Lec 34 Multi-Threading Part 5 :

This lesson explains how race conditions occur in multithreading when multiple threads try to change shared data at the same time, leading to unpredictable results. Synchronization is introduced as a solution to control access, ensuring only one thread can modify critical data at a time, using either object-level or class-level locks for thread safety.

Understanding Race Conditions

A race condition happens when two or more threads access and change shared data simultaneously, causing unexpected or incorrect results. This typically occurs because thread operations overlap, and both threads believe they've completed their tasks, but only one set of changes is correctly applied.

Preventing Race Conditions: Synchronization

Synchronization is a technique that restricts access to critical code so only one thread can execute it at a time, preventing race conditions. This can be done by marking methods or code blocks as synchronized,

putting other threads on hold until the current one finishes.

Thread Safety Explained

Thread safety means that shared data can be safely accessed or changed by multiple threads without causing errors or unexpected behavior. Synchronization is the main way to achieve thread safety, but other methods exist too.

Synchronized Methods vs Synchronized Blocks

Synchronized methods lock entire methods, while synchronized blocks allow you to lock only specific parts of code. This helps control exactly which code is protected, improving performance by limiting the locked section.

Built-in Thread Safety: StringBuffer Example

Some Java classes, like StringBuffer, are already thread-safe because their methods are synchronized internally. Using such classes avoids the need for extra synchronization when multiple threads append data.

Demonstrating Race Condition with Counters

When two threads increment a shared counter without synchronization, the final value can be less than expected due to overlapping operations. This proves that unsynchronized code can lead to lost updates.

Using Synchronized Blocks for Custom Scope

Synchronized blocks allow you to protect only the specific code that needs it, rather than the whole method. By locking on the object (using this), you prevent other threads from entering the critical section until the current thread is done.

Object-Level Locking

When a thread enters a synchronized block locked on an object, no other thread can enter any synchronized block or method of that object until the lock is released. This ensures only one thread accesses the critical code of that object at a time.

Selective Synchronization and Race Condition Detection

By synchronizing only some variables and not others, you can observe where race conditions occur and measure thread overlap. This is useful for debugging and understanding thread behavior.

Real-World Use Cases for Synchronization

In real applications, only critical operations (like updating a bank balance) should be synchronized. Non-critical or independent tasks (like downloading a file) should run without synchronization to improve performance.

Object-Level vs Class-Level Synchronization

Synchronization can be applied at the object level (protecting individual instances) or class level (protecting all instances of a class). Object-level locks allow different objects to be accessed by different threads simultaneously, while class-level locks restrict access across all instances.

Simple Printer Example for Synchronization

A printer resource is shared between two users (threads). Without synchronization, both can print at the same time, causing mixed output. Making the

printer method synchronized ensures one user prints all copies before the next user starts.

Launch.java

```java
package com.mainapp;
public class Launch {

    public static void main(String[] args) {

        Resource resource = new Resource();

        User u1 = new User(resource);
        User u2 = new User(resource);

        u1.setName("USER1");
        u2.setName("USER2");

        u1.start();
        u2.start();
    }
}
```

User.java / Resource.java

```java
package com.mainapp;

public class User extends Thread {

    private Resource resource;

    public User(Resource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        resource.printer(name);
    }
}
```

Console

```
<terminated> Launch (16) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Oct 5, 2024,
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
```

Launch.java

```java
package com.mainapp;
public class Launch {

    public static void main(String[] args) {

        Resource resource = new Resource();

        User u1 = new User(resource);
        User u2 = new User(resource);

        u1.setName("USER1");
        u2.setName("USER2");

        u1.start();
        u2.start();
    }
}
```

User.java / Resource.java

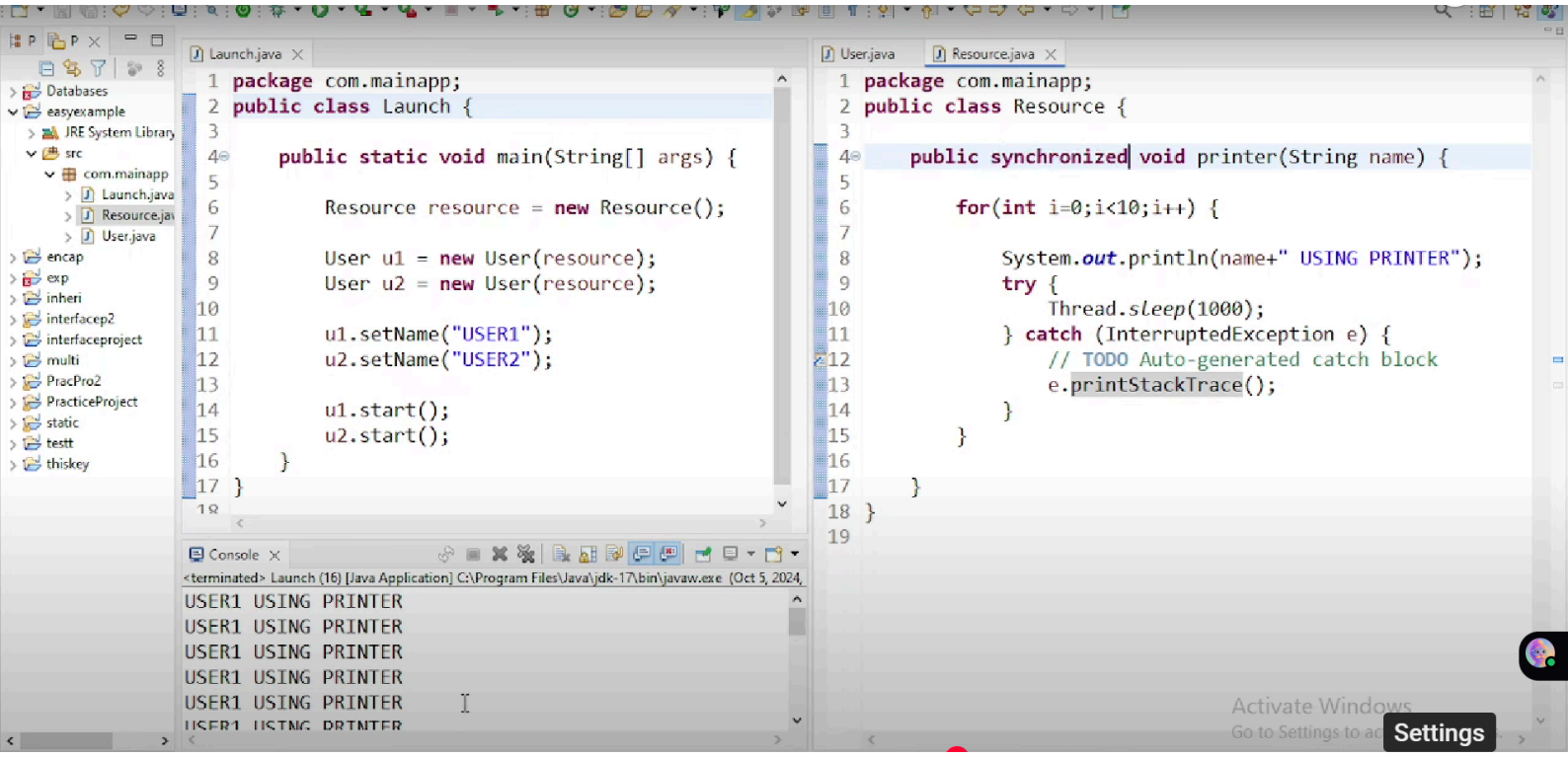
```java
package com.mainapp;
public class Resource {

    public synchronized void printer(String name) {

        for(int i=0;i<10;i++) {

            System.out.println(name+" USING PRINTER");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

    }
}
```

Console

```
<terminated> Launch (16) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Oct 5, 2024,
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
USER1 USING PRINTER
```

Object Locking Demonstrated with Multiple Objects

If two threads use different resource objects, each can execute synchronized methods independently. Locking is specific to each object, so one thread's lock doesn't block access to other objects.

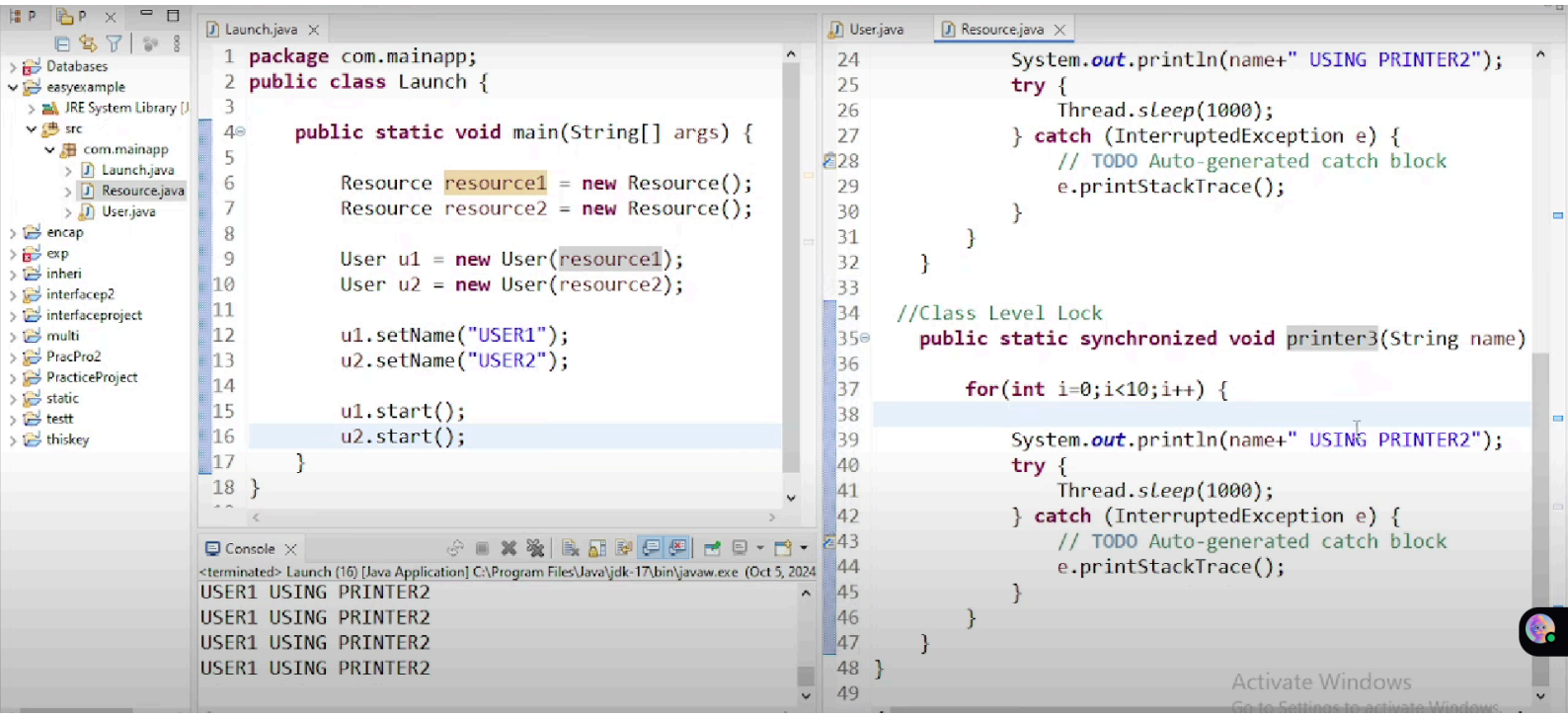


- In above code, we have 2 threads(u1 and u2) which have same resource object.
- When we start the threads, u1 and u2 encounters printer method from Resource class which is synchronized means, whoever calls it will finish it's execution first then will allow others the same. Output will be user1 first then user2 or vice versa.
- Its like putting lock.
- It is an OBJECT LEVEL LOCK.
- Suppose in this same example, if we have separate resources for u1 and u2 and then we call this method then → output will be user1 followed by user2 followed by user1 (means it will be zigzag output).

- Because though the method is synchronized but u1 and u2 threads will run non-synchronized. Context switch between them will occur.
- u1 has lock on r1 / u2 has lock on r2.

Static Methods and Class-Level Locks

Static methods belong to the class, not to instances. Synchronizing a static method or using a synchronized block with the class object (ClassName.class) ensures that only one thread can access the method across all instances, enforcing a class-level lock.



- Now in above code, there are changes. There are 2 separate resources, both u1 and u2 has it's own resources.
- Now we will use the printer3 method for our example.

- This method is static and synchronized.
- Static makes the method common for all instances of this class which is why, when u1 puts lock on this printer3 method then only user1 will use it.
- It doesn't matter if they have separate resource allocated, because the printer3 method is made static which is common for all instances.
- This is called CLASS LEVEL LOCK.

Synchronized Block with Object and Class References

By using a synchronized block with this, you get object-level locking; by using ClassName.class, you get class-level locking. This allows fine control over whether critical code is protected per object or across all instances.

Summary and Practical Advice

Synchronization is crucial for preventing race conditions in multithreaded programs. Use object-level locks for per-instance protection and class-level locks for shared resources across instances. Only synchronize necessary code to avoid performance issues.