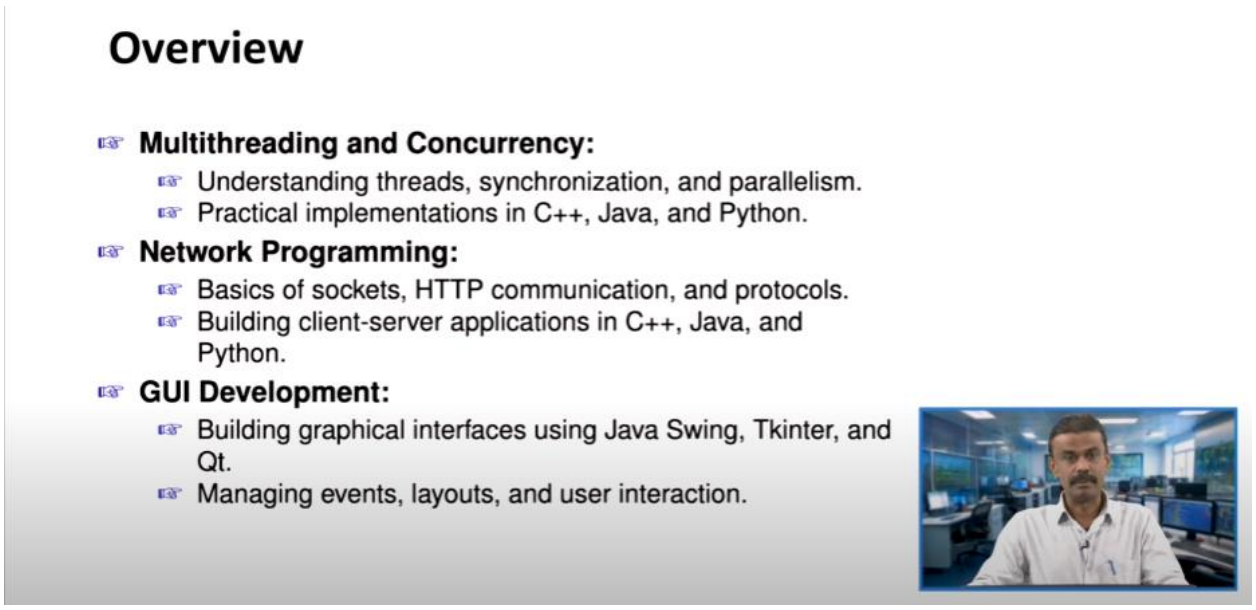
FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture51

Lecture 51: Advanced Topics - Multithreading and Concurrency

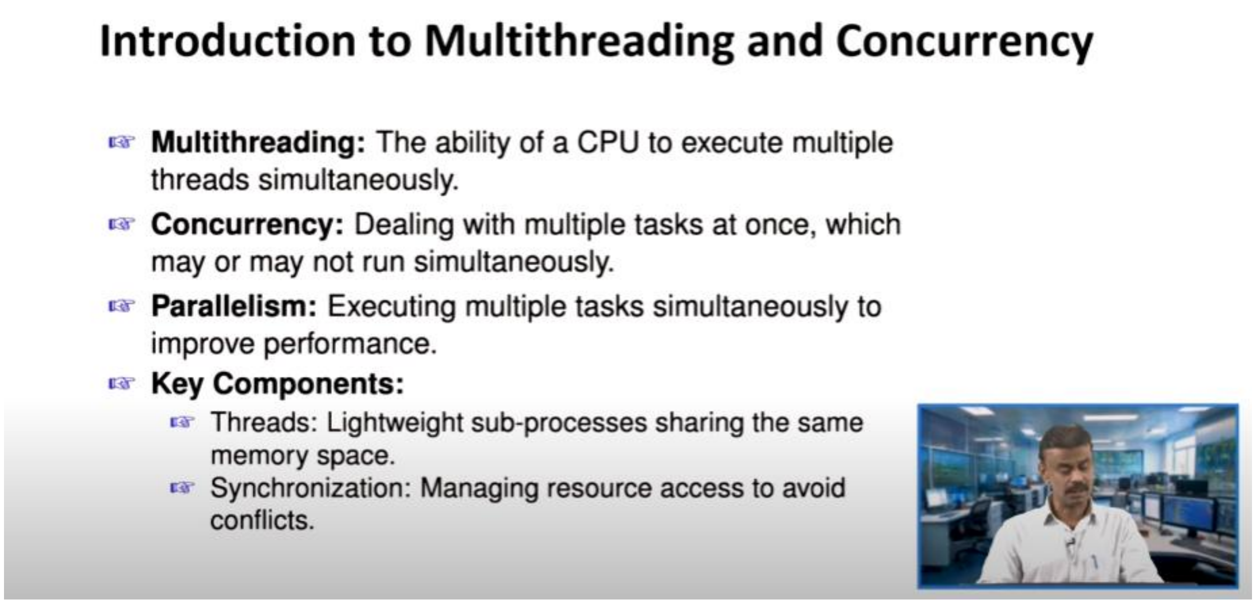


Overview

- **Multithreading and Concurrency:**
  - Understanding threads, synchronization, and parallelism.
  - Practical implementations in C++, Java, and Python.
- **Network Programming:**
  - Basics of sockets, HTTP communication, and protocols.
  - Building client-server applications in C++, Java, and Python.
- **GUI Development:**
  - Building graphical interfaces using Java Swing, Tkinter, and Qt.
  - Managing events, layouts, and user interaction.

So, welcome to lecture number 51. So, in today's lecture, we are going to see advanced topics in object oriented programming. So, under which, so here this is the overview. So, we are going to see the concept of multi-threading and concurrency. So, that means, we are going to understand what do you mean by threads, what do you mean by synchronization and what do you mean by the parallel processing or parallelism, right.

So, we will see the practical implementation in C++, Java and Python. So, once you are familiar with multi-threading and concurrency we will go to network programming right. So, under this we are going to see the basics of sockets, HTTP communication and protocols right. So, then we will talk about the client server applications in C++ right. So, this we will build and we will talk about the client server applications in C++.



Introduction to Multithreading and Concurrency

- **Multithreading:** The ability of a CPU to execute multiple threads simultaneously.
- **Concurrency:** Dealing with multiple tasks at once, which may or may not run simultaneously.
- **Parallelism:** Executing multiple tasks simultaneously to improve performance.
- **Key Components:**
  - Threads: Lightweight sub-processes sharing the same memory space.
  - Synchronization: Managing resource access to avoid conflicts.

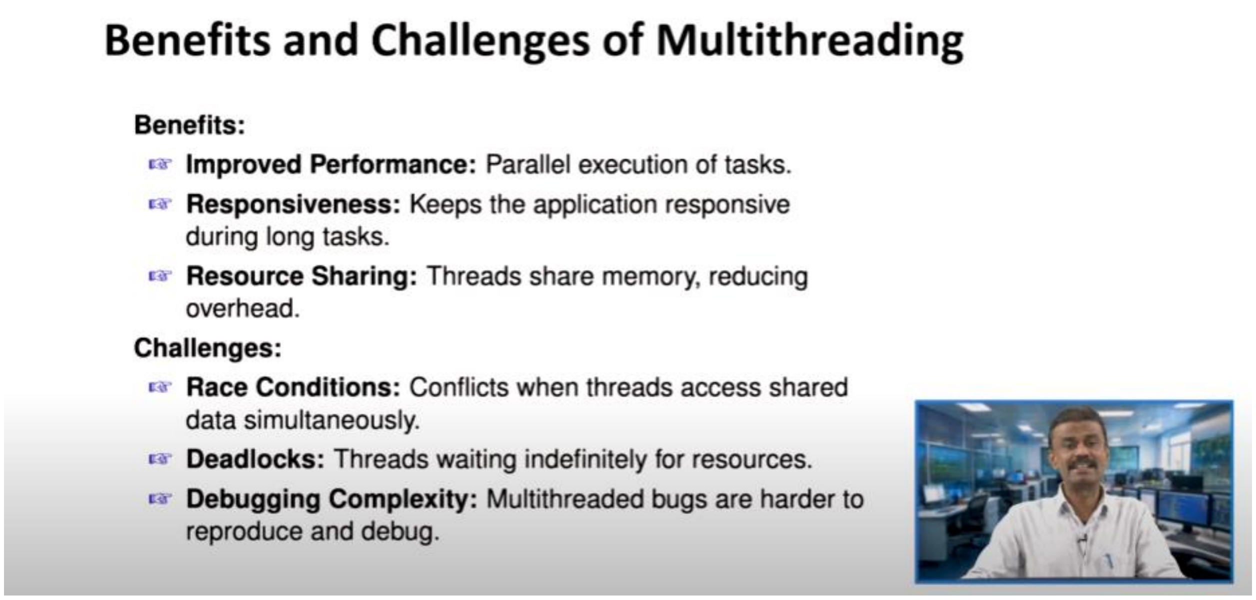
I mean not only C++ we will talk about Java and Python as well. So, then we have to familiar with graphical user interface development right. So, let us have the graphical interfaces using java swing right kinter and qt. So, then once we are familiar with the graphical interfaces we will talk about managing events layouts and user interaction. So, what do you mean by multi threading and concurrency?

right. So, threads we are going to see an example right. So, suppose assume that we have a multiple threads right. So, in that case the ability of a central processing unit to execute the multiple threads simultaneously. Suppose there are more than one thread right exists

So, in that case how CPU is going to execute these multiple threads in a same time or simultaneously. So, this you call it as multithreading right. So, this we are going to study. In fact, we are going to talk about I mean we are going to see the practical applications in C++ and Java right and a concurrency. What do you mean by concurrency?

So, concurrency is nothing, but I mean when you are having a multiple tasks right. So, dealing these multiple tasks at once right. So, which may or may not run simultaneously right. So, this you call it as concurrency right.

So, there is a difference between multithreading and concurrency right. So, the multiple tasks will be there. So, at the same time we have to deal and we may get the result or we may not get the result right. So, that you call it as concurrency parallelism right the parallel programming. So, here in this case assume that again you have multiple tasks



Benefits and Challenges of Multithreading

Benefits:

- **Improved Performance:** Parallel execution of tasks.
- **Responsiveness:** Keeps the application responsive during long tasks.
- **Resource Sharing:** Threads share memory, reducing overhead.

Challenges:

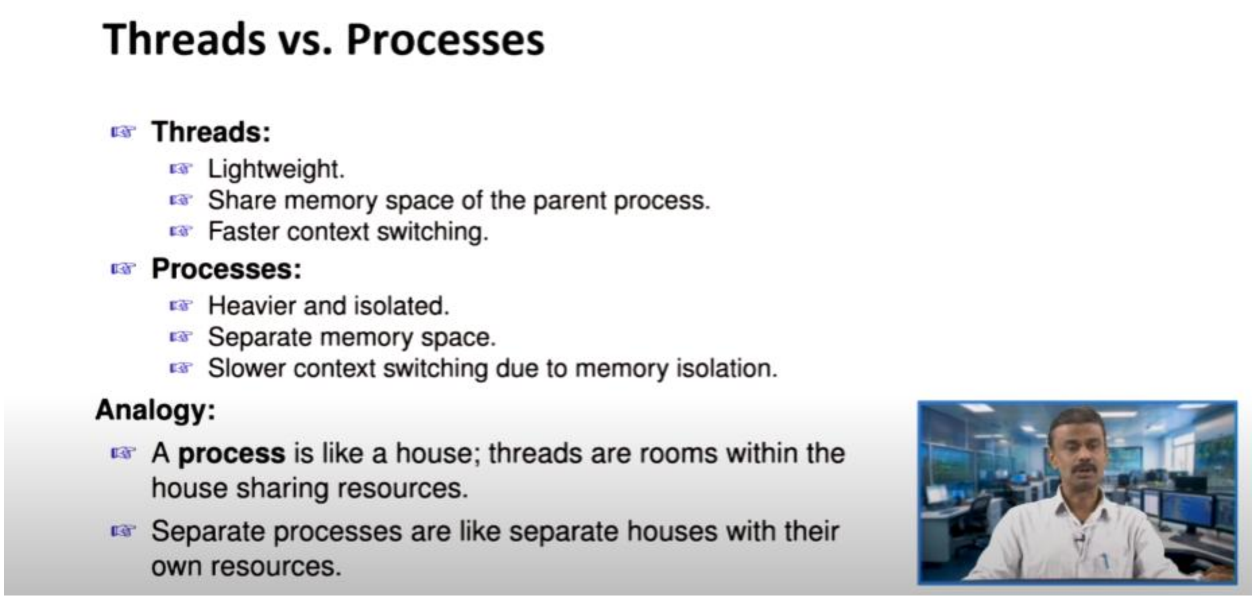
- **Race Conditions:** Conflicts when threads access shared data simultaneously.
- **Deadlocks:** Threads waiting indefinitely for resources.
- **Debugging Complexity:** Multithreaded bugs are harder to reproduce and debug.

So, we have to execute the multiple tasks simultaneously to improve the performance, right. So, parallel programming. So, your task is to do one program, but you can use, right, I mean you can do or you can execute the multiple tasks or several tasks, right. So, simultaneously we are doing this. So, you call this as parallelism.

So, that say for example, some of the programs may run right if you are using a single CPU machine. So, it may run several hours right. So, is it possible to do all right with the several CPUs together and get the output right. So, you call it as a parallel programming or parallelism. So, the key components what we are going to study as I already said threads.

So, threads are nothing but a lightweight sub processes sharing the same memory space. We will see an example in the program right. So, the next one and the next key component is the synchronization. So, here we are going to manage the resource access to avoid the conflicts right. So, these are all the basic introduction that we are going to see and in fact, practically when you learn

right, you know what is the difference between multithreading, concurrency and parallelism. So, when I am talking about multithreading, what are all the benefits and challenges that you will face, right? So, definitely we have the improved performance, right? So, when you are doing the parallel execution of tasks, so my performance will be improved. I mean, in fact, if I do with a single machine, right, I may have, right, or if I do with a single thread,



Threads vs. Processes

- **Threads:**
  - Lightweight.
  - Share memory space of the parent process.
  - Faster context switching.
- **Processes:**
  - Heavier and isolated.
  - Separate memory space.
  - Slower context switching due to memory isolation.

Analogy:

- A **process** is like a house; threads are rooms within the house sharing resources.
- Separate processes are like separate houses with their own resources.

I mean, it takes a lot of time, right, to complete the program, all right, or to get the output, all right. So, here when I do the parallel execution of task, so I will get the improved performance. And next point is the responsiveness, all right. So, this multi-threading, so which keeps the application responsive during the longer larger task, all right. And another important benefit is the resource sharing, all right.

So, threads which share memory, and reducing the overhead, right. So, these are all the main benefits of multithreading. And what are all the challenges you face? So, we have the important technical term called the race condition, right.

So, the race condition is nothing but conflicts when threads access shared data, right. So, assume that there are two threads accessing the shared data at the same time simultaneously, right. So, in that process, there will be a conflict. So, the conflicts occur when threads access shared data simultaneously. So, this is called a race condition.

And the second one is deadlock. So, assume that two trains are on the same track, opposite sides. So, you have to have some solution to get rid of this. So, same thing. So here, instead of trains, I'm talking in terms of threads.

So threats waiting indefinitely for resources. So this you call it as deadlock. And third challenge you will face, debugging complexities. So, here you have multi-threaded bugs, right. So, which are very harder to reproduce and debug, right.

So, these are all the main challenges we face in the case of multi-threading. At the same time, so we have a improved performance, responsiveness and resource sharing. So, which are all the main benefits of multi-threading. So, this is a very famous interview question, threads versus processes. So, how do you define a thread?

How do you define a processes? So, thread already we had seen it is a lightweight. So, also it shares the memory space of the parent process. So, analogy we will take we will take we will give a house example. And threads which are faster context switching.

So, when you have these three points, you call these the threads. What about the processor? So, a thread, I said, is lightweight. So, here, processes are heavier and isolated, right? So, this has separate memory space, and threads, as we have seen, allow faster context switching, right?

So, whereas it is the opposite—due to memory isolation, these processes are slow, right? Or you can call it slower context switching, right? So, we will take a practical example. The process, which is nothing but a house, right? So, threads are nothing but the rooms in the house, right? So, when I am talking about the rooms in the house, they will share the resources, right?

So, a process is like a house, and the threads are the rooms inside that house, right? So, which are sharing the resources. So, when I am talking about separate processes, right? So, assume that you have N processes, right? Right, so it is nothing but like you have N separate houses, and each house has its own resources.

so this is how we can have the difference so process means it is like a house right you can think it is like a house and inside the house of course you have rooms right the rooms are nothing but the threads so every threads are having the owned resources for example if I take kitchen so the kitchenery item will be there if I have a bedroom The cot, pillow, everything will be there. So, that means individually it shares the resources. So, similarly the threads, alright.

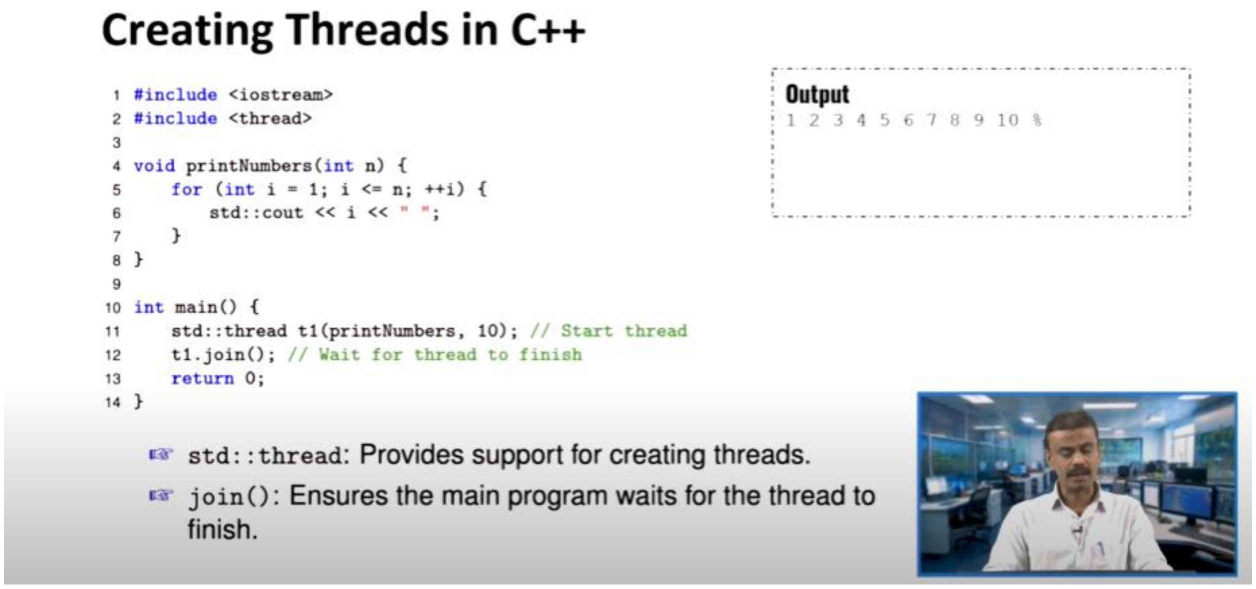
So, threads they are nothing but the rooms. So, they I mean like house we have the sharing resources. So, here you will have the sharing resources. So this is a major difference between the threads and processes and this is a very important interview course in what is the difference between threads and processes. So now what is the life cycle of a thread?

So thread states. So we start with new. So thread is created but not started. We are going to see some of the inbuilt functions. So, when you want to run the program we are going to see some of the inbuilt functions.

So, the state new which is nothing, but the thread is being created, but it is not being started. The next state is a runnable state right. So, thread is ready to run, but waiting for CPU scheduling all right. So, I mean you can say that it is ready right for running. But it will wait for CPU scheduling.

Now next state is running. So now thread is actively executing. So this state the running state meaning is thread is actively executing. The fourth state is blocked or waiting state. So here

The thread is waiting for the resource or resources right. So thread is waiting for a resource and then the terminated state that means thread has completed the execution. So this is the lifestyle of a thread right. So start with new then runnable right and running so that means it is actively executing. So the fourth state you can call it as a blocked or waiting state that means thread is waiting for the resource.



Creating Threads in C++

```cpp
#include <iostream>
#include <thread>

void printNumbers(int n) {
    for (int i = 1; i <= n; ++i) {
        std::cout << i << " ";
    }
}

int main() {
    std::thread t1(printNumbers, 10); // Start thread
    t1.join(); // Wait for thread to finish
    return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10 %
```

- std::thread: Provides support for creating threads.
- join(): Ensures the main program waits for the thread to finish.

And the last one is terminated, which means the thread has completed the execution. So now, let us consider how you are creating threads in C++, right? So what I will do here is write this program, and I am asking you to execute it because we have done several executions, right? Maybe in the case of Java, I will do that. So, in the case of C++, because it is not very difficult, we are already including all the header files.

Right. So, we will see this program and how you are going to create the threads in C++. Right. So, let us have the function: void print_numbers(int n). The usual one: for (i = 1; i <= n; i++) { cout << i; }. Right. So, now we can create the thread.

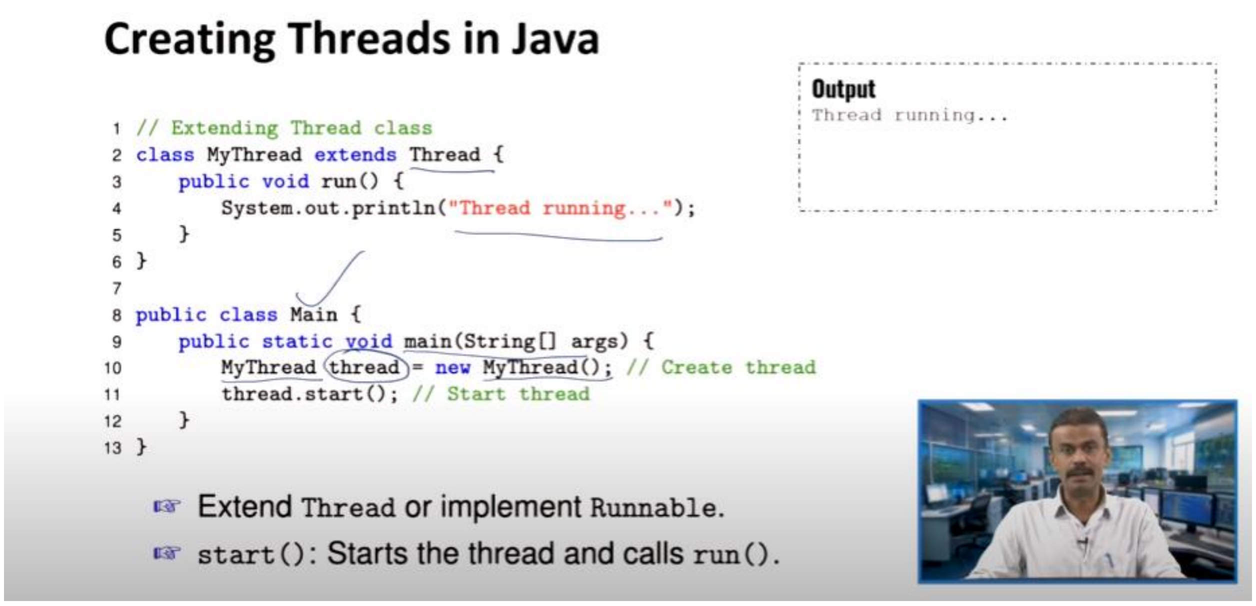
Right. So, for creating a thread. So, here you have. Right. I have included thread.

Right. The header file has include thread. Line number 2. So, now let us go to the main program. So, in the main program, we have thread t1.

So, t1 is a thread. And then you are passing 2 parameters. It is a 2-argument thread. So, what are the arguments? Print numbers.

You are passing print numbers and 10. What is print numbers? Print numbers is nothing but a function. And 10, n will be assigned to 10. n will be assigned to 10. That is the meaning.

So that means when I am comparing with the states, so I can say it is runnable and running, right? So it is start, right? So thread has been started. That is the meaning, right? So ready to run, but waiting for CPU scheduling.



Creating Threads in Java

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

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread(); // Create thread
        thread.start(); // Start thread
    }
}
```

Output

```
Thread running...
```

- Extend Thread or implement Runnable.
- start(): Starts the thread and calls run().

And the next stage is thread is actively executing, right? So the next line, the t1 is invoking join, right? So join, which is already available in the header file thread. So that means it is waiting for thread to finish. So here the join function, the meaning of the join function, wait for thread to finish.

So this is what exactly the block slash waiting. Thread is waiting for a resource. So wait for thread to finish. So when I run this code, so return 0. When I run this code, right, so what will happen?

The thread will start, right, and will be executed. And it is also waiting for thread to finish. That means you have to give t1.join, all right. So, now I am asking you. Right.

You can put a common statement on line number 12. Right. And see what will happen. Right. So you will get the runtime error.

Right. You will get the runtime error. So put t1.join the common statement. Right. So once you remove the common statement, that means the wait for thread to finish.

Suppose I do not have line number 12. Right. So it is runnable and running. Right. So but what about the wait?

So thread should wait for a resource. right and then it has to be terminated. So, waiting and terminated we have to have line number 12 right. So, what I will give a small exercise remove line number 12 and see right. So, if I put line number 12 right wait and terminated right return 0 you are putting.

So, you will get the output all right. So, this is your output. So, now you can see I use line number 11 thread right. So, this is providing. So, I have created t1.

Right. So, this provides support for creating the thread. So, in fact, I said. Right. The line number 11.

So, you have over here runnable and run. Right. You have created runnable and running. So, all these three states. Right.

And then next one join. So, this ensures the main program waits for the thread. To finish. Right. So this is how the basic example in C++.

Right. For the creation of threads. Right. So first, what you do. You try this program completely.

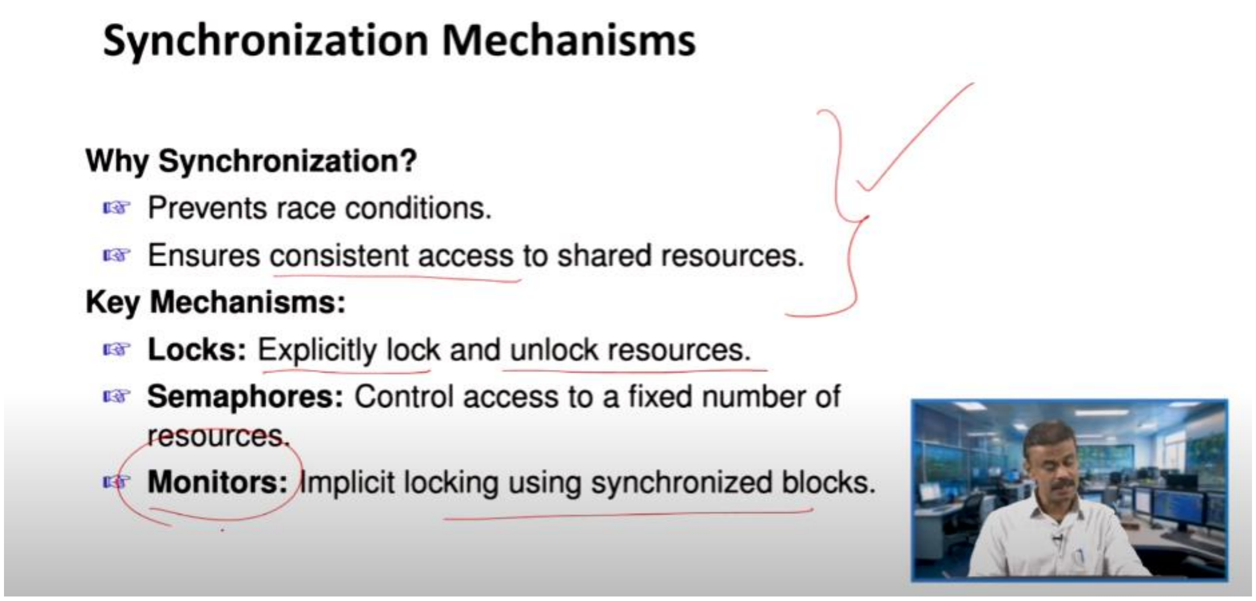
And try to comment on line number 12. All right. So, you will understand. So, certain states will not be executed if t1 does not call this join function. Right.

So, the join function is for waiting for the thread to finish. So, maybe we will see it when I run Java. All right. So, here I have the MyThread class, MyThread, which is extending Thread. Right.

Inbuilt. So, here you have one run method: System.out.println. Thread running. All right, so like what we had seen in the previous program, so similar, all right. So here we have the main, that means a driver class: public class Main, which is a driver class, and here you go, the main method, all right. So here I am creating the thread, right? The object thread under MyThread class, MyThread, right? So you are allocating memory with a constructor, MyThread.

So, this is the syntax for creating the thread, right. So, this is how exactly we have done in line number 11, equivalent, right. So, slightly changing in C++, it is expected. So, here it is just like you are creating an object under the class. On the right-hand side, you have the memory allocation and the constructor. Right.

So, you have created the thread, and the next line: thread.start, right. So, we know the same meaning as what we have done in the case of right here, right. So, this will start. So, here in this case, this will start. So, that means it will start executing, right.



Synchronization Mechanisms

Why Synchronization?

- Prevents race conditions.
- Ensures consistent access to shared resources.

Key Mechanisms:

- **Locks:** Explicitly lock and unlock resources.
- **Semaphores:** Control access to a fixed number of resources.
- **Monitors:** Implicit locking using synchronized blocks.

See for example, here we have written right extend thread or implement runnable right. So, here you have extend thread or implement runnable right so this is what we have talked about the states right so runnable in the sense thread is ready to run but waiting for cpu scheduling right and the start line number 11 start so it starts the thread and calls run right so once the object thread is invoking start

the meaning is the function run will be executed right so that means it is calling run function run method and we know when the run method is being executed so then we will get the output thread running right so this is what exactly happening so that means line number 10 you are creating a thread and when I have that object thread is invoking start so that means It is starting the thread and it is calling the run method, right? So, this is what exactly happening.

So, we know once it is calling run method. So, run method you have system.out.println statement which is printing thread running, right? So, that is what we are getting the output. Maybe what we can do in Java, we can try this. C++ I gave a exercise in Java we will try this.

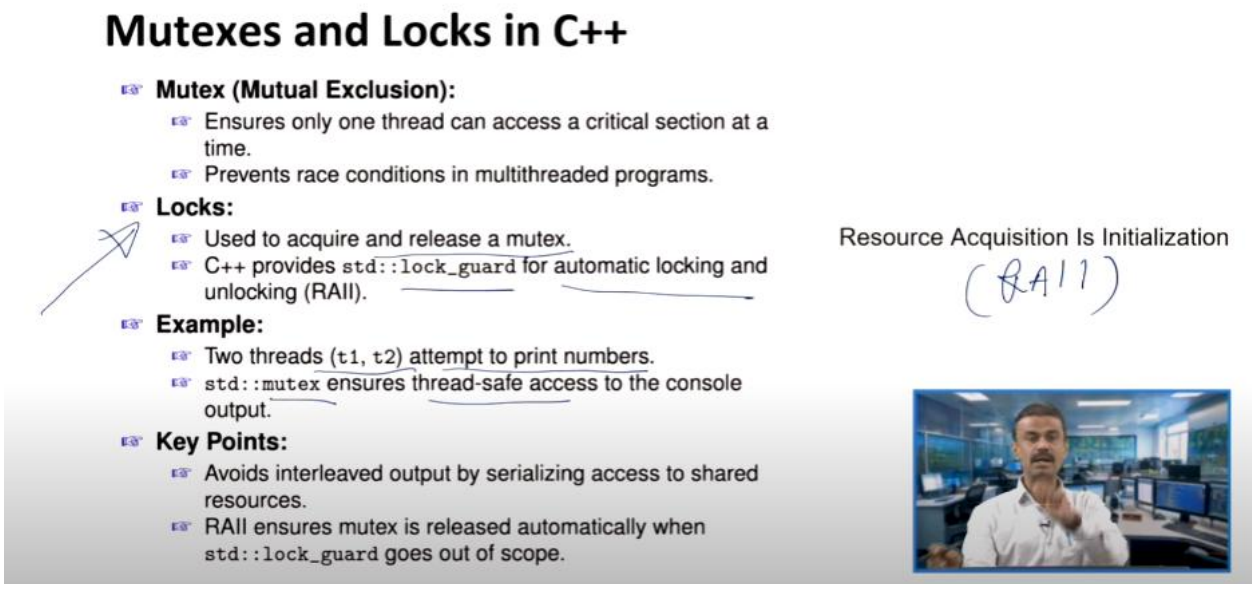
I will go to the Java here ok the same program. So, now we will run the code. So, if we run the code Java MyThread. So, you can see the output right thread running followed by three dots. So, the next concept synchronization mechanism right what do you mean by synchronization and why synchronization is required right.

So, synchronization I we talked about the race conditions. So, this will prevent the race condition right race condition or race conditions right. Also the synchronization which is ensuring consistent access to the shared resources right. So, these two reasons these two main reasons we require synchronization right. The key mechanisms are locks, semaphores and monitors all right.

So the locks which are nothing but so you are having a explicitly locked and unlocked resources right. So lock mechanism the key mechanisms of synchronization is nothing but locks. So the locks are explicitly locked and unlocked resources and the next one is semaphores right. So the semaphores Control access to a fixed number of resources, all right.

So, you can have the control access to a fixed number of resources. So, you call it as a semaphores, all right. And the third one is monitors, right. The third key mechanism is the monitor. So, we talked about the explicit lock, all right.

Explicit lock and explicit unlock resources. So, here monitors will implicit, when it is having the implicit locking. Now, implicit locking using synchronized blocks. So, you call it as monitors all right. So, these are all the three key mechanisms locks, semaphores and monitors under the synchronization.



Mutexes and Locks in C++

- **Mutex (Mutual Exclusion):**
  - Ensures only one thread can access a critical section at a time.
  - Prevents race conditions in multithreaded programs.
- **Locks:**
  - Used to acquire and release a mutex.
  - C++ provides `std::lock_guard` for automatic locking and unlocking (RAII).
- **Example:**
  - Two threads (`t1`, `t2`) attempt to print numbers.
  - `std::mutex` ensures thread-safe access to the console output.
- **Key Points:**
  - Avoids interleaved output by serializing access to shared resources.
  - RAII ensures mutex is released automatically when `std::lock_guard` goes out of scope.

Resource Acquisition Is Initialization

(RAII)

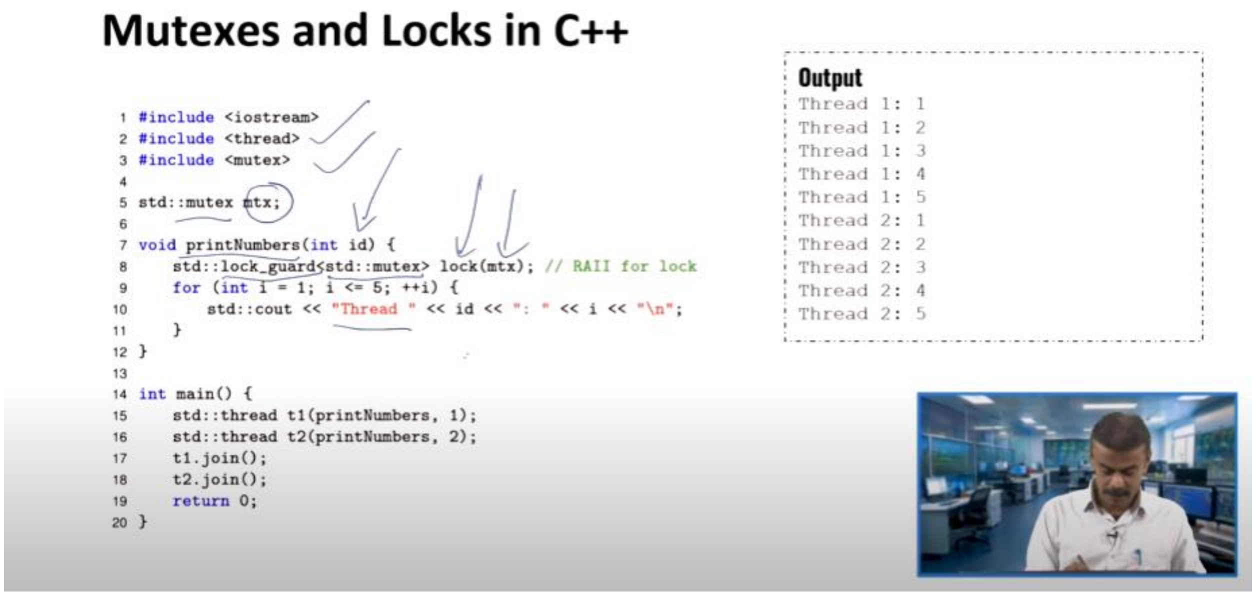
So, these are all helpful to prevent the race condition and it is also ensuring that you can have the consistent access to shared resources. So, the next concept we are going to see mutual exclusions and locks all right. You call it as a mutex. Mutex is nothing but mutual exclusion. So, we are going to study about mutual exclusion and locks in C++.

So, what do you mean by mutex, right? I mean, as we have already seen. So, this mutex, or mutual exclusion, prevents race conditions in multi-threaded programs, right? Also, this ensures only one thread can access a critical section. at a time, right?

So, you call this mutual exclusion or simply mutex. So, the next concept is locks, right? The locks are useful to acquire and release a mutex—acquire and release a mutex. So, in fact, we are going to see with the help of lock_guard, right? For automatic locking and unlocking, it is like RAII. Right. So, what do you mean by RAII?

It is nothing but resource acquisition is initialization. So, you call this RAII, right? So, it is equivalent to that. So, the lock_guard. So, that is for automatic locking and unlocking.

So, let us consider right with an example. Assume that your two threads in fact this example we are going to see in C++. So, assume that so the two threads t1 and t2 they are attempting to print the numbers right. So, now with the help of mutex right. So, in fact the mutex will be ensuring that thread safe access right to the console output that means it will print one by one.



Mutexes and Locks in C++

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;

void printNumbers(int id) {
    std::lock_guard<std::mutex> lock(mtx); // RAII for lock
    for (int i = 1; i <= 5; ++i) {
        std::cout << "Thread " << id << ": " << i << "\n";
    }
}

int main() {
    std::thread t1(printNumbers, 1);
    std::thread t2(printNumbers, 2);
    t1.join();
    t2.join();
    return 0;
}
```

Output

```
Thread 1: 1
Thread 1: 2
Thread 1: 3
Thread 1: 4
Thread 1: 5
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
Thread 2: 5
```

So, either T1 assume that it is printing let us say 10 numbers. So, t1 10 numbers will be printed then the thread 2s 10 numbers will be printed or vice versa. t2 will print the numbers and t1 will print the number. So, that means the mutex is ensuring that right. So, key points logs avoid interleaved output right by serializing access to shared resources right.

Also the resource acquisition is serialization RIII right. So, which ensures mutex is released automatically when you have the class right the lock underscore guard this we are going to see right the lock underscore guard goes out of scope right. So, these are all about your mutexes and locks and in fact practically how we can implement. So, we will see in this particular program all right.

So, here we are including thread and mutex right. So, here we are including thread and mutex. I have the object mtx under mutex class it is available in your hash include mutex. I have the usual function let us say print numbers. So, this is what I talked about this in the theory right.

The print numbers your argument is id whose data type is integer. So, now here right RAII for lock right. So, I here I have the lock got class. So, this is what we talked about. So, lock underscore got

of mutex right and here i have the lock object with mutex call it as mtx that object right the lock so when i am creating this object so automatically you are passing mtx that means the corresponding constructor will be called right so this is for lock line number 8 statement is for lock right and then here you have for int i is equal to 1 i less than or equal to 5 plus plus i We are trying to print thread, right? This will be printed and id colon and i is running from 1 to 5, right?

So, this we are expecting. So, now we will go to the main program. Like the first program that we had seen, here I have two threads. Line number 15, I have t1, thread t1, where the parameters are print numbers. Print numbers is nothing but the function.

And then you are passing 1. That means the ID is 1 now, right? ID is 1 now. So, T1 is the thread, class thread, right? So, like you know, when you are starting this, right?

So, we know that the thread has been started, right? So, this is what we studied. The first program, if you recall, we have started, we did only one, right? Thread t1, right? print numbers and 10 we had given if you recall and then t1.join.

So, now what I will do is I have another thread, right? So, we are talking about the multi-thread, another thread t2, right? So, here you have print numbers now, ID is 2, right? So, t2 ID is 2. So, you have two threads, t1 and t2, running simultaneously, right? They have been started simultaneously.

Now we know the meaning of t1.join and t2.join, right? The last three states—five states we have studied—the first two are right, based on the thread t1 when you are creating or starting, right? So now, t1.join and t2.join, so it will go up to the termination, and we know that, right? We are going to get, let us say, either thread 1 first or thread 2 first, right? So, it depends on the compiler. You have to run it in various compilers, right? So here, in this case, what I have got is first I have got thread 1.

Right. So, the meaning is one by one it will happen. So first, thread one, and that is nothing but the ID. So, this ID will be printed. Then the I values, I is equal to one to five, will be printed.

Right. So next that is what we had told over here. Right. So two threads t1 and t2 attempt to print the numbers. Right.

So that is what the meaning here. Line number 15 and 16. So two threads t1 and t2 attempts to print numbers. So now this class mutex. Right.

So which is ensuring. Right. So here you go. Right. So this mutex.

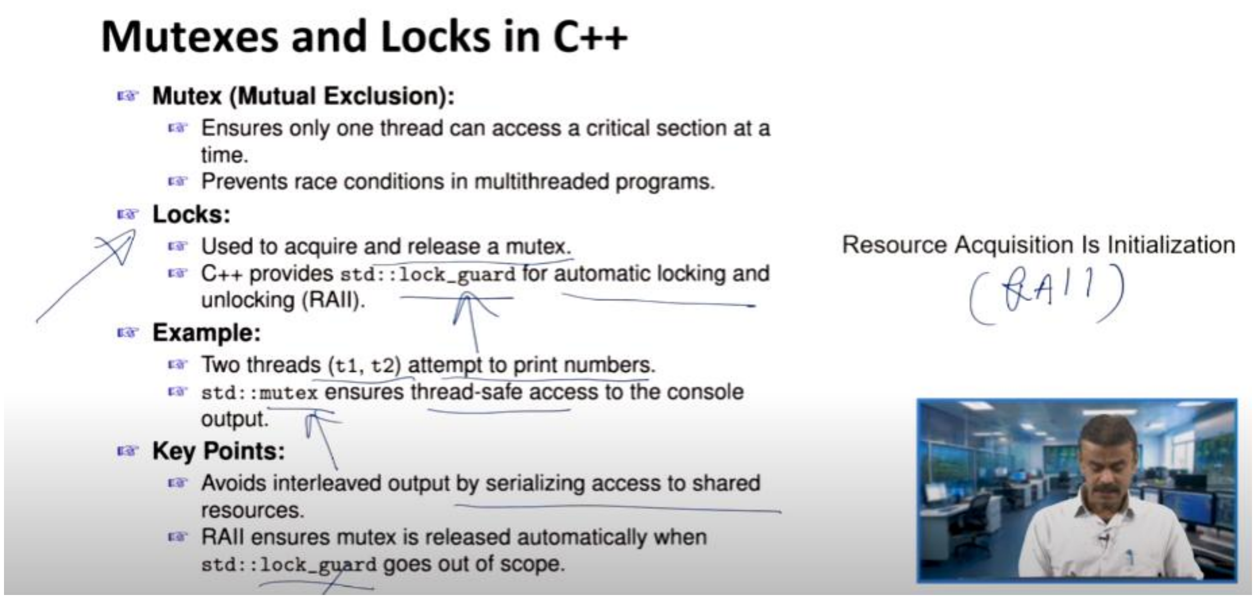
So which is ensuring. Right. So the thread safe access to the console output. So here I have got thread 1 first. So there is always possibility that thread 2 may come first.

Yes it is possible. So one of them will be printed because of mutex. So, you may get some of you may get right. So, you run in various compilers right. Some of you may get thread 2 first.

So, thread 2, 1 2 3 4 5 first and then thread 1, 1 2 3 4 5 next possible right. So, the same meaning the t1.join and t2.join we had seen only one case we had seen in the first program. So, that means the rest states right that runnable up to termination right. So, that will be done and then you will get the output right.

So, here idea is you have through two threads t1 and t2 both are trying to print right these output at the same time right. So, but you want the output one by one right. So, for that what will you do you have line number 8, line number 8 you have mutex. So, this mutex right STD scope resolution operator mutex. Exactly we are using over here.

So this is useful right for the thread safe access and then it is printing the number 1 by 1 right. 1 by 1 in the sense thread by thread. First thread t1 and then t2 we have got right. in some of the compilers t2 first and then t1 next right. So, this is also possible.



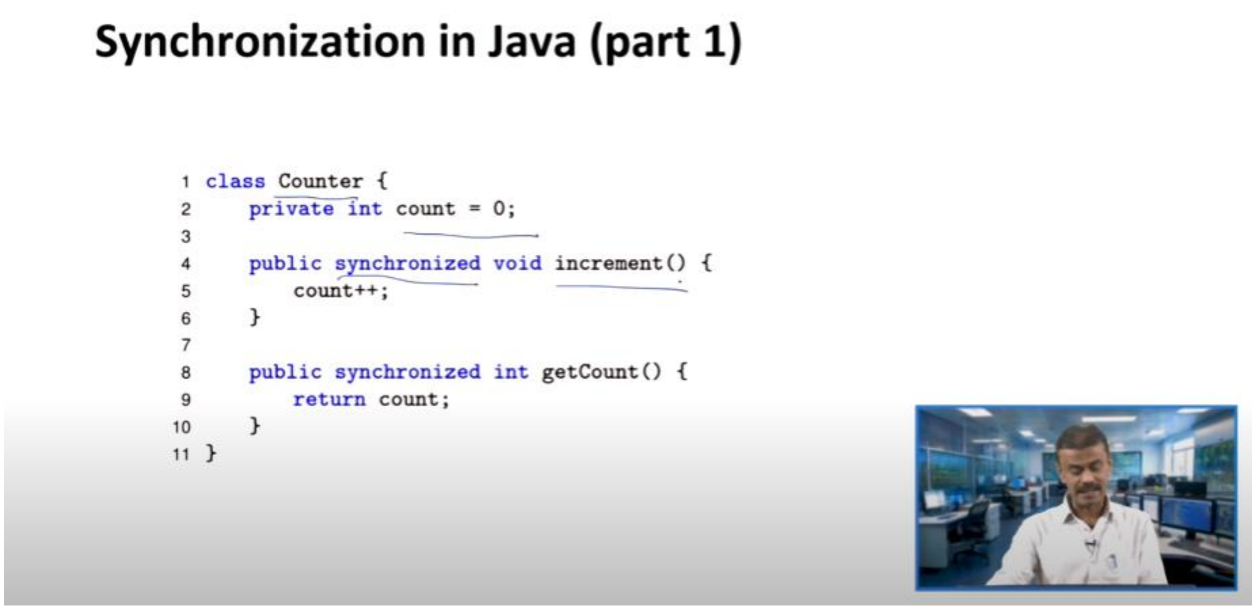
Mutexes and Locks in C++

- Mutex (Mutual Exclusion):
  - Ensures only one thread can access a critical section at a time.
  - Prevents race conditions in multithreaded programs.
- Locks:
  - Used to acquire and release a mutex.
  - C++ provides `std::lock_guard` for automatic locking and unlocking (RAII).
- Example:
  - Two threads (`t1`, `t2`) attempt to print numbers.
  - `std::mutex` ensures thread-safe access to the console output.
- Key Points:
  - Avoids interleaved output by serializing access to shared resources.
  - RAII ensures mutex is released automatically when `std::lock_guard` goes out of scope.

Resource Acquisition Is Initialization

(RAII)

So, the next example we will see synchronization in Java right. So, let us consider class counter right let us consider class counter. So, under class counter you have the private member data count 0 right. So, count equal to 0 private member data we consider count and then you have a public first time we are using synchronized void increment So, the reason here is suppose there are two threads.



Synchronization in Java (part 1)

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}
```

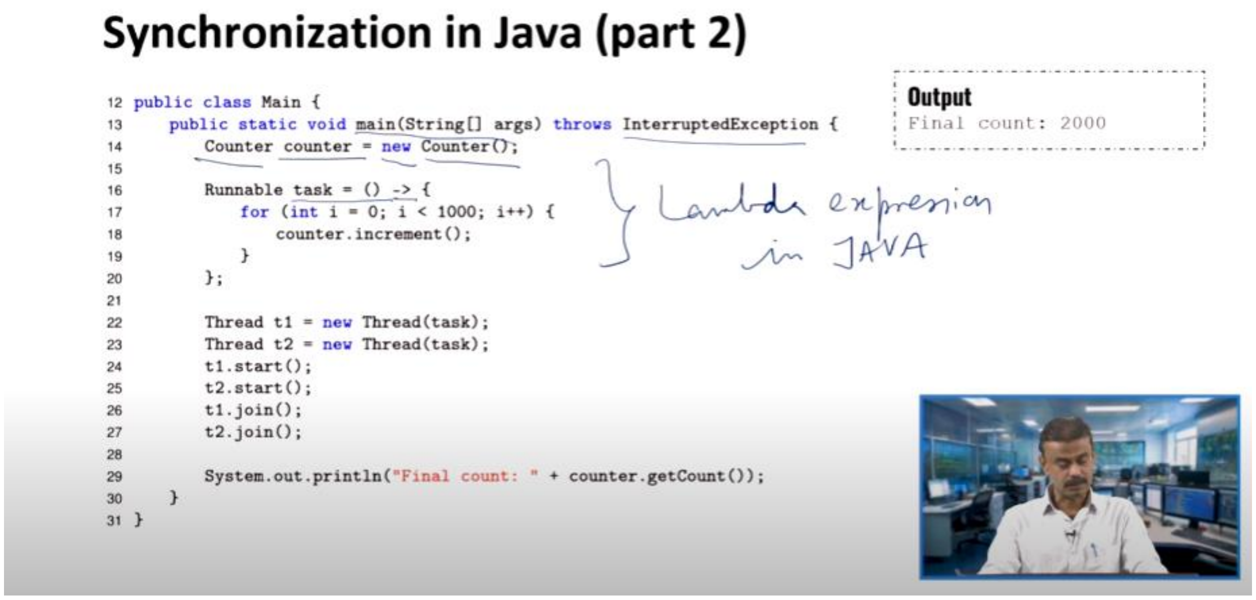
So, we have to synchronize, and then we have to do the increment. So, let us say one thread is running for i equal to 1 to 10. And then you are doing the increment. The second thread, i is equal to 1 to 10. You are doing the increment.

So, in total, you should get 20. So, we use this with the help of the keyword called synchronize. Increment is doing count plus plus. And again synchronize, because two threads will produce a result. So, the result we are printing is return count.

So, for this, we require a getCount function, right? The getCount method. So, this is your class counter. So, now we will go to the driver class. The driver class is the main. You have the main program, the main method over here.

So, for the safer side, it throws an interrupted exception. So, if any exception occurs, So, it will throw the exception, right? So, now you have a counter object; the class is Counter, and then the memory will be allocated with the constructor Counter, OK? So, when I am creating this,

So, the Counter constructor will be called. So, we do not have any default. So, we will proceed further. So, here, line number 16, this is nothing but the lambda expression in Java. All right, lambda expression in Java, all right.



Synchronization in Java (part 2)

```
12 public class Main {
13     public static void main(String[] args) throws InterruptedException {
14         Counter counter = new Counter();
15 
16         Runnable task = () -> {
17             for (int i = 0; i < 1000; i++) {
18                 counter.increment();
19             }
20         };
21 
22         Thread t1 = new Thread(task);
23         Thread t2 = new Thread(task);
24         t1.start();
25         t2.start();
26         t1.join();
27         t2.join();
28 
29         System.out.println("Final count: " + counter.getCount());
30     }
31 }
```

Output

```
Final count: 2000
```

Lambda expression in JAVA

So, the Counter constructor will be called. So, we do not have any default. So, we will proceed further. So, here, line number 16, this is nothing but the lambda expression in Java. All right, lambda expression in Java, all right.

So, here, this is a Runnable; the name is Runnable, the task. So, these are all the syntax, right, followed by an arrow, right. And then here, you have for int i is equal to 0, i less than 1000, i plus plus, right. So, the lambda expression is useful when you assume that you have two threads. In fact, we are going to have two threads.

So, let us assume it is running. So, we know in the previous program in the C++. So, we have got the output separately right. One is the thread 1 and another one is thread 2. So, here in this case thread 1.

So, which will write the calling this counter increment every time for i is equal to 0 to 999 that means 1000 times correct and then another thread T2 right. So, we have to do the synchronization. Correct. So, that is why we have used the keywords in connection. So, two threads both produce the 1000.

Right. And then we will see what is the output that we are going to get. So, here I have created. So, is over counter is invoking increment function. All right.

And it is over. Right. The for loop is over, and your lambda expression is over here. Right. So, in the main class, we have created two threads.

t1 is a thread. t2 is also a thread. And then here, you can see the thread constructor passing the task. So, here you have the task. And then you are starting t1.start and t2.start.

So, the previous program will help you. In C++, which we had seen. So, t1 will run, and t2 will run, and then we have to do the synchronization. When the first thread runs—t1 runs—the increment will be 1000, or whichever runs; t2 can also run first, right. So, one of the threads—without loss of generality, let us say t1, right—the increment will be how much?

1000, right. Next thread, t2, increment will be 1000. So, next thread when it starts, so it will come, it will go from 1001, 2003, etc., right. So, the total increment will be 2000, right. So, this is your usual syntax you are starting and then T1 and T2 are running then you are completing with t1 and t2 join right.

So, just refer the first program in Java that we had seen right you use start and then join right. So, here simultaneously two threads are running and counter dot increment when it is calling it is a synchronized function or synchronized method increment. So for t1 right it will give 1000 and t2 it will give 1000 or other way around right assume that first t2 is running right. So t2 will produce 1000 and t1 so on the whole the total should be invariant right. So 1000 plus 1000 we should get the output 2000 let us see right final count this will be the output right because it is in the double quote and then counter is calling getCount right. So, we have already have 2000. So, your getCount function when it is calling it will return count. So, the count is 2000. So, 2000 will be the output, right.

So, 2000 will be the output. So, what we can do? We can run this code and we will see we will go to the java console, right and we will run the code. So, java right count main you can see 2000, ok. So, the same code

So, maybe what we can do we can change to 500 500 let us see we open the Java code right. So, here what I will do I will change to the loop let us say 500 just a check and then save the code right. So, now again I will run here Java I have to compile first successful Java right. So, here you can see the output 1000 right. So, I have changed the loop to 500.

So, 500 plus 500, it should be 1000. So, this is how the synchronization is working in Java. So, in the next lecture, I will start from the deadlocks. The causes and prevention, we will see in the next class. Thank you very much.