SDU

Lesson 7: Threads and Synchronization

Agenda

1. Thread Class Methods & Properties
    a. *Yield(), Sleep(milliseconds)*
    b. *Join(), Interrupt()*
    c. *IsAlive, IsBackground, Priority, ThreadState*
2. *Thread synchronization*
    a. *Using Synchronised attribute*
    b. *Using Lock statement*
    c. *Using Monitor class*

Objectives

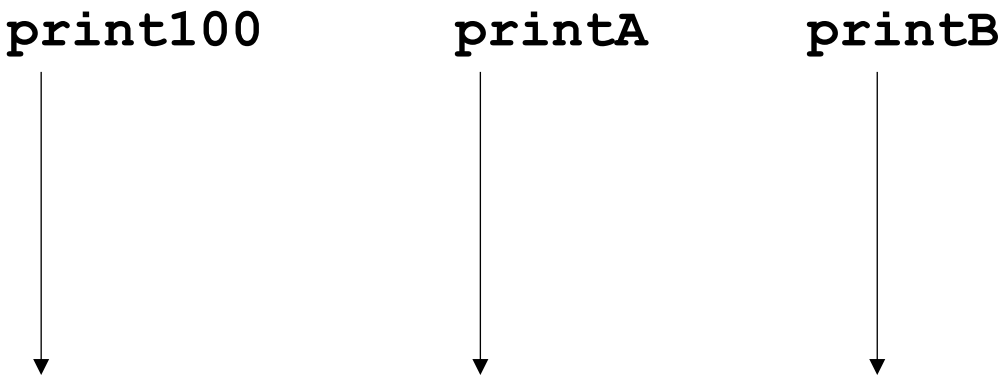
1. Thread Class and Methods
    a. *Use Thread class methods and properties to control Task execution*
2. *Thread Synchronization and Locks*
    a. *Synchronize access to shared objects using Synchonised attribute*
    b. *Synchronize access to shared objects using Lock statement*
    c. *Synchronize access to shared objects using Monitor class*

The Thread Class

Method	Description
`public void Start();`	Starts the execution of the thread
`public bool IsAlive{get;}`	Gets a value indicating whether the thread is currently alive and executing.
`public ThreadState ThreadState{get;}`	Gets a value containing the state of the current thread.
`public bool IsBackground{get;set;}`	Gets or sets a value indicating whether or not a thread is a background thread.
`public ThreadPriority Priority{get;set;}`	Gets or sets a value indicating the scheduling priority of a thread.
`public static void Yield();`	Causes the calling thread to yield execution to another thread that is ready to run
`public static void Sleep(int milli);`	Suspends the current thread for the specified number of milliseconds
`public void Join();`	Blocks the calling thread until this instance's thread terminates,
`public void Interrupt();`	Interrupts a thread that is in the WaitSleepJoin thread state.

The Thread Methods: Yield()

- The static method `Thread.Yield()` method temporarily releases time for other threads



Run method of "PrintNum (print100)" Class

```
public void Run() {
  for (int i = 1; i <= lastNum; i++) {
    Console.Write(" " + i);
    Thread.Yield();
  }
}
```


SDU

The Thread Methods: Sleep()

- The static method `Thread.Sleep(milliseconds)` method puts the thread to sleep for a specified millisecond.

```
public void Run()
{
    try
    {
        for (int i = 1; i <= lastNum; i++)
        {
            Console.Write(" " + i);

            if (i >= 50)
            {
                Thread.Sleep(1000);
            }
        }
    }
    catch (ThreadInterruptedException ex)
    {
        Console.WriteLine("Thread interrupted: " + ex.Message);
    }
}
```

Run method of “PrintNum” Class

SDU

The Thread Methods: Join()

- The method `Join()` can be used to force one thread to wait for another thread to finish.

Run method of "print100" class

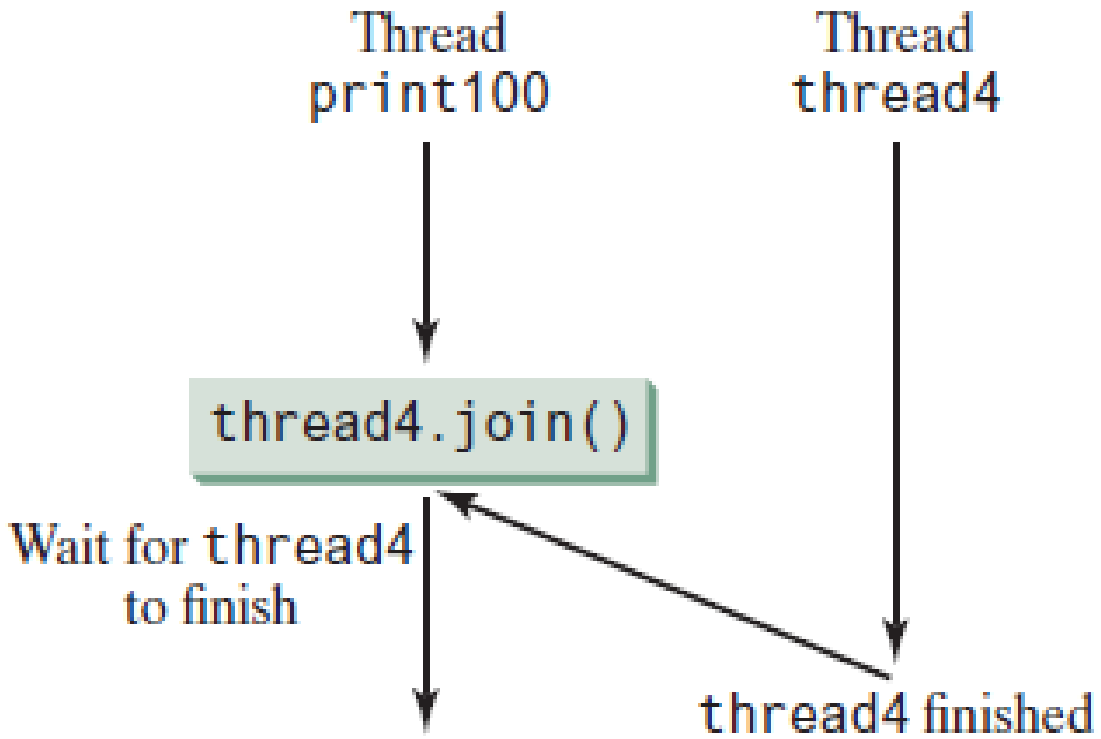
Thread of "PrintChar" class

Thread of "PrintChar" class

```
public void Run()
{
    PrintChar printChar = new PrintChar('c', 40);
    Thread thread4 = new Thread(printChar.Run);
    thread4.Start();

    try
    {
        for (int i = 1; i <= lastNum; i++)
        {
            Console.Write(" " + i);

            if (i == 50)
            {
                thread4.Join();
            }
        }
    }
    catch (ThreadInterruptedException ex)
    {
        Console.WriteLine("Thread interrupted: " + ex.Message);
    }
}
```



The Thread Methods: Interrupt()

- The instance method `Interrupt()` interrupts a thread if it is currently in the Wait, Sleep, Join or Running State.
- If the thread is currently blocked, it is awaken into a Ready State and it throws an `ThreadInterruptedException`.
- If the thread is not blocked at the time of calling `Interrupt()`, the exception will be thrown the next time it enters a blocking state.

SDU

The Thread Property: Priority

- C# assigns priority to every thread. Thread priority in C# is represented by the ThreadPriority enumeration.
- Thread class defines five enum constants for setting thread priorities:
  - `enum ThreadPriority{Lowest, BelowNormal, Normal, AboveNormal, Highest}`
- We can get the priority of a thread using the `Priority` property

Default Priority

  - `thread.Priority.ToString();`
- We can also set the priority of a thread using:

```
thread.Priority = ThreadPriority.Highest
```

- By default, a new thread inherits the priority of the thread that spawned it.

SDU

The Thread Property: IsBackground()

- If the value of IsBackground is set to be true, then it means the thread is a background thread.
- If the value of IsBackground is set to be false, then it means the thread is a foreground thread.

SDU

Background Threads

- Background thread is a thread that is subordinate to the thread that creates it.
- When the thread that created the background thread ends, background thread dies with it.
- Typically used for threads that run infinite time.

Foreground Threads

- A thread that is not a background thread is called a Foreground thread
- The **Main()** method is a **foreground thread (non-background thread)**.
- Typically used for threads that run finite time.
- It must be explicitly stopped or destroyed, or its run method must return.
- You can only call IsBackground for a thread before it starts

SDU

FAQs

- If all threads have equal priorities, each is assigned an equal portion of CPU time in a circular queue.
- This circular queue is called round-robin scheduling.
- Starvation can happen when higher priority threads or same-priority thread do not yield.
- To avoid starvation, threads with higher priority should frequently invoke Yield() or Sleep() methods.

Threads Methods & Properties

From Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> Misc

- ✓ PrintChar.cs
- ✓ PrintNum.cs
- ✓ PrintNum2.cs
- ✓ Program.cs

SDU

Multithreading & Avalonia

Threads and updating the UI

- Avalonia UI applications have one main thread, and this handles the UI.
  - You may have multiple threads running different processes, and in some of these you may want to update the UI. So how do you that if the UI is running on its own thread?
- Avalonia has a dispatcher service that can be used to access the UIThread: → **`Dispatcher.UIThread`**
  - It has two methods **`Post`** and **`InvokeAsync`** that are used to run a process on the UI thread.

Method	Description
`Post`	When you want to start a job, but not wait for it to complete. *"Fire-and-forget"*
`InvokeAsync`	When you need to wait for the result of the job.

In this example we want to be able to update the images from another thread, while the program is running. Therefore, we use the **`Dispatcher.UIThread.Post`**:

```
<StackPanel>
  <Image Height="200" Width="200" Name="Image1"/>
  <Image Height="200" Width="200" Name="Image2"/>
</StackPanel>
```

```
public void UpdateImages(Image image1, Image image2)
{
  Dispatcher.UIThread.Post(() =>
  {
    Image1.Source = image1.Source;
    Image2.Source = image2.Source;
  });
}
```


https://docs.avaloniaui.net/docs/guides/development-guides/accessing-the-ui-thread

Threads and updating the UI


```
private void StartHandler(object sender, RoutedEventArgs e)
{
    thread = new Thread(IncrementCounter);
    thread.IsBackground = true;
    thread.Start();
}
```

```
private void StopHandler(object sender, RoutedEventArgs e)
{
    thread.Interrupt();
}
```

```
private void IncrementCounter()
{
 try
 {
  while (true)
  {
   string currentCounter = "";
   Dispatcher.UIThread.InvokeAsync(() => currentCounter = counterText.Text).Wait();
   counter = int.Parse(currentCounter);
   counter = counter + 1;
   Console.WriteLine("Counter: " + counter);
   Dispatcher.UIThread.InvokeAsync(() => counterText.Text = counter.ToString()).Wait();
   Thread.Sleep(waitTime);
  }
 }
 catch (ThreadInterruptedException ex)
 {
  Console.WriteLine("Interrupted");
 }
}
```

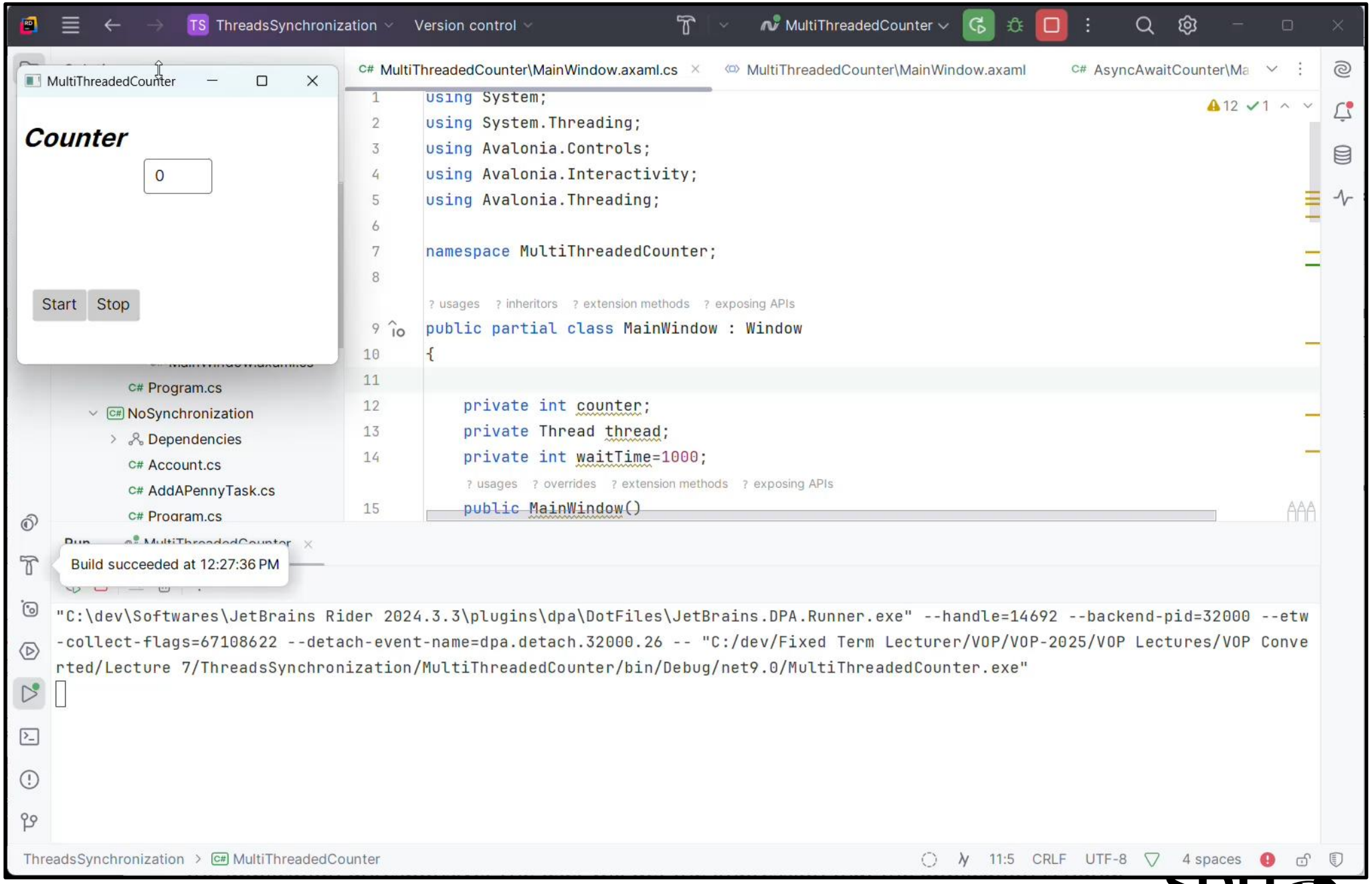

https://docs.avaloniaui.net/docs/guides/development-guides/accessing-the-ui-thread

Threads and updating the UI

From Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> MultiThreadedCounter

- ✓ MainWindow.axaml.cs
- ✓ MainWindow.axaml.cs

Output


MultiThreadedCounter
Counter
0
Start
Stop
ThreadsSynchronization
Version control
MultiThreadedCounter
MultiThreadedCounter\MainWindow.axaml.cs
MultiThreadedCounter\MainWindow.axaml
AsyncAwaitCounter\Ma
using System;
using System.Threading;
using Avalonia.Controls;
using Avalonia.Interactivity;
using Avalonia.Threading;
namespace MultiThreadedCounter;
? usages ? inheritors ? extension methods ? exposing APIs
public partial class MainWindow : Window
{
private int counter;
private Thread thread;
private int waitTime=1000;
? usages ? overrides ? extension methods ? exposing APIs
public MainWindow()
Program.cs
NoSynchronization
Dependencies
Account.cs
AddAPennyTask.cs
Program.cs
Build succeeded at 12:27:36 PM
"C:\dev\Softwares\JetBrains Rider 2024.3.3\plugins\dpa\DotFiles\JetBrains.DPA.Runner.exe" --handle=14692 --backend-pid=32000 --etw-collect-flags=67108622 --detach-event-name=dpa.detach.32000.26 -- "C:/dev/Fixed Term Lecturer/VOP/VOP-2025/VOP Lectures/VOP Converted/Lecture 7/ThreadsSynchronization/MultiThreadedCounter/bin/Debug/net9.0/MultiThreadedCounter.exe"
ThreadsSynchronization > MultiThreadedCounter
11:5
CRLF
UTF-8
4 spaces

Async Tasks and updating the UI
Self Study

CancellationTokenSource (CTS) is used to manage task **cancellation** in an **asynchronous and cooperative** manner.

```
private CancellationTokenSource cts;
```

```
private void StartHandler(object sender, RoutedEventArgs e)
{
    cts = new CancellationTokenSource();
    IncrementCounter(cts.Token);

}
```

```
private void StopHandler(object sender, RoutedEventArgs e)
{
    cts.Cancel();
}
```

```
private async Task IncrementCounter(CancellationToken token)
{
    try
    {
        while (!token.IsCancellationRequested)
        {
            string currentCounter = await Dispatcher.UIThread.InvokeAsync(() => counterText.Text);
            counter=int.Parse(currentCounter);
            counter++;
            Console.WriteLine("Counter: "+counter);
            await Dispatcher.UIThread.InvokeAsync(() => counterText.Text=counter.ToString());
            await Task.Delay(waitTime,token);
        }
    }
    catch(OperationCanceledException)
    {
        Console.WriteLine("Counter Stopped.");
    }
}
```

Async Tasks and updating the UI

From Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> AsyncAwaitCounter

- ✓ MainWindow.axaml.cs
- ✓ MainWindow.axaml.cs

BREAK (10 min)

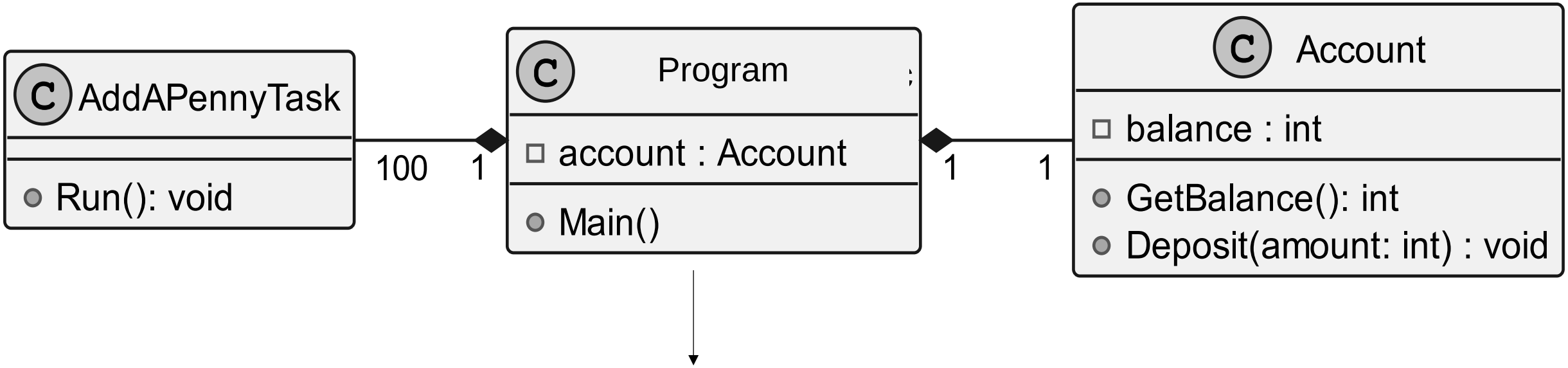
SDU

Threads Synchronization

Thread Synchronization

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
- For example, 100 unsynchronized threads are depositing into the same bank account.
- This will create a race condition, where some tasks will overwrite the result of other tasks.

Case Study: Bank Account



If opening balance = 0, then
closing balance should be 100

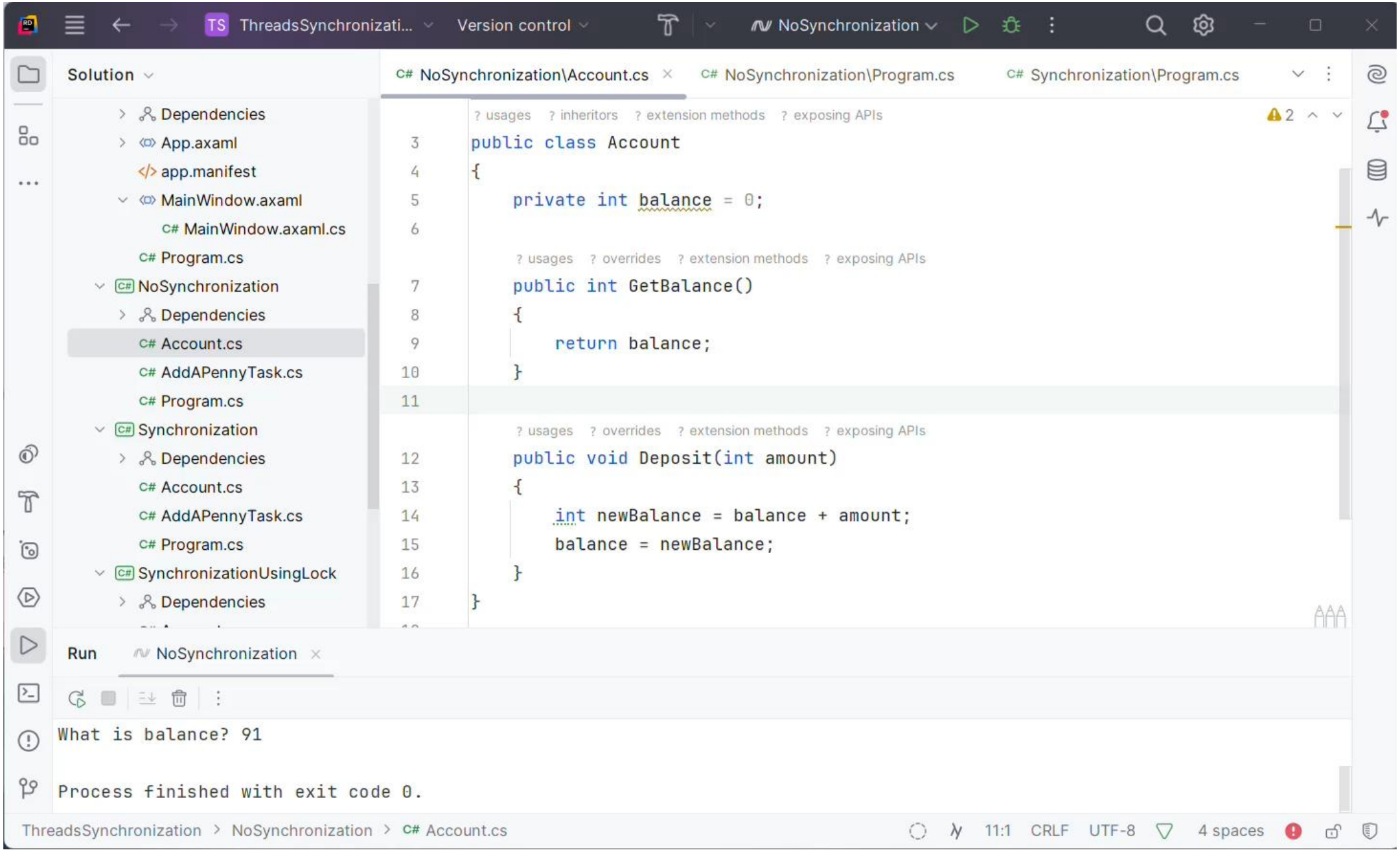
SDU

Depositing into Account: No Synchronization

From Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> NoSynchronization

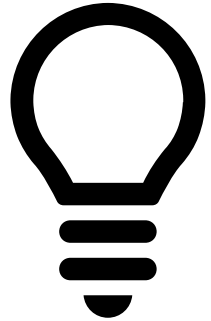
- ✓ Account.cs
- ✓ AddAPennyTask.cs
- ✓ Program.cs

Output



Problem:
Race condition
Data Inconsistency

What does it imply???



Although, Multithreading is a powerful feature but we cannot afford data inconsistency

To avoid this situation, We need each thread to deposit money in a sequential manner

SDU

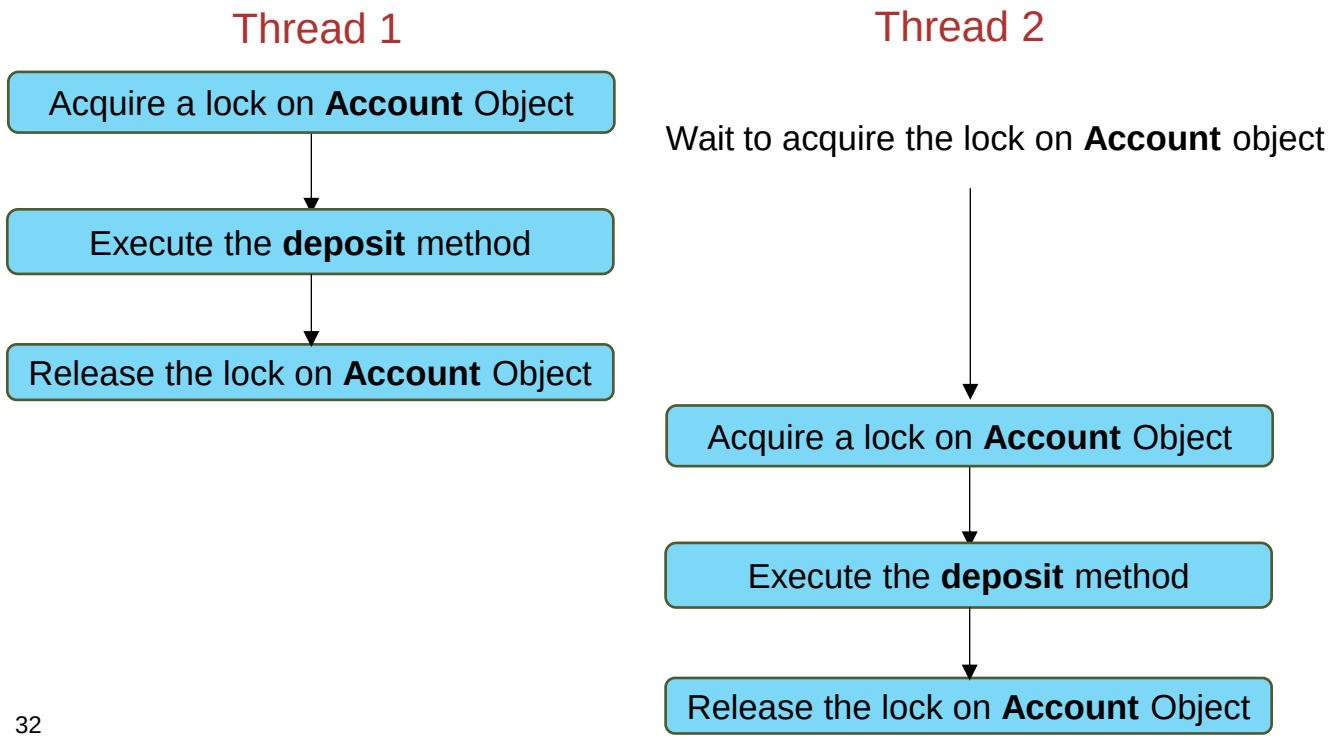
SDU

Threads Synchronization

a. *Using Synchronized attribute*
b. *Using Lock statement*
c. *Using Monitor class*

Synchronization using `Synchronized` attribute

- A `Synchronized` attribute can be used to avoid race condition and develop thread-safe classes.
- If one thread enters a method with **Synchronized** attribute, the lock of that object (in this case Account object) is acquired first, then the method is executed, and finally the lock is released.
- For instance, we can simply place `[MethodImpl(MethodImplOptions.Synchronized)]` as an attribute of the `deposit(int amount)` method.



Account.cs

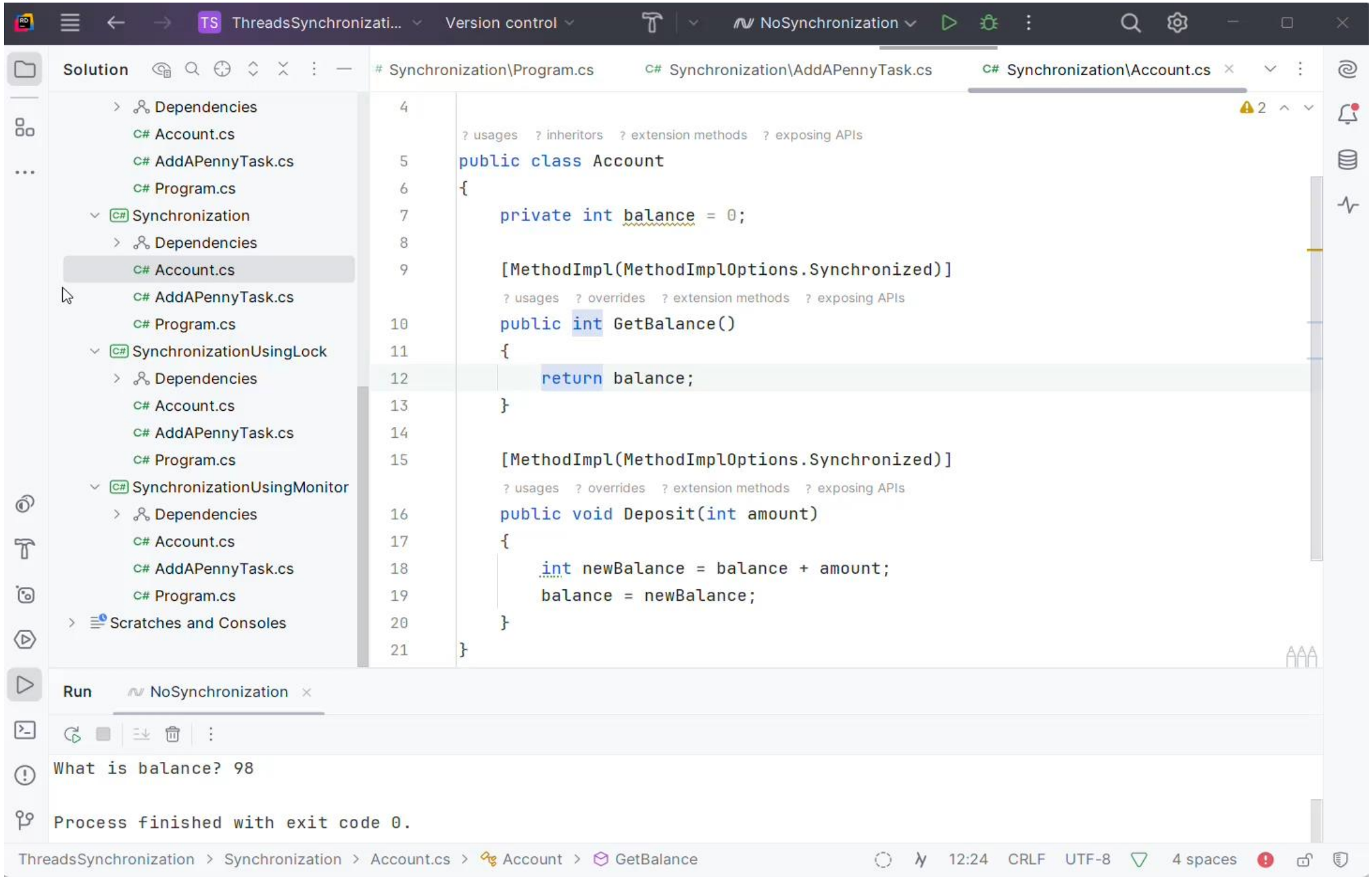
```
[MethodImpl(MethodImplOptions.Synchronized)]
public void deposit(int amount)
{
    int newBalance = balance + amount;
    balance = newBalance;
}
```

Depositing into Account: Synchronization Attribute

From Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> Synchronization

- ✓ Account.cs
- ✓ AddAPennyTask.cs
- ✓ Program.cs

Output



Problem Solved:

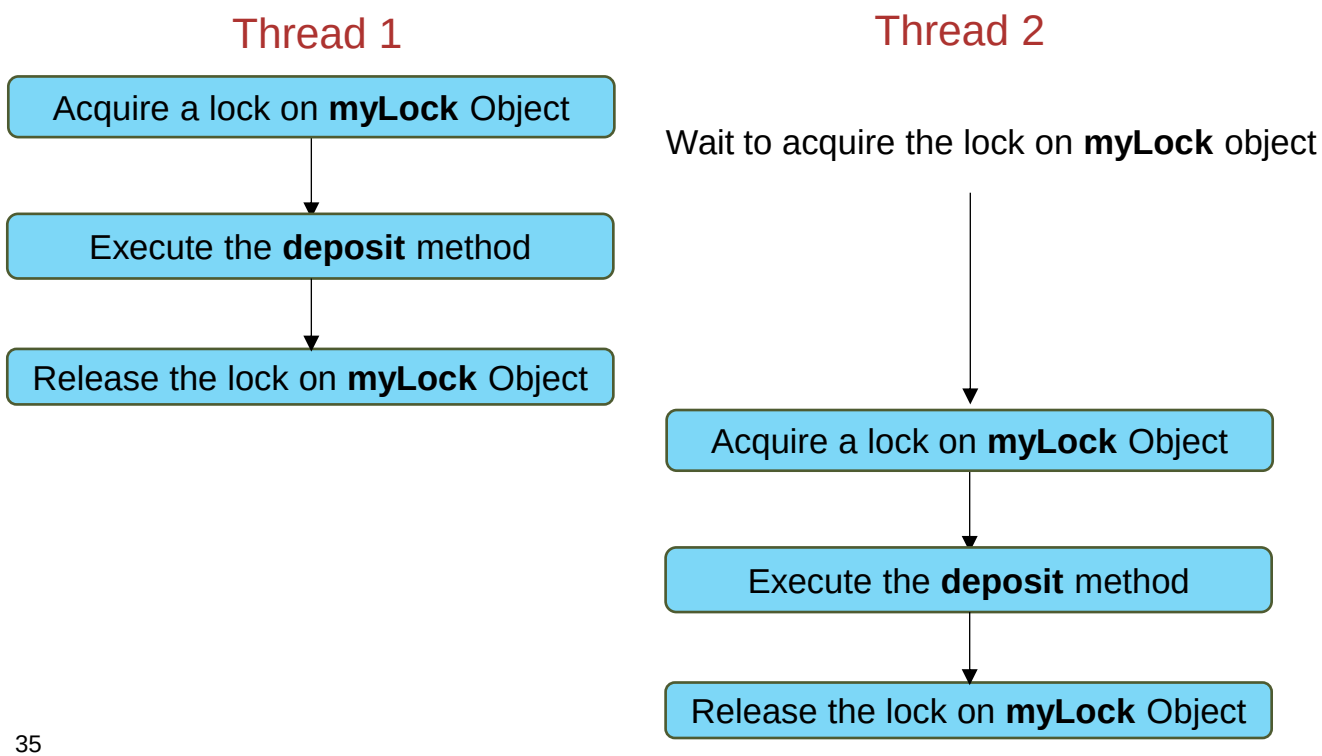
- Simple approach
- No Race condition
- No Data Inconsistency

But

- Reduce concurrency
- Less Control

Synchronization using lock statement

- A `lock` statement can be used to avoid race condition and develop thread-safe classes.
- It ensures that only one thread can enter a critical section of codes. Other threads that try to be the owner of the `lock` would be suspended until the first thread releases the `lock`.
- For instance, we can simply place `lock` statement inside the code of the `deposit(double amount)` method.



Account.cs

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

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

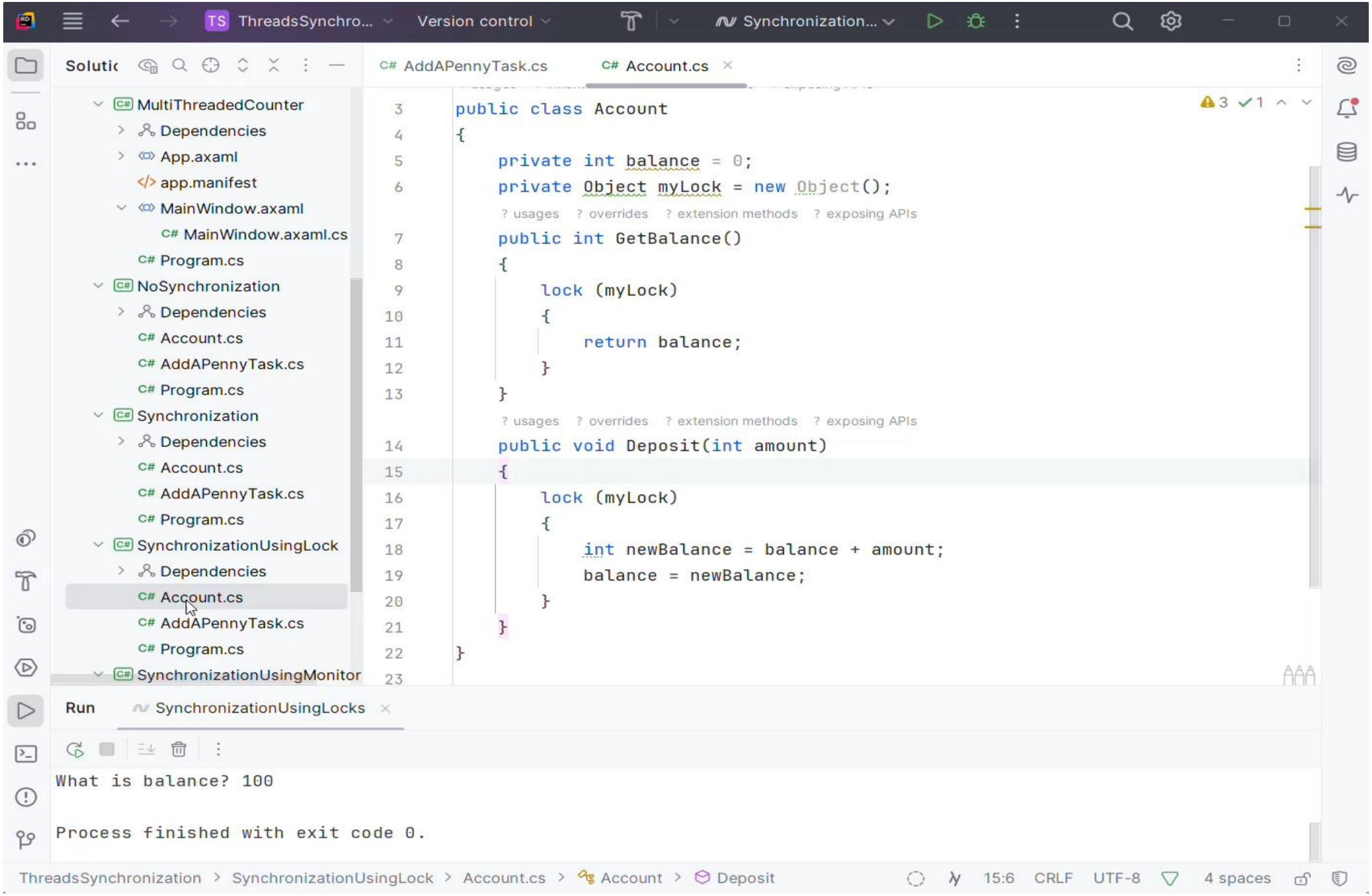

https://kevinsalimi.medium.com/synchronization-by-lock-and-monitor-statements-in-c-28395fcc5daa

Depositing into Account: Synchronization using Lock

Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> SynchronizationUsingLock

- ✓ Account.cs
- ✓ AddAPennyTask.cs
- ✓ Program.cs

Output



Problem Solved:

- No Race condition
- No Data Inconsistency
- Better concurrency

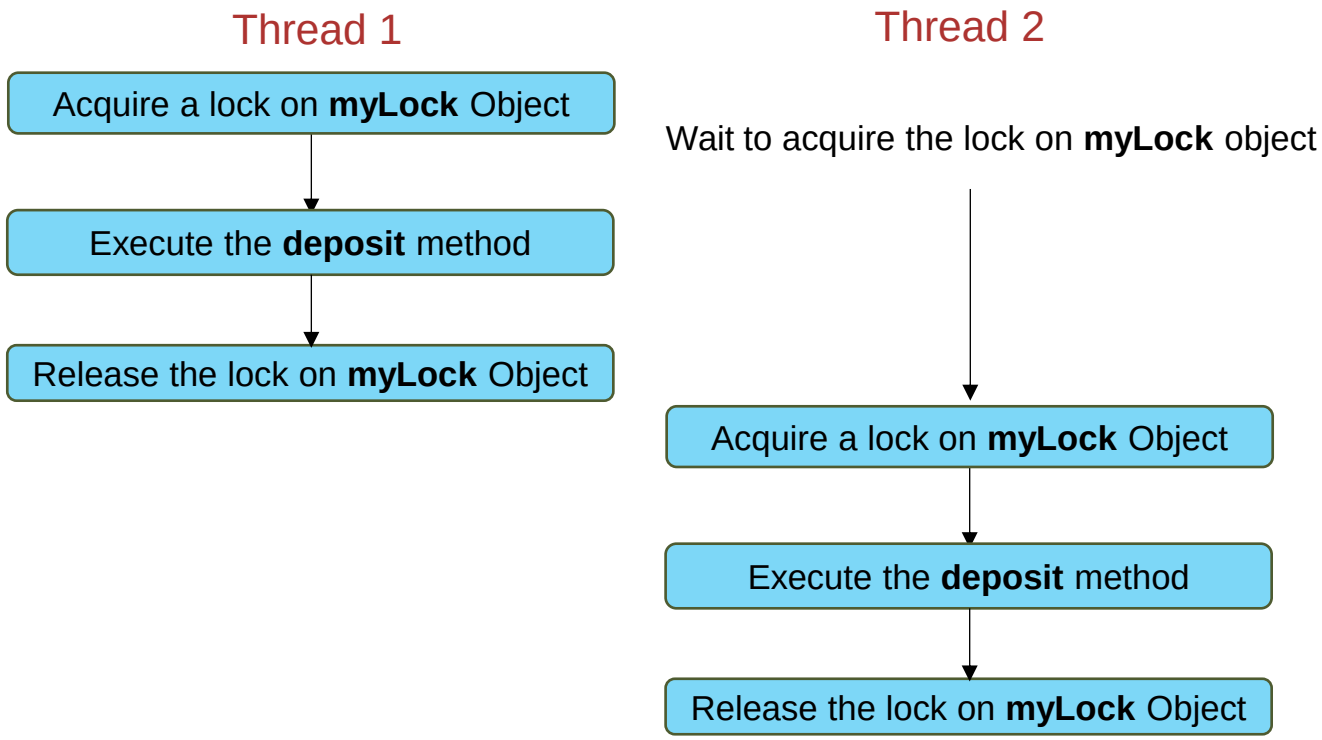
But

- Prone to deadlock

Synchronization using Monitor class

- Provides a mechanism that synchronizes access to objects.
- When a thread calls Monitor.Enter(), it tries to acquire the lock. Once a thread has acquired the lock, other threads that try to acquire the same lock will be blocked until the lock is released by calling Monitor.Exit().

Enter(Object)	Acquires an exclusive lock on the specified object.
Exit(Object)	Releases an exclusive lock on the specified object.



Account.cs

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

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

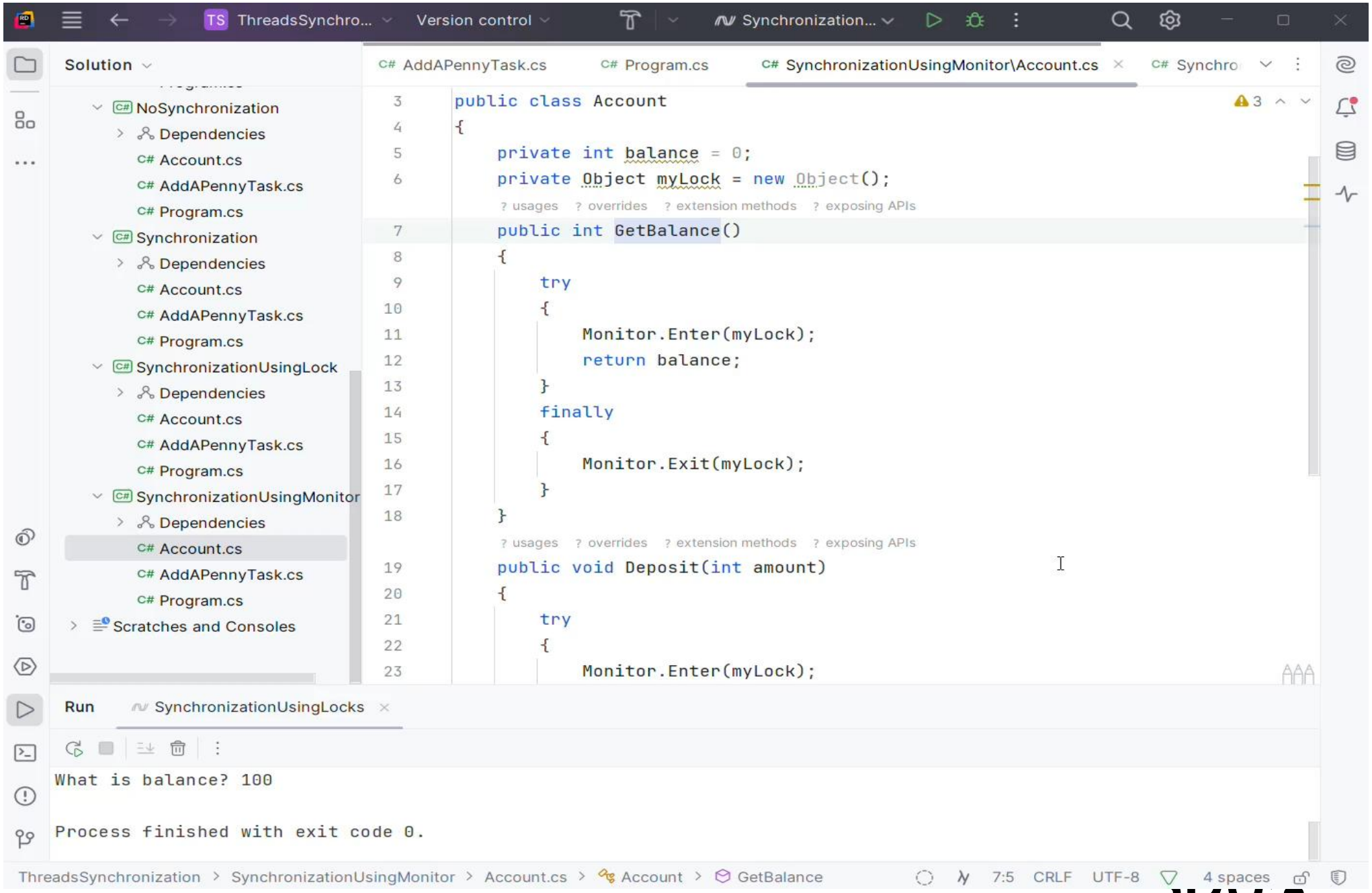

https://learn.microsoft.com/en-us/dotnet/api/system.threading.monitor?view=net-8.0

Depositing into Account: Synchronization using Monitor

Plans-> VOP-7->VOP-7(Lecture)-> Resources and Activities-> ThreadSynchronization.zip -> SynchronizationUsingMonitor

- ✓ Account.cs
- ✓ AddAPennyTask.cs
- ✓ Program.cs

Output



Problem Solved:

- Recommended approach
- No Race condition
- No Data Inconsistency
- Better concurrency
- Avoid deadlock

MCQs Quiz

Go to Plans -> VOP-7 -> VOP-7 (Lecture) -> Lecture-7 Test

Good Luck ☺