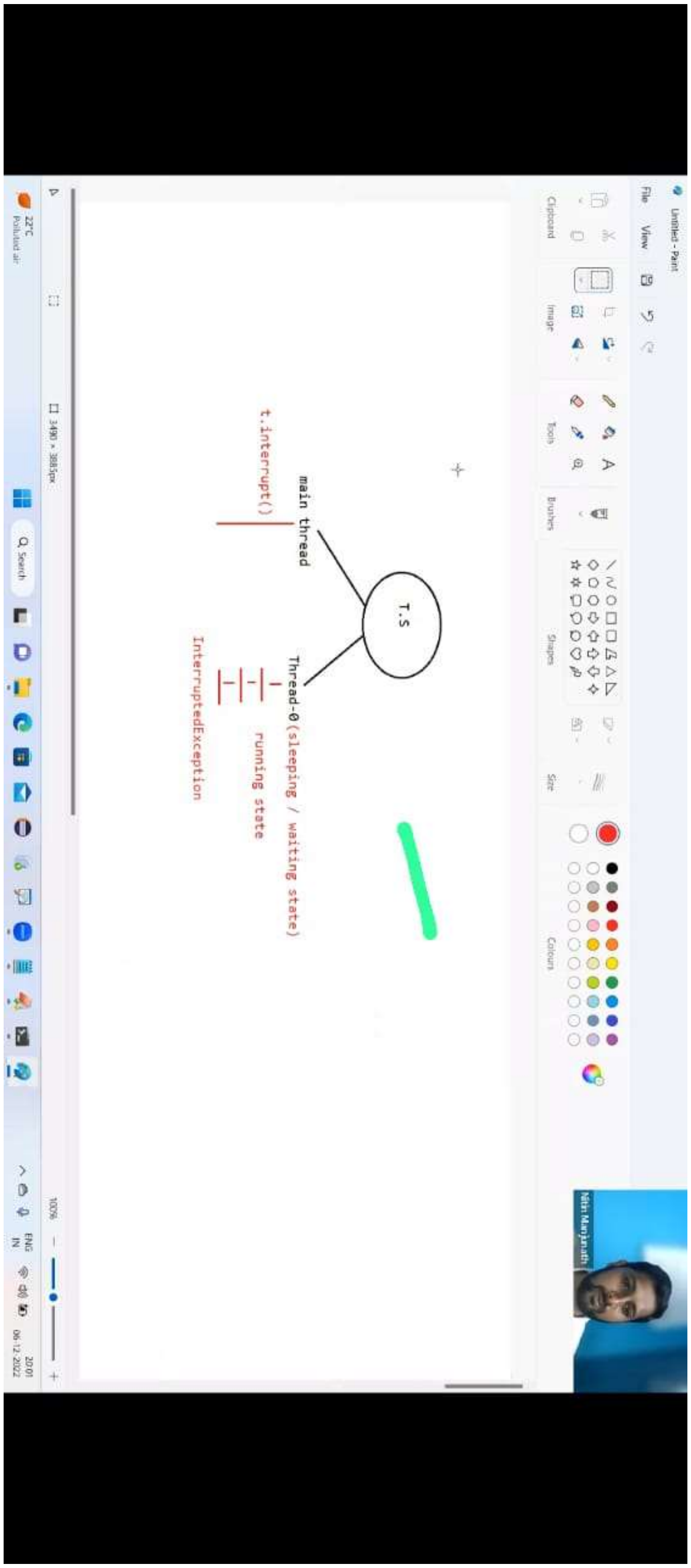
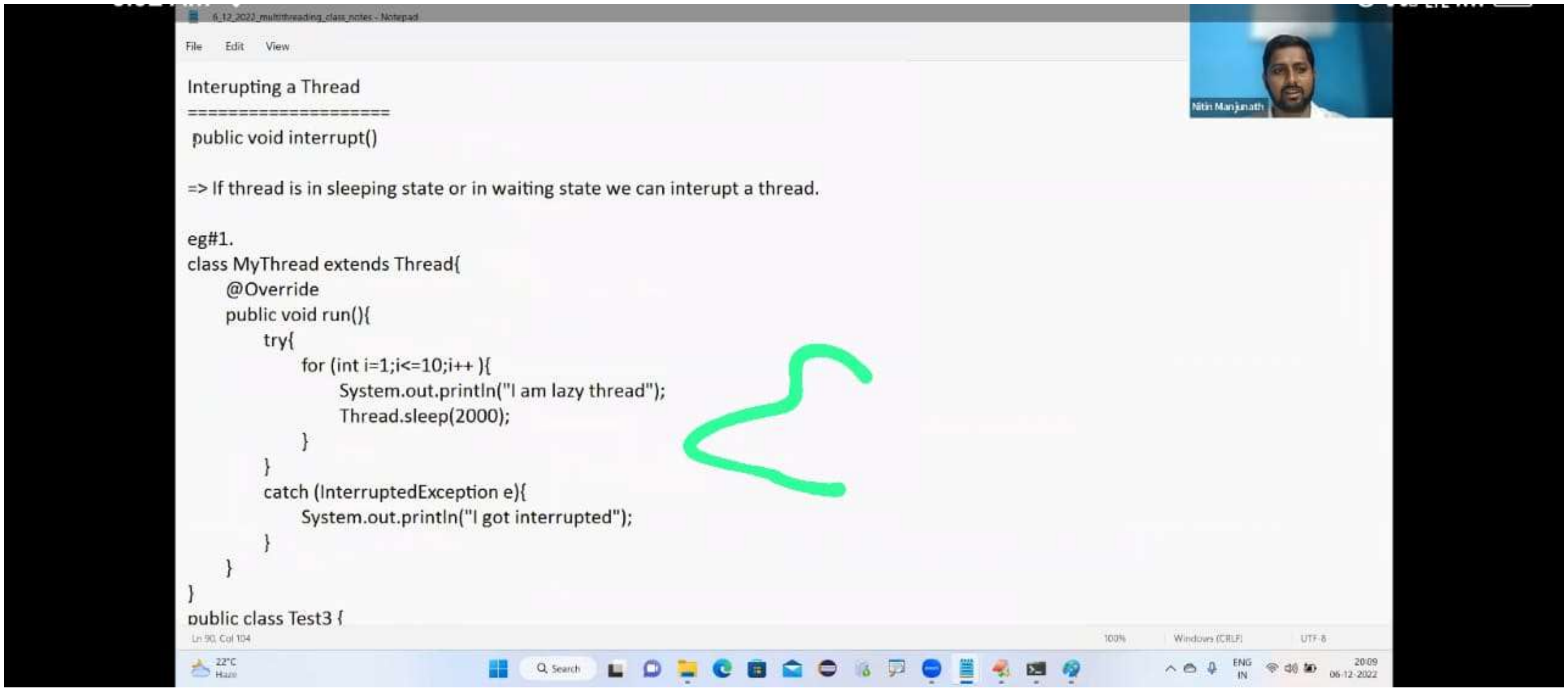
Java Multithreading Part4



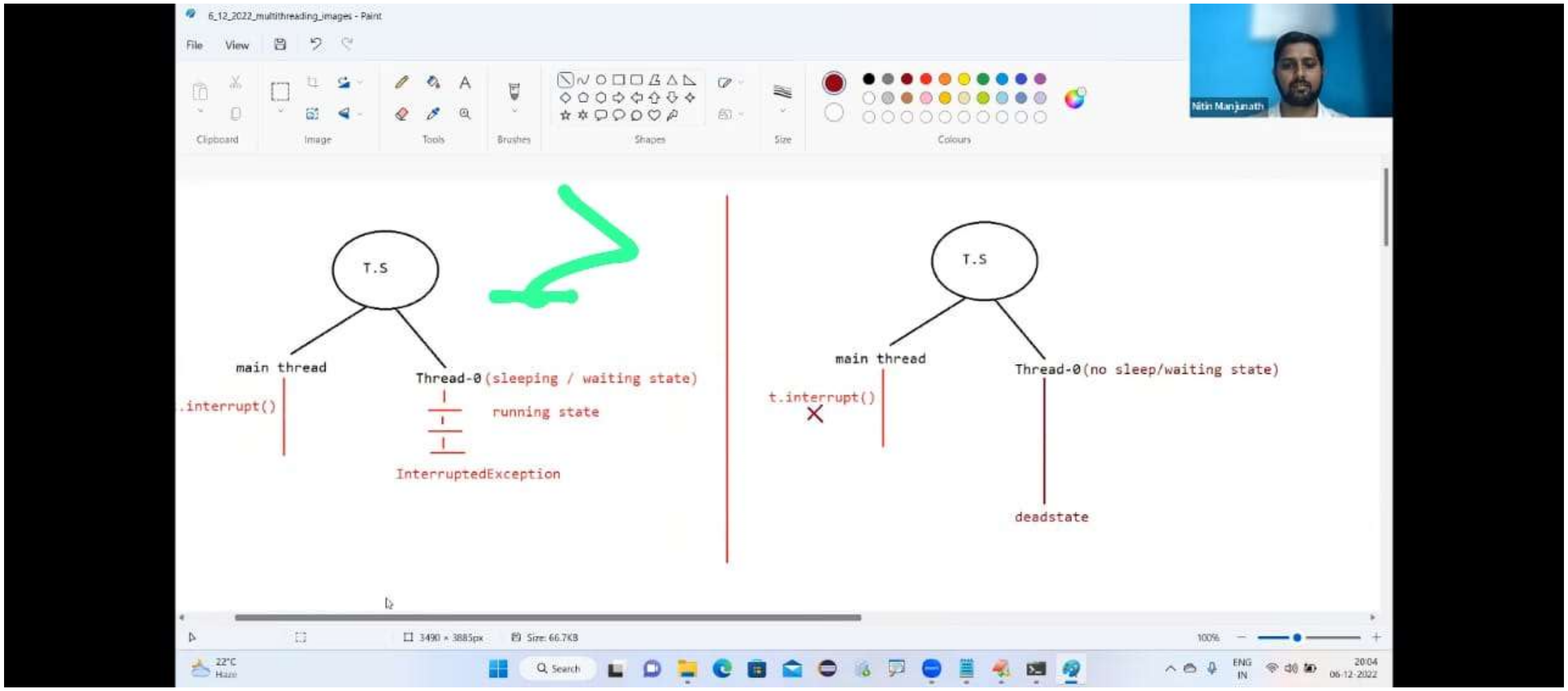


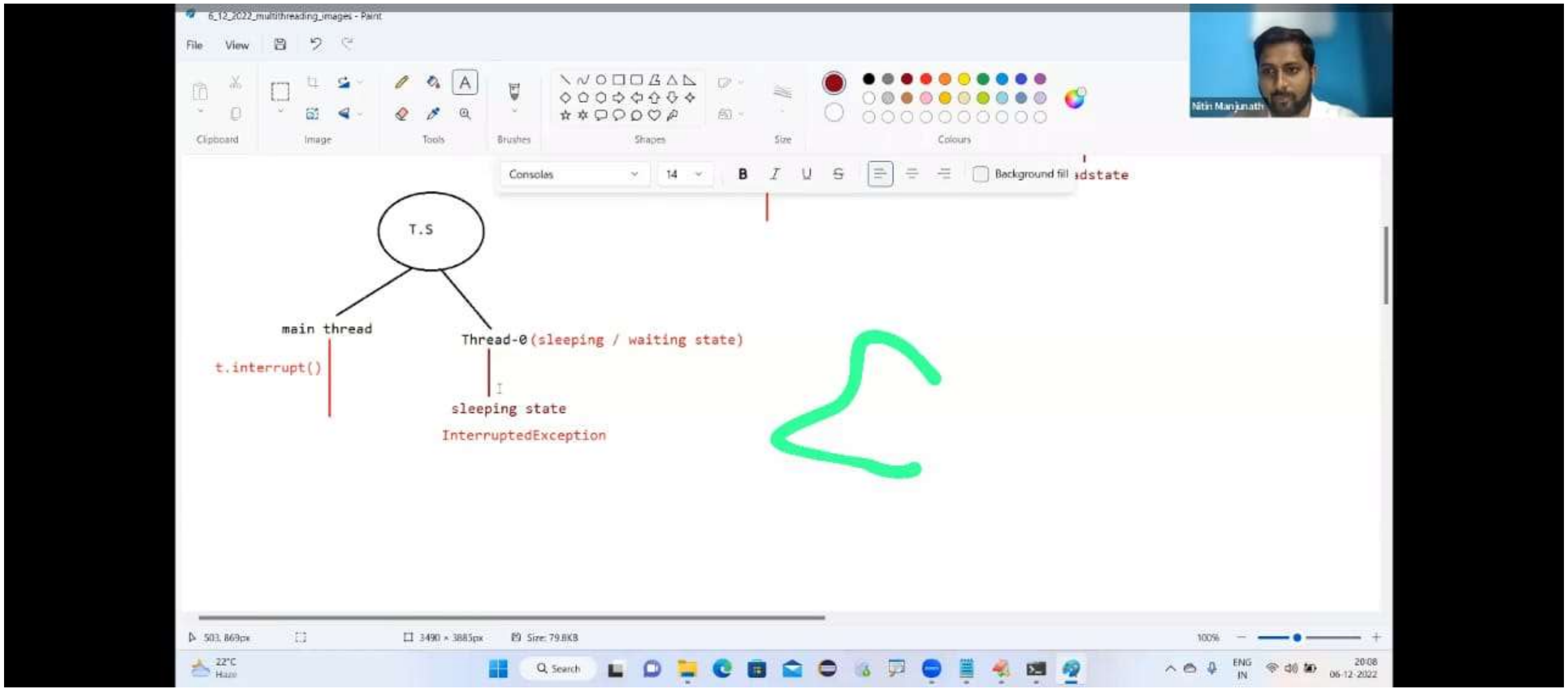
Interupting a Thread
====================
public void interrupt()

=> If thread is in sleeping state or in waiting state we can interupt a thread.

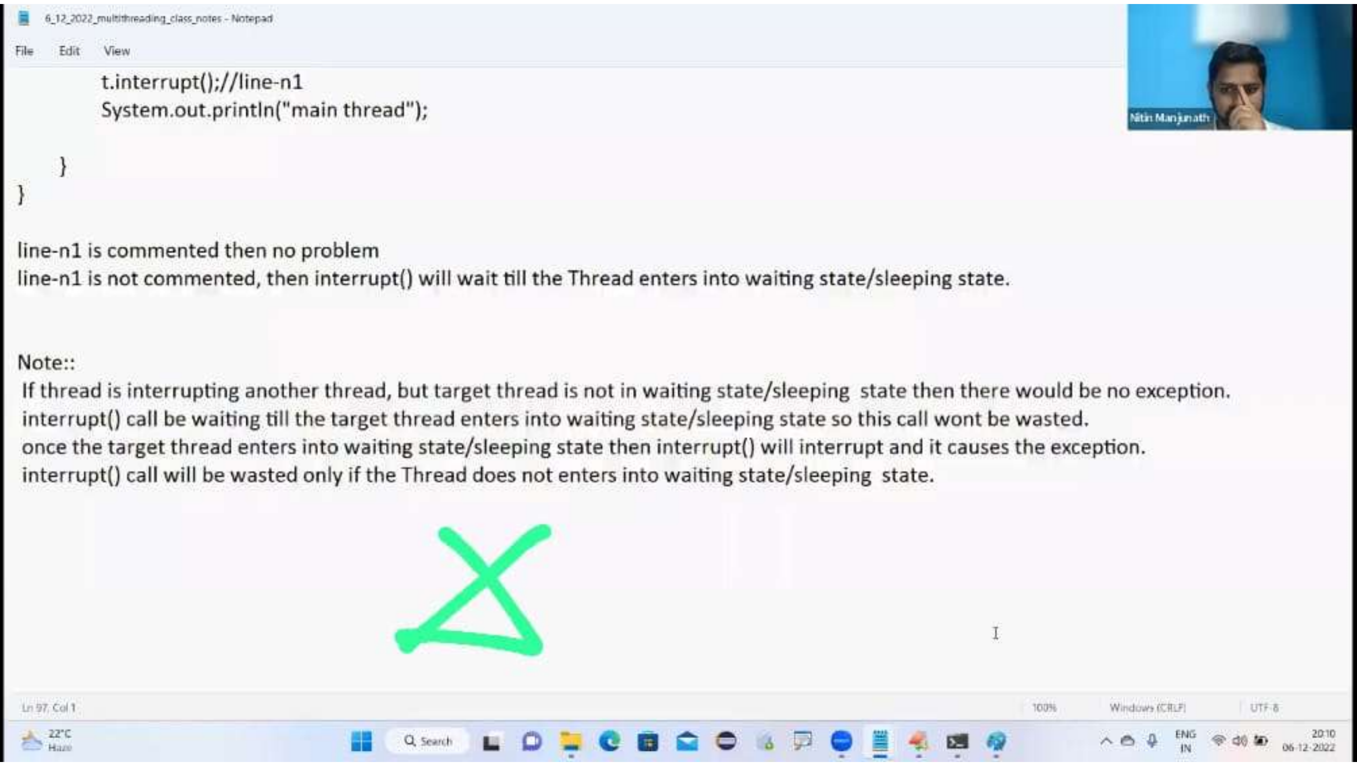
eg#1.

```
class MyThread extends Thread{
    @Override
    public void run(){
        try{
            for (int i=1;i<=10;i++ ){
                System.out.println("I am lazy thread");
                Thread.sleep(2000);
            }
        }
        catch (InterruptedException e){
            System.out.println("I got interrupted");
        }
    }
}
public class Test3 {
```


6_12_2022_multithreading_images - Paint
T.S
main thread
.interrupt()
Thread-0 (sleeping / waiting state)
running state
InterruptedException
T.S
main thread
t.interrupt()
Thread-0(no sleep/waiting state)
deadstate
Nitin Manjunath



```
T.S
main thread
t.interrupt()
Thread-0(sleeping / waiting state)
sleeping state
InterruptedException
adstate
```

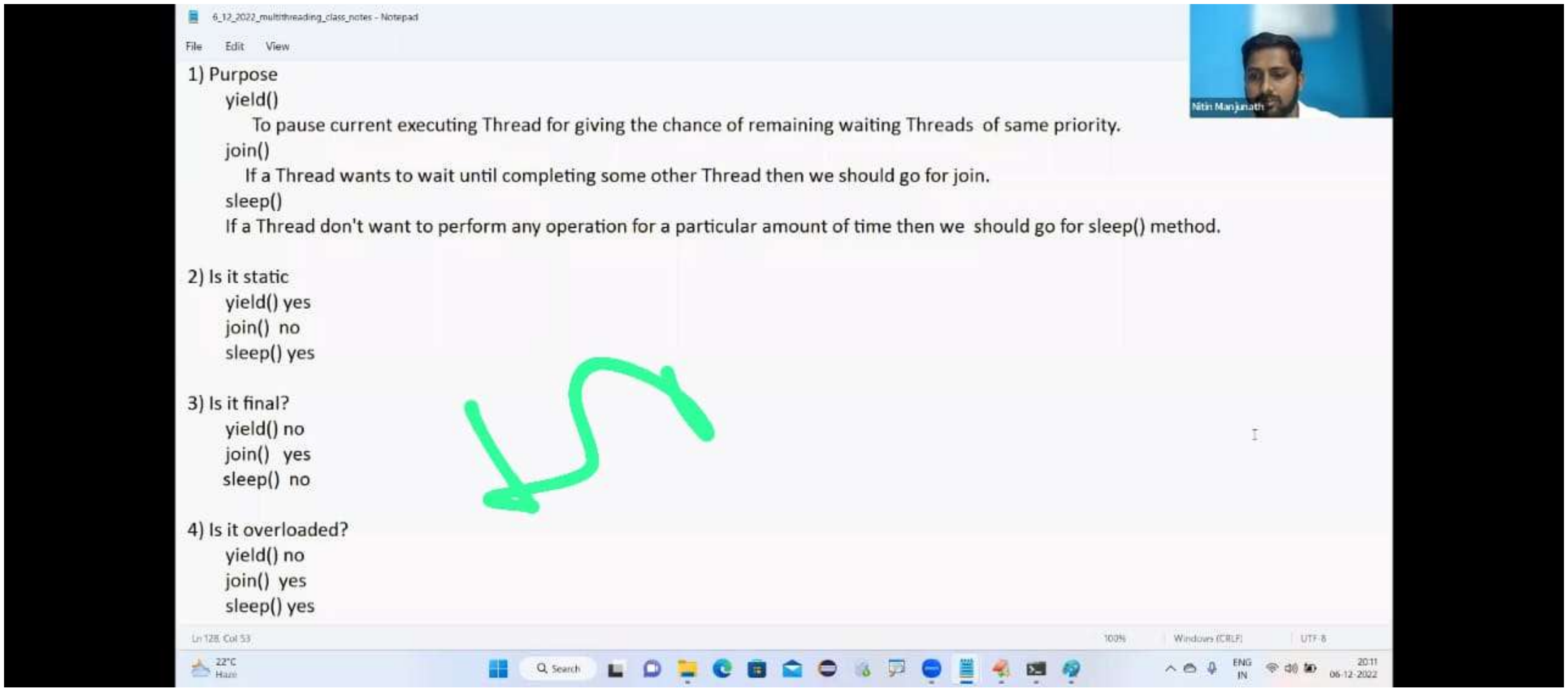


6_12_2022_multithreading_class_notes - Notepad

```
        t.interrupt();//line-n1
        System.out.println("main thread");

    }
}
```

line-n1 is commented then no problem
line-n1 is not commented, then interrupt() will wait till the Thread enters into waiting state/sleeping state.

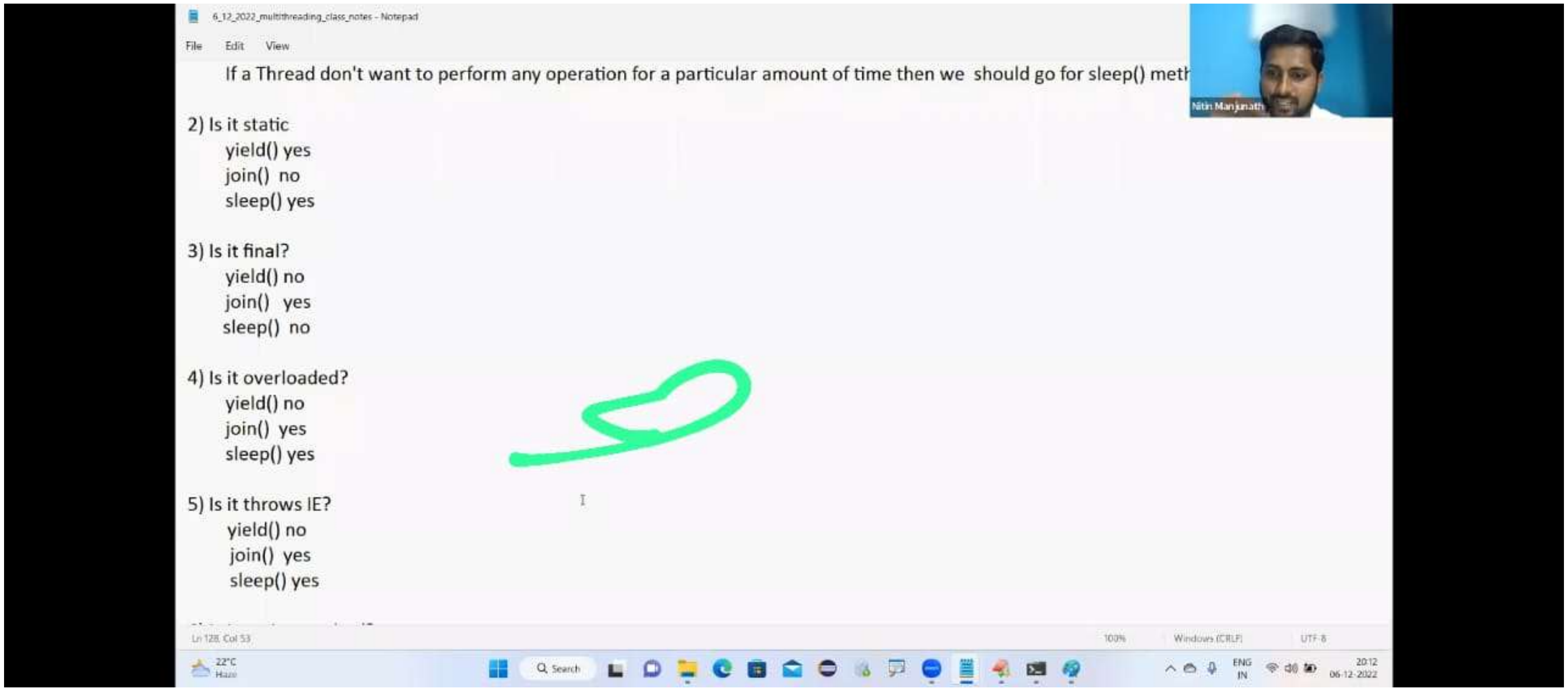
Note::
If thread is interrupting another thread, but target thread is not in waiting state/sleeping state then there would be no exception.
interrupt() call be waiting till the target thread enters into waiting state/sleeping state so this call wont be wasted.
once the target thread enters into waiting state/sleeping state then interrupt() will interrupt and it causes the exception.
interrupt() call will be wasted only if the Thread does not enters into waiting state/sleeping state.



6_12_2022_multithreading_class_notes - Notepad

```
1) Purpose
    yield()
       To pause current executing Thread for giving the chance of remaining waiting Threads  of same priority.
    join()
      If a Thread wants to wait until completing some other Thread then we should go for join.
    sleep()
    If a Thread don't want to perform any operation for a particular amount of time then we  should go for sleep() method.

2) Is it static
    yield() yes
    join()  no
    sleep() yes

3) Is it final?
    yield() no
    join()  yes
    sleep()  no

4) Is it overloaded?
    yield() no
    join()  yes
    sleep() yes
```

5



6_12_2022_multithreading_class_notes - Notepad

File Edit View

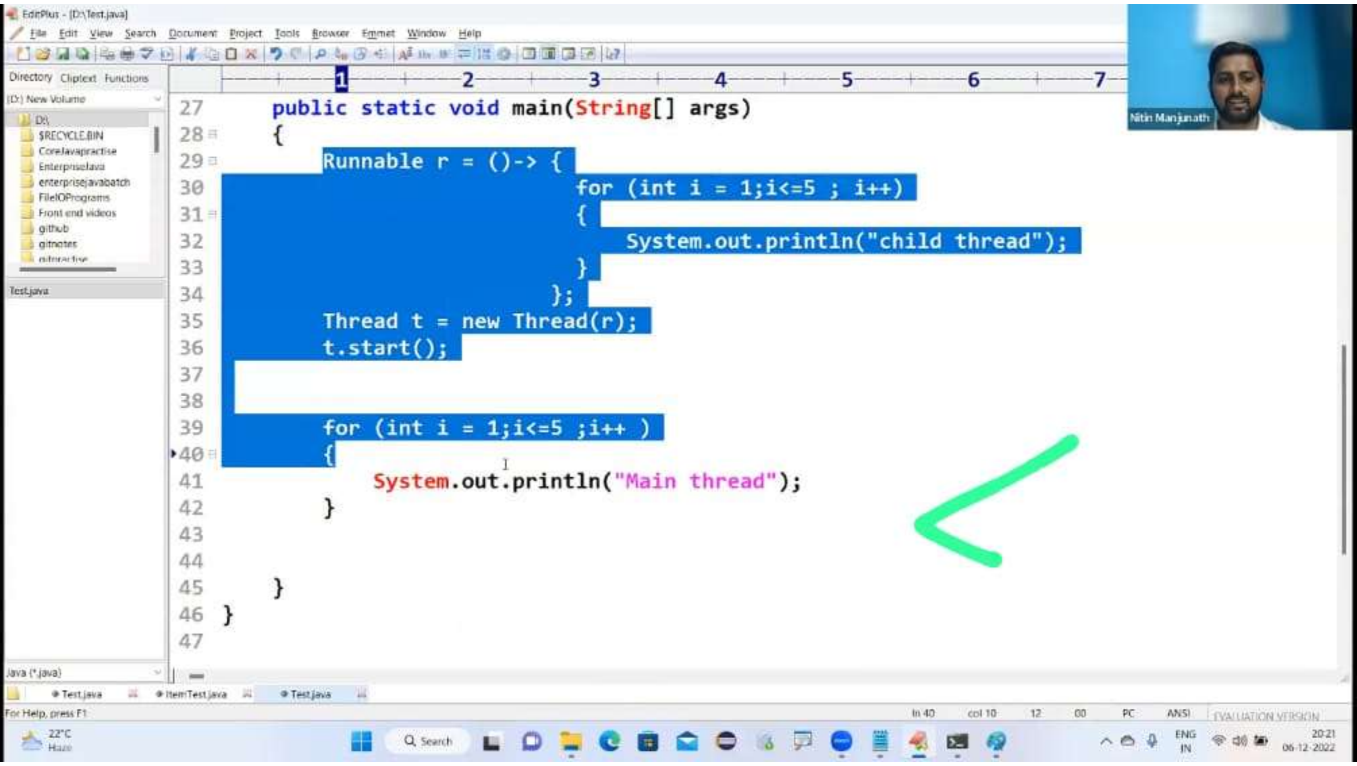
If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() meth

2) Is it static
    yield() yes
    join() no
    sleep() yes

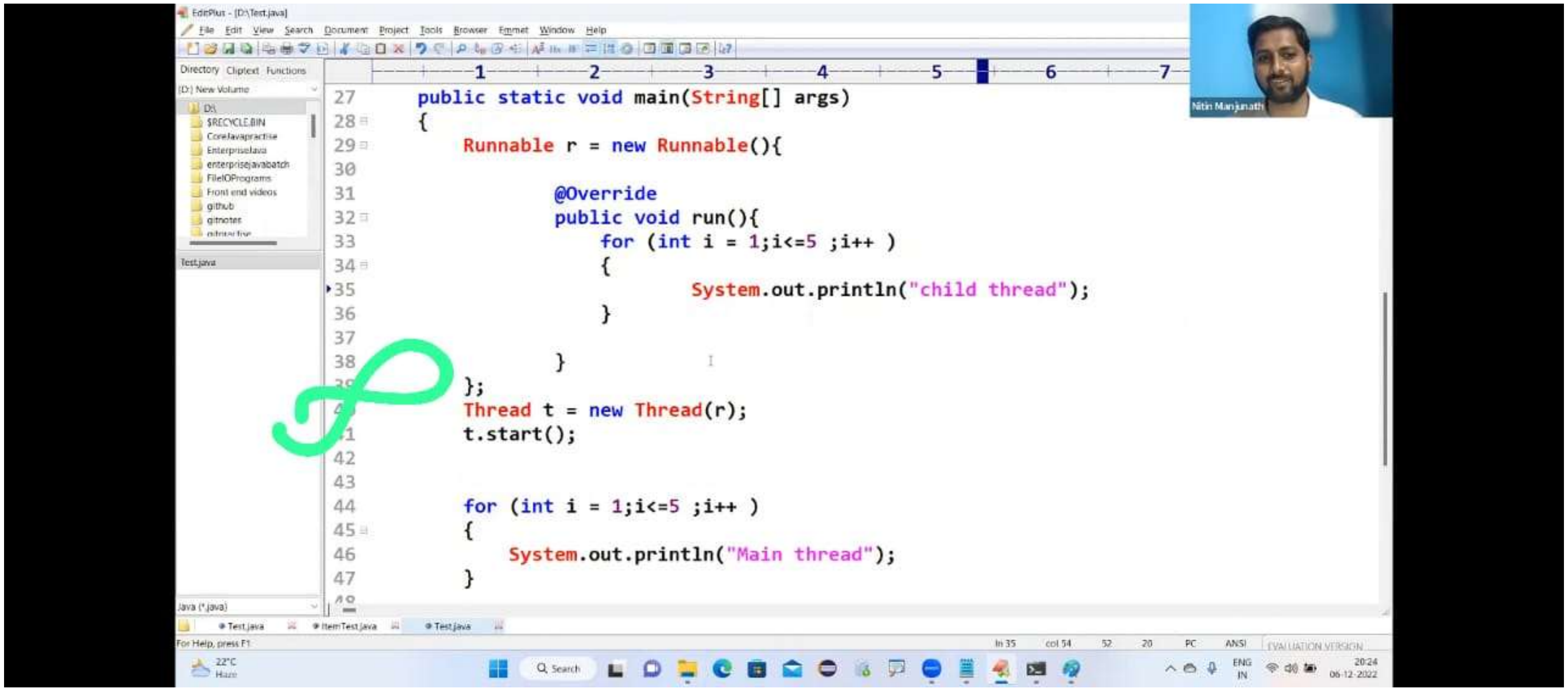
3) Is it final?
    yield() no
    join() yes
    sleep() no

4) Is it overloaded?
    yield() no
    join() yes
    sleep() yes

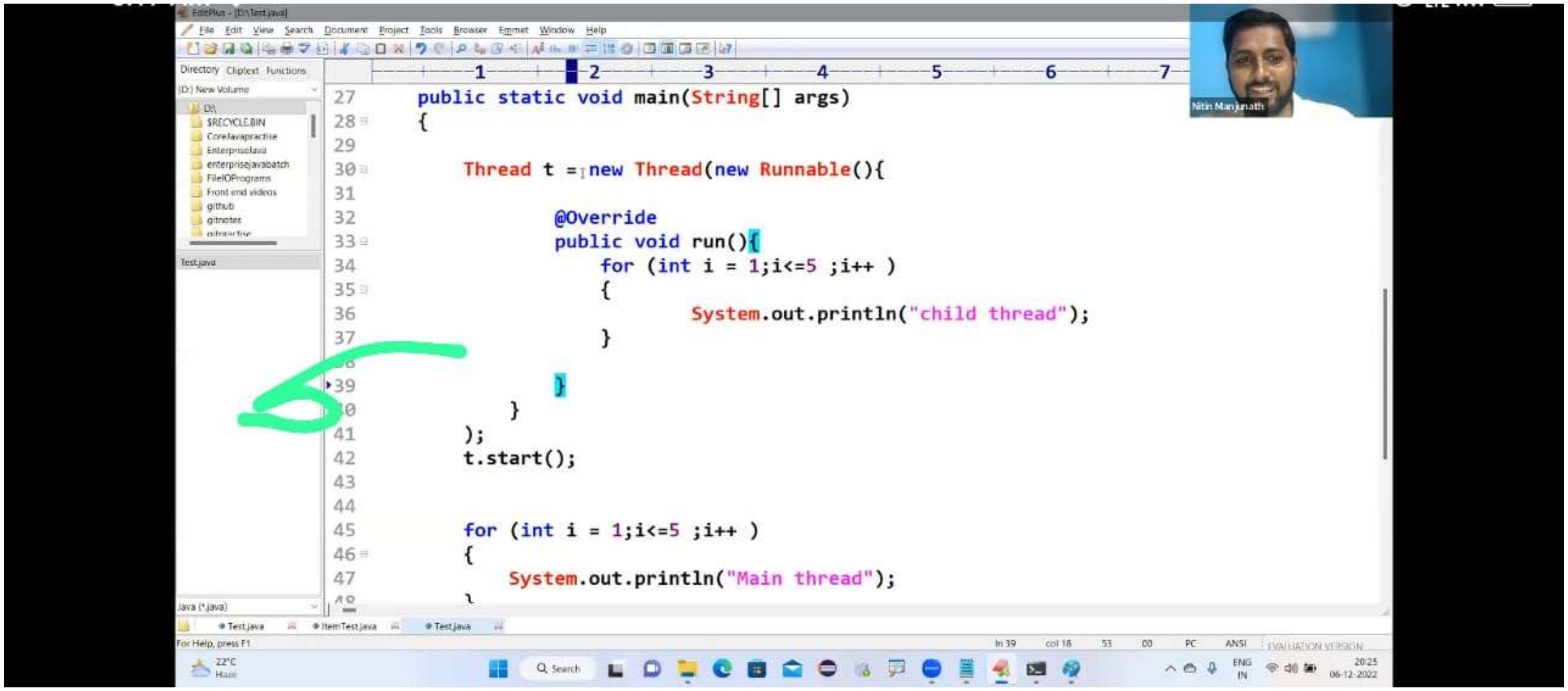
5) Is it throws IE?
    yield() no
    join() yes
    sleep() yes



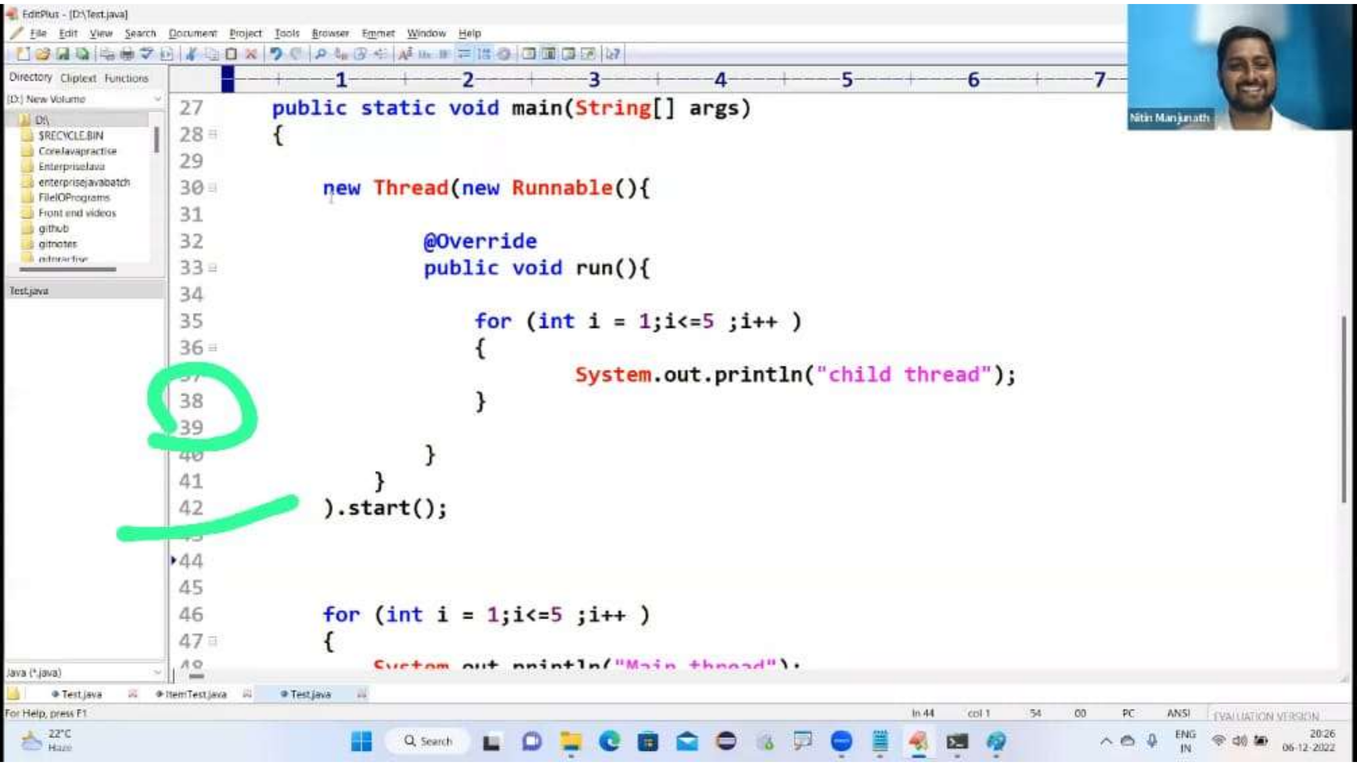
```
    public static void main(String[] args)
    {
        Runnable r = ()-> {
                          for (int i = 1;i<=5 ; i++)
                          {
                              System.out.println("child thread");
                          }
                        };
        Thread t = new Thread(r);
        t.start();


        for (int i = 1;i<=5 ;i++ )
        {
            System.out.println("Main thread");
        }


    }
}
```



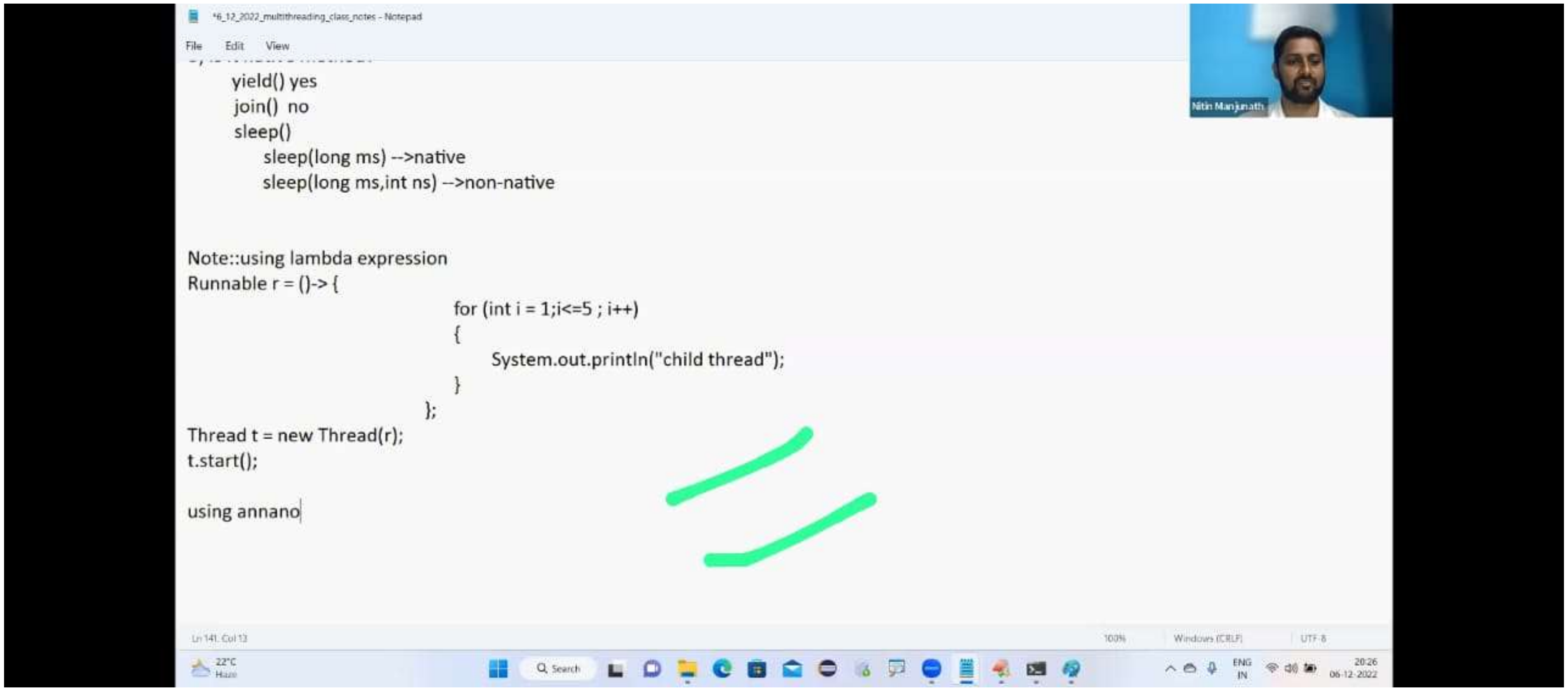
```java
public static void main(String[] args)
{
    Runnable r = new Runnable(){

            @Override
            public void run(){
                for (int i = 1;i<=5 ;i++ )
                {
                        System.out.println("child thread");
                }

            }
    };
    Thread t = new Thread(r);
    t.start();


    for (int i = 1;i<=5 ;i++ )
    {
        System.out.println("Main thread");
    }
```



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

    Thread t = new Thread(new Runnable(){

            @Override
            public void run(){
                for (int i = 1;i<=5 ;i++ )
                {
                        System.out.println("child thread");
                }

            }
        }
    );
    t.start();


    for (int i = 1;i<=5 ;i++ )
    {
        System.out.println("Main thread");
```

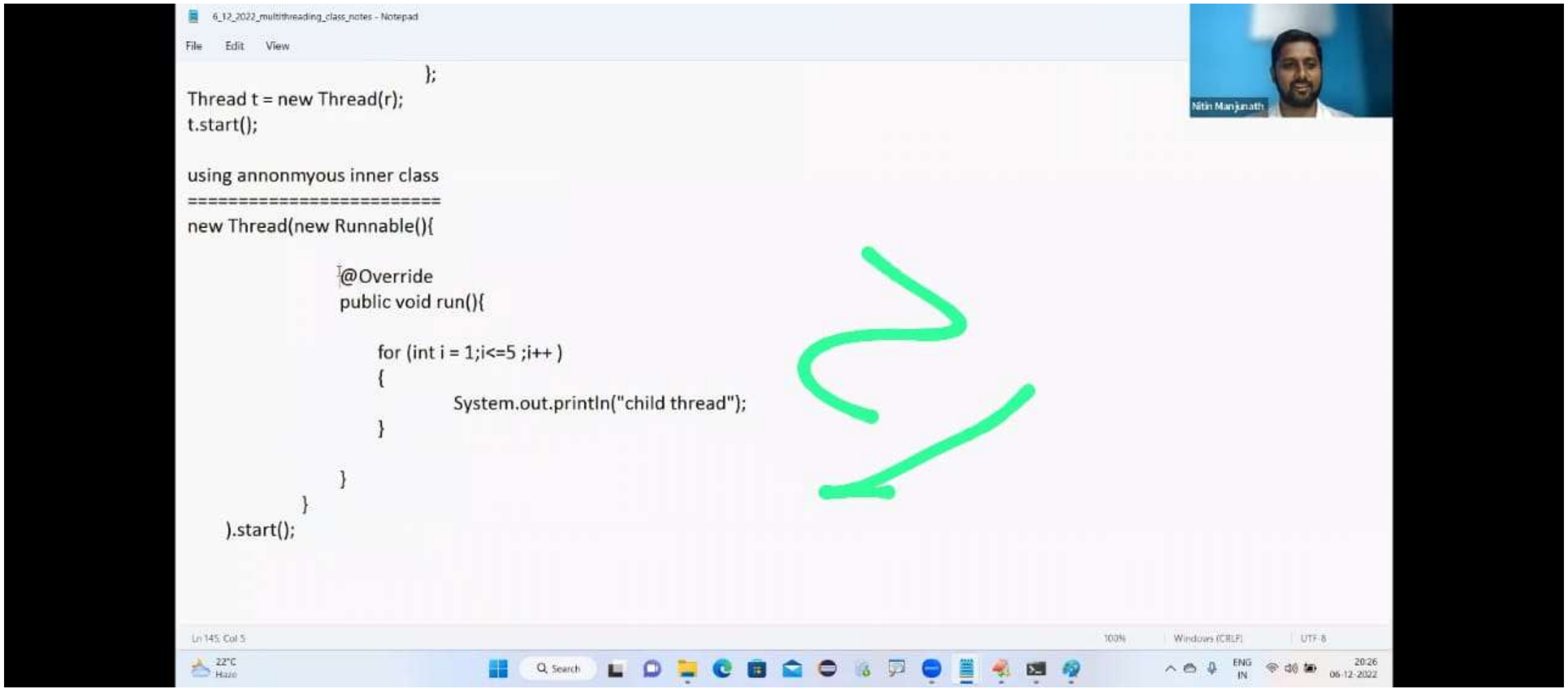


```java
public static void main(String[] args)
{

    new Thread(new Runnable(){

            @Override
            public void run(){

                for (int i = 1;i<=5 ;i++ )
                {
                        System.out.println("child thread");
                }

            }
        }
    ).start();



    for (int i = 1;i<=5 ;i++ )
    {
        System.out.println("Main thread");
```



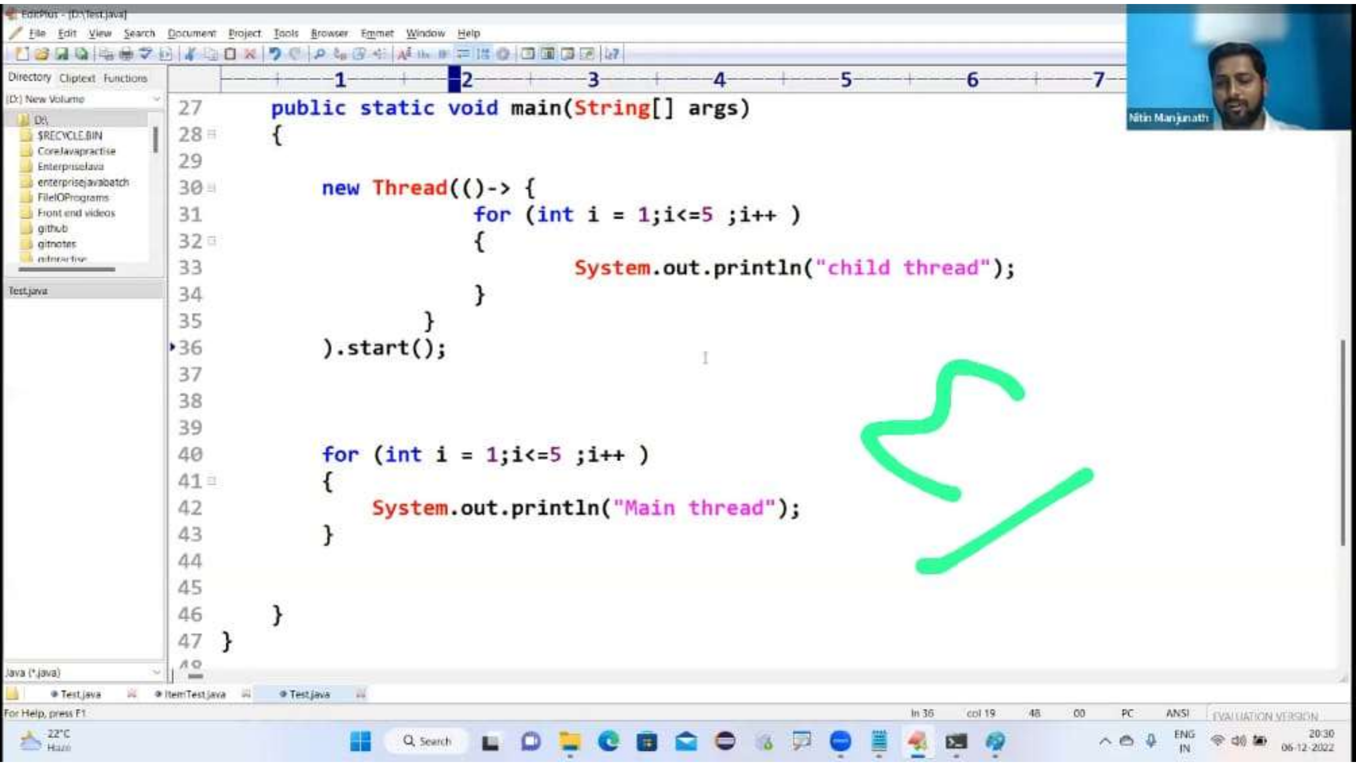
*6_12_2022_multithreading_class_notes - Notepad

```
    yield() yes
    join()  no
    sleep()
        sleep(long ms) -->native
        sleep(long ms,int ns) -->non-native


Note::using lambda expression
Runnable r = ()-> {
                        for (int i = 1;i<=5 ; i++)
                        {
                            System.out.println("child thread");
                        }
                    };
Thread t = new Thread(r);
t.start();

using annano
```



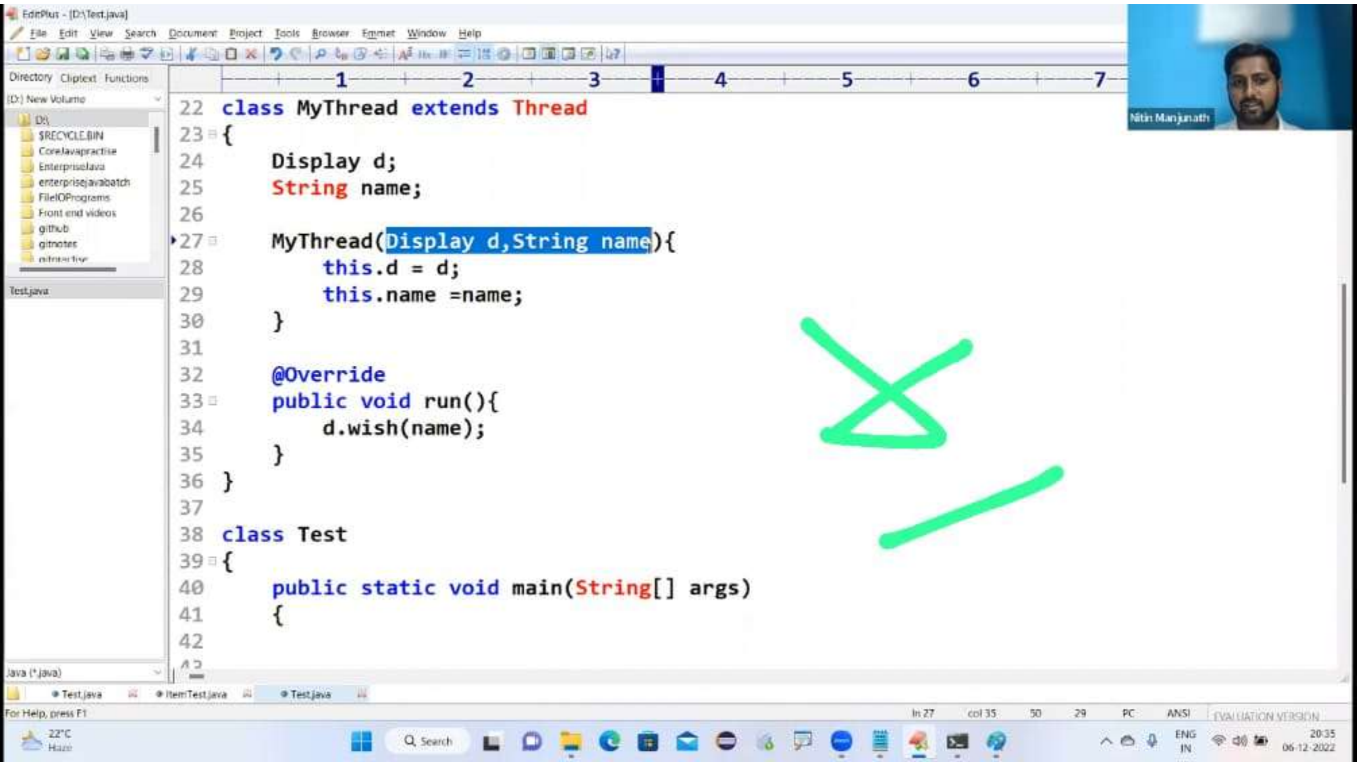
```
                    };
Thread t = new Thread(r);
t.start();

using annonmyous inner class
==========================
new Thread(new Runnable(){

            @Override
            public void run(){

                for (int i = 1;i<=5 ;i++ )
                {
                        System.out.println("child thread");
                }

            }
        }
    ).start();
```

12

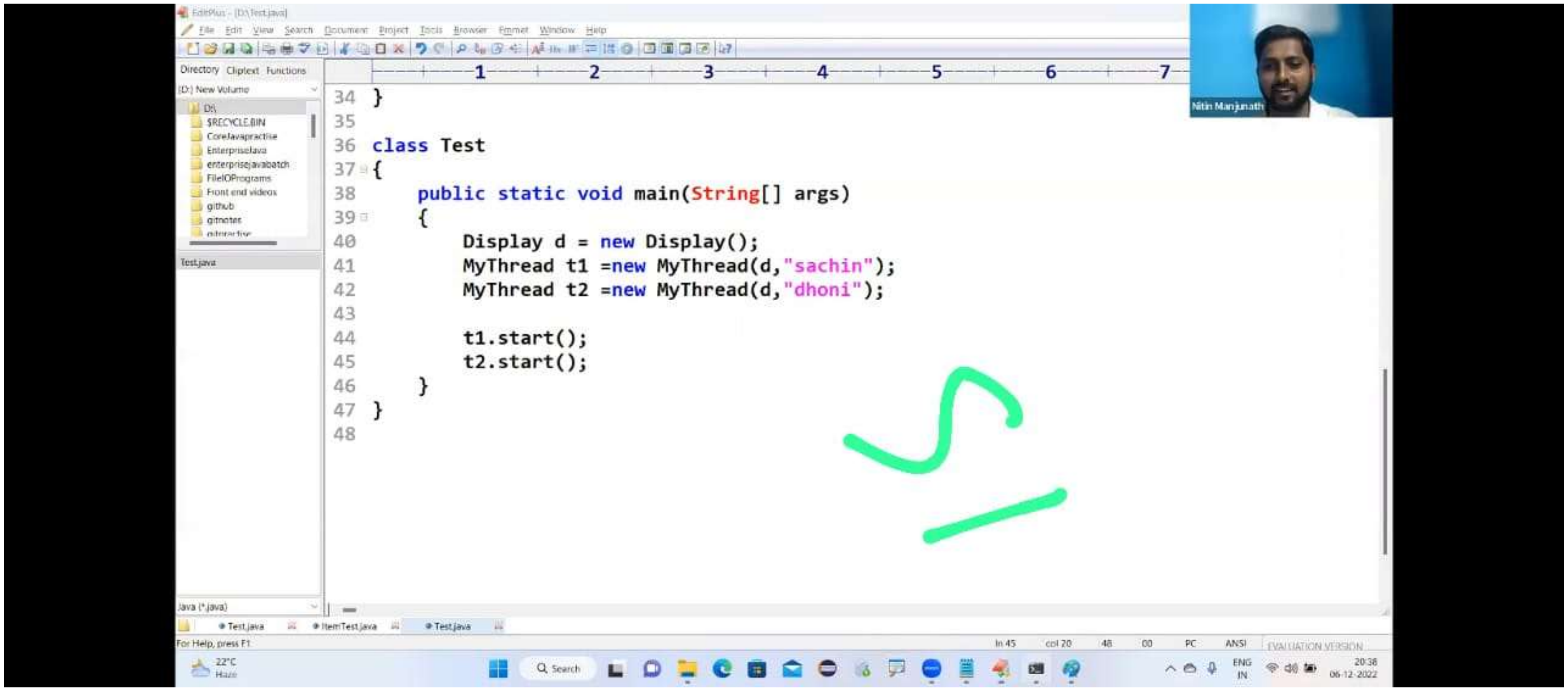


```java
    public static void main(String[] args)
    {

        new Thread(()-> {
                    for (int i = 1;i<=5 ;i++ )
                    {
                            System.out.println("child thread");
                    }
                }
        ).start();


        for (int i = 1;i<=5 ;i++ )
        {
            System.out.println("Main thread");
        }


    }
}
```

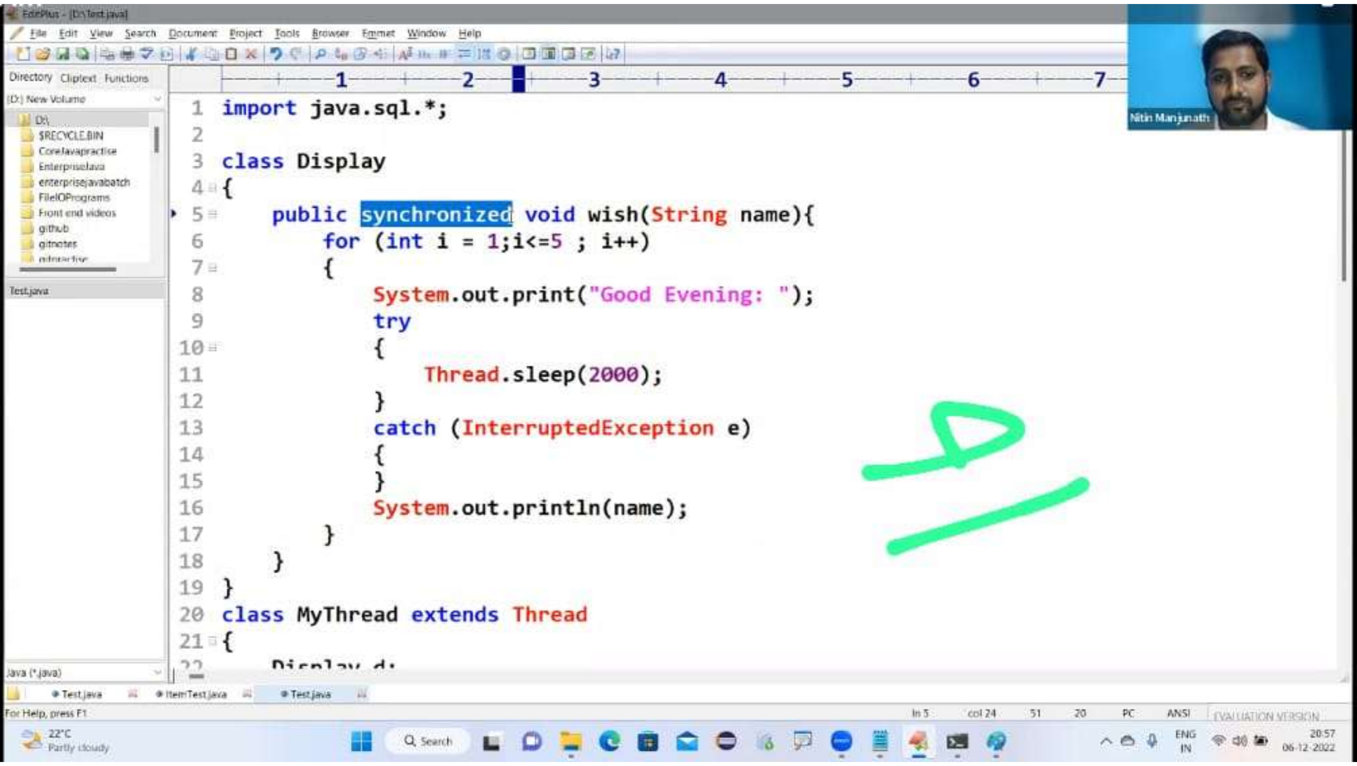
13



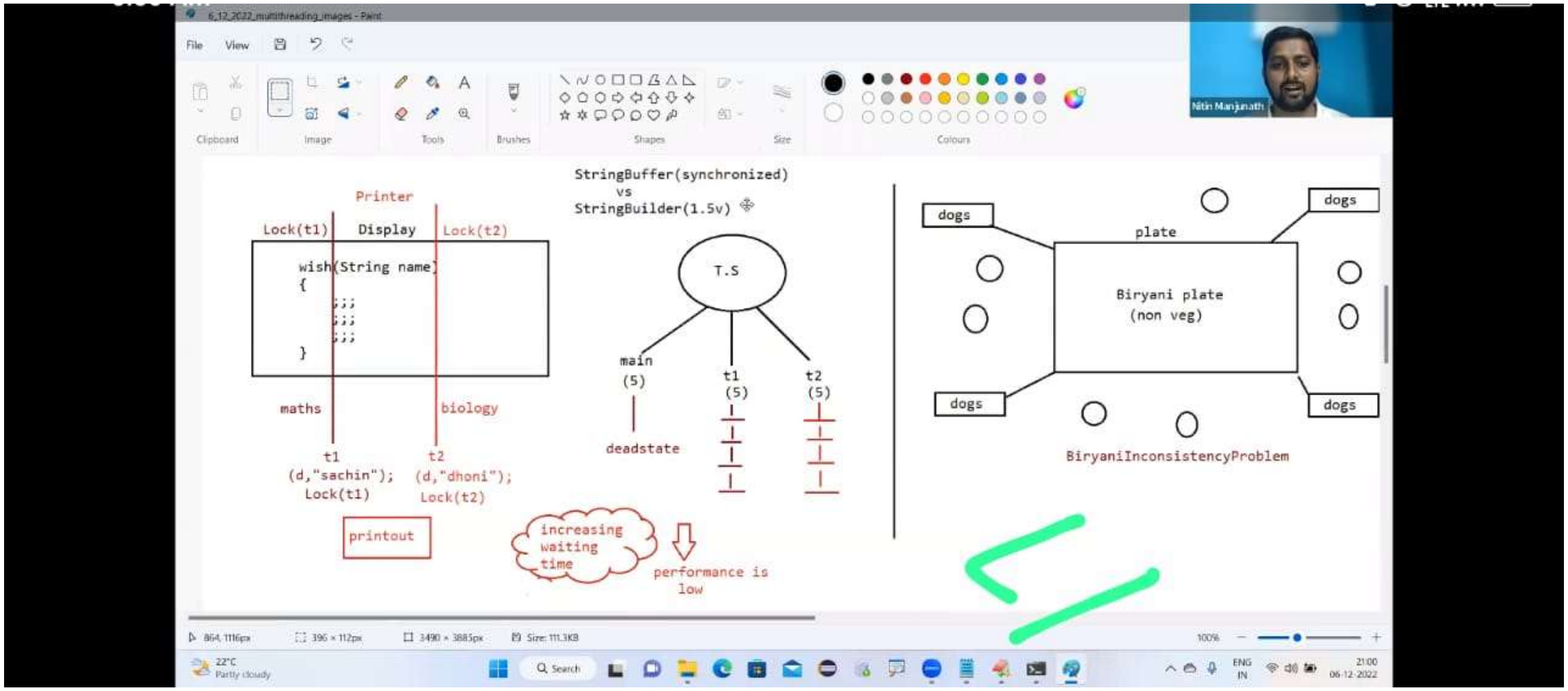
```
class MyThread extends Thread
{
    Display d;
    String name;

    MyThread(Display d,String name){
        this.d = d;
        this.name =name;
    }

    @Override
    public void run(){
        d.wish(name);
    }
}

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

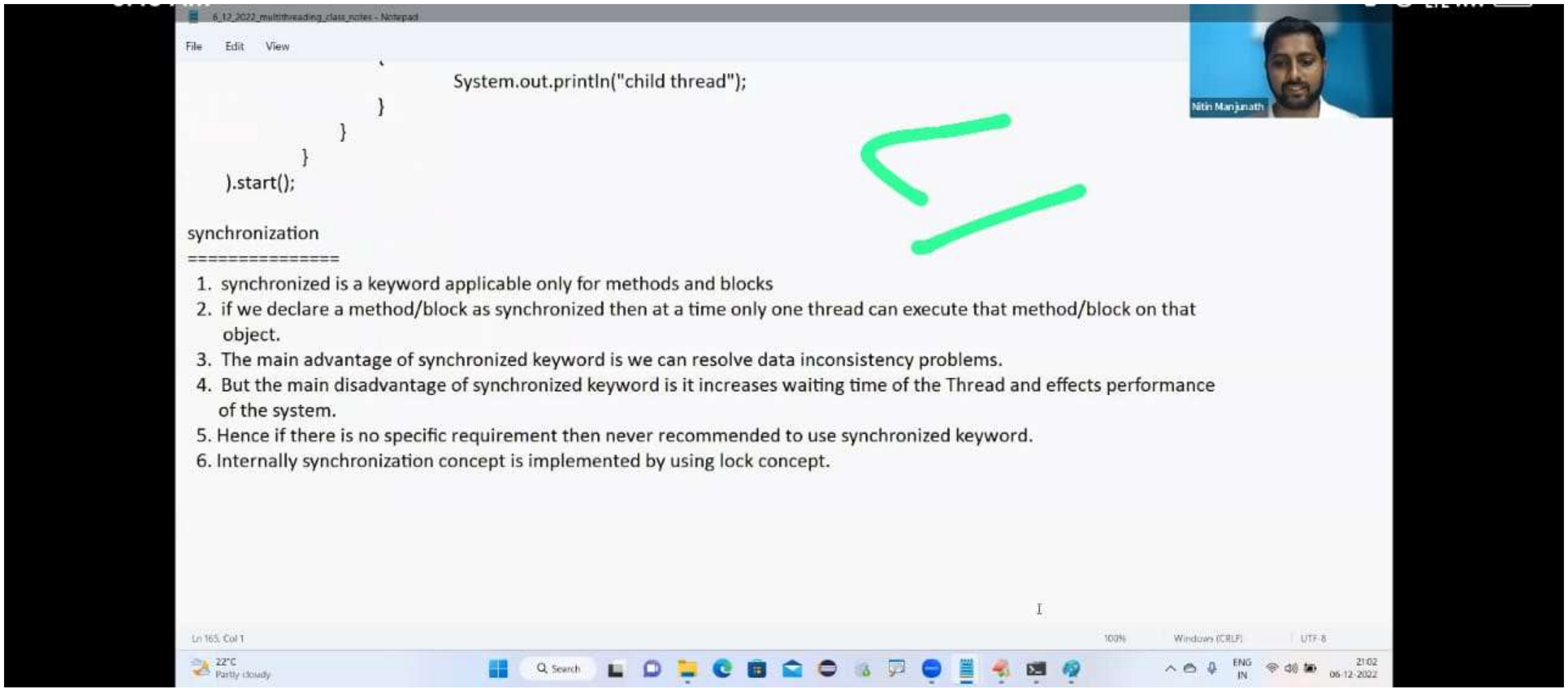


```
}

class Test
{
    public static void main(String[] args)
    {
        Display d = new Display();
        MyThread t1 =new MyThread(d,"sachin");
        MyThread t2 =new MyThread(d,"dhoni");

        t1.start();
        t2.start();
    }
}
```



```java
import java.sql.*;

class Display
{
    public synchronized void wish(String name){
        for (int i = 1;i<=5 ; i++)
        {
            System.out.print("Good Evening: ");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e)
            {
            }
            System.out.println(name);
        }
    }
}
class MyThread extends Thread
{
    Display d;
```

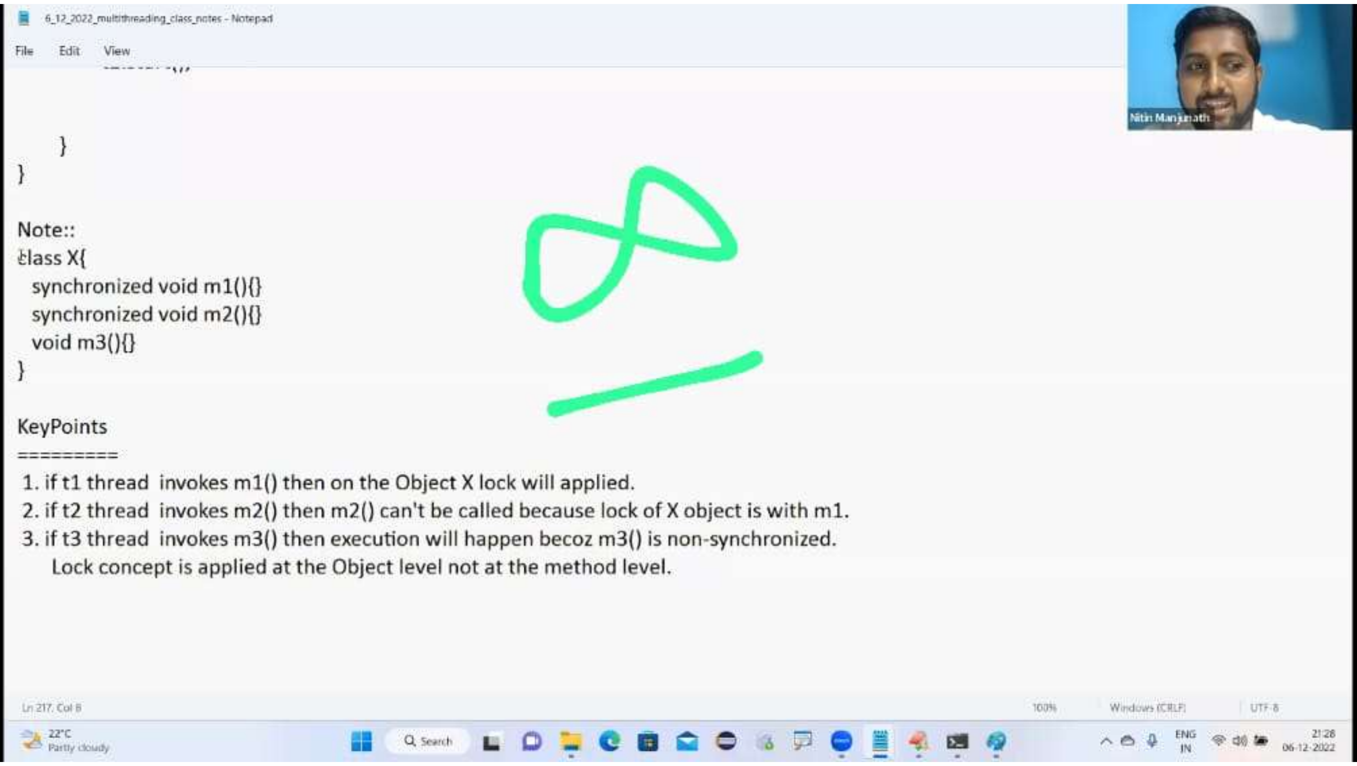

6_12_2022_multithreading_images - Paint
Printer
Lock(t1)
Display
Lock(t2)
wish(String name)
{
;;;
;;;
;;;
}
maths
biology
t1
(d,"sachin");
Lock(t1)
t2
(d,"dhoni");
Lock(t2)
printout
increasing waiting time
performance is low
StringBuffer(synchronized)
vs
StringBuilder(1.5v)
T.S
main
(5)
t1
(5)
t2
(5)
deadstate
dogs
plate
Biryani plate
(non veg)
BiryaniInconsistencyProblem
Nitin Manjunath



```
                        System.out.println("child thread");
                }
            }
        }
    ).start();
```

synchronization
===============

1. synchronized is a keyword applicable only for methods and blocks
2. if we declare a method/block as synchronized then at a time only one thread can execute that method/block on that object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. Internally synchronization concept is implemented by using lock concept.

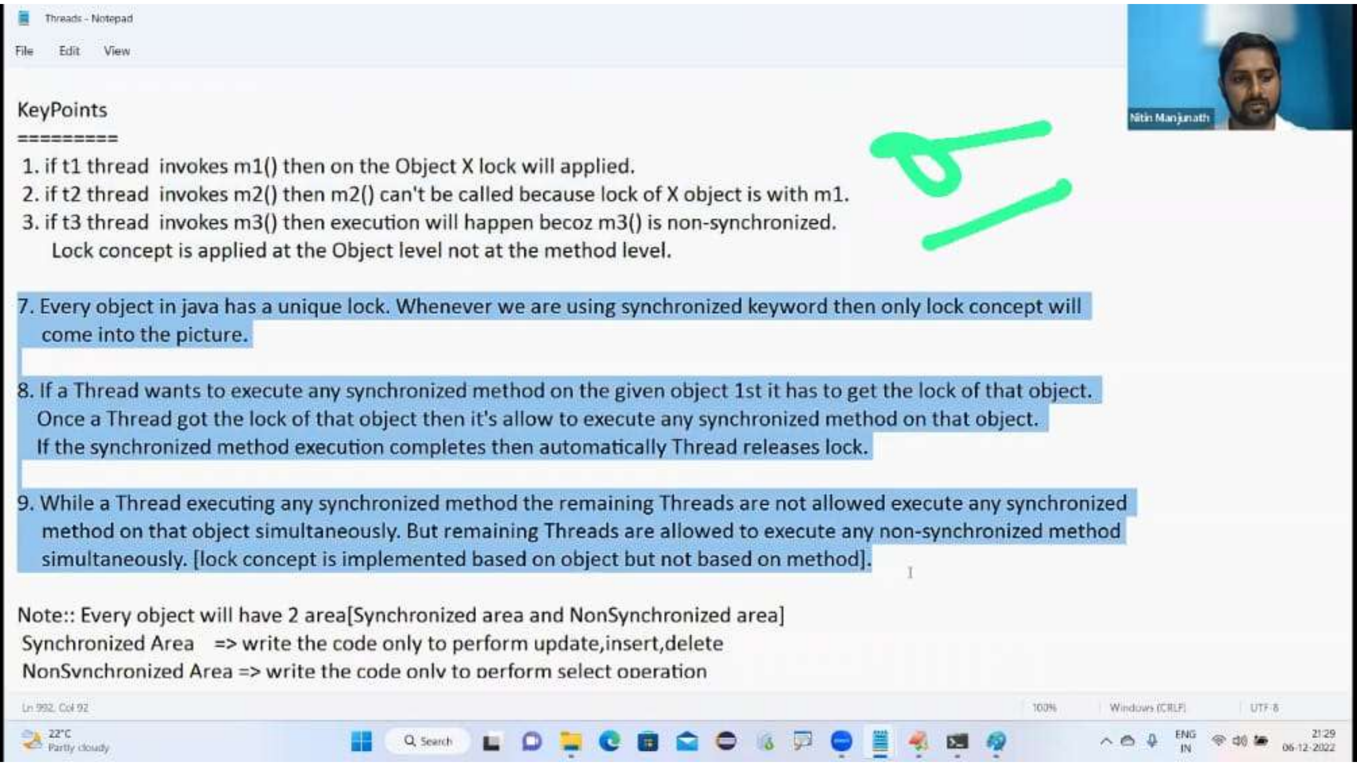


6_12_2022_multithreading_class_notes - Notepad

```
      }
}

Note::
class X{
  synchronized void m1(){}
  synchronized void m2(){}
  void m3(){}
}

KeyPoints
=========
 1. if t1 thread  invokes m1() then on the Object X lock will applied.
 2. if t2 thread  invokes m2() then m2() can't be called because lock of X object is with m1.
 3. if t3 thread  invokes m3() then execution will happen becoz m3() is non-synchronized.
     Lock concept is applied at the Object level not at the method level.
```

18

19



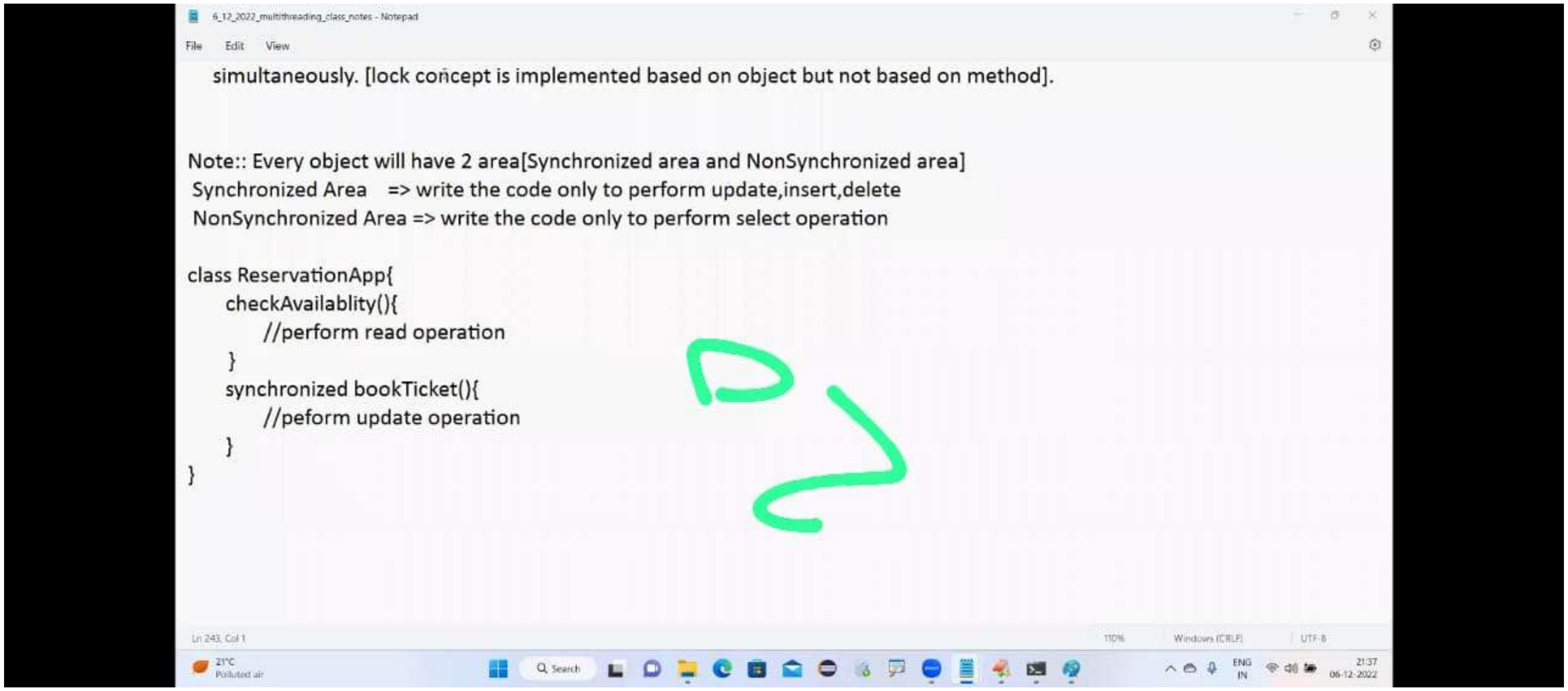
KeyPoints
=========
1. if t1 thread invokes m1() then on the Object X lock will applied.
2. if t2 thread invokes m2() then m2() can't be called because lock of X object is with m1.
3. if t3 thread invokes m3() then execution will happen becoz m3() is non-synchronized.
   Lock concept is applied at the Object level not at the method level.

7. Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.

8. If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object.
Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object.
If the synchronized method execution completes then automatically Thread releases lock.

9. While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].

Note:: Every object will have 2 area[Synchronized area and NonSynchronized area]
Synchronized Area   => write the code only to perform update,insert,delete
NonSynchronized Area => write the code only to perform select operation

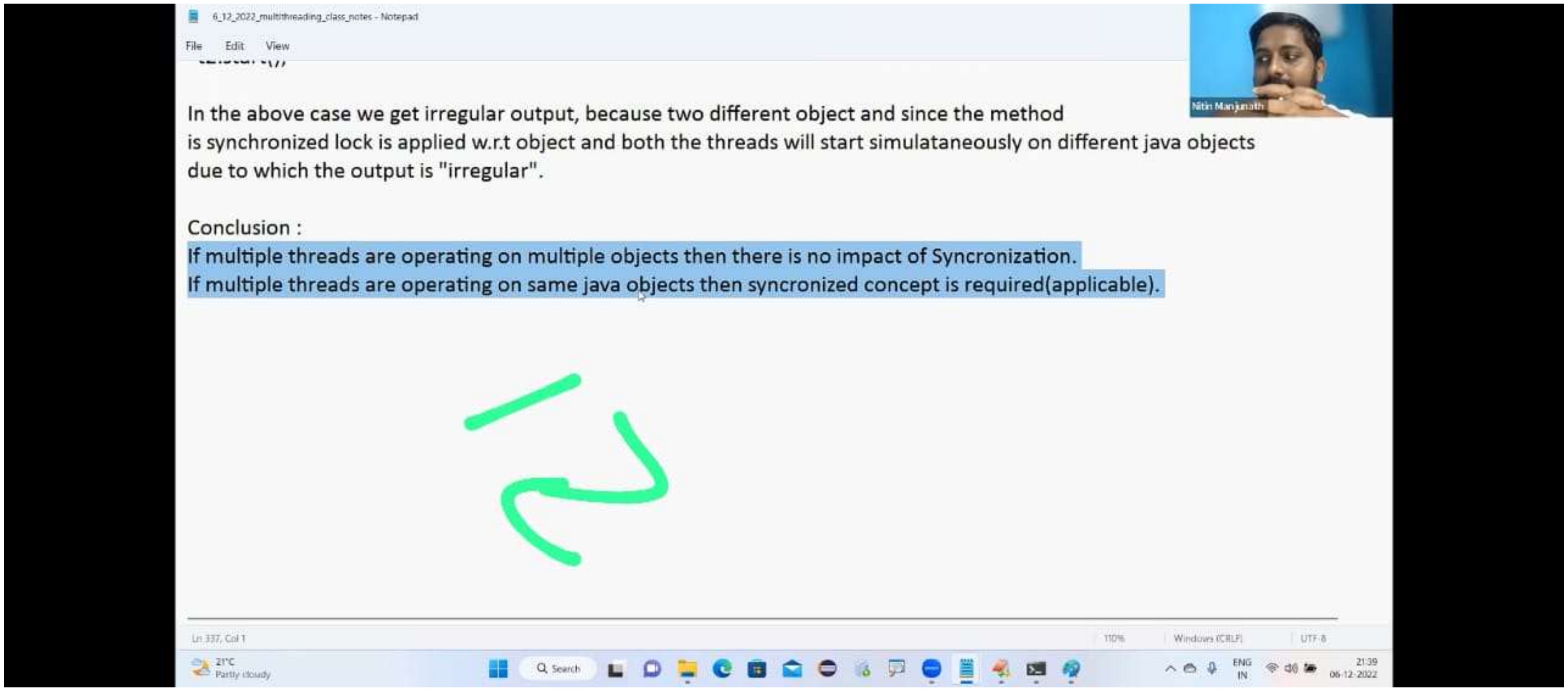


simultaneously. [lock coñcept is implemented based on object but not based on method].

Note:: Every object will have 2 area[Synchronized area and NonSynchronized area]
Synchronized Area   => write the code only to perform update,insert,delete
NonSynchronized Area => write the code only to perform select operation

```
class ReservationApp{
     checkAvailablity(){
          //perform read operation
     }
     synchronized bookTicket(){
          //peform update operation
     }
}
```

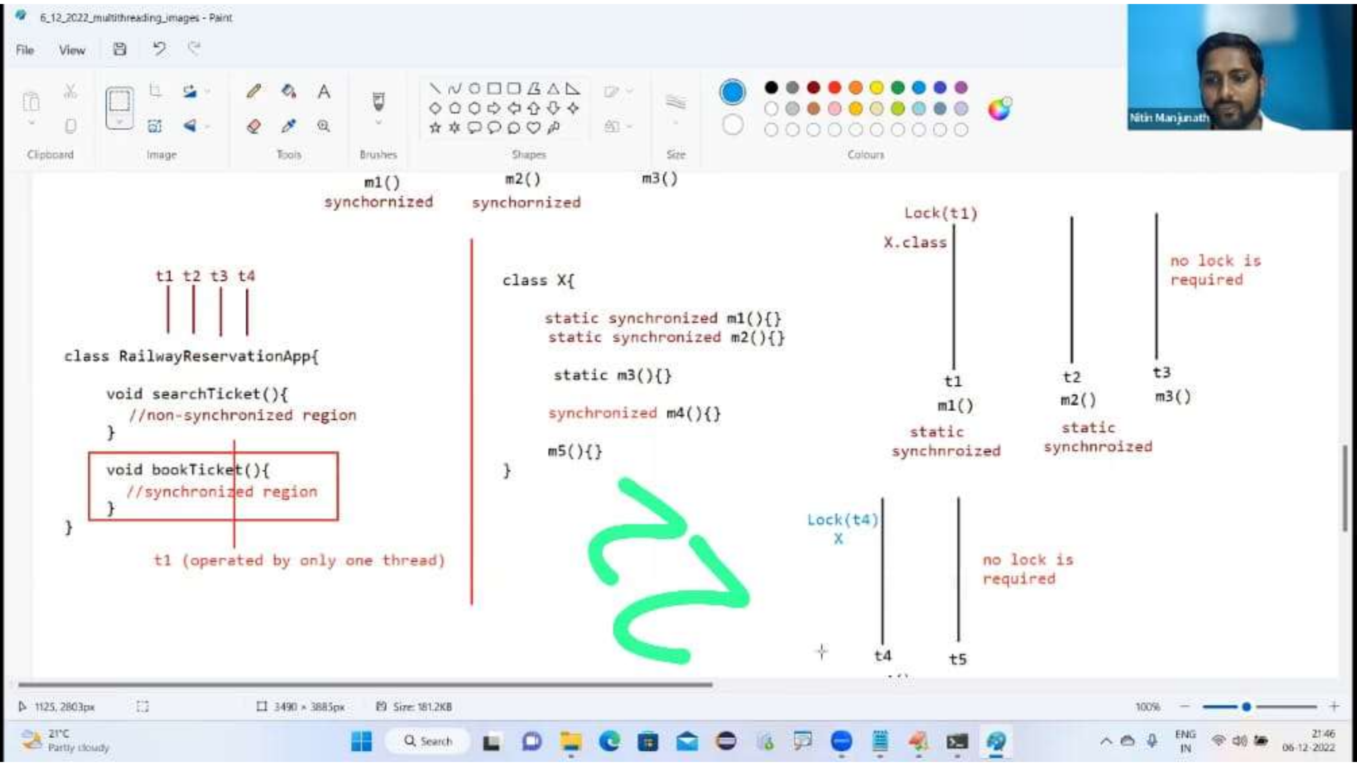
20



In the above case we get irregular output, because two different object and since the method
is synchronized lock is applied w.r.t object and both the threads will start simulataneously on different java objects
due to which the output is "irregular".

Conclusion :
If multiple threads are operating on multiple objects then there is no impact of Syncronization.
If multiple threads are operating on same java objects then syncronized concept is required(applicable).

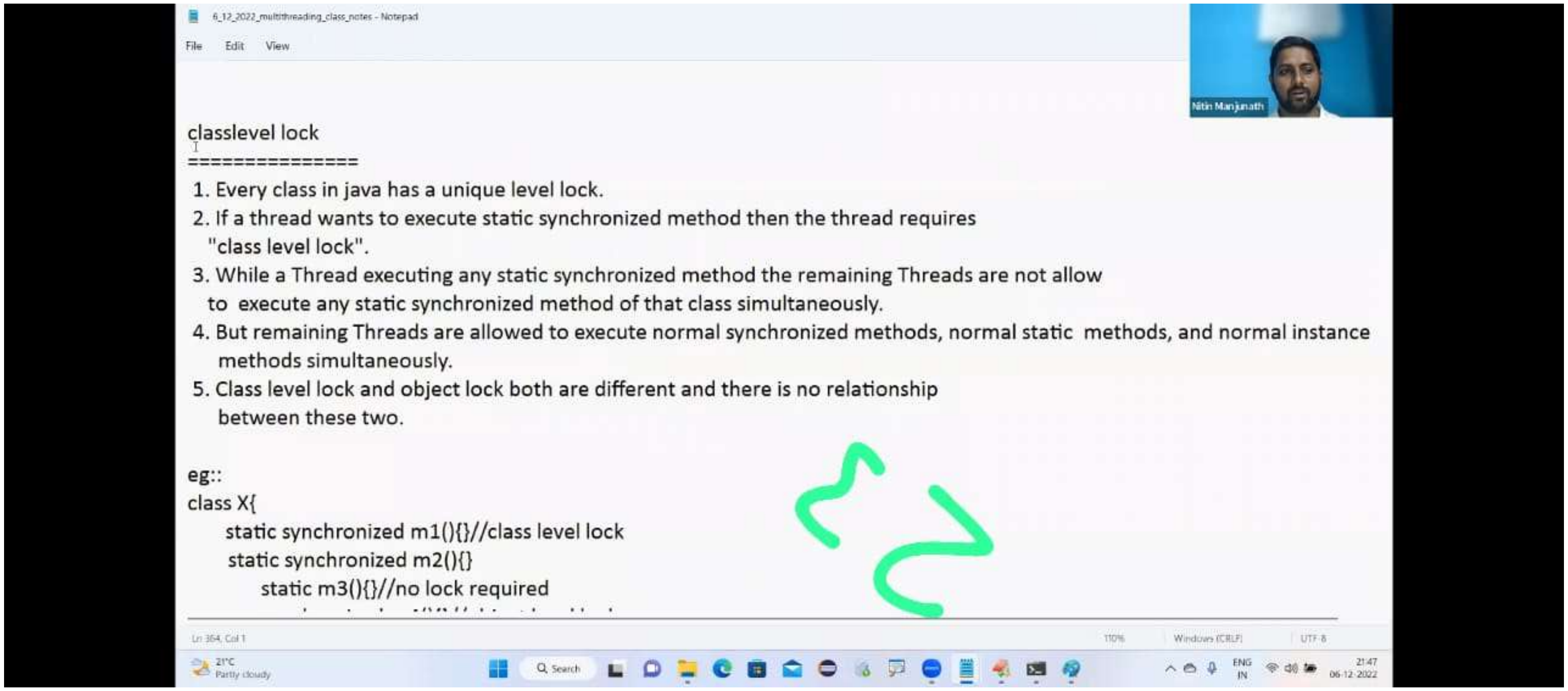
21



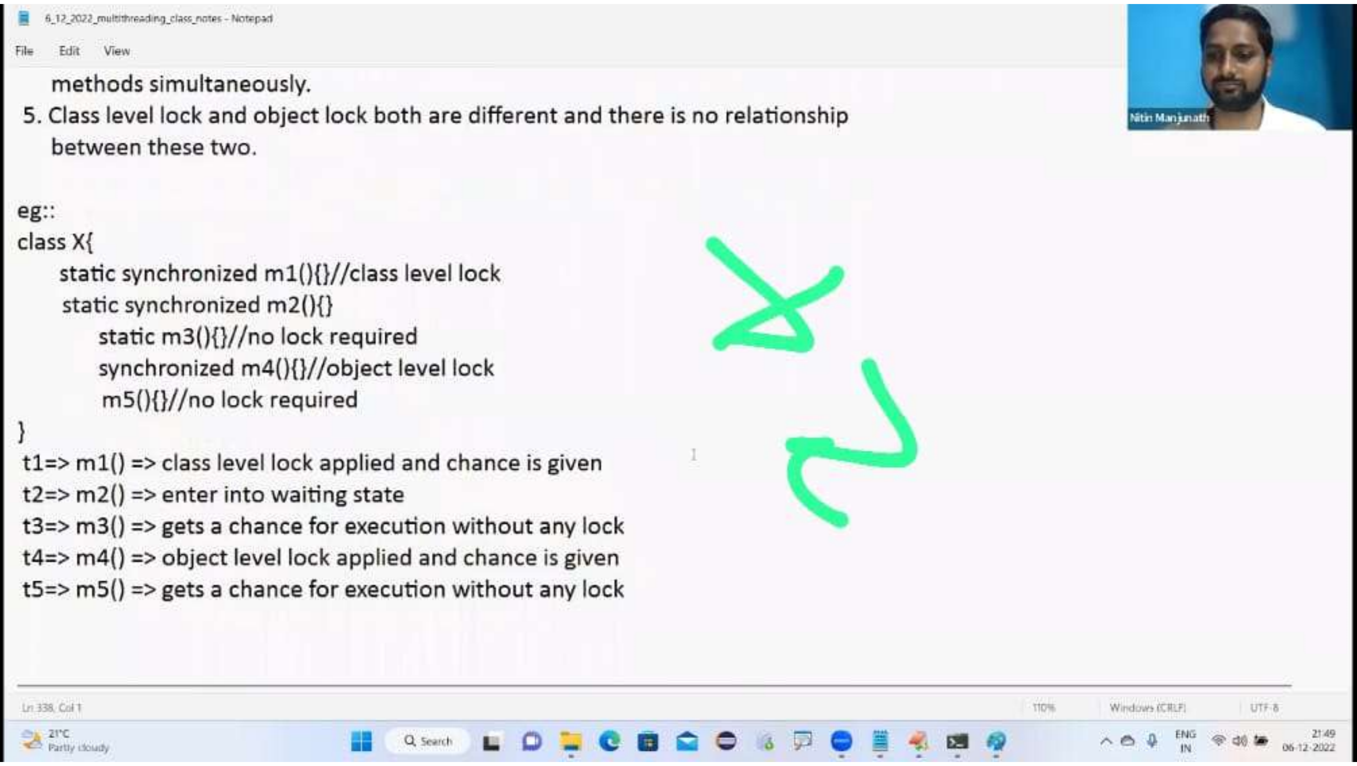
```
m1()              m2()              m3()
synchornized      synchornized

t1 t2 t3 t4

class RailwayReservationApp{

    void searchTicket(){
      //non-synchronized region
    }

    void bookTicket(){
      //synchronized region
    }
}
     t1 (operated by only one thread)
```

```
class X{

    static synchronized m1(){}
    static synchronized m2(){}

    static m3(){}

    synchronized m4(){}

    m5(){}
}
```

```
Lock(t1)
X.class                               no lock is
                                      required

      t1              t2          t3
      m1()            m2()        m3()
    static          static
  synchnroized    synchnroized

Lock(t4)
  X                 no lock is
                    required

     t4        t5
```

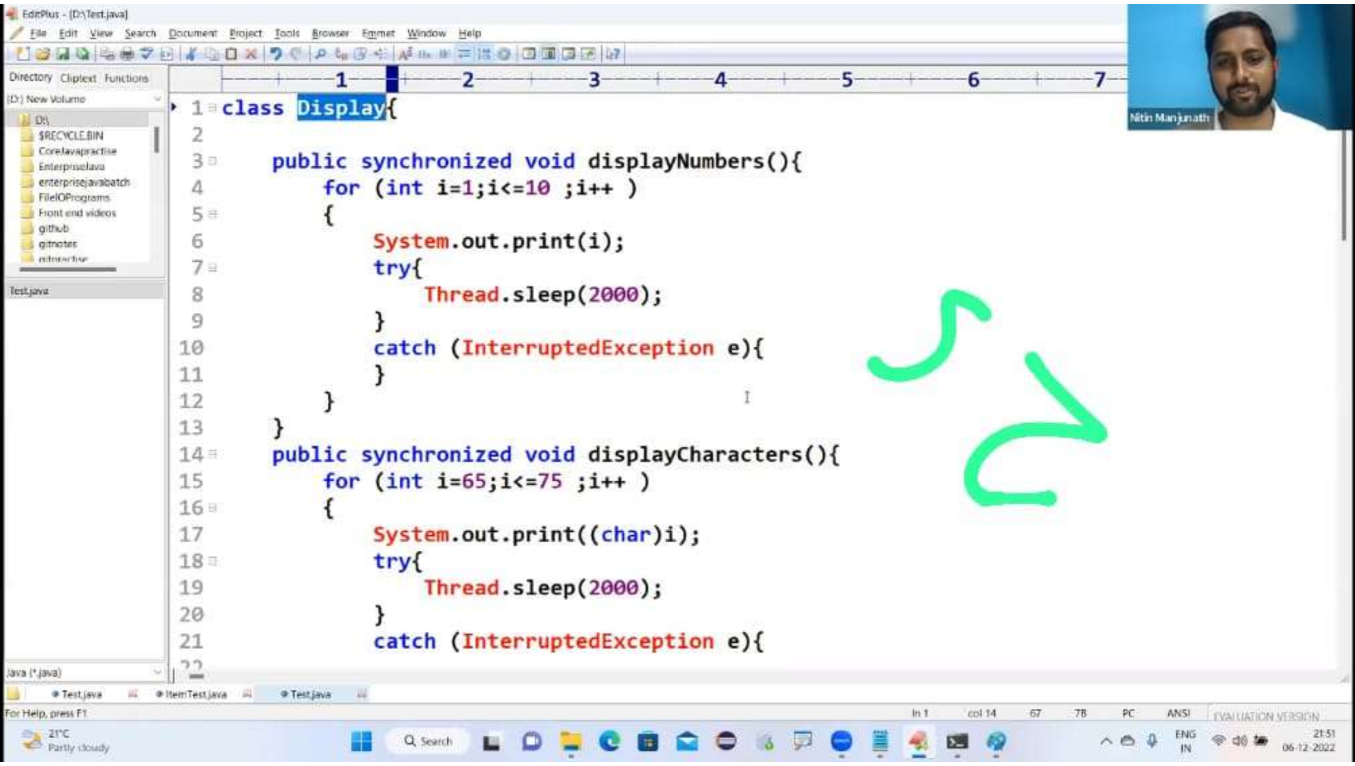


```
classlevel lock
===============
 1. Every class in java has a unique level lock.
 2. If a thread wants to execute static synchronized method then the thread requires
   "class level lock".
 3. While a Thread executing any static synchronized method the remaining Threads are not allow
   to  execute any static synchronized method of that class simultaneously.
 4. But remaining Threads are allowed to execute normal synchronized methods, normal static  methods, and normal instance
    methods simultaneously.
 5. Class level lock and object lock both are different and there is no relationship
    between these two.

eg::
class X{
     static synchronized m1(){}//class level lock
     static synchronized m2(){}
          static m3(){}//no lock required
```



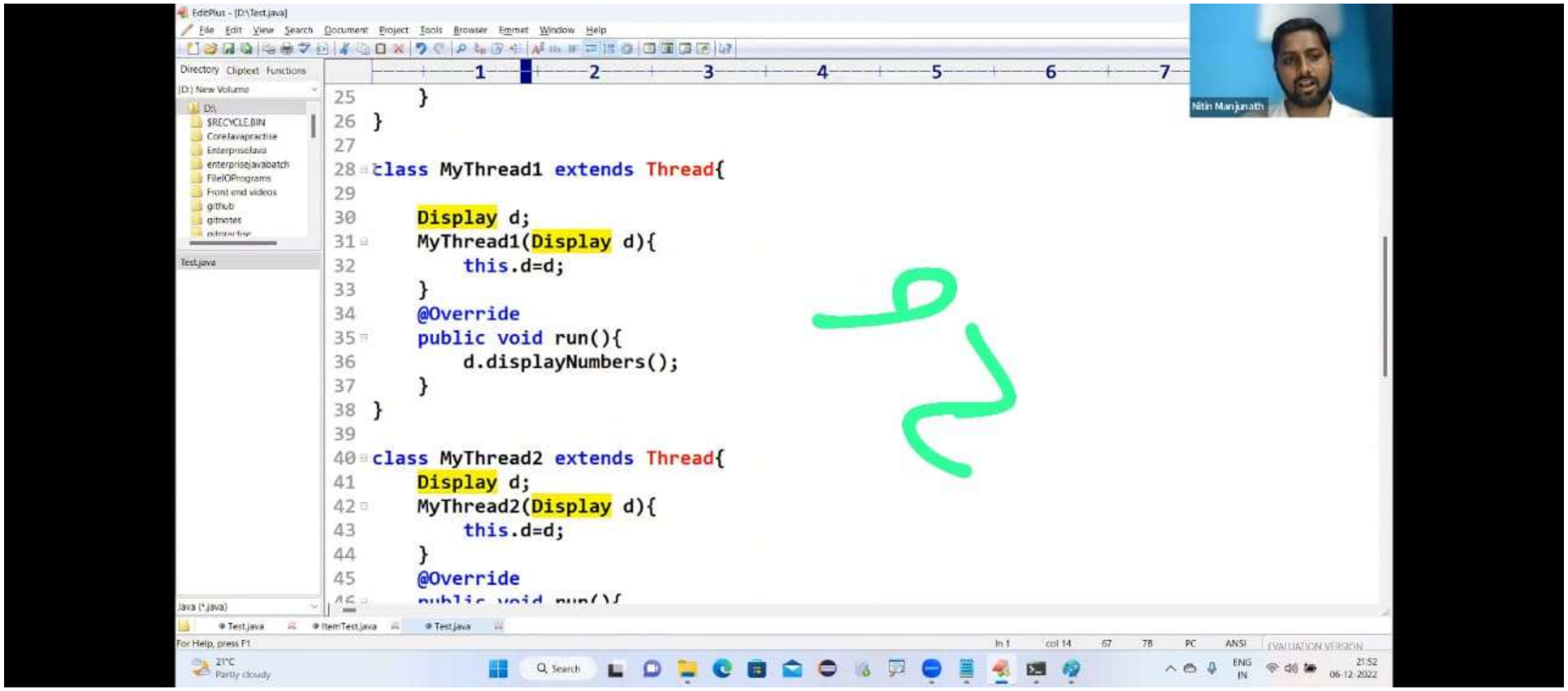
```
    methods simultaneously.
 5. Class level lock and object lock both are different and there is no relationship
    between these two.

eg::
class X{
     static synchronized m1(){}//class level lock
      static synchronized m2(){}
          static m3(){}//no lock required
          synchronized m4(){}//object level lock
           m5(){}//no lock required
}
 t1=> m1() => class level lock applied and chance is given
 t2=> m2() => enter into waiting state
 t3=> m3() => gets a chance for execution without any lock
 t4=> m4() => object level lock applied and chance is given
 t5=> m5() => gets a chance for execution without any lock
```

24

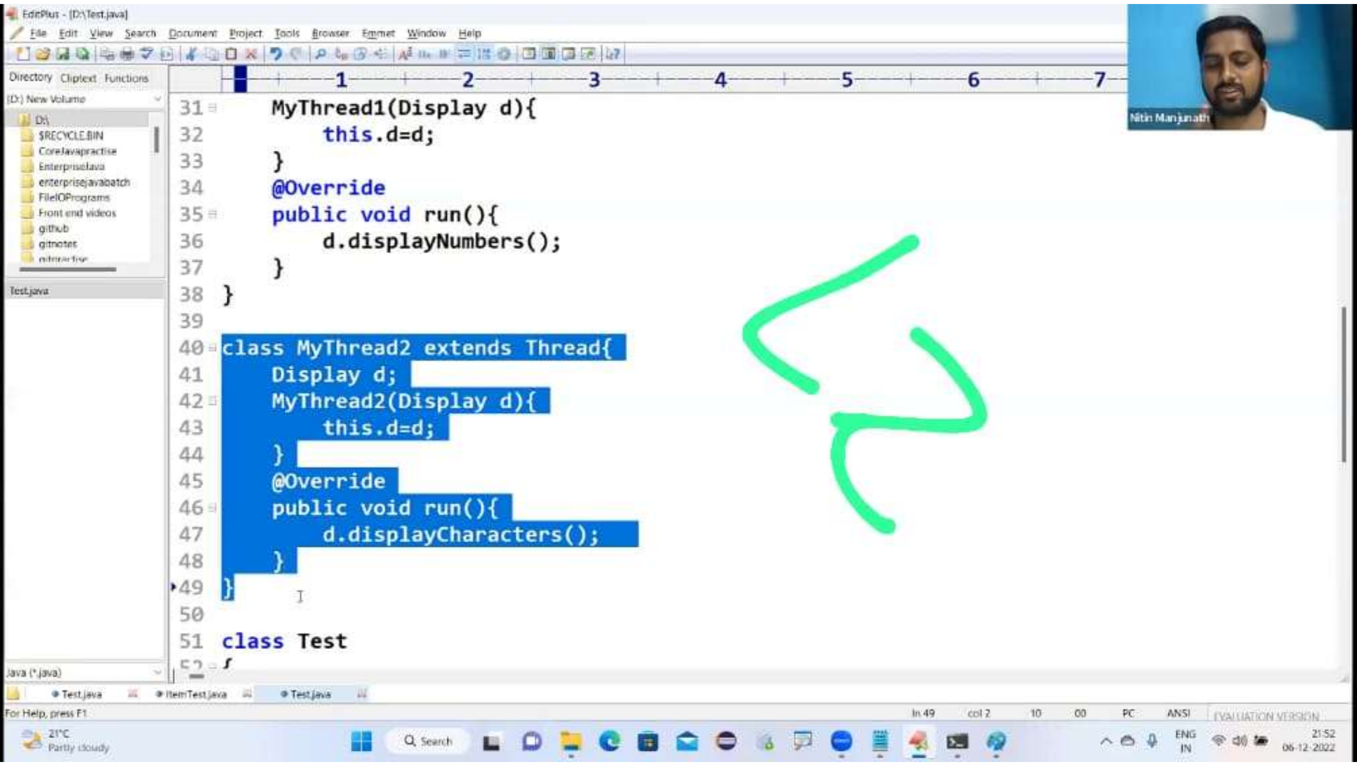


```
class Display{

    public synchronized void displayNumbers(){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print(i);
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
        }
    }
    public synchronized void displayCharacters(){
        for (int i=65;i<=75 ;i++ )
        {
            System.out.print((char)i);
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
```

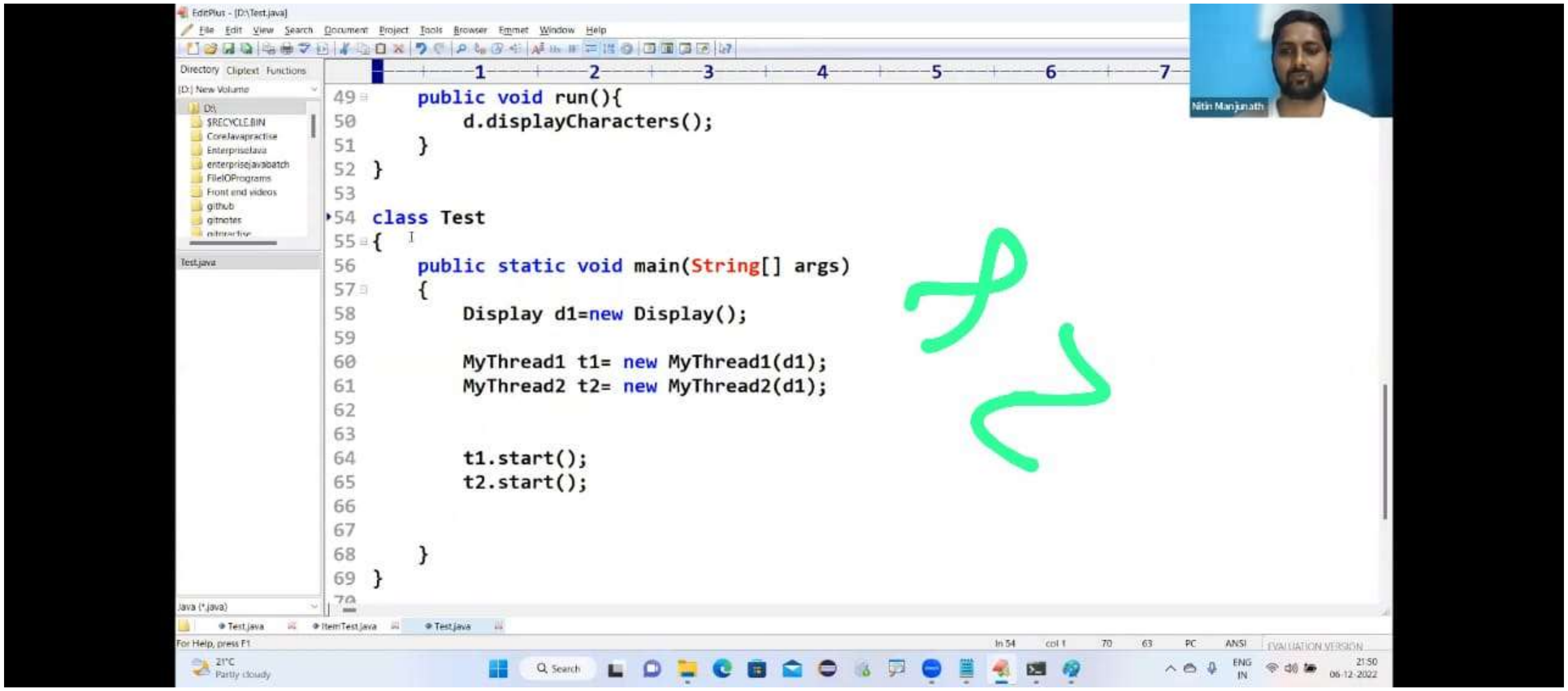
2s



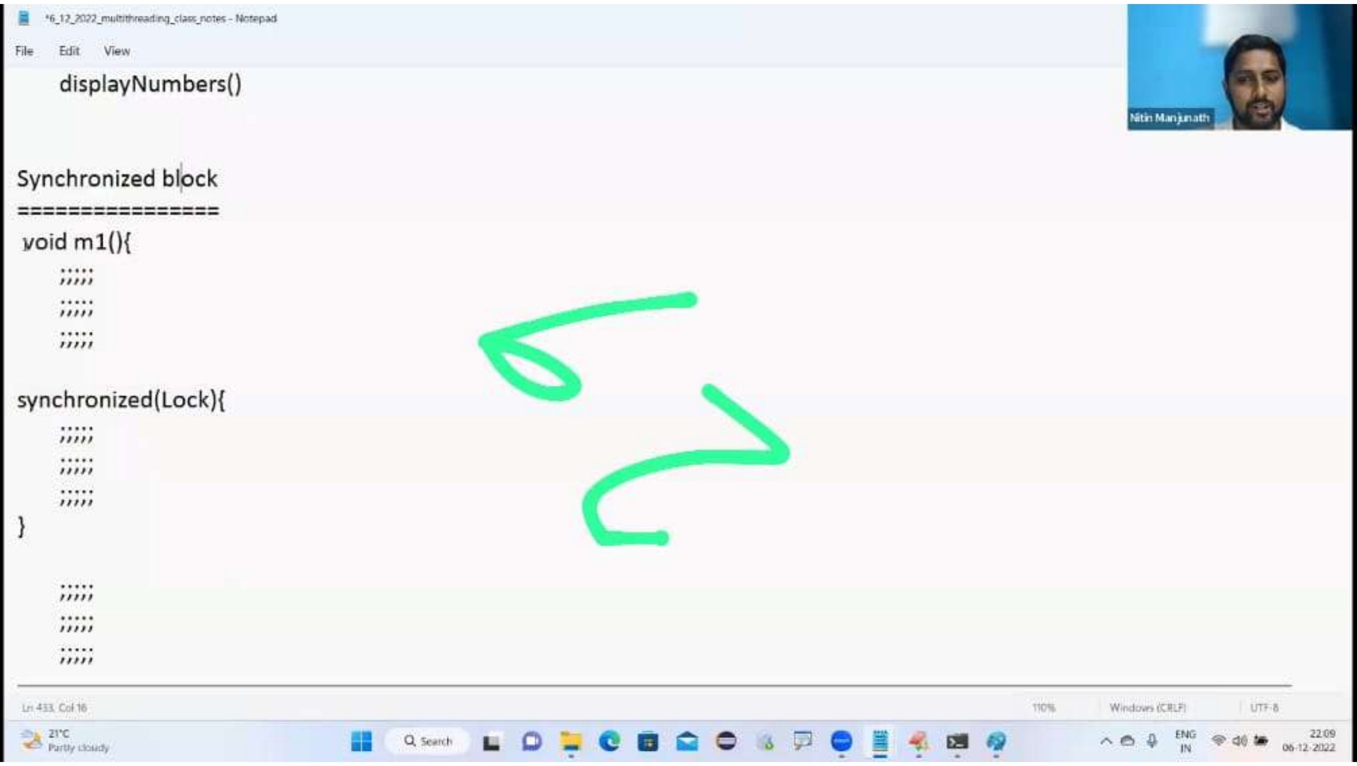
```
    }
}

class MyThread1 extends Thread{

    Display d;
    MyThread1(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayNumbers();
    }
}

class MyThread2 extends Thread{
    Display d;
    MyThread2(Display d){
        this.d=d;
    }
    @Override
    public void run(){
```



```java
    MyThread1(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayNumbers();
    }
}

class MyThread2 extends Thread{
    Display d;
    MyThread2(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayCharacters();
    }
}

class Test
{
```

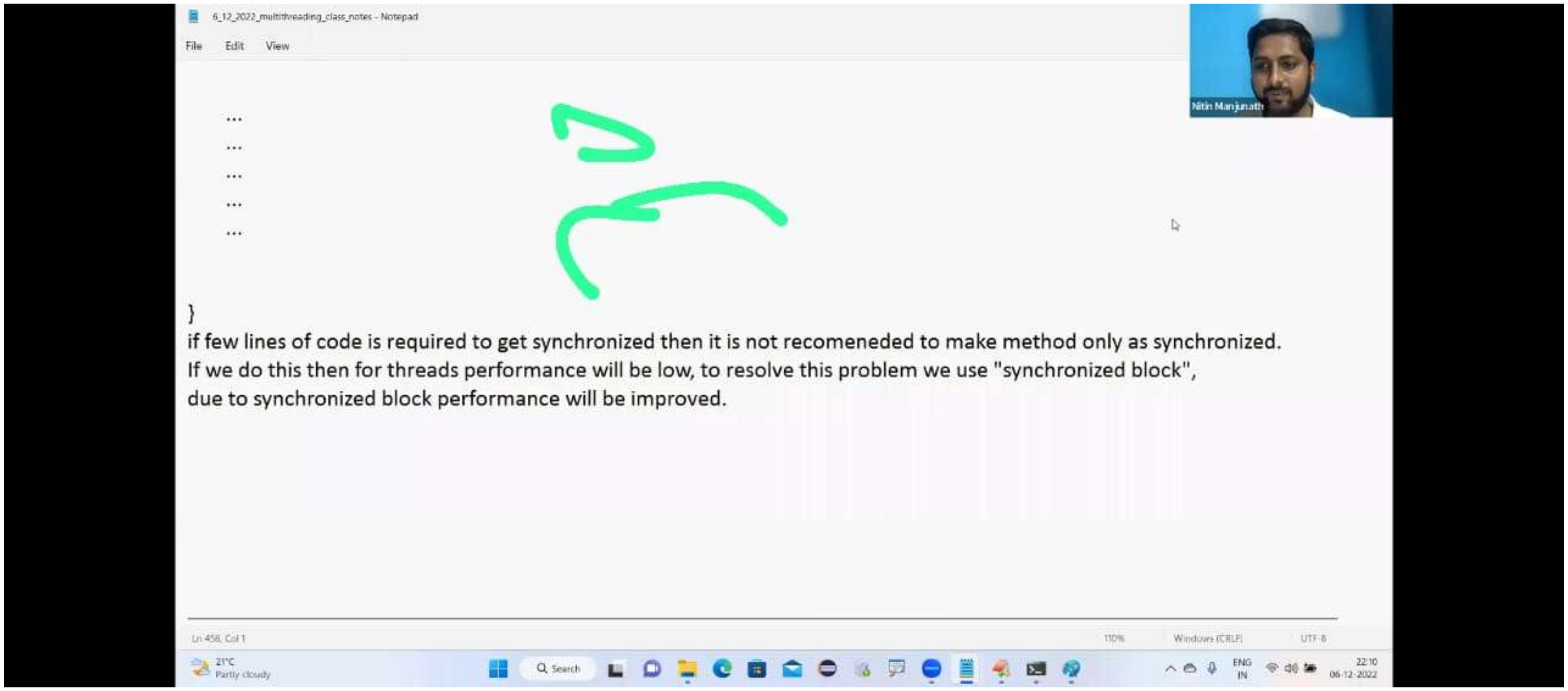


```
    public void run(){
        d.displayCharacters();
    }
}

class Test
{
    public static void main(String[] args)
    {
        Display d1=new Display();

        MyThread1 t1= new MyThread1(d1);
        MyThread2 t2= new MyThread2(d1);


        t1.start();
        t2.start();


    }
}
```



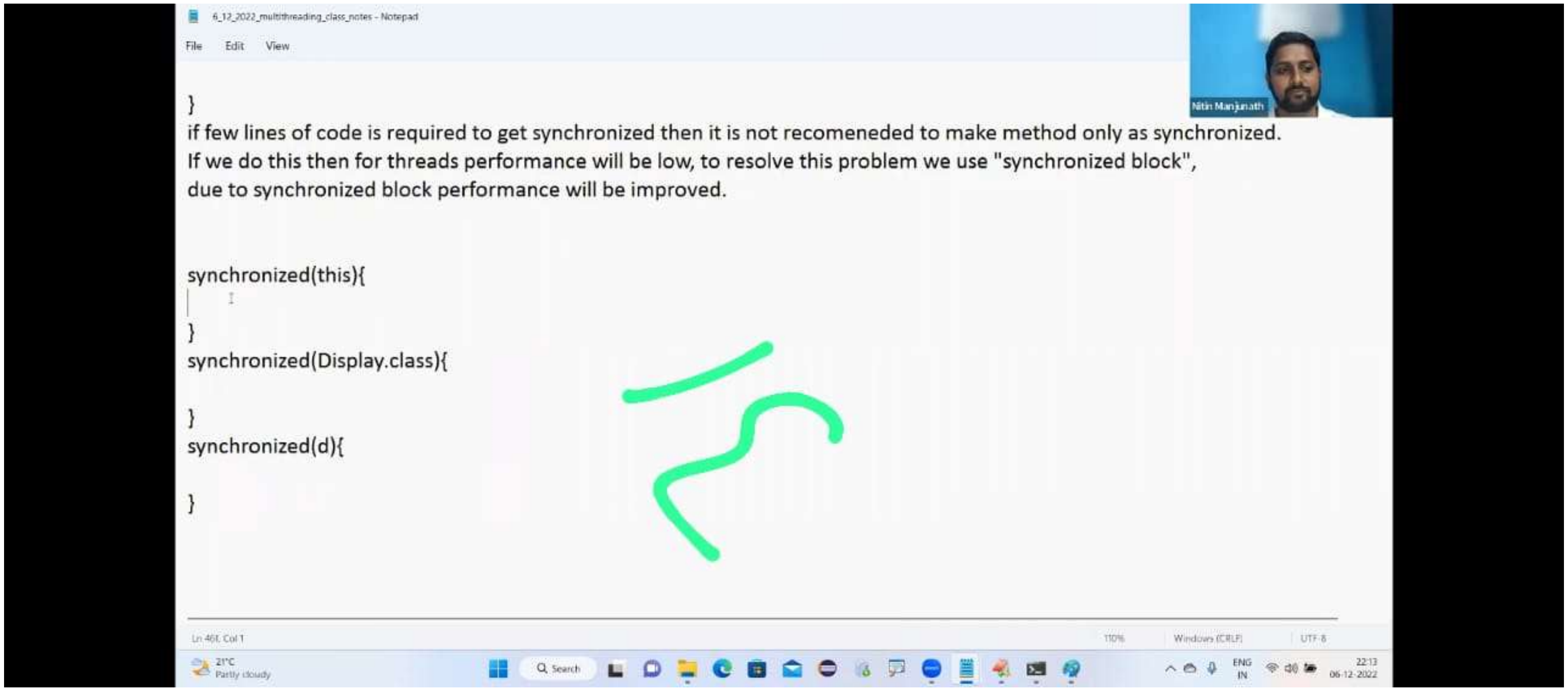
```
    displayNumbers()


Synchronized block
================
 void m1(){
    ;;;;;
    ;;;;;
    ;;;;;

synchronized(Lock){
    ;;;;;
    ;;;;;
    ;;;;;
}

    ;;;;;
    ;;;;;
    ;;;;;
```

29

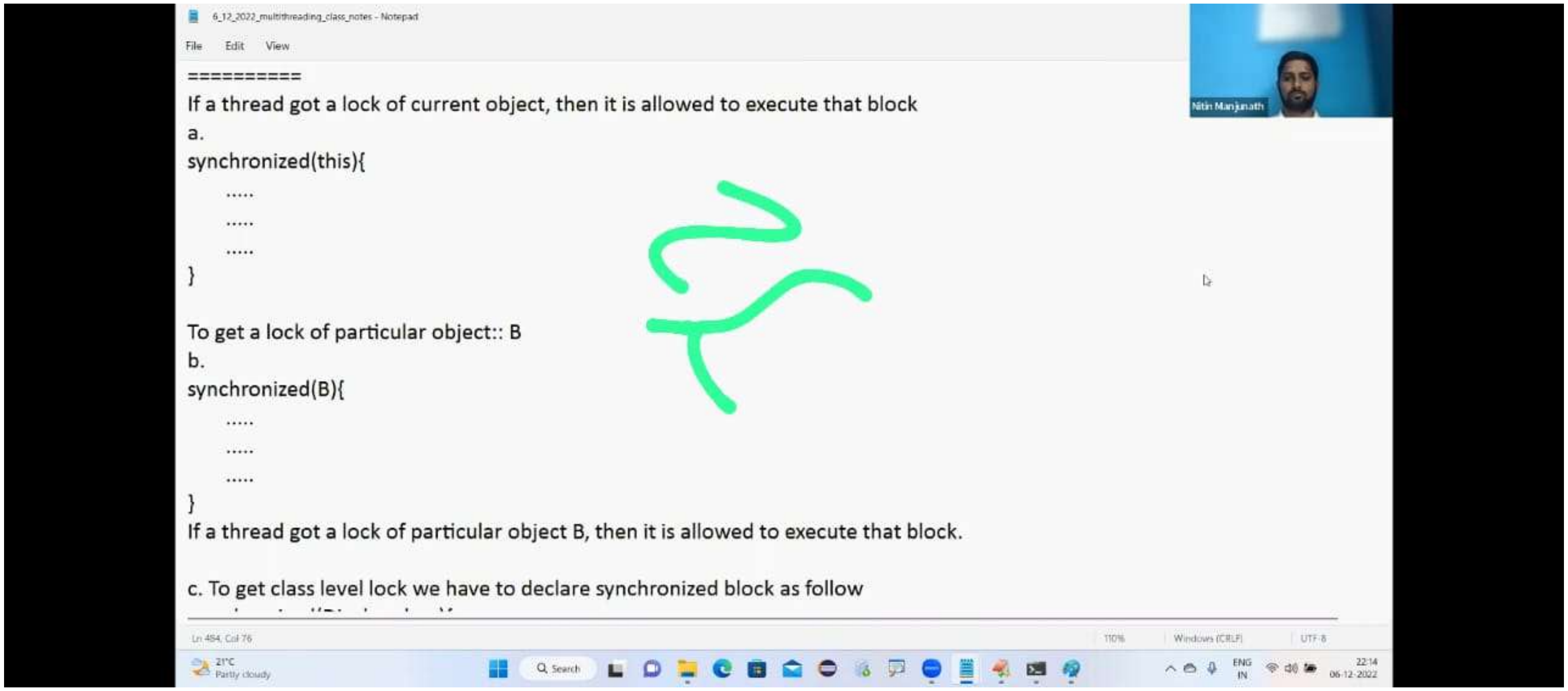


```
    ...
    ...
    ...
    ...
    ...

}
```

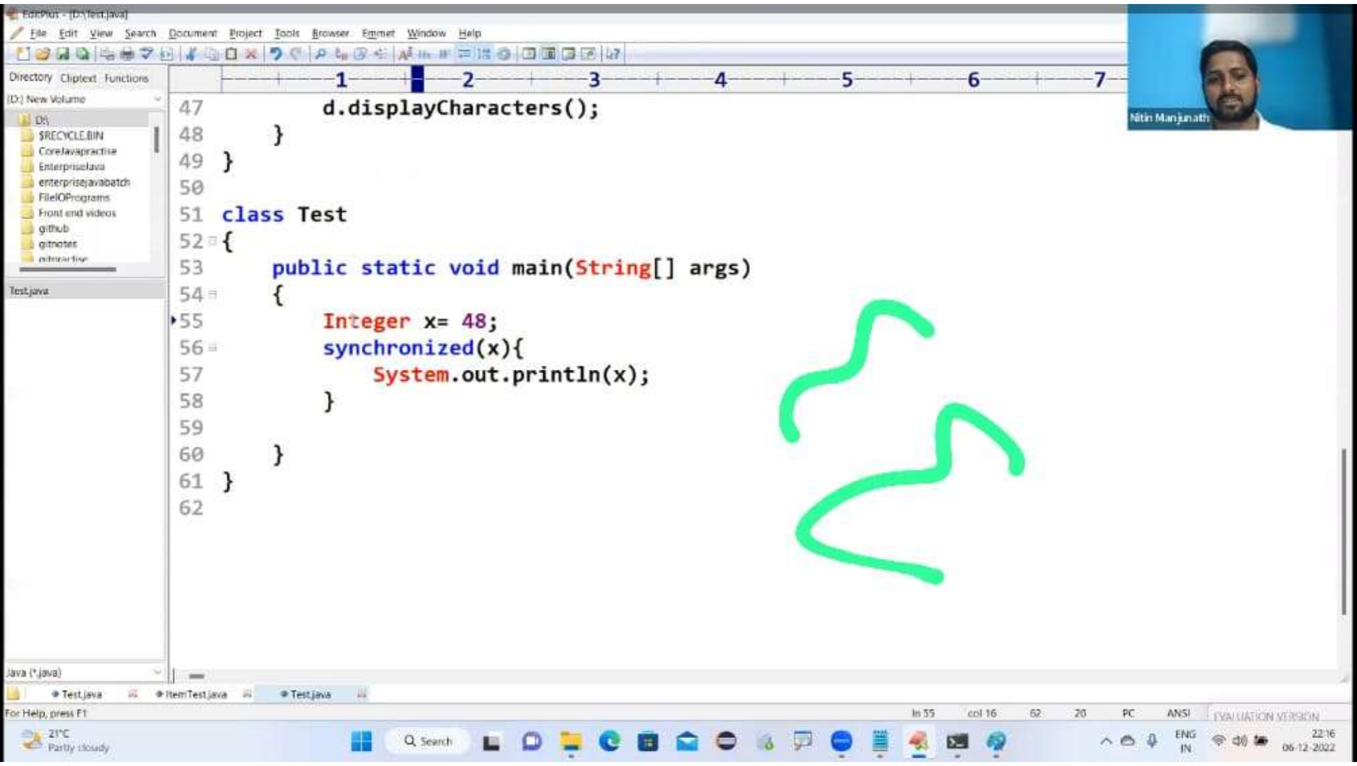
if few lines of code is required to get synchronized then it is not recomeneded to make method only as synchronized.
If we do this then for threads performance will be low, to resolve this problem we use "synchronized block",
due to synchronized block performance will be improved.



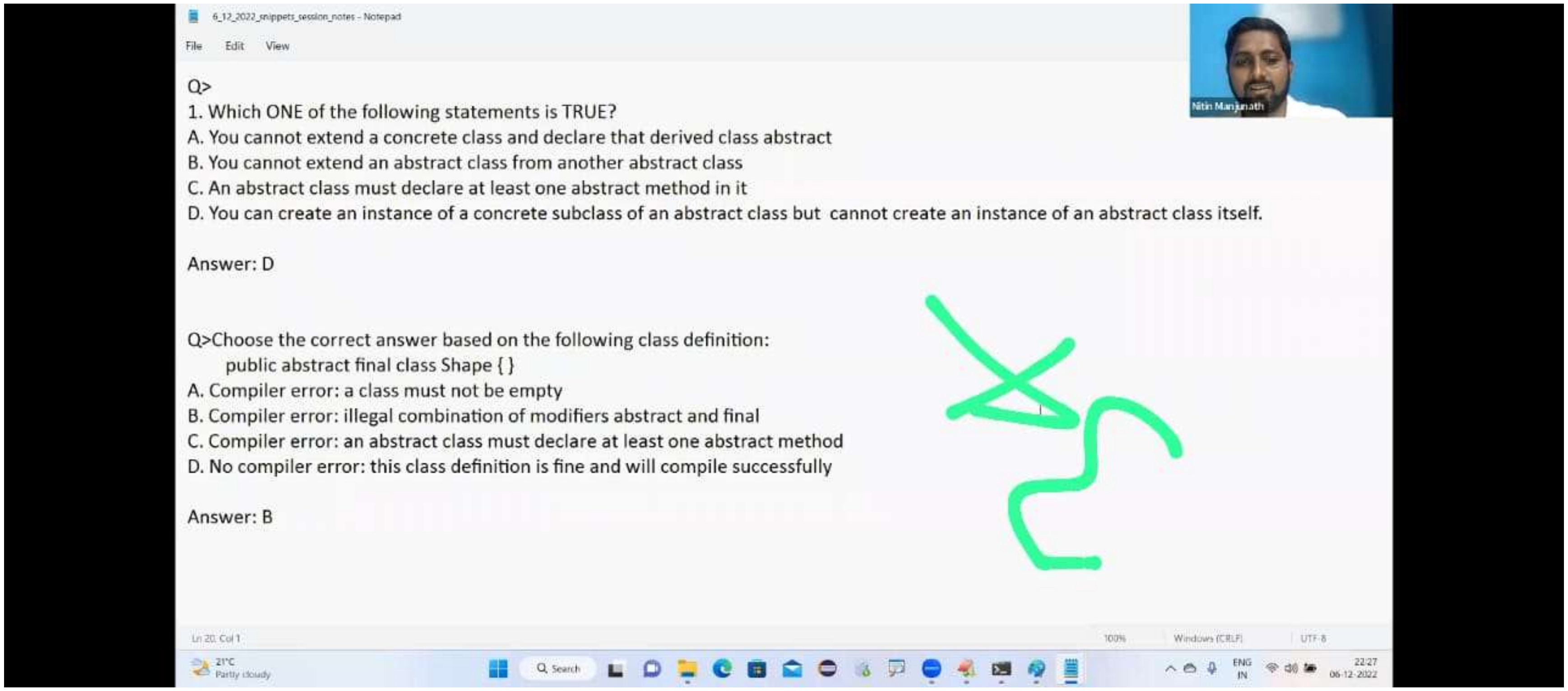
```
}
if few lines of code is required to get synchronized then it is not recomeneded to make method only as synchronized.
If we do this then for threads performance will be low, to resolve this problem we use "synchronized block",
due to synchronized block performance will be improved.


synchronized(this){

}
synchronized(Display.class){

}
synchronized(d){

}
```


6_12_2022_multithreading_class_notes - Notepad

```
==========
If a thread got a lock of current object, then it is allowed to execute that block
a.
synchronized(this){
	.....
	.....
	.....
}

To get a lock of particular object:: B
b.
synchronized(B){
	.....
	.....
	.....
}
If a thread got a lock of particular object B, then it is allowed to execute that block.

c. To get class level lock we have to declare synchronized block as follow
```



```java
        d.displayCharacters();
    }
}

class Test
{
    public static void main(String[] args)
    {
        Integer x= 48;
        synchronized(x){
            System.out.println(x);
        }

    }
}
```



6_12_2022_snippets_session_notes - Notepad

Q>
1. Which ONE of the following statements is TRUE?
A. You cannot extend a concrete class and declare that derived class abstract
B. You cannot extend an abstract class from another abstract class
C. An abstract class must declare at least one abstract method in it
D. You can create an instance of a concrete subclass of an abstract class but cannot create an instance of an abstract class itself.

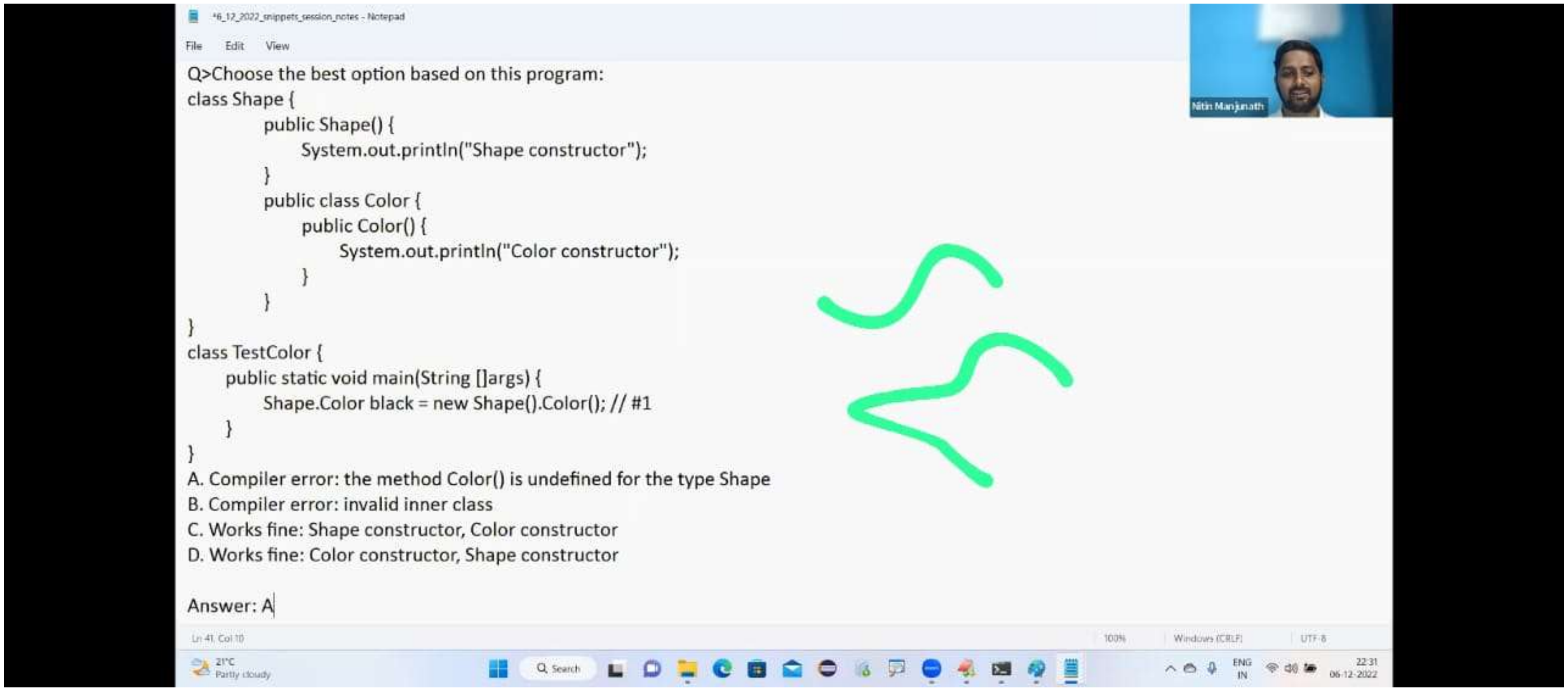
Answer: D

Q>Choose the correct answer based on the following class definition:

```
      public abstract final class Shape { }
```

A. Compiler error: a class must not be empty
B. Compiler error: illegal combination of modifiers abstract and final
C. Compiler error: an abstract class must declare at least one abstract method
D. No compiler error: this class definition is fine and will compile successfully

Answer: B

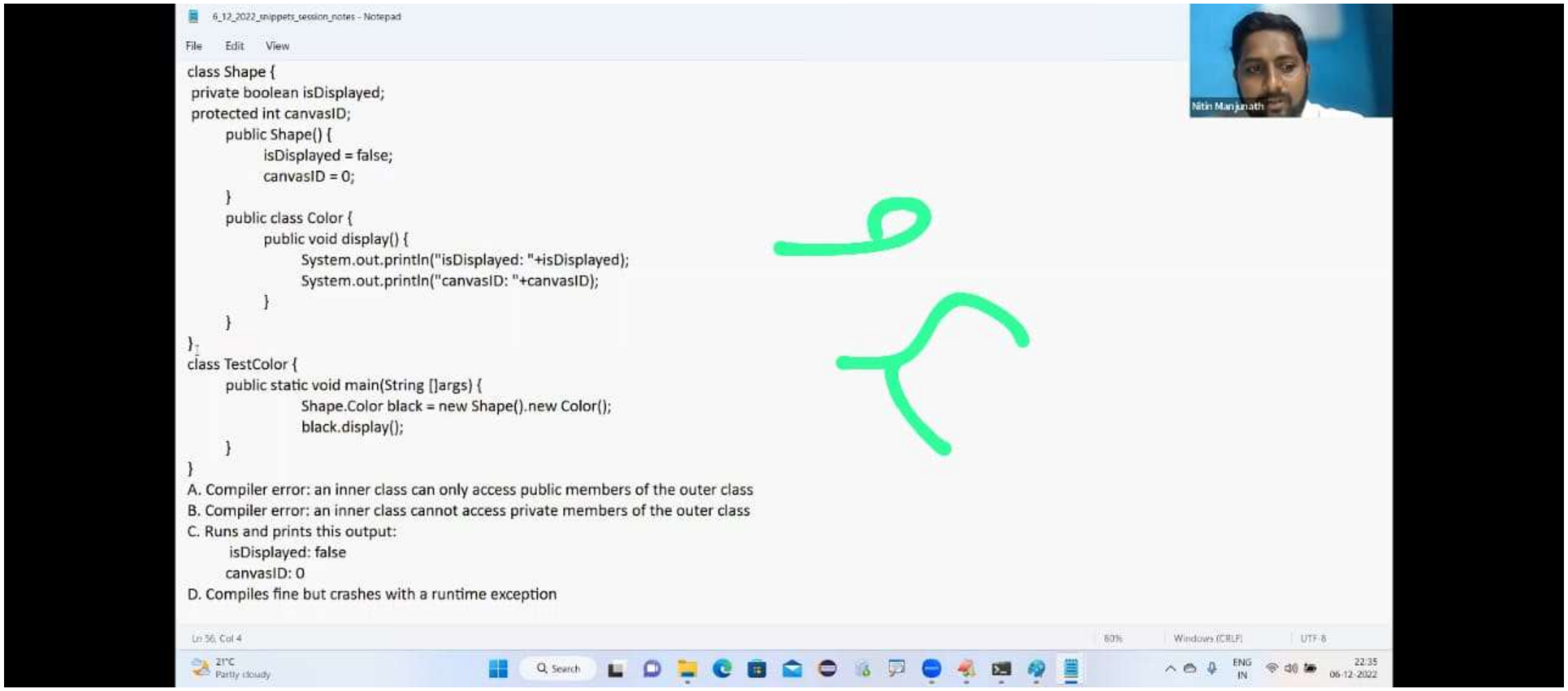


Q>Choose the best option based on this program:

```
class Shape {
            public Shape() {
                  System.out.println("Shape constructor");
            }
            public class Color {
                  public Color() {
                        System.out.println("Color constructor");
                  }
            }
}
class TestColor {
      public static void main(String []args) {
            Shape.Color black = new Shape().Color(); // #1
      }
}
```

A. Compiler error: the method Color() is undefined for the type Shape
B. Compiler error: invalid inner class
C. Works fine: Shape constructor, Color constructor
D. Works fine: Color constructor, Shape constructor

Answer: A

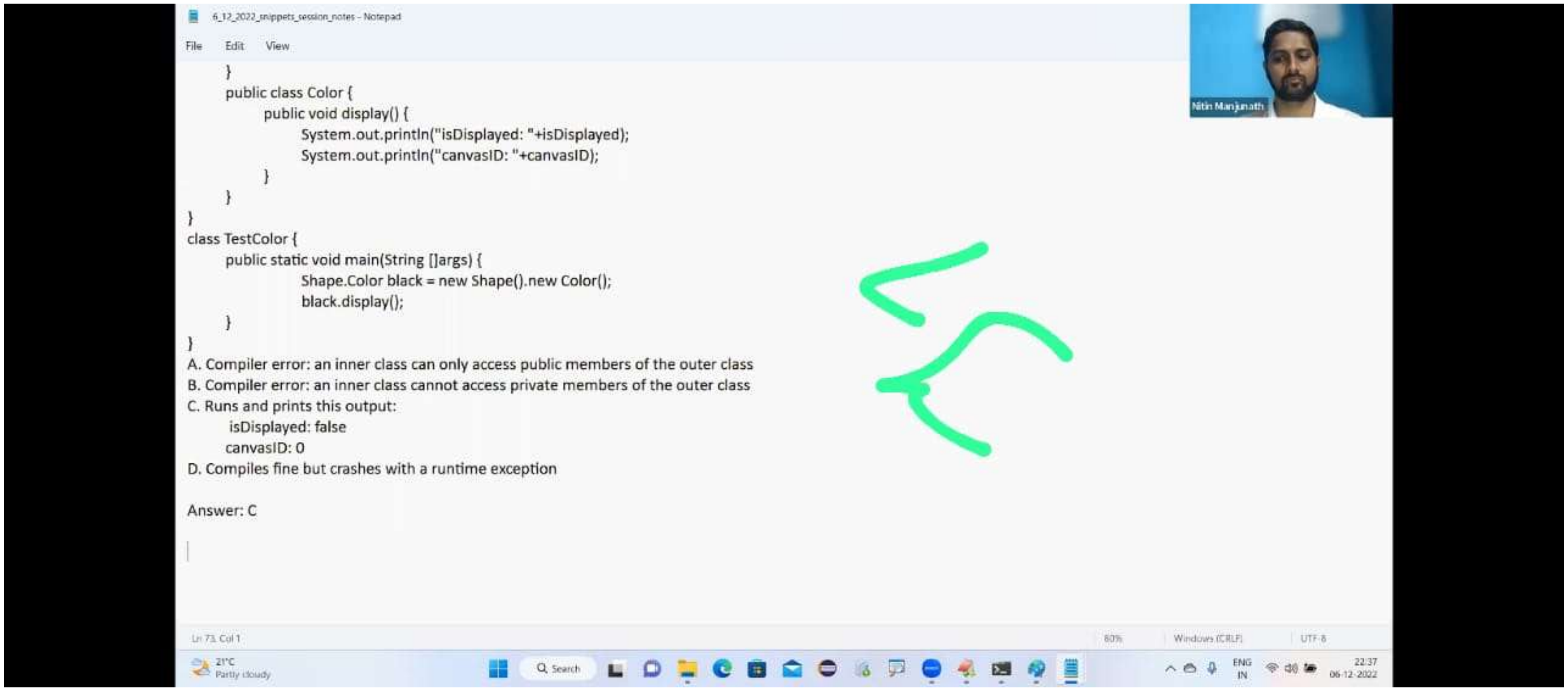


```
class Shape {
 private boolean isDisplayed;
 protected int canvasID;
	public Shape() {
		isDisplayed = false;
		canvasID = 0;
	}
	public class Color {
		public void display() {
			System.out.println("isDisplayed: "+isDisplayed);
			System.out.println("canvasID: "+canvasID);
		}
	}
}
class TestColor {
	public static void main(String []args) {
			Shape.Color black = new Shape().new Color();
			black.display();
	}
}
```

A. Compiler error: an inner class can only access public members of the outer class
B. Compiler error: an inner class cannot access private members of the outer class
C. Runs and prints this output:

```
	 isDisplayed: false
	canvasID: 0
```

D. Compiles fine but crashes with a runtime exception



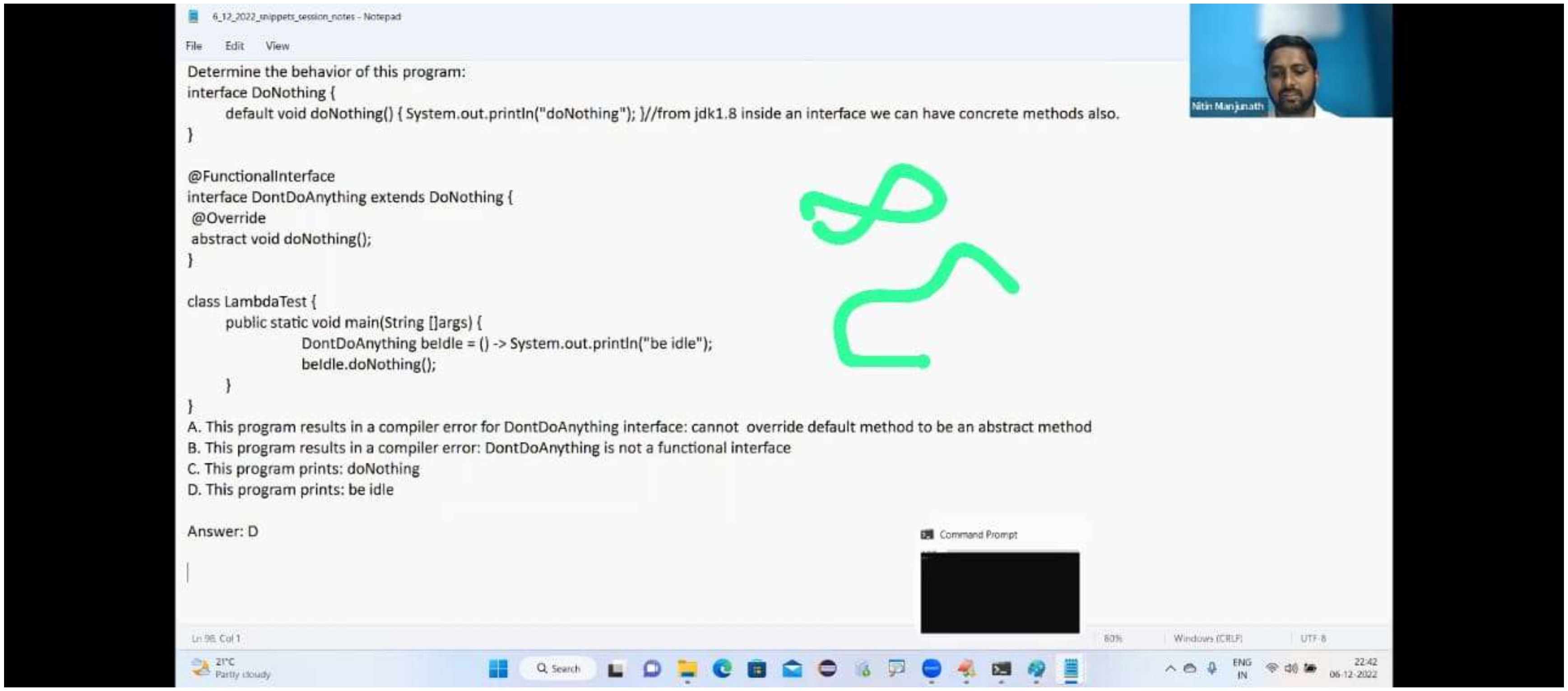
```
    }
    public class Color {
        public void display() {
            System.out.println("isDisplayed: "+isDisplayed);
            System.out.println("canvasID: "+canvasID);
        }
    }
}
class TestColor {
    public static void main(String []args) {
            Shape.Color black = new Shape().new Color();
            black.display();
    }
}
```

A. Compiler error: an inner class can only access public members of the outer class
B. Compiler error: an inner class cannot access private members of the outer class
C. Runs and prints this output:

```
 isDisplayed: false
canvasID: 0
```

D. Compiles fine but crashes with a runtime exception

Answer: C

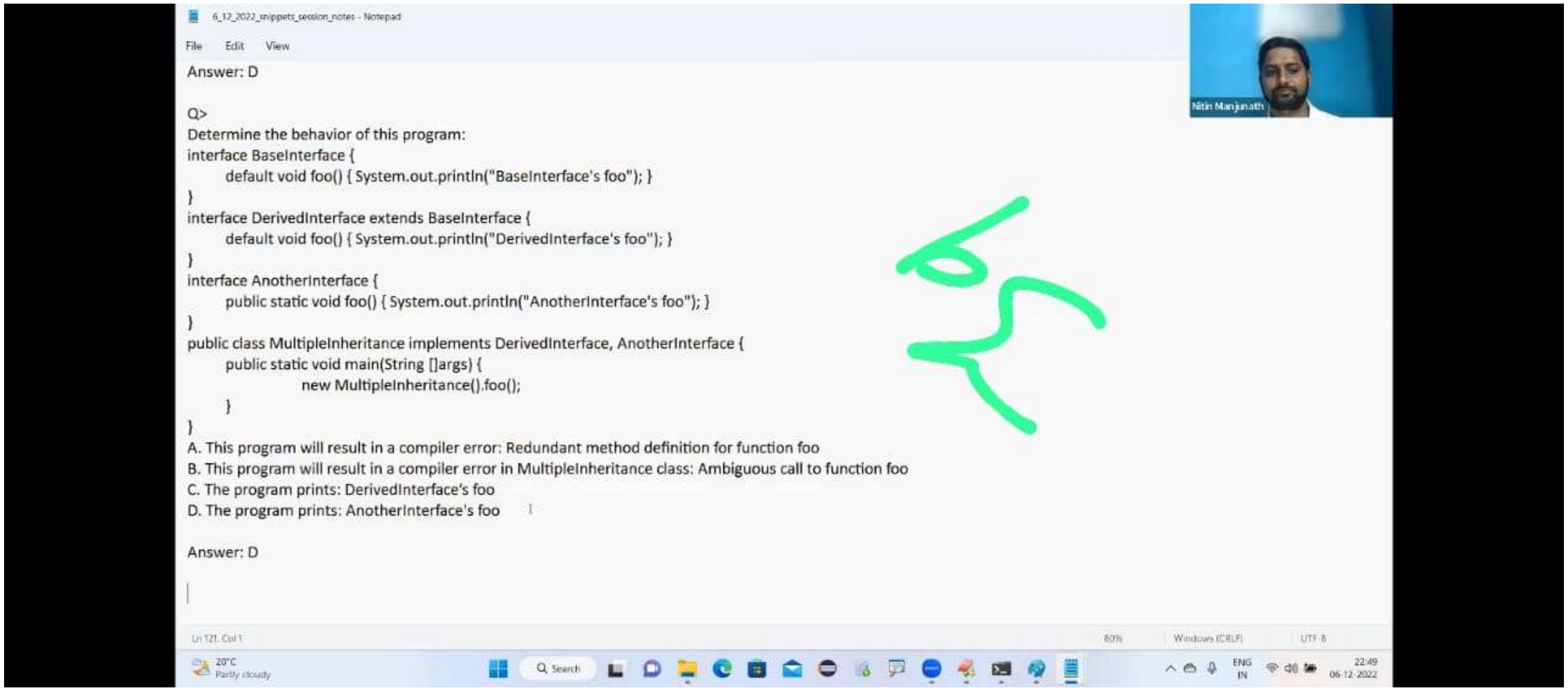


Determine the behavior of this program:

```
interface DoNothing {
	default void doNothing() { System.out.println("doNothing"); }//from jdk1.8 inside an interface we can have concrete methods also.
}

@FunctionalInterface
interface DontDoAnything extends DoNothing {
 @Override
 abstract void doNothing();
}

class LambdaTest {
	public static void main(String []args) {
			DontDoAnything beIdle = () -> System.out.println("be idle");
			beIdle.doNothing();
	}
}
```

A. This program results in a compiler error for DontDoAnything interface: cannot override default method to be an abstract method
B. This program results in a compiler error: DontDoAnything is not a functional interface
C. This program prints: doNothing
D. This program prints: be idle

Answer: D


Answer: D

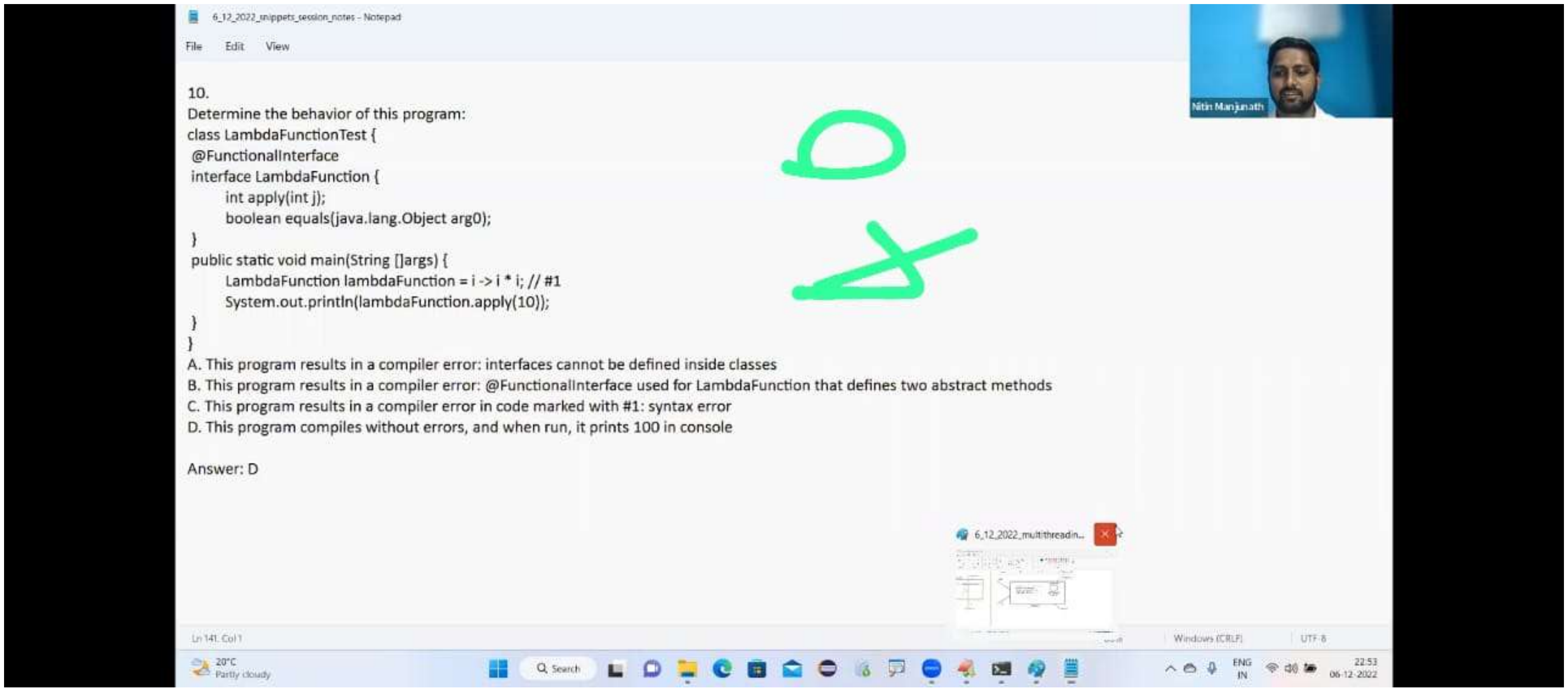
Q>
Determine the behavior of this program:

```
interface BaseInterface {
	default void foo() { System.out.println("BaseInterface's foo"); }
}
interface DerivedInterface extends BaseInterface {
	default void foo() { System.out.println("DerivedInterface's foo"); }
}
interface AnotherInterface {
	public static void foo() { System.out.println("AnotherInterface's foo"); }
}
public class MultipleInheritance implements DerivedInterface, AnotherInterface {
	public static void main(String []args) {
			new MultipleInheritance().foo();
	}
}
```

A. This program will result in a compiler error: Redundant method definition for function foo
B. This program will result in a compiler error in MultipleInheritance class: Ambiguous call to function foo
C. The program prints: DerivedInterface's foo
D. The program prints: AnotherInterface's foo

Answer: D

39



6_12_2022_snippets_session_notes - Notepad

10.
Determine the behavior of this program:

```
class LambdaFunctionTest {
 @FunctionalInterface
 interface LambdaFunction {
       int apply(int j);
       boolean equals(java.lang.Object arg0);
 }
 public static void main(String []args) {
       LambdaFunction lambdaFunction = i -> i * i; // #1
       System.out.println(lambdaFunction.apply(10));
 }
}
```

A. This program results in a compiler error: interfaces cannot be defined inside classes
B. This program results in a compiler error: @FunctionalInterface used for LambdaFunction that defines two abstract methods
C. This program results in a compiler error in code marked with #1: syntax error
D. This program compiles without errors, and when run, it prints 100 in console

Answer: D

40