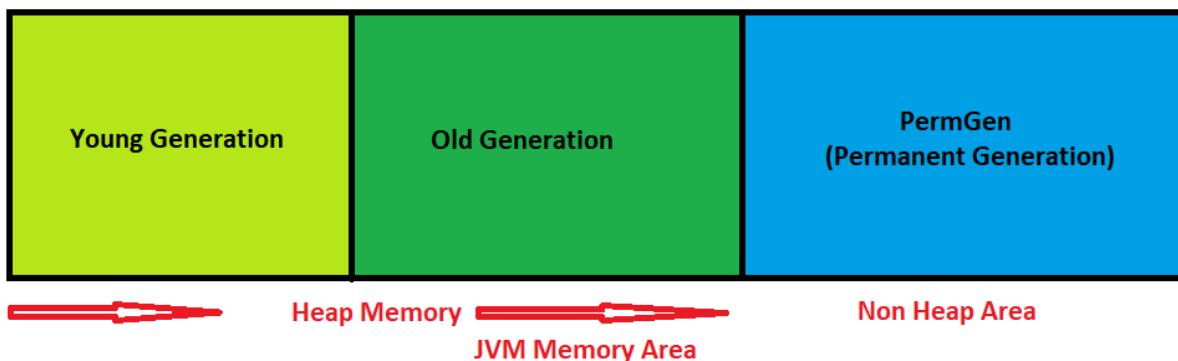


Here, the heap area is one of the most important memory areas of JVM. Here, all the [java objects](#) are stored. The heap is created when the JVM starts. The heap is generally divided into two parts. That is:

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?

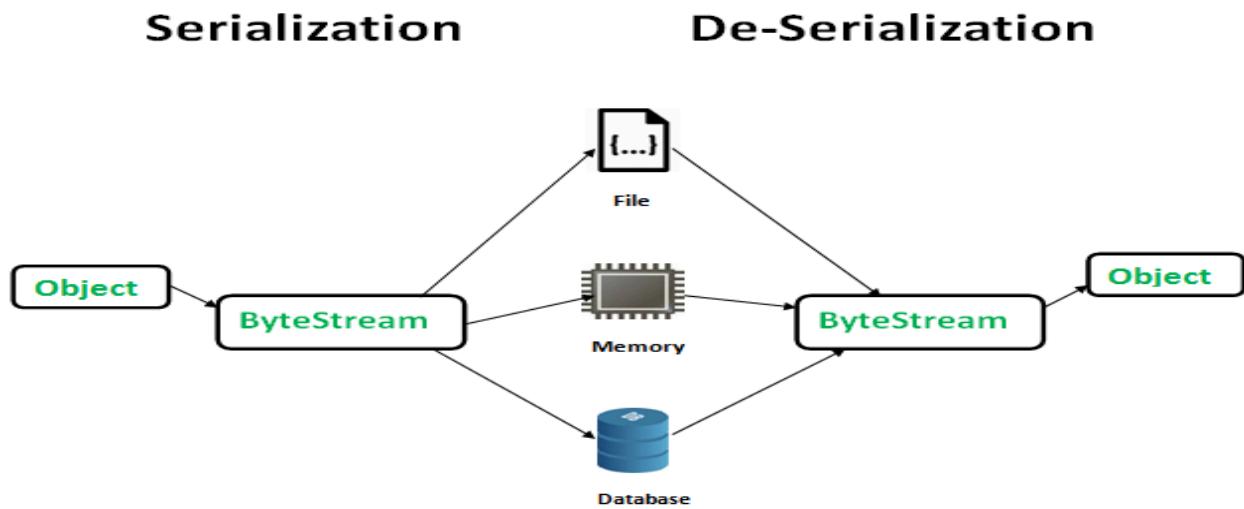
Serialization in Java is the process of converting an object into a byte stream so that it can be:

- 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 serializedPerson = (Person) ois.readObject();
            System.out.println("Deserialization Successful!");
            serializedPerson.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:

### 1 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)`

## 2 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
    }
}

```

```
}
```

```
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 monolithic architecture and drawbacks?

- All features are in one application
- Application is deployed as single unit
- Any changes to features require complete deployment
- Scaling of individual component is difficult
- Application becomes big and complex over time
- Bug in one module can bring down the entire application



## 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

Product Information

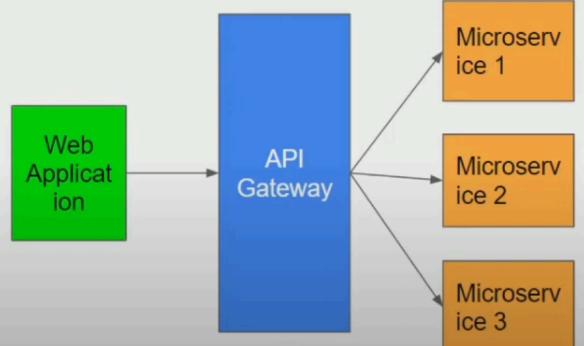
Pricing Information

Customer Review

User

## What is API Gateway?

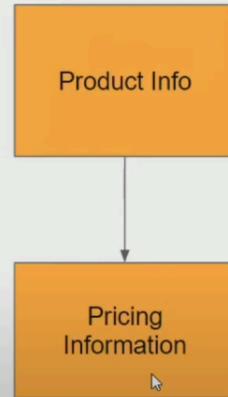
- API Gateway sits between client and microservices
- It acts as an entry point to the microservices
- All HTTP calls shall go through API gateway
- Common cross cutting concerns like Authentication, Authorization, Rate limiting can be handled
- For high availability you can deploy multiple instances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

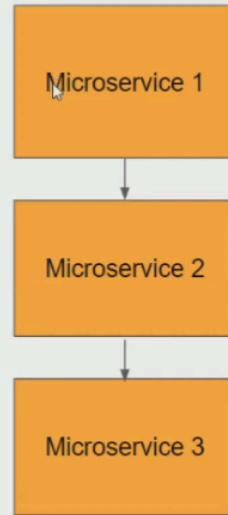
http://10.15.20.25:8080



http://10.20.30.40:9090

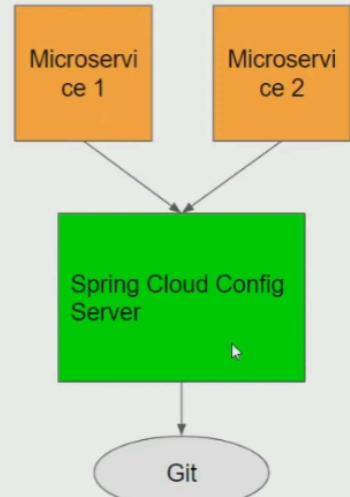
## How do you handle fault tolerance in microservices?

- 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



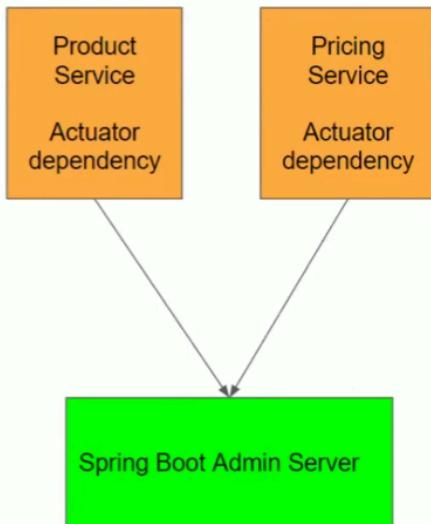
## What is spring cloud config server?

- Externalize configuration in a distributed system
- For example you can read the properties from git project



## How do you monitor your microservices?

- Use Spring Boot admin server to monitor services
- Each microservice must have actuator dependency in it
- Each microservice should register with spring boot admin server
- Spring boot admin server provide the web user interface to monitor microservices



## 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.println(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("Plaintrome");
        else
            System.out.println(" not Plainrome");
    // 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

Accenture Interview Question:

Employee has two fields name and city.

There is a list of employees with below details.

You need to group by employees with the city.

Input:

```
Name = Amar City = Pune  
Name = Raj City = Pune  
Name = Neha City = Mumbai  
Name = Sam City = Mumbai
```

Output:

```
Pune = Amar, Raj  
Mumbai = Neha, Sam
```

```
package com.ram;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Map;  
import java.util.stream.Collectors;  
class Employeeess {  
    private String name;  
    private String city;  
    public Employeeess(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<Employeeess> list = new ArrayList<Employeeess>();
        list.add(new Employeeess("Amar", "Pune"));
        list.add(new Employeeess("Raj", "Pune"));
        list.add(new Employeeess("Neha", "Mumbai"));
        list.add(new Employeeess("Sam", "Mumbai"));
        System.out.println("Before grouping");
        for (Object o : list) {
            System.out.println(o);
        }
        // After
        System.out.println("After grouping");
        Map<String, List<Employeeess>> 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** I  
125

**Output**  
125 is not Armstrong number

```
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() {  
        counter = 20;  
    }  
    Test(int x){  
        counter = x;  
    }  
    public static void main(String[] args) {  
        Test Test = new Test();  
        System.out.println(counter);  
    }  
}
```

Object name is test instead of Test  
**Test test = new Test();**

```
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

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

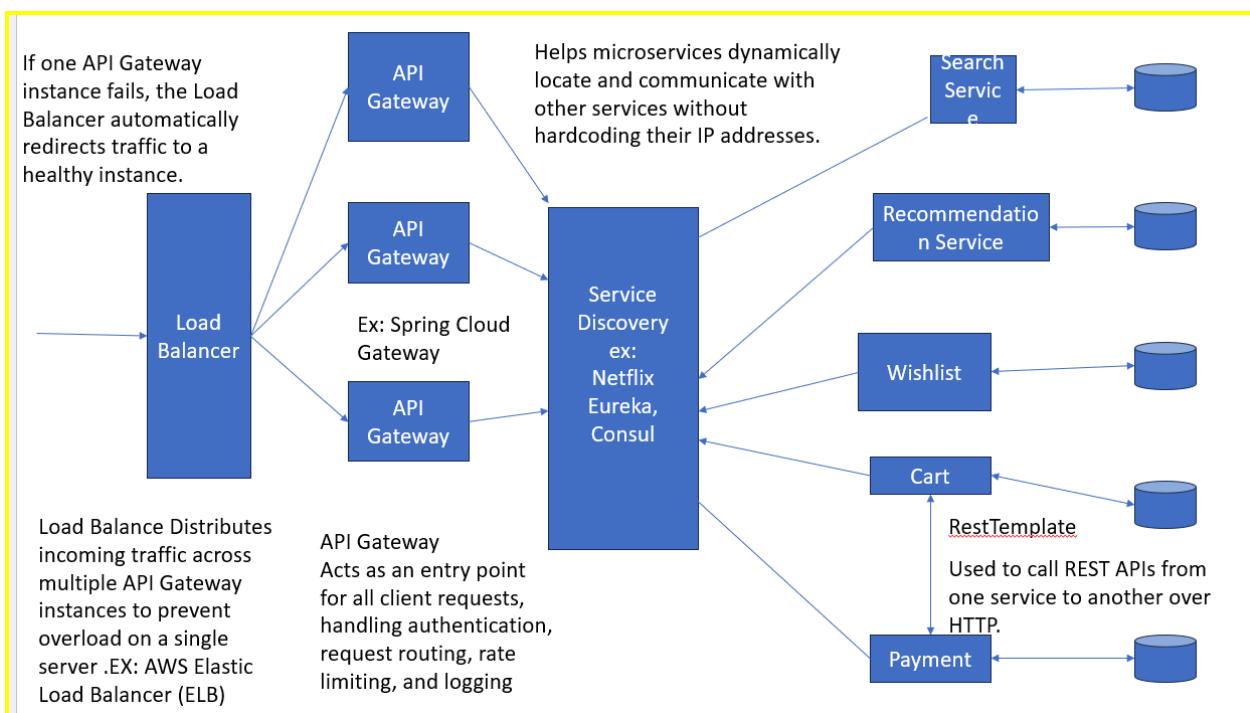
```
package com.ram;  
import java.util.Arrays;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Set;  
import java.util.stream.Collectors;  
public class DuplicateNumbersusingJava7ndJava8 {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(10, 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]

```
5 public class Program1 {
6     public static void main(String[] args) {
7         List<Integer> numbers = Arrays.asList(10, 20, 20, 25, 25, 27, 27, 30);
8
9         Set<Integer> hs = new HashSet<Integer>();
10
11         numbers.stream().filter(n -> !hs.add(n)).forEach(System.out::println);
12
13     }
14
15 }
```

## 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

### 1 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.
-

## 2 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.
- 

## 3 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.
- 

## 4 Search Service

### ♦ What is it?

Fetches products from the database.

- Returns results based on user queries.

## 5 Recommendation Service

### ♦ What is it?

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

## 6 Wishlist Service

### ♦ What is it?

Allows users to save products for future purchases.

## 7 Cart Service

### ♦ What is it?

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

## 8 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.

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

```

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.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.

The screenshot shows the Eclipse IDE interface with the following details:

- Package Explorer:** Shows the project structure under "InterviewQuestions". The "src" folder contains numerous Java files, with `TryWithResource.java` currently selected.
- TryWithResource.java Content:** The code implements the `Closeable` interface and uses a `try-with-resources` statement. It prints "sample method" when the resource is initialized and "closing method" when it is closed.
- Execution Output:** The bottom console shows the output of running the application: "sample method" followed by "closing method".

```
1 package com.basic;
2
3 import java.io.Closeable;
4 import java.io.IOException;
5
6 class MyResource implements Closeable {
7     public void sampleMethod() {
8         System.out.println("sample method");
9     }
10
11     @Override
12     public void close() throws IOException {
13         System.out.println("closing method");
14     }
15 }
16
17 }
18
19 public class TryWithResource {
20
21     public static void main(String[] args) {
22         try (MyResource mr = new MyResource()) {
23             mr.sampleMethod();
24
25         } catch (Exception e) {
26             // e.printStackTrace();
27             System.out.println("Exception caught: " + e.getMessage());
28         }
29     }
30 }
31
32 }
```

Problems @ Javadoc Declaration Console <terminated> TryWithResource [Java Application] C:\Users\jakkula.ramesh\Downloads\eclipse-java-2021-06-R-win32\sample method closing method

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

```
package com.smartwatch;

public interface StepCounter {
    void countSteps();
}

package com.smartwatch;

public interface HeartRateTracker {
    void keepTrackOfHeartRate();
}

package com.smartwatch;

public interface NotificationReceiver {
    void receiveNotification();
}

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.

```

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() {

```

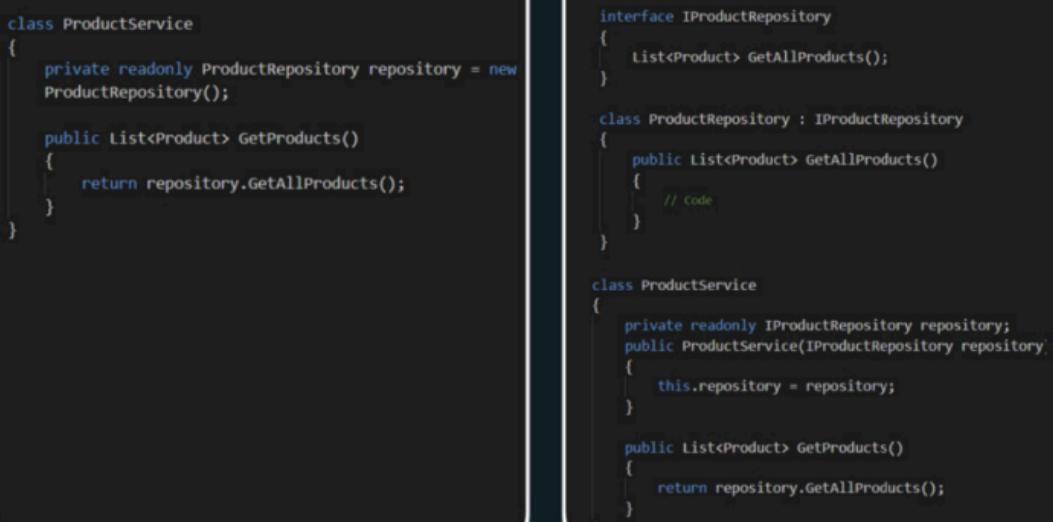
```
        System.out.println("Human is working...");  
    }  
  
    public void eat() {  
        System.out.println("Human is eating...");  
    }  
  
}  
  
class RobotWorker implements Workable {  
    public void work() {  
        System.out.println("Robot is working...");  
    }  
}
```

## **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).

# Dependency Inversion Principle

*High-level modules should depend on abstractions, not on concretions.*



```
class ProductService
{
    private readonly ProductRepository repository = new ProductRepository();

    public List<Product> GetProducts()
    {
        return repository.GetAllProducts();
    }
}
```

```
interface IProductRepository
{
    List<Product> GetAllProducts();
}

class ProductRepository : IProductRepository
{
    public List<Product> GetAllProducts()
    {
        // Code
    }
}

class ProductService
{
    private readonly IProductRepository repository;
    public ProductService(IProductRepository repository)
    {
        this.repository = repository;
    }

    public List<Product> GetProducts()
    {
        return repository.GetAllProducts();
    }
}
```

## 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.
  - o `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

The screenshot shows an IDE interface with a code editor and a console window. The code in the editor is:

```
1 package com.random;
2
3 import java.util.Random;
4
5 public class RandomExample2 {
6
7     public static void main(String[] args) {
8
9         Random random = new Random();
10        System.out.println("Random Integer: " + random.nextInt(10,100));
11        System.out.println("Random Integer (0 to 99): " + random.nextInt(100));
12        System.out.println("Random Double: " + random.nextDouble());
13        System.out.println("Random Boolean: " + random.nextBoolean());
14    }
15
16 }
17
```

The console window shows the output of the program:

```
@ 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
1 package com.random;
2
3 import java.util.Random;
4
5 public class RandomExample {
6
7     public static void main(String[] args) {
8         Random rand = new Random();
9         rand.ints(5,10,100).forEach(System.out::println);
10        //if you want sort them
11        //rand.ints(5,10,100).sorted().forEach(System.out::println);
12    }
13 }
14
```

<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
1 package com.random;
2
3 import java.util.Random;
4 import java.util.stream.IntStream;
5
6 public class RandomExample {
7
8     public static void main(String[] args) {
9         Random rand = new Random();
10        IntStream randomInts = rand.ints(5, 0, 100);
11        randomInts.forEach(System.out::println);
12    }
13 }
14
```

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

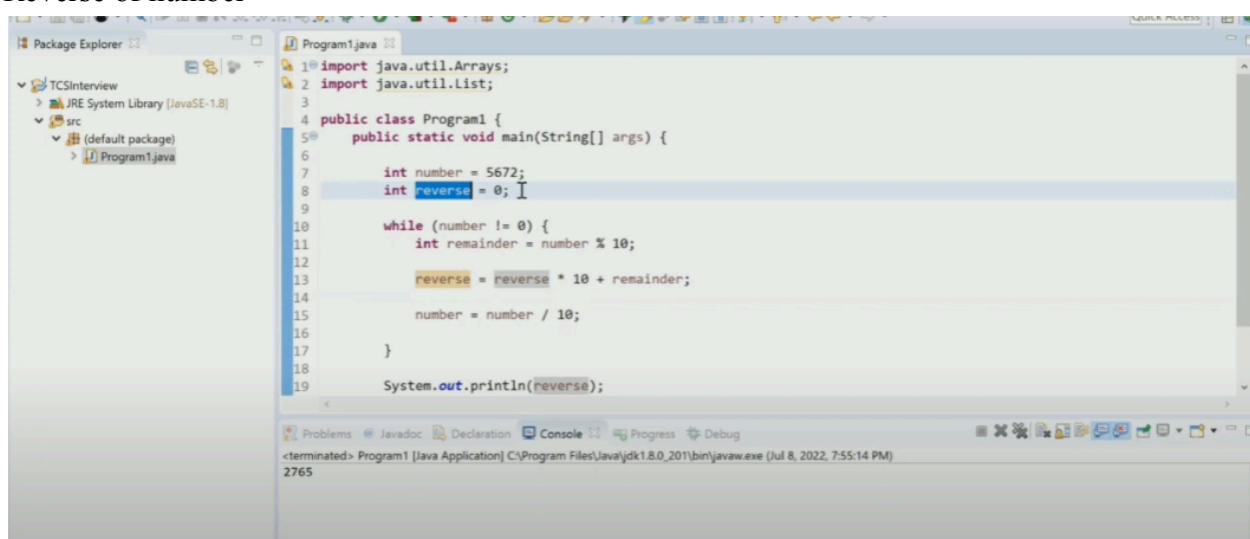
```

**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



The screenshot shows the Eclipse IDE interface with the following details:

- Package Explorer:** Shows a project named "TCSInterview" with a "src" folder containing a file named "Program1.java".
- Editor:** Displays the Java code for "Program1.java". The code defines a class "Program1" with a "main" method. It initializes an integer "number" to 5672 and an integer "reverse" to 0. A while loop runs as long as "number" is not zero. Inside the loop, it calculates the remainder of "number" divided by 10 and adds it to "reverse" multiplied by 10. Then, it divides "number" by 10. Finally, it prints the reversed number using `System.out.println(reverse);`.
- Console:** Shows the output of the program: `2765`.

## 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<Employee> employees = Arrays.asList(
            new Employee("Emp1", "CS", 10000),
            new Employee("Emp2", "CS", 15000),
            new Employee("Emp3", "IT", 20000),
            new Employee("Emp4", "IT", 25000)
        );
        Map<Object, Optional<Employee>> 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<Employee> employees = Arrays.asList(
            new Employee("Emp1", "CS", 10000),
            new Employee("Emp2", "CS", 15000),
            new Employee("Emp3", "IT", 20000),
            new Employee("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}

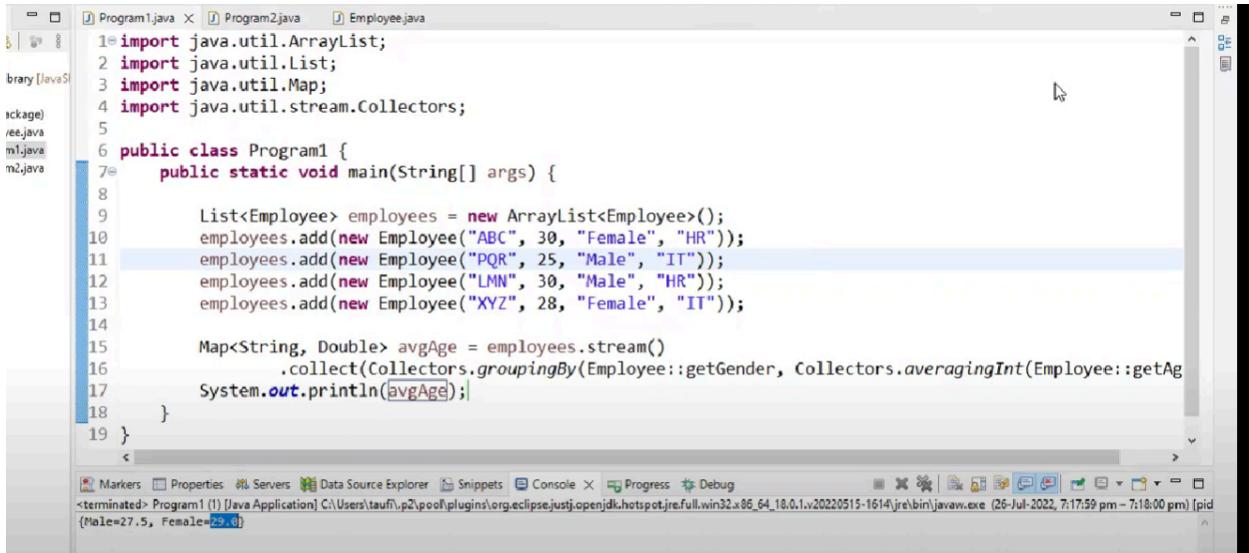
    }
}

```

CS  
IT  
4

{CS=2, IT=2}

And also Average of ages of genders



The screenshot shows the Eclipse IDE interface with the following code in Program1.java:

```
1 import java.util.ArrayList;
2 import java.util.List;
3 import java.util.Map;
4 import java.util.stream.Collectors;
5
6 public class Program1 {
7     public static void main(String[] args) {
8
9         List<Employee> employees = new ArrayList<Employee>();
10        employees.add(new Employee("ABC", 30, "Female", "HR"));
11        employees.add(new Employee("PQR", 25, "Male", "IT"));
12        employees.add(new Employee("LMN", 30, "Male", "HR"));
13        employees.add(new Employee("XYZ", 28, "Female", "IT"));
14
15        Map<String, Double> avgAge = employees.stream()
16            .collect(Collectors.groupingBy(Employee::getGender, Collectors.averagingInt(Employee::getAge)));
17        System.out.println(avgAge);
18    }
19 }
```

The code uses Stream API methods like `stream()`, `groupingBy()`, and `averagingInt()` to calculate the average age for each gender. The output in the console shows: `[Male=27.5, Female=29.0]`.

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	<code>stream()</code> (Sequential Stream)	<code>parallelStream()</code> (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 <code>ForkJoinPool</code> (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

The screenshot shows an IDE interface with two files open: Program1.java and Patient.java. The Program1.java file contains the following code:

```
1 import java.util.Arrays;
2 import java.util.List;
3
4 public class Program1 {
5
6     public static void main(String[] args) {
7
8         Patient p1 = new Patient("P1", 20, "Corona", 18000);
9         Patient p2 = new Patient("P2", 25, "Fever", 6000);
10        Patient p4 = new Patient("P4", 29, "Corona", 8000);
11        Patient p3 = new Patient("P3", 23, "Corona", 12000);
12
13        List<Patient> patients = Arrays.asList(p1, p2, p3, p4);
14
15        patients.stream().filter(p -> p.getDisease().equals("Corona")).forEach(System.out::println);
16
17    }
18
19 }
```

The Patient.java file defines a Patient class with attributes name, age, disease, and amount.

In the IDE's console tab, the output of the program is shown:

```
<terminated> Program1 (1) [Java Application] C:\Program Files\Java\jdk1.8.0_201\bin\javaw.exe (Aug 21, 2022, 10:46:38 AM)
Patient [name=P1, age=20, disease=Corona, amount=18000]
Patient [name=P3, age=23, disease=Corona, amount=12000]
Patient [name=P4, age=29, disease=Corona, amount=8000]
```

Average

```
1 import java.util.Arrays;
2 import java.util.List;
3 import java.util.stream.Collectors;
4
5 public class Program1 {
6
7     public static void main(String[] args) {
8
9         Patient p1 = new Patient("P1", 20, "Corona", 18000);
10        Patient p2 = new Patient("P2", 26, "Fever", 6000);
11        Patient p4 = new Patient("P4", 29, "Corona", 8000);
12        Patient p3 = new Patient("P3", 23, "Corona", 12000);
13
14        List<Patient> patients = Arrays.asList(p1, p2, p3, p4);
15
16        Double averageBillPaid = patients.stream().filter(p -> p.getDisease().equals("Corona"))
17            .collect(Collectors.averagingDouble(Patient::getAmount));
18
19        System.out.println("Average Bill Paid " + averageBillPaid);
    }
}

```

Problems 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

```
1 Program1.java X Patient.java
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 public class Program1 {
7
8     public static void main(String[] args) {
9
10        Patient p1 = new Patient("P1", 20, "Corona", 18000);
11        Patient p2 = new Patient("P2", 26, "Fever", 6000);
12        Patient p4 = new Patient("P4", 29, "Corona", 8000);
13        Patient p3 = new Patient("P3", 23, "Corona", 12000);
14
15        List<Patient> patients = Arrays.asList(p1, p2, p3, p4);
16
17        patients.stream().filter(p -> p.getDisease().equals("Corona") && p.getAge() < 25).forEach(System.out::println);
18
19    }
}

```

Problems 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:**

```

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:**

```

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 )

## ① 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
}

```

---

## ② 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
}
```

---

## ③ 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
}
```

---

## ④ 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

Copy Edit

```

<?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 <code>@Value("value")</code>	✗ No	✗ No	When values are constant.
<code>@Value("\${property}")</code>	✗ No	✓ Yes	When values need to be configurable.
SpEL with <code>@Value("\${property:default}")</code>	✗ 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
<code>none</code>	Disables schema generation	Use in production when schema is managed manually
<code>update</code>	Updates schema (adds missing columns but doesn't remove existing data)	Use in development & testing
<code>create</code>	Drops and recreates tables on every startup	Good for testing but <b>destroys data</b>
<code>create-drop</code>	Same as <code>create</code> , but also drops tables when the app stops	Useful for temporary in-memory DBs
<code>validate</code>	Just checks if the schema matches entities, but <b>does not modify</b> 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)

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

```
<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

```
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

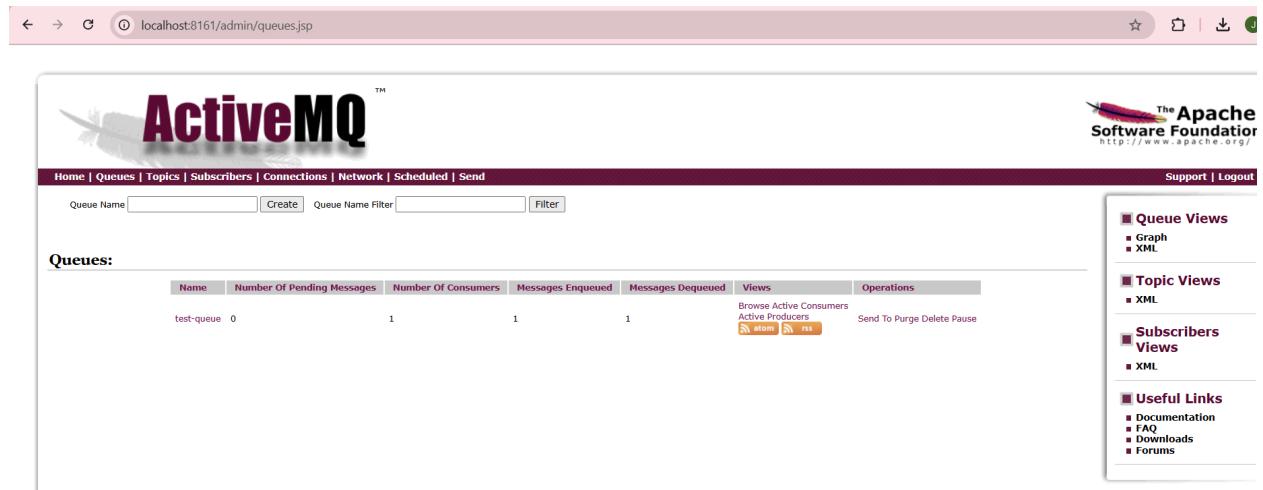
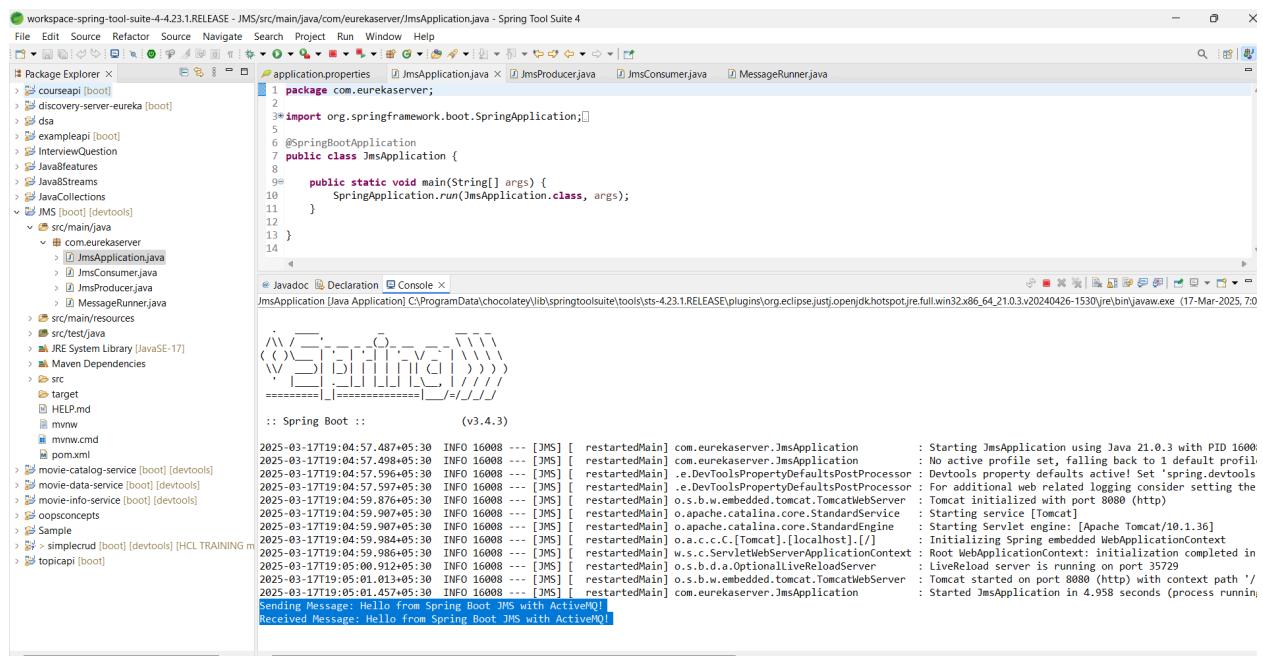
We will use **CommandLineRunner** to trigger the message sending.

```

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

```

```
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?

- ① A **producer** sends messages to an **exchange**.
- ② The **exchange** routes messages to appropriate **queues**.
- ③ 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

### 1 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.

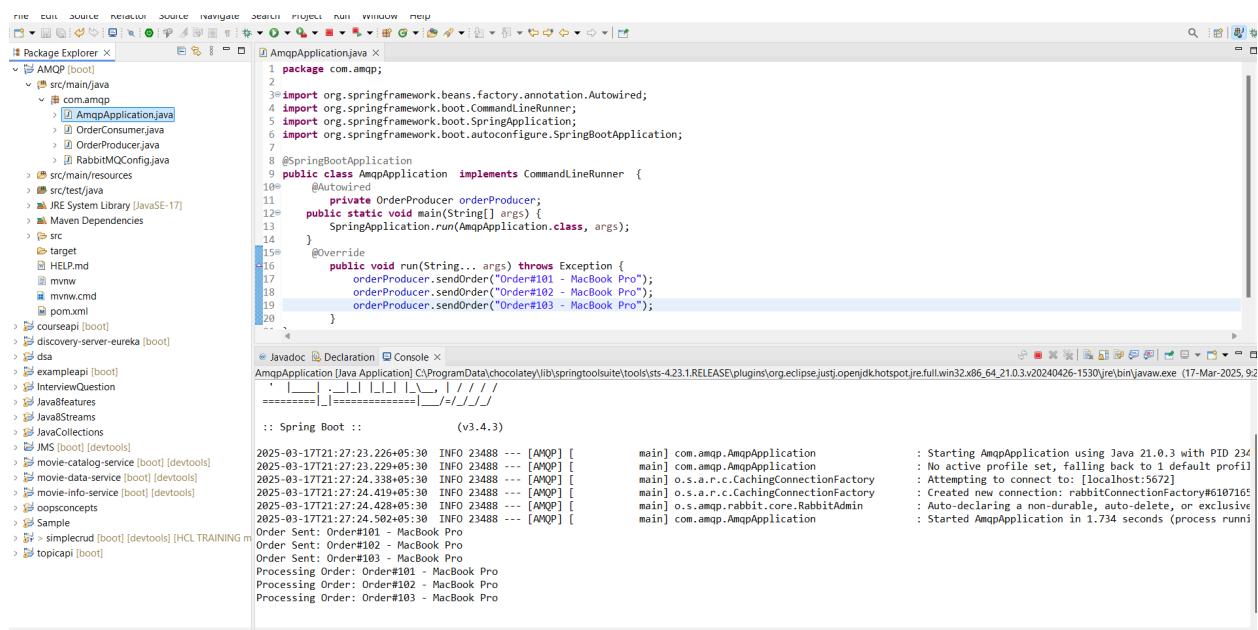
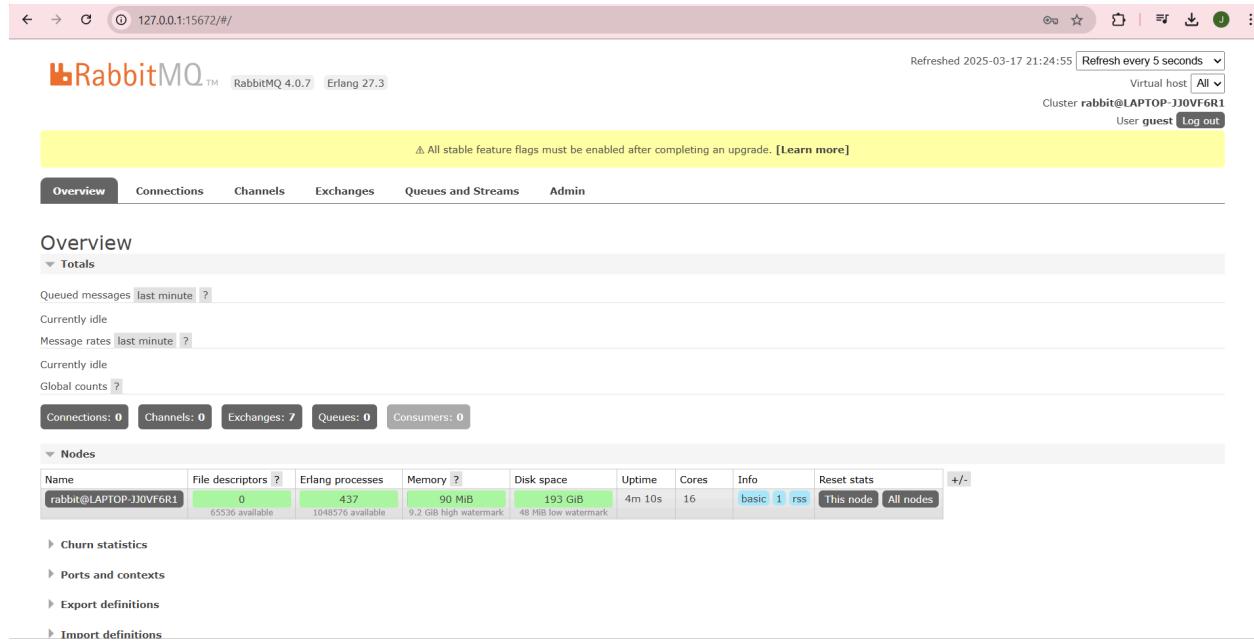
```

Open your browser and go to:

👉 <http://localhost:15672/>

Default login credentials:

- Username: **guest**
- Password: **guest**



All stable feature flags must be enabled after completing an upgrade. [Learn more]  
All deprecated features are being used. [Learn more]

Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	order-queue	classic	Args	idle	0	0	0	0.00/s	0.00/s	0.00/s

## Apache Kafka

What is Kafka?

- Kafka is a distributed event streaming platform.
- Used for real-time processing of large volumes of data.
- 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:

- ① A producer sends messages to a Kafka topic.
- ② Kafka stores messages in partitions inside the topic.
- ③ 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

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

### Step 2: Configure `application.properties`

properties

 Copy  Edit

```
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:**

① The **Order Service** sends a message to RabbitMQ:

Order #1234 placed by User A

- ② The **Payment Service** listens for payment-related messages.  
③ The **Restaurant Service** receives order notifications.  
④ 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:

1 Drivers send their GPS locations every few seconds.

Driver #5678 → Latitude: 12.9716, Longitude: 77.5946

2 Kafka partitions these location updates for **load balancing**.

3 Ride Matching Service reads Kafka topics to find the nearest driver.

4 Pricing Service calculates surge pricing based on real-time demand.

5 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. Example:

### 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();
}
}
```