Lec 35 Multi-Threading Part 6 :

- Daemon Thread → Garbage Collector thread
- Resource r1 = new Resource(); // this will be cleaned up by garbage collector
- r1 = new Resource(); // re-initialized

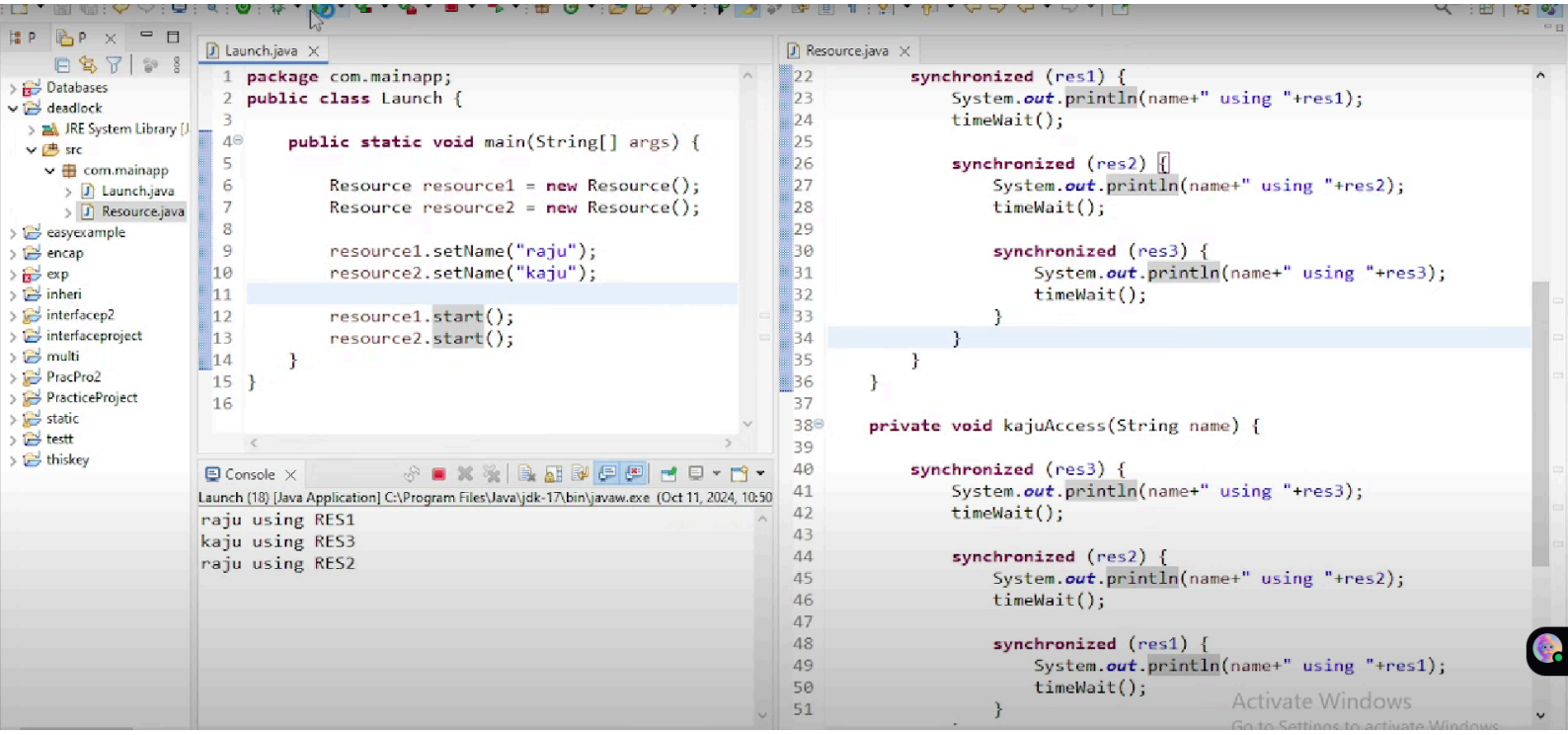
This video explains how synchronization in Java prevents multiple threads from accessing shared resources at the same time, and how improper synchronization can cause deadlocks, where threads are stuck waiting for each other forever. It also covers inter-thread communication using wait, notify, and notifyAll to coordinate actions between threads, especially in producer-consumer scenarios.

Synchronized Methods and Custom Locking

Synchronization in Java can be applied to entire methods (locking the whole object or class) or to specific code blocks (custom locking using synchronized blocks). Object-level and class-level locks control which threads can access particular resources at a time.

Introduction to Deadlock

Deadlock occurs when two or more threads are each waiting for resources held by the other, causing all of them to wait forever. This is common in synchronization and thread communication if resource access is not managed carefully.



Launch.java

```
1  package com.mainapp;
2  public class Launch {
3
4      public static void main(String[] args) {
5
6          Resource resource1 = new Resource();
7          Resource resource2 = new Resource();
8
9          resource1.setName("raju");
10         resource2.setName("kaju");
11
12         resource1.start();
13         resource2.start();
14     }
15 }
16
```

Console

```
Launch (18) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Oct 11, 2024, 10:50
raju using RES1
kaju using RES3
raju using RES2
```

Resource.java

```
22         synchronized (res1) {
23             System.out.println(name+" using "+res1);
24             timeWait();
25
26             synchronized (res2) {
27                 System.out.println(name+" using "+res2);
28                 timeWait();
29
30                 synchronized (res3) {
31                     System.out.println(name+" using "+res3);
32                     timeWait();
33                 }
34             }
35         }
36     }
37
38     private void kajuAccess(String name) {
39
40         synchronized (res3) {
41             System.out.println(name+" using "+res3);
42             timeWait();
43
44             synchronized (res2) {
45                 System.out.println(name+" using "+res2);
46                 timeWait();
47
48                 synchronized (res1) {
49                     System.out.println(name+" using "+res1);
50                     timeWait();
51                 }
```


- Deadlock is occuring because of access to resources are not in order.
- When raju → res1 then kaju is free to use res3.
- When raju moves to get res2 (raju still has lock on res1 till here cause res1 block is not exited) and same as this kaju has lock on res3 and is moving towards res2.
- Here raju gets res2 and it needs res3.
- And kaju has lock on res3 but needs res2.
- This is called Deadlock.

Deadlock Example with Multiple Resources

When multiple threads access several shared resources, if each thread locks one resource and then waits for another locked by a different thread, a deadlock can occur. For example, Thread A holds Resource 1 and waits for Resource 2, while Thread B holds Resource 2 and waits for Resource 1.

Avoiding Deadlock by Resource Order

To avoid deadlock, always acquire locks for resources in a consistent order across all threads. If threads acquire resources in different orders, circular waiting can happen, causing deadlock.

Implementing a Deadlock Scenario in Code

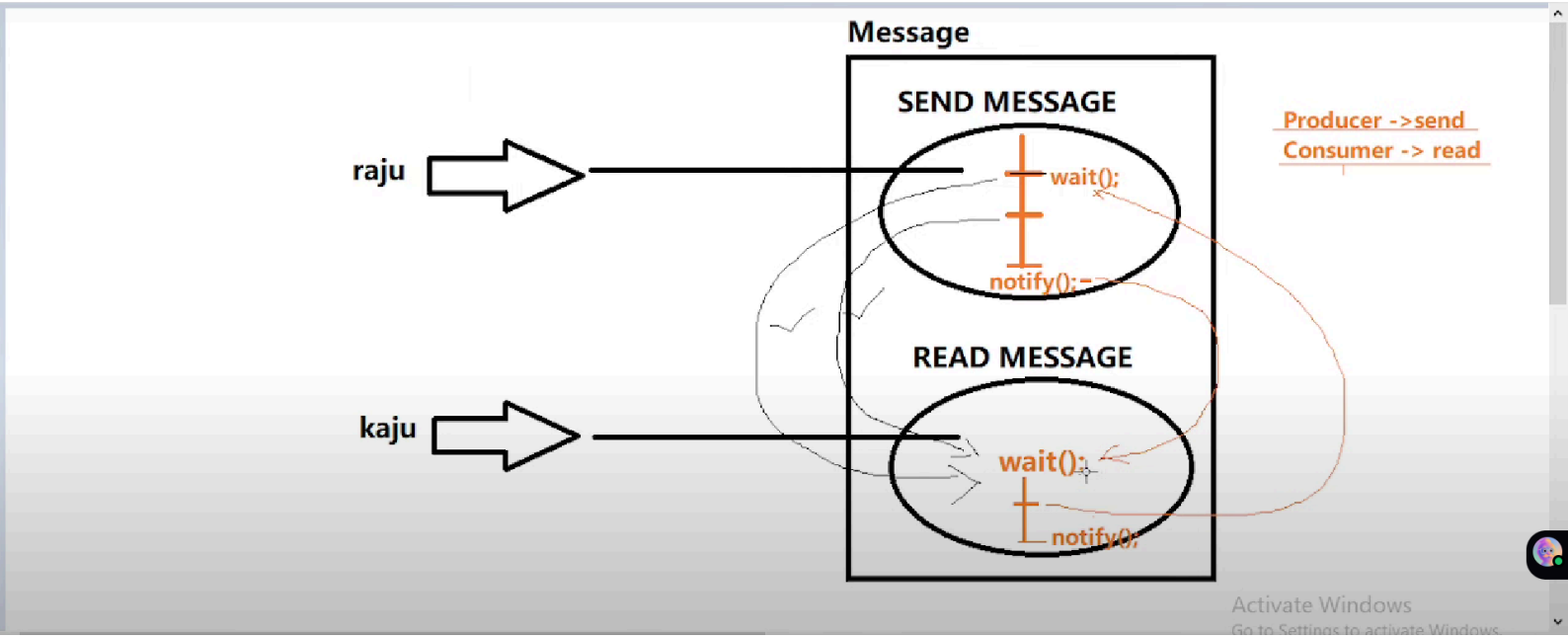
The video demonstrates creating threads and resources in Java, showing how improper locking can lead to deadlock. Nested synchronized blocks are used to show how holding multiple locks at once can cause problems if not managed correctly.

Real-World Applications and Importance of Synchronization

Synchronization and deadlock prevention are crucial in applications like banking, where multiple operations might need exclusive access to resources like accounts. Ensuring only one thread can modify a resource at a time prevents data corruption and fraud.

Inter-Thread Communication Basics

Inter-thread communication allows threads to coordinate their actions using shared objects, typically with wait, notify, and notifyAll methods. This prevents issues like one thread reading data before another has written it.



Producer-Consumer Problem and Wait/Notify

The producer-consumer problem is a classic example where one thread produces data and another consumes it. Proper use of wait and notify ensures the consumer waits for the producer to provide data, and the producer waits until the consumer is ready.

Implementing Wait-Notify with Flags

Flags (boolean variables) are used along with wait and notify to signal the state of data between threads. A while loop checks the flag condition, and threads wait or proceed based on the flag's value, preventing missed or repeated actions.

Correct Usage Order for Notify and Flag Updates

It's important to update the flag (such as setting “message sent” to true) before calling notify, to ensure the waiting thread sees the correct state when it wakes up. Otherwise, threads may wake up and go back to waiting unnecessarily.

Producer-Consumer Problem Solved with Inter-Thread Communication

When wait and notify are used properly, the producer and consumer threads can alternate actions without

missing or duplicating messages, solving the coordination problem and preventing data loss.

Recap and Practical Advice

The session emphasizes practicing these concepts in code to truly understand synchronization, deadlock, and inter-thread communication. Real implementation helps clarify the logic and pitfalls.