SDU

Lecture 08: Deadlocks & Thread Communication

The slides from this lecture are taken from
https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-c-sharp/
https://dotnettutorials.net/lesson/inter-thread-communication-in-csharp/

Agenda

1. The Thread Lifecycle
2. Synchronization of Threads (Recap)
3. Deadlock
4. Deadlock Prevention using Monitor
5. Case Study: *Producer/ Consumer*
6. Mid Evaluation

Objectives

1. The Thread lifecycle
    a. *Understand the different states of a Thread*
2. Synchronization of Threads
    a. *Understand the pitfalls of synchronization*
3. Deadlock Prevention
    a. *Utilize the Monitor to avoid deadlock*
4. Case Study: *Producer/ Consumer*
5. Discuss Mid Evaluation Feedback ☺

Test Statistics ☹

Lecture 1

Students not submitted:	**58** (of 182)	Average result score:	**13,85** (69,23%)
Maximum test score:	**20**	Highest result score:	**19** (95%)
Manual assessment questions:	**0**	Lowest result score:	**7** (35%)

Lecture 2

Students not submitted:	**118** (of 182)	Average result score:	**10,59** (81,49%)
Maximum test score:	**13**	Highest result score:	**13** (100%)
Manual assessment questions:	**0**	Lowest result score:	**5** (38,46%)

Lecture 3

Students not submitted:	**126** (of 182)	Average result score:	**6,54** (81,7%)
Maximum test score:	**8**	Highest result score:	**8** (100%)
Manual assessment questions:	**0**	Lowest result score:	**4** (50%)

Lecture 4

Students not submitted:	**150** (of 182)	Average result score:	**8,91** (89,06%)
Maximum test score:	**10**	Highest result score:	**10** (100%)
Manual assessment questions:	**0**	Lowest result score:	**7** (70%)

Lecture 4

Students not submitted:	**162** (of 182)	Average result score:	**7,35** (81,67%)
Maximum test score:	**9**	Highest result score:	**9** (100%)
Manual assessment questions:	**0**	Lowest result score:	**3** (33,33%)

Lecture 6

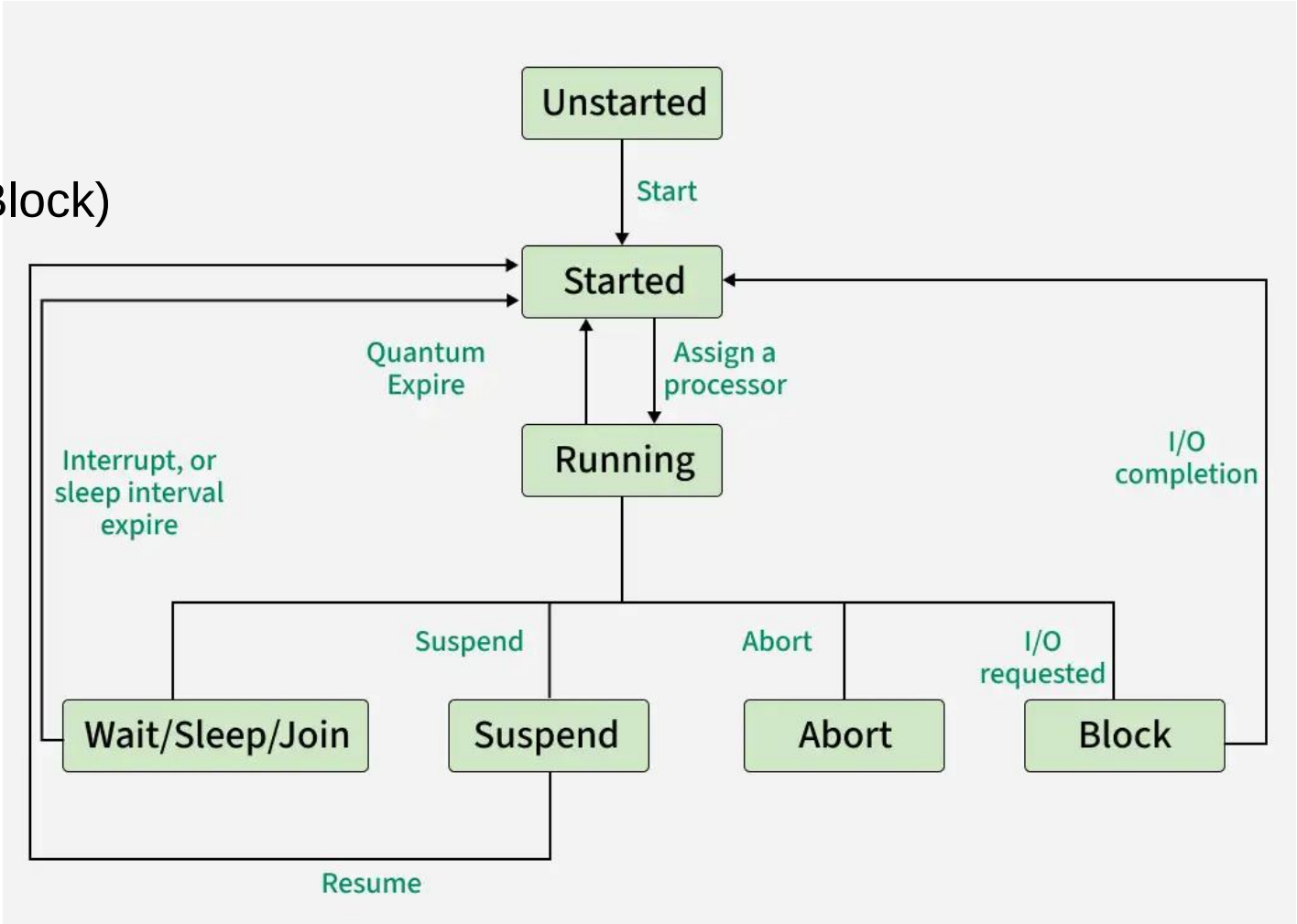
Students not submitted:	**154** (of 182)	Average result score:	**4,64** (77,38%)
Maximum test score:	**6**	Highest result score:	**6** (100%)
Manual assessment questions:	**0**	Lowest result score:	**2** (33,33%)

Lecture 7

Students not submitted:	**168** (of 182)	Average result score:	**11,57** (89,01%)
Maximum test score:	**13**	Highest result score:	**13** (100%)
Manual assessment questions:	**0**	Lowest result score:	**8** (61,54%)

Thread Lifecycle

- A thread can be in one of these 5 states:
- Unstarted
- Runnable (Started)
- Running
- Not Runnable (Wait/Sleep/Join/Suspend/Block)
- Dead (Terminated/Abort)



https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-c-sharp/

Do you recall synchronization mechanism in C#???

How

SDU

Threads Synchronization

a. *Using Synchronized attribute*

b. *Using Lock statement*

c. *Using Monitor class*

```
[MethodImpl(MethodImplOptions.Synchronized)]
public void deposit(int amount)
{
    int newBalance = balance + amount;
    balance = newBalance;
}
```

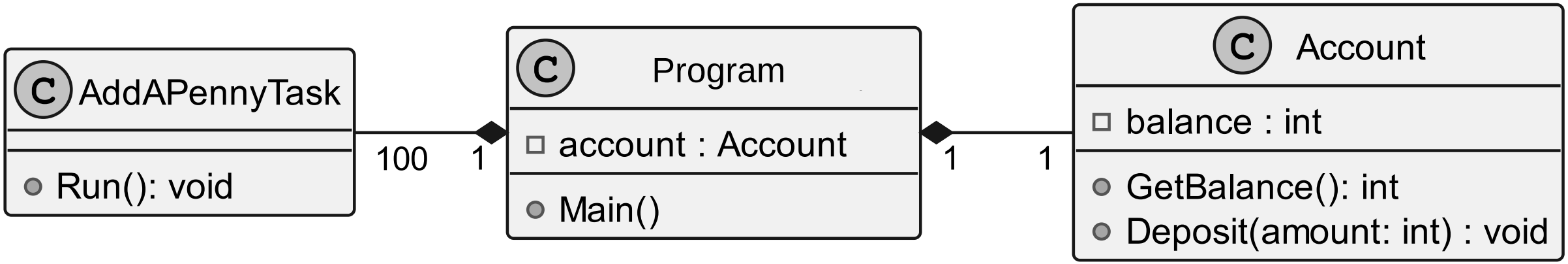
```
private Object myLock = new Object();

public void deposit(int amount)
{
    try
    {
        Monitor.Enter(myLock);
        int newBalance = balance + amount;
        balance = newBalance;
    }
    finally
    {
        Monitor.Exit(myLock);
    }
}
```

```
private Object myLock = new Object();

public void deposit(int amount)
{
    lock (myLock)
    {
        int newBalance = balance + amount;
        balance = newBalance;
    }
}
```

Case Study: Bank Account

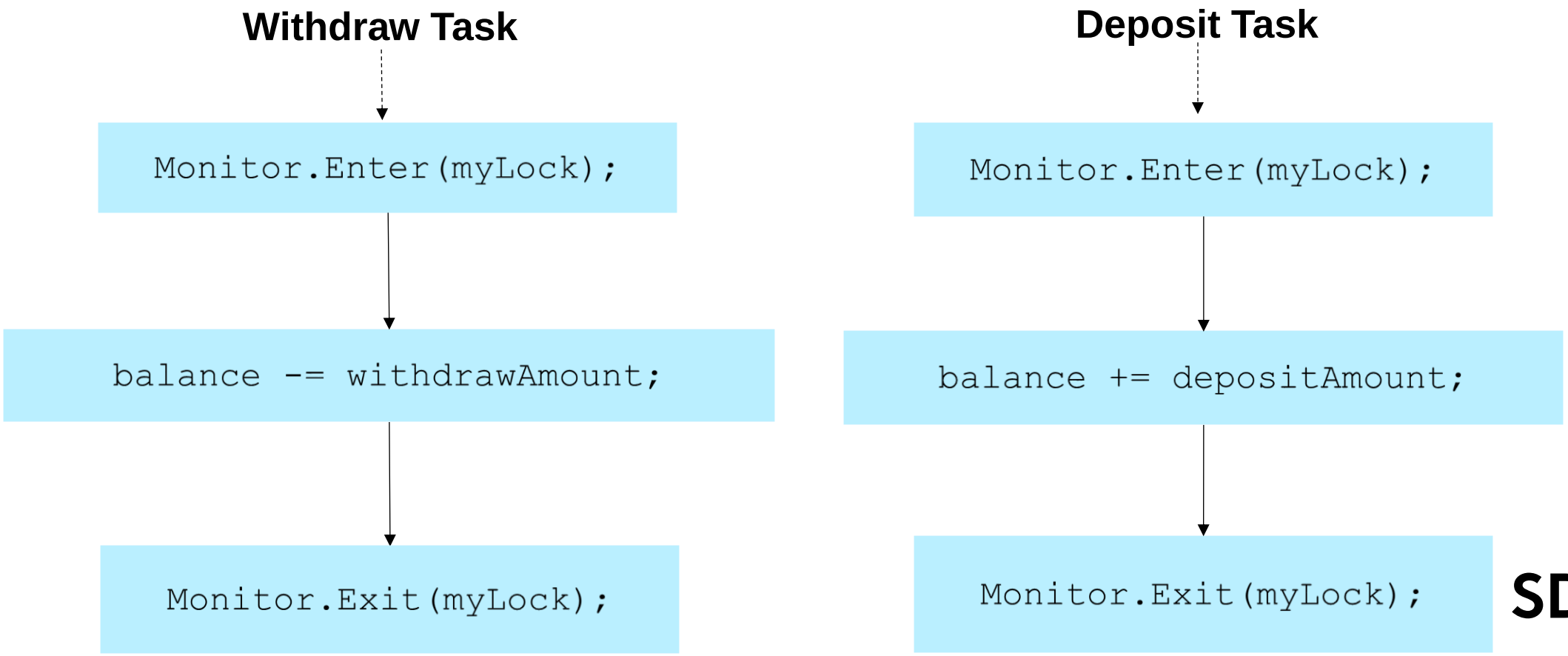


SDU

Extended Case Study: Bank Account with Monitor

- In this case study, we extend the account object with a `Withdraw(int amount)` method.
- We define 100 deposit and withdraw tasks to add and subtract from the account balance respectively.

```
Object myLock = new Object();
```



Without Deadlock

Plans-> VOP-8->VOP-8 (Lecture)-> Resources and Activities -> DeadlockPrevention.zip-> ExtendedCaseStudy

- Account.cs
- DepositTask.cs
- WithdrawTask.cs
- Program.cs

Output

Problem:
It allows **negative** balance →

Deadlocks

- `Lock` can lead to deadlocks or deadly embrace.
- This happens when one thread acquires a lock and then waits for another thread to do some essential work. If that other thread is currently waiting to acquire the same lock, then neither of the two threads can proceed
- For example, If we change our **`Withdraw()`** method to disallow negative balance.

```
private Object myLock = new Object();

public void Withdraw(int amount)
{
    int newBalance = 0;
    try
    {
        Monitor.Enter(myLock);
        while (balance < amount)
        {
            continue;
        }
        newBalance = balance - amount;
        balance = newBalance;
        Console.WriteLine("Balance after Withdraw " + balance);
    }
    finally
    {
        Monitor.Exit(myLock);
    }
}
```

With Deadlock

Plans-> VOP-8->VOP-8 (Lecture)-> Resources and Activities -> DeadlockPrevention.zip-> DeadlockDemo

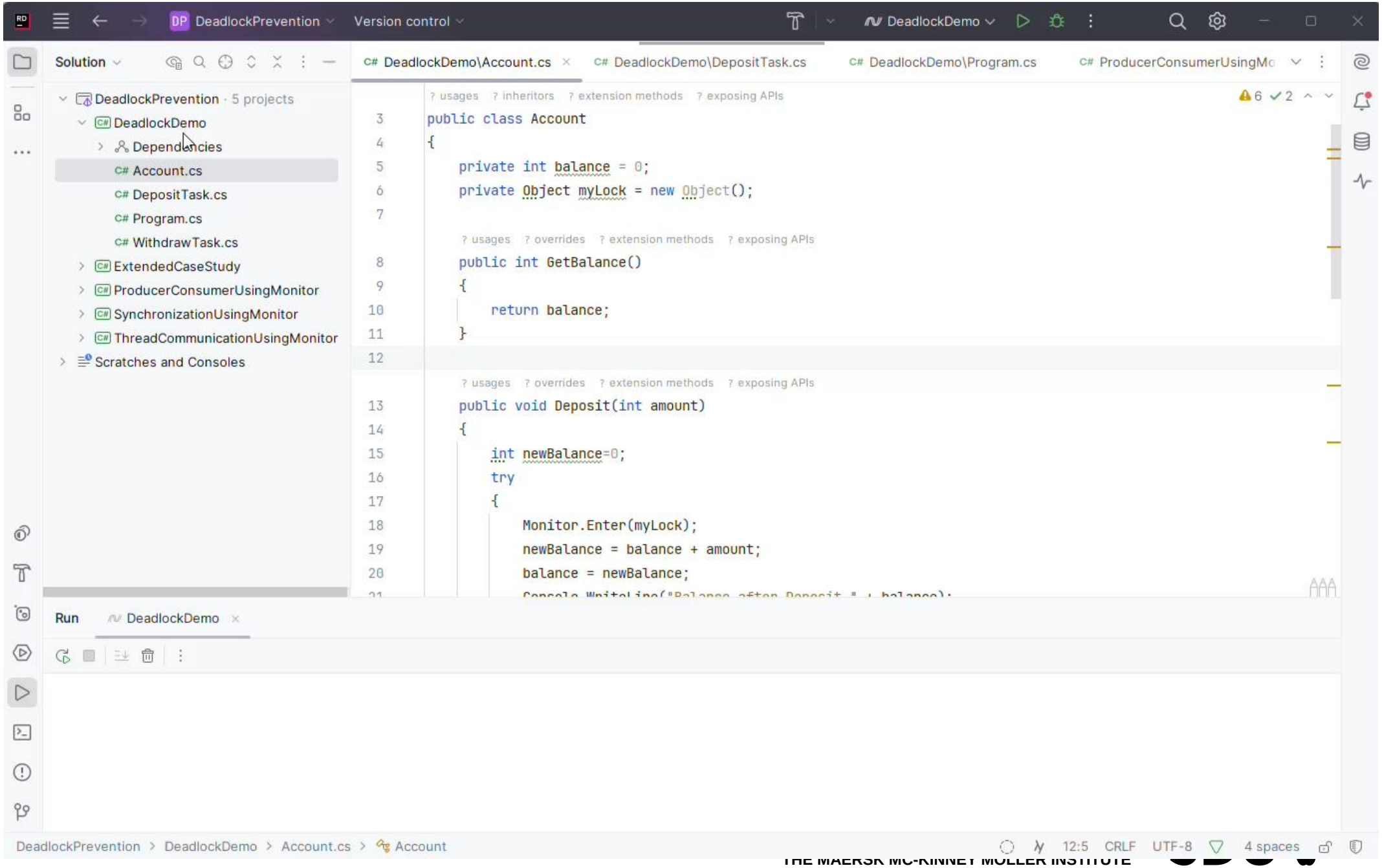
- Account.cs
- DepositTask.cs
- WithdrawTask.cs
- Program.cs

Output

Can you explain the output?

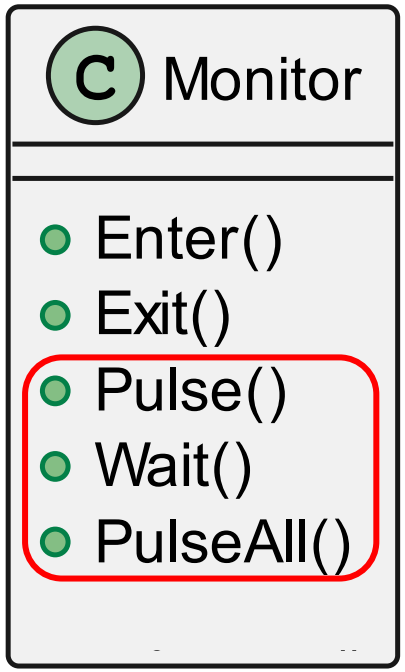
Problem:

It disallows **negative** balance but leads to deadlock →



Avoiding Deadlocks

- We overcome this problem with by utilizing the Monitor methods to facilitate communications between threads.
- We can use **`Wait()`**, **`Pulse()`**, and **`PulseAll()`** methods for thread communications.



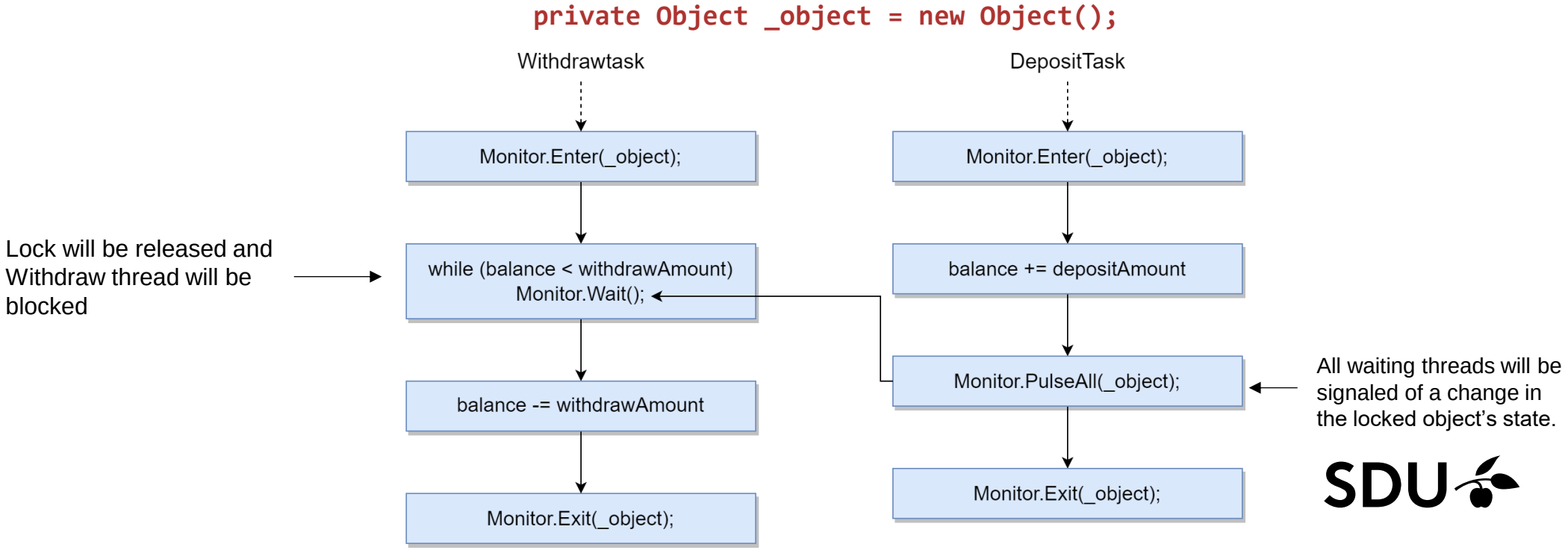
1.Wait(): When the Monitor class's Wait method is invoked, it releases the lock on an object and blocks the current thread until it reacquires the lock.

2.Pulse(): When the Pulse method is invoked of the Monitor class, it sends a signal to a thread in the waiting queue of a change in the locked object's state.

1.PulseAll(): When the Monitor class's PulseAll method is invoked, it sends signals to all waiting threads about a change in the locked object's state.

Thread Communications

- Threads communicate via `Wait()`, `Pulse()`, and `PulseAll()` methods of the Monitor.
- If the balance < amount to be withdrawn, the `WithdrawTask` will wait for the `DepositTask` to add money in the account made.
- When the `DepositTask` adds money to the account, it signals the `WithdrawTask` to try again.
- The `Wait()`, `Pulse()`, and `PulseAll()` methods should be called in a synchronized method or block to avoid `IllegalMonitorStateException`.

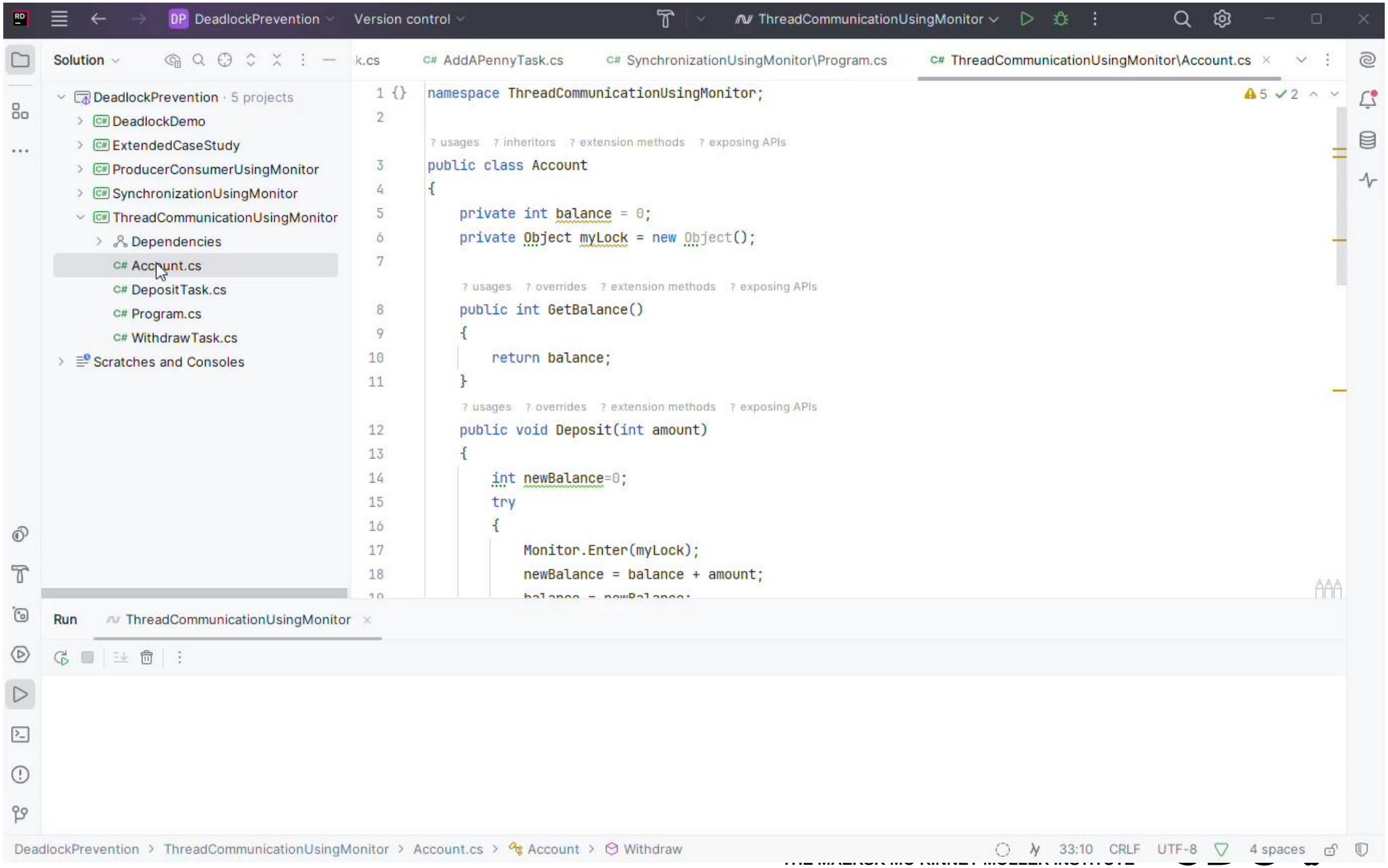


Deadlock Prevention

Plans-> VOP-8->VOP-8 (Lecture)-> Resources and Activities -> DeadlockPrevention.zip-> ThreadCoomunicationUsingMonitor

- Account.cs
- DepositTask.cs
- WithdrawTask.cs
- Program.cs

Output

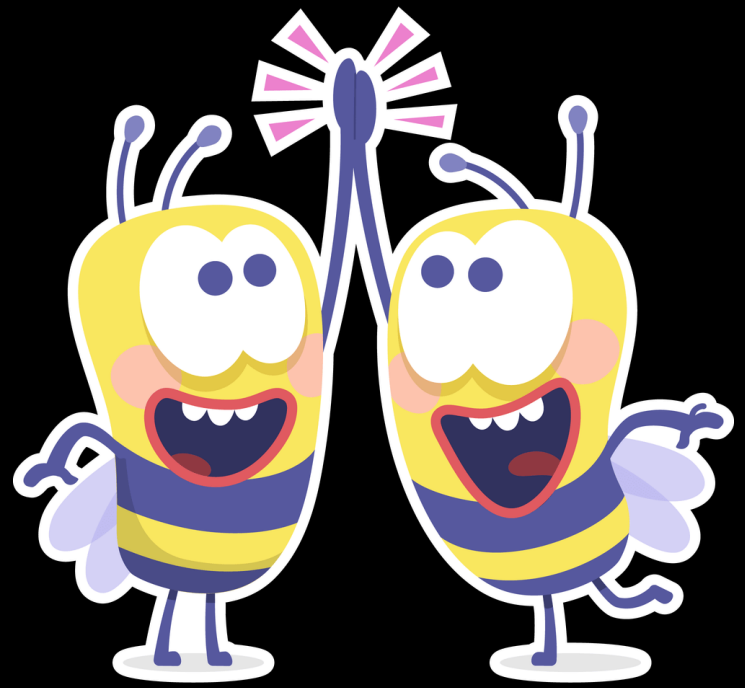


Problem Solved:

No **negative** balance

No deadlock →

Break (10 min)

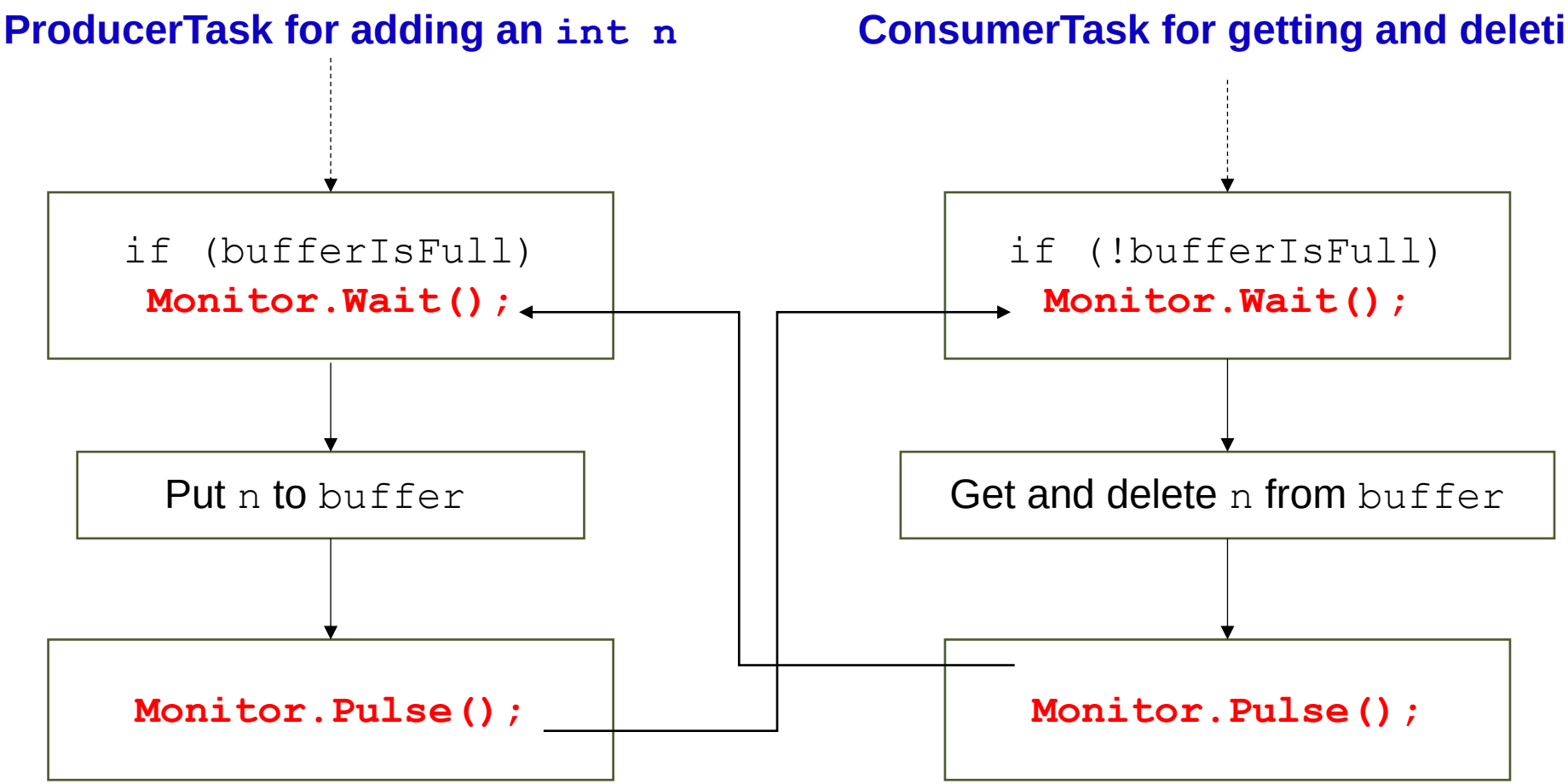


Case Study: Producer/Consumer (Using Monitors)

- In this case study, we have a ValueBuffer `buffer` to store an integer value.
- ValueBuffer `buffer` provides a synchronized method to `Put(int n)` to add an int value
- ValueBuffer `buffer` also provides a synchronized method `Get()` to read and delete an int value from `buffer`.
- A `ProducerTask` for adding values to the ValueBuffer `buffer`
- A `ConsumerTask` for getting values from the ValueBuffer `buffer`
- When ValueBuffer `buffer` is empty, the `Get()` method waits for a *ProducerTask* to `Put(int n)` into `buffer`.
- When ValueBuffer `buffer` is full, the `Put(int n)` method waits for a *ConsumerTask* to `Get()` value `n`.

SDU

Case Study: Producer/Consumer (Using Monitors)

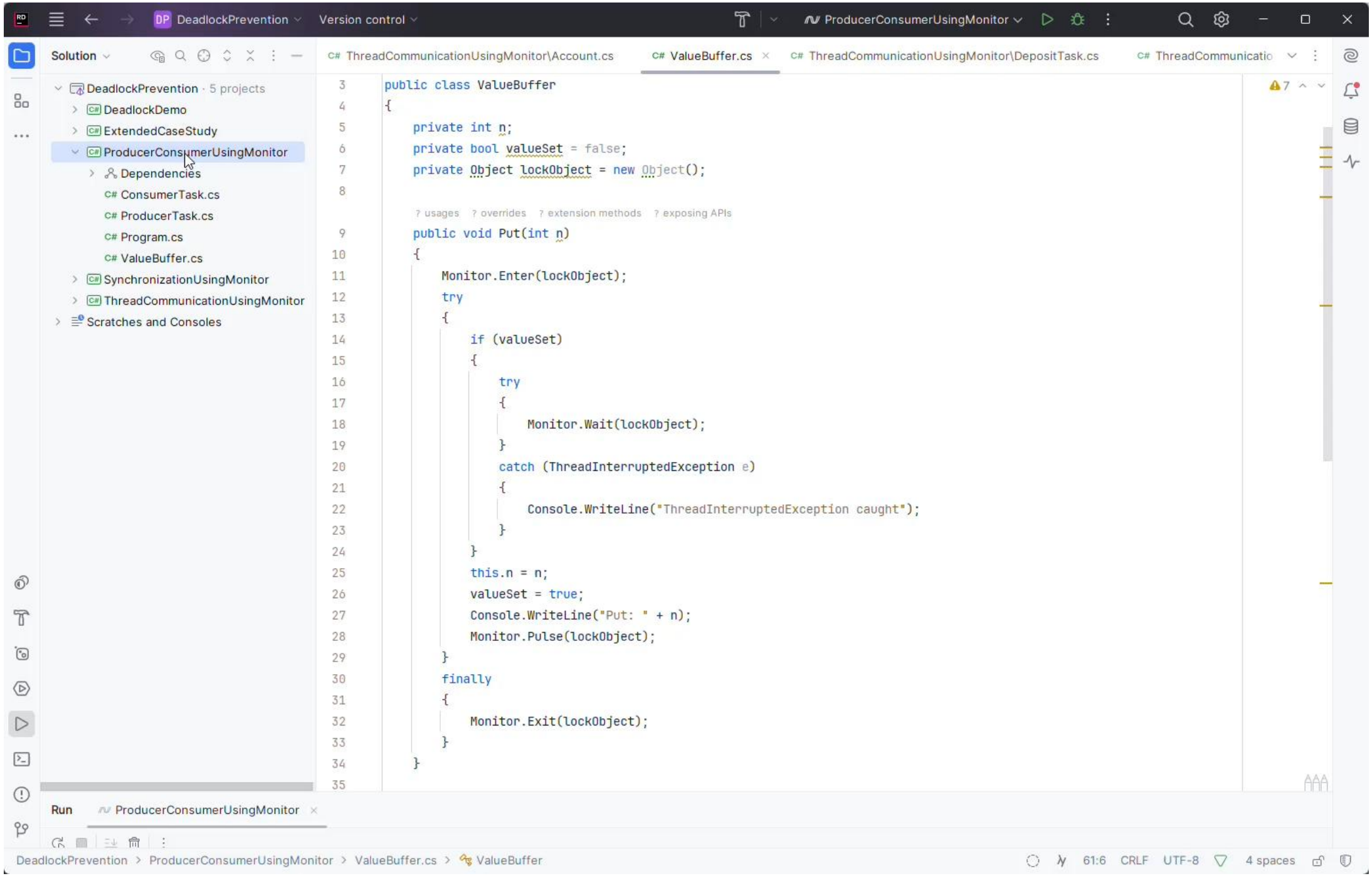


Producer Consumer Case Study

Plans-> VOP-8->VOP-8 (Lecture)-> Resources and Activities -> DeadlockPrevention.zip-> ProducerConsumerUsingMonitor

- ValueBuffer.cs
- ProducerTask.cs
- ConsumerTask.cs
- Program.cs

Output



Clean Synchronous Behavior →

MCQs Quiz

Go to Plans -> VOP-8 -> VOP-8 (Lecture) -> Lecture-8 Test

Good Luck ☺