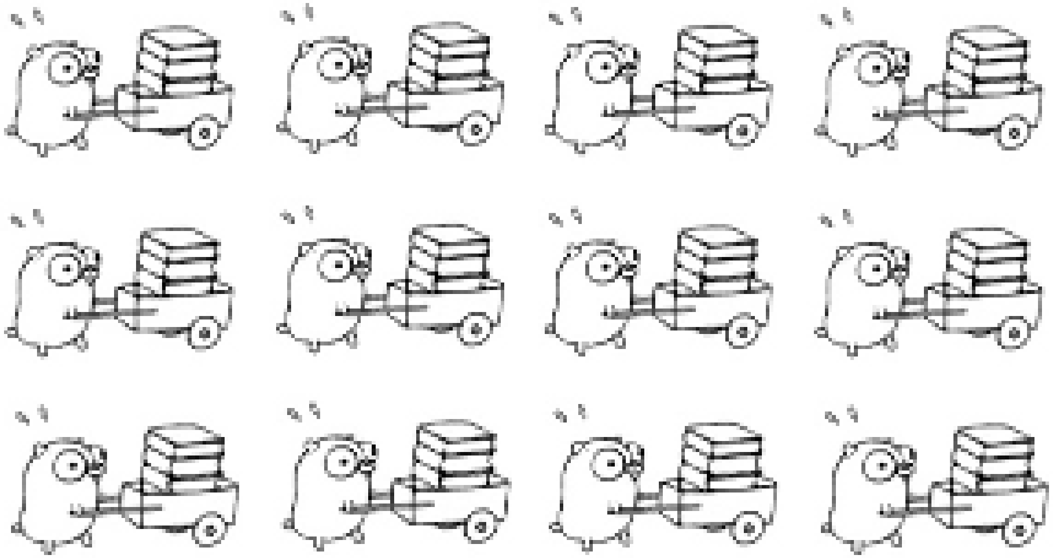
Parallel Programming

Thread

Threadpool

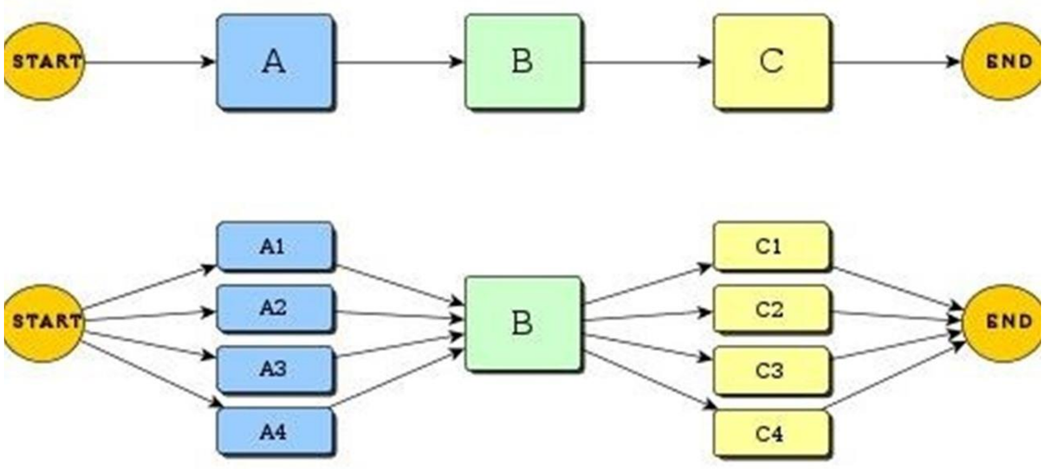
Multithreading

Synchronisation



Content

- Parallel Programming
- Process & Thread
- Problem: Race Condition
- Solution: Thread Synchronisation
  - Join
  - Lock
  - Monitor
  - Semaphore



Parallel Programming

is a type of computation in which many calculations or the execution of processes are carried out simultaneously.

Multithreading

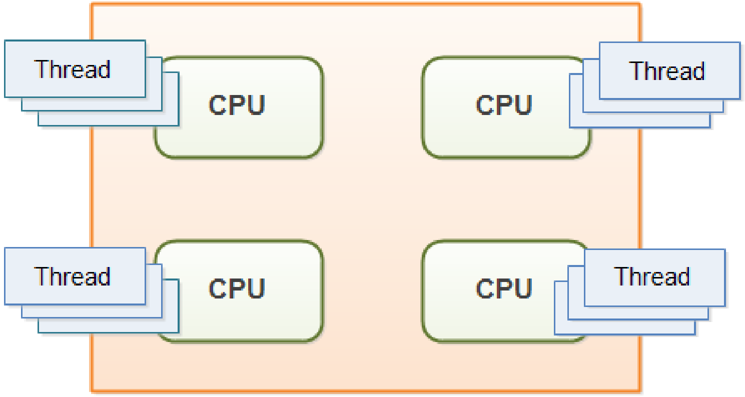
- **Single thread:** one thread can run as a single sequential thread
- **Multiple threads:** in a single program all running at the same time and performing different tasks

referred as Multithreading

Thread

- lightweight process
- it runs within the context of a program
- takes advantage of resources allocated for that program

- By default, C# Console Applications run “Single-Threaded”
  - = One process with one thread
  - Sufficient for small Programs
  - Only one action at a Time



Multithreading

Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process.

Task Manager

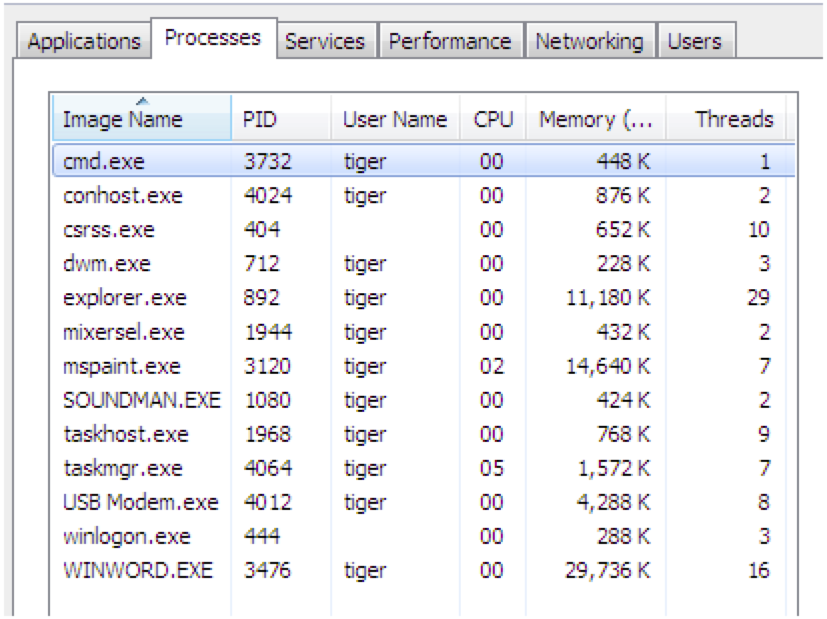
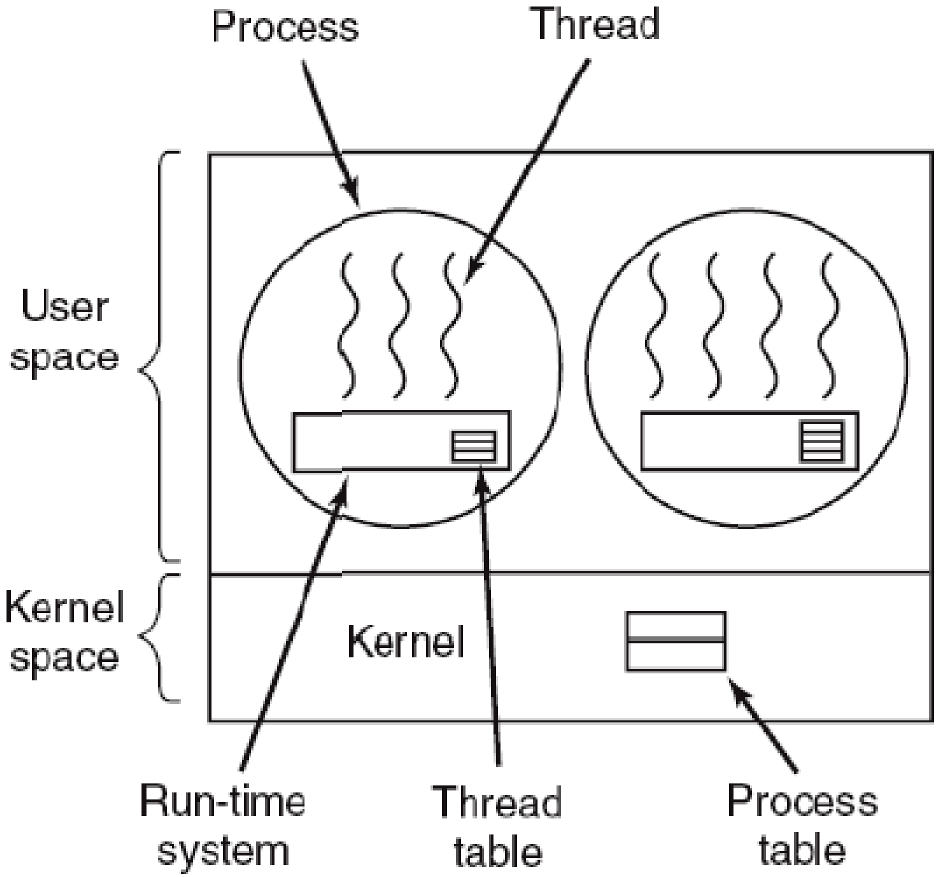
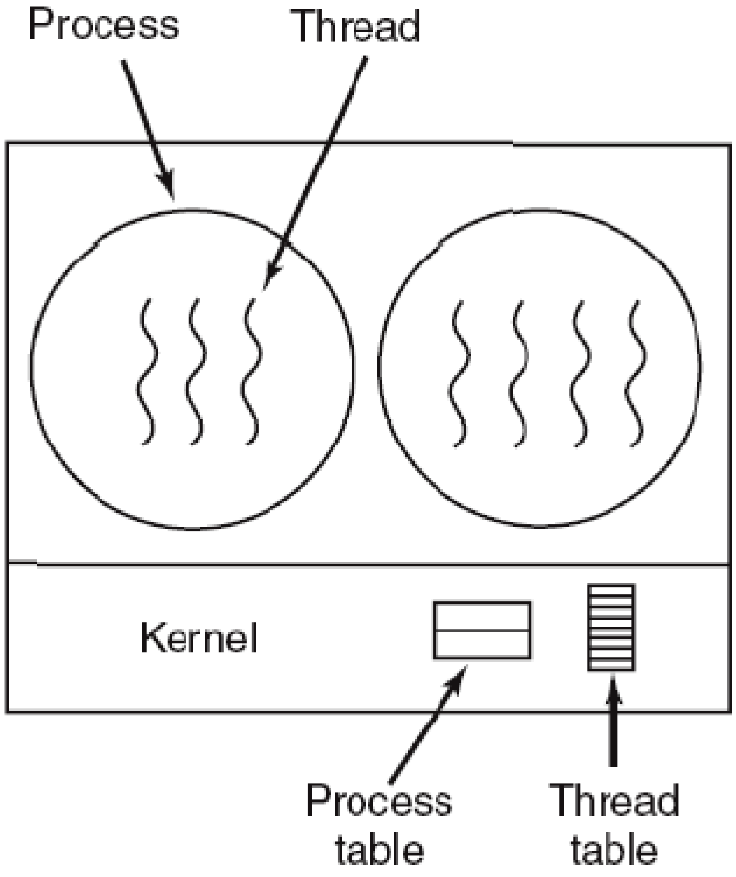


Image Name	PID	User Name	CPU	Memory (...	Threads
cmd.exe	3732	tiger	00	448 K	1
conhost.exe	4024	tiger	00	876 K	2
csrss.exe	404		00	652 K	10
dwm.exe	712	tiger	00	228 K	3
explorer.exe	892	tiger	00	11,180 K	29
mixersel.exe	1944	tiger	00	432 K	2
mspaint.exe	3120	tiger	02	14,640 K	7
SOUNDMAN.EXE	1080	tiger	00	424 K	2
taskhost.exe	1968	tiger	00	768 K	9
taskmgr.exe	4064	tiger	05	1,572 K	7
USB Modem.exe	4012	tiger	00	4,288 K	8
winlogon.exe	444		00	288 K	3
WINWORD.EXE	3476	tiger	00	29,736 K	16

- Turn on Thread column
- See the processes and the number of threads for every process
- Notice that only cmd.exe is running inside a single thread
- All other applications use multipe threads

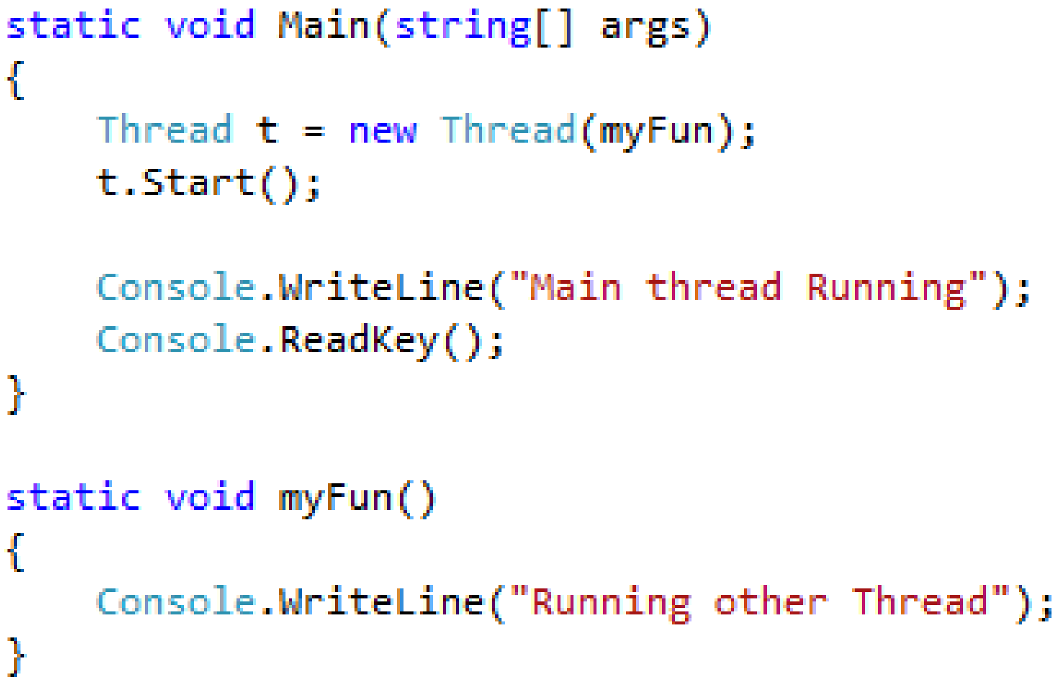
Process vs Thread





Simple Thread Creation

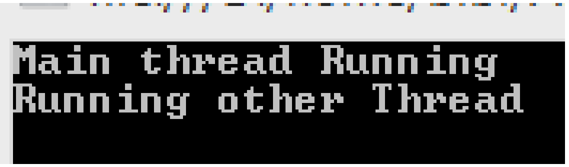
- Thread class accepts a delegate parameter
- Starting Thread t with t.Start()



```
static void Main(string[] args)
{
    Thread t = new Thread(myFun);
    t.Start();

    Console.WriteLine("Main thread Running");
    Console.ReadKey();
}

static void myFun()
{
    Console.WriteLine("Running other Thread");
}
```



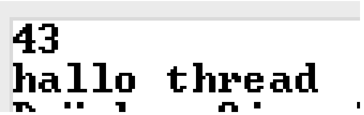
```
Main thread Running
Running other Thread
```

Threads mit Parameter

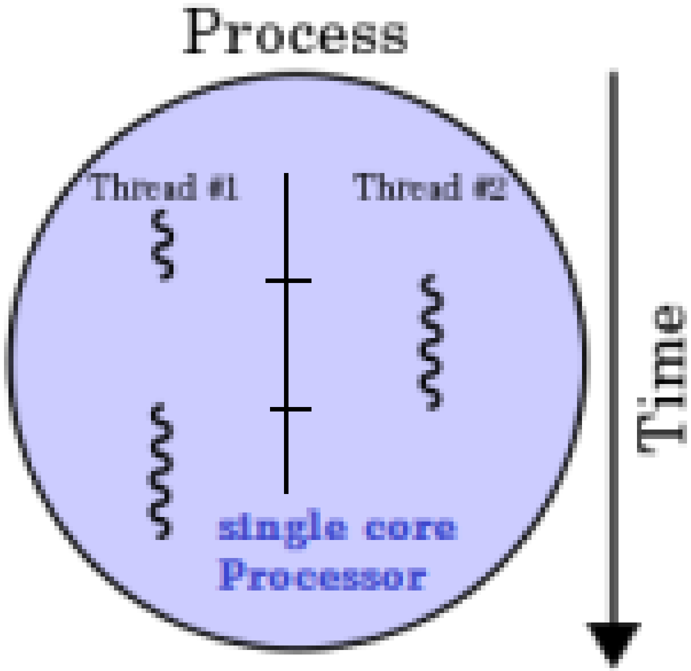
```
class ParameterThreads03
{
    public static void Main(String[] args)
    {
       // ParameterThreads03 p = new ParameterThreads03();
        //Parametrisierte Threads starten
        ParameterizedThreadStart pts = new ParameterizedThreadStart(methode);
        Thread thread = new Thread(pts);
        thread.Start(43);

        //Eine weitere Möglichkeit
        Thread thread2 = new Thread(delegate() { methode("hallo thread"); });
        thread2.Start();
    }

    //Beispielmethode:
    private static void methode(Object parameter)
    {
        Console.WriteLine(parameter);
    }
}
```



```
43
hallo thread
```



Class Thread

& Namespace System.Threading

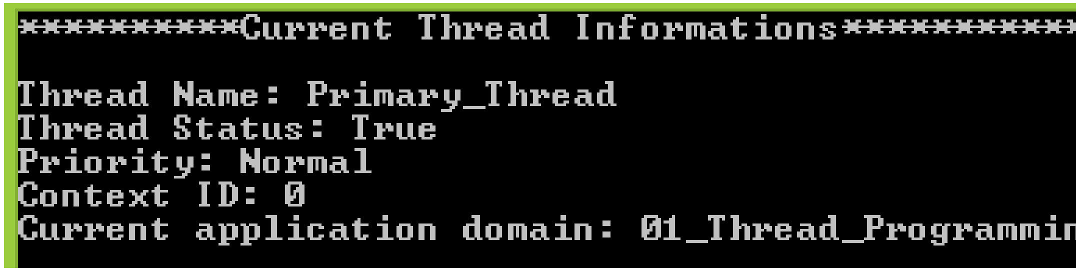
Thread Informations

```
using System;
using System.Threading;

namespace _01_Thread_Programming
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("**********Current Thread Informations***************\n");
            Thread t = Thread.CurrentThread;
            t.Name = "Primary_Thread";

            Console.WriteLine("Thread Name: {0}", t.Name);
            Console.WriteLine("Thread Status: {0}", t.IsAlive);
            Console.WriteLine("Priority: {0}", t.Priority);
            Console.WriteLine("Context ID: {0}", Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Current application domain: {0}", Thread.GetDomain().Friendly

            Console.ReadKey();
        }
    }
}
```



```
**********Current Thread Informations***********

Thread Name: Primary_Thread
Thread Status: True
Priority: Normal
Context ID: 0
Current application domain: 01_Thread_Programmin
```

IsAlive: true if this thread has been started and has not terminated normally or aborted; therwise, false.

System.Threading.Thread class

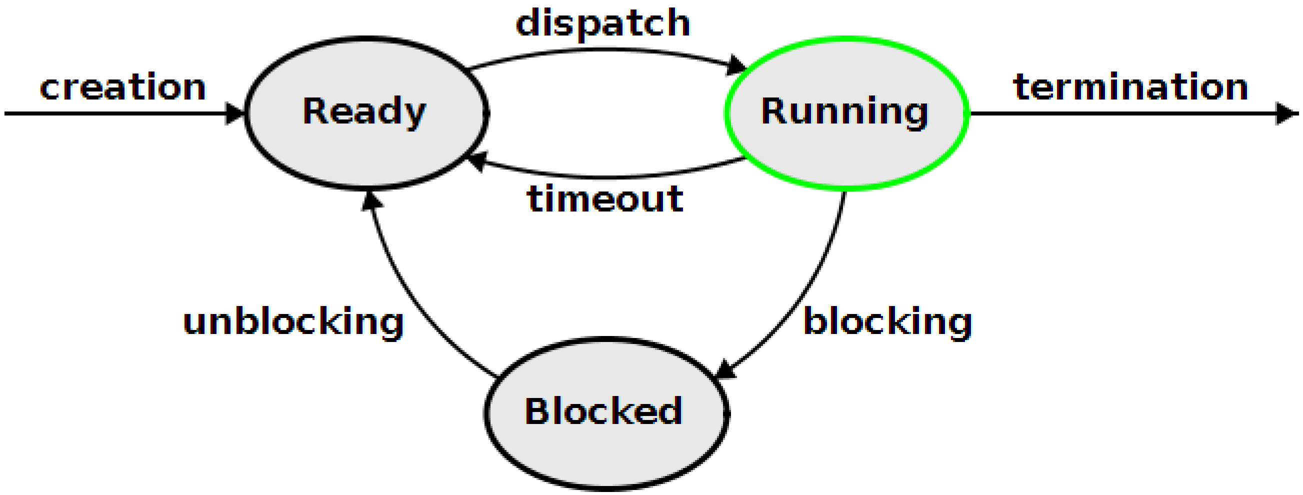
Member	Type	Description
CurrentThread	Static	Return a reference of current running thread.
Sleep	Static	Suspend the current thread for a specific duration.
GetDoamin	Static	Return a reference of current application domain.
CurrentContext	Static	Return a reference of current context in which the thread currently running.
Priority	Instance level	Get or Set the Thread priority level.
IsAlive	Instance level	Get the thread state in form of True or False value.
Start	Instance level	Instruct the CLR to start the thread.
Suspend	Instance level	Suspend the thread.
Resume	Instance level	Resume a previously suspended thread.
Abort	Instance level	Instruct the CLR to terminate the thread.
Name	Instance level	Allows establishing a name to thread.

System.Threading Namespace

Type	Description
Thread	It represents a thread that execute within the CLR. Using this, we can produce additional threads in application domain.
Mutex	It is used for synchronization between application domains.
Monitor	It implements synchronization of objects using Locks and Wait.
Smaphore	It allows limiting the number of threads that can access a resource concurrently.
Interlock	It provides atomic operations for variables that are shared by multiple threads.
ThreadPool	It allows you to interact with the CLR maintained thread pool.
ThreadPriority	This represents the priority level such as High, Normal, Low.

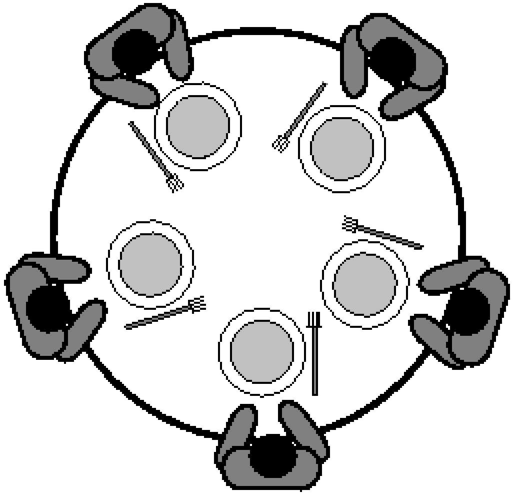
Three States of a Thread

- Ready State Thread.Start() has been called
- Running State Run() is being executed
- Blocked State Waiting for an event to occur

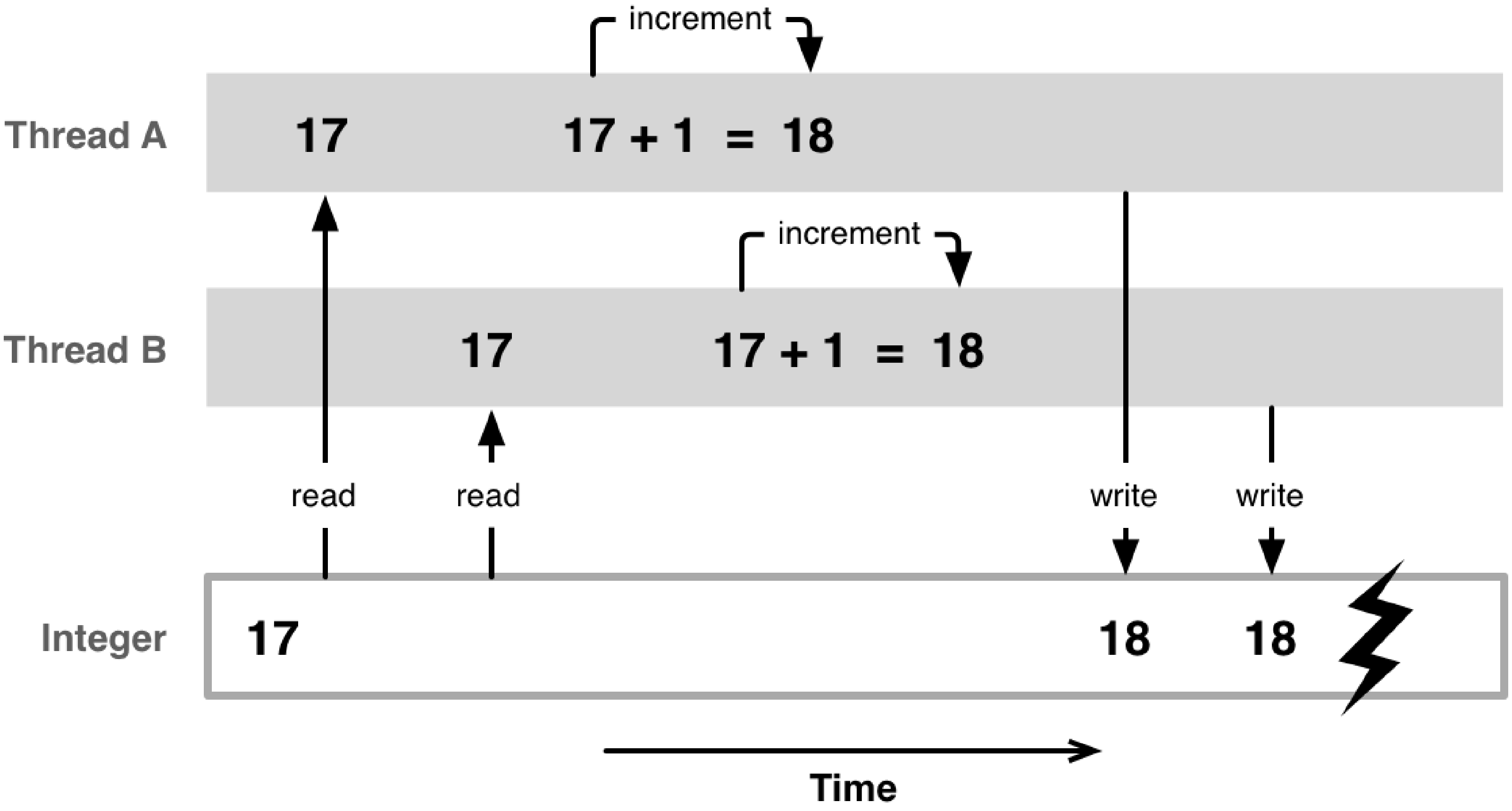


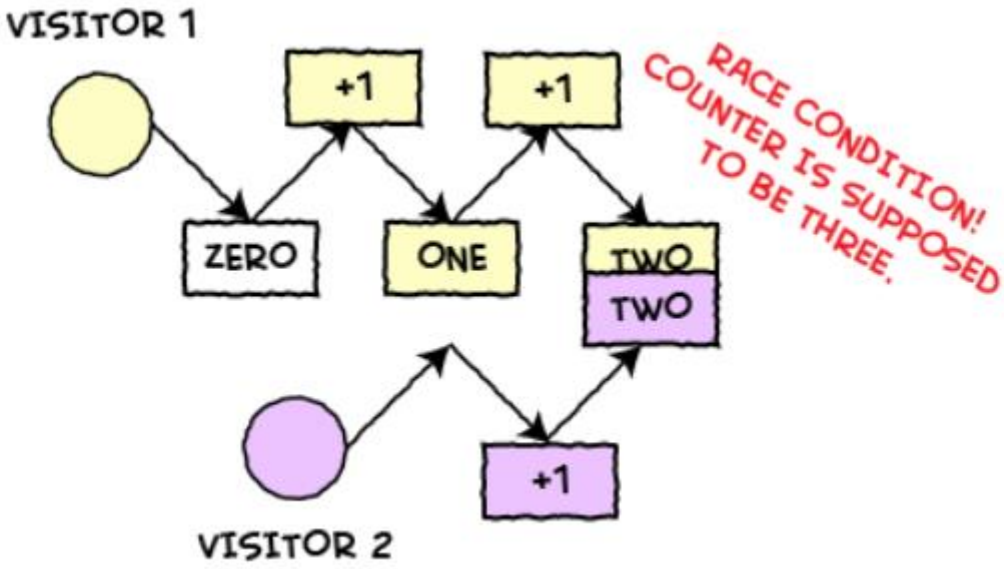
Philosophen Problem

- The dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them



Race Condition

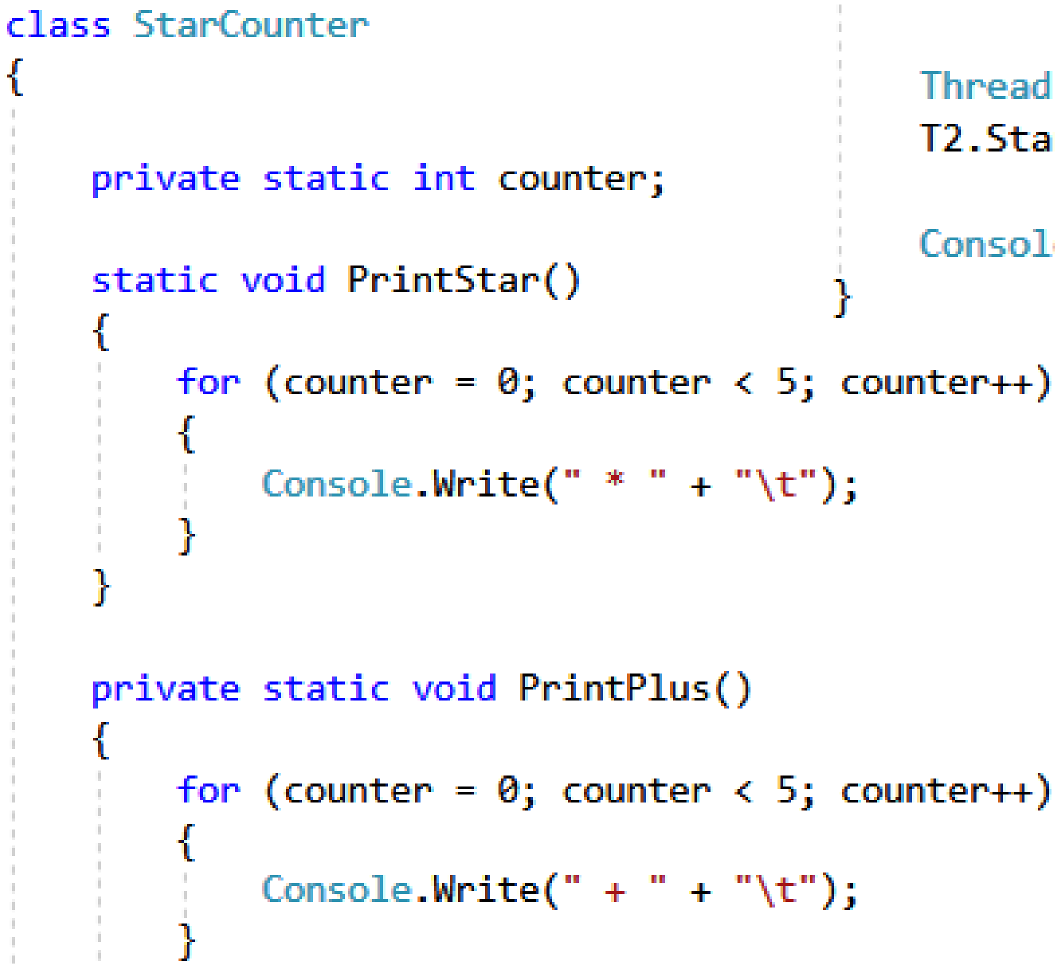




Race Condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time.

Race Condition
Star & Plus

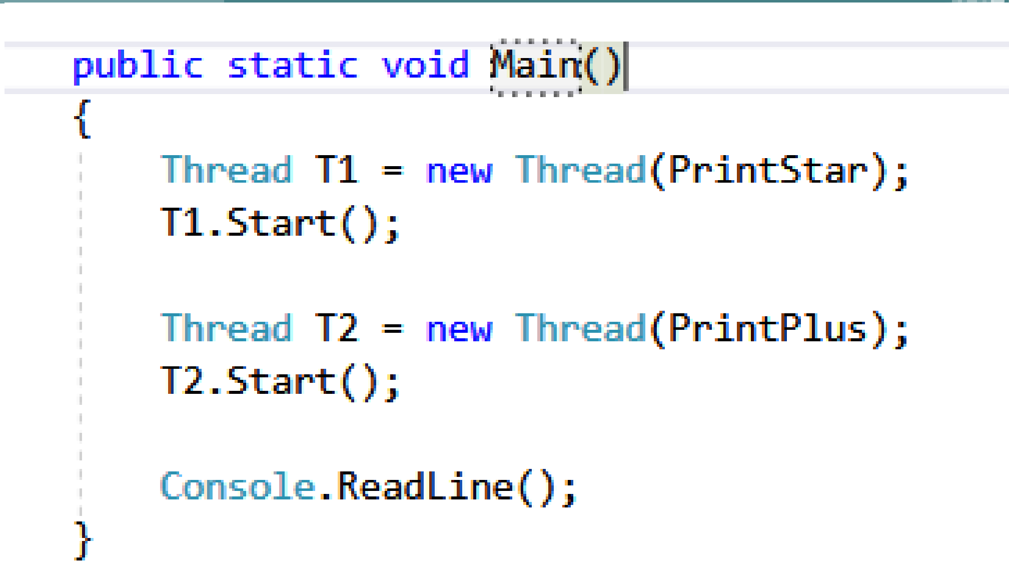


```
class StarCounter
{

    private static int counter;

    static void PrintStar()
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" * " + "\t");
        }
    }

    private static void PrintPlus()
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" + " + "\t");
        }
    }
```



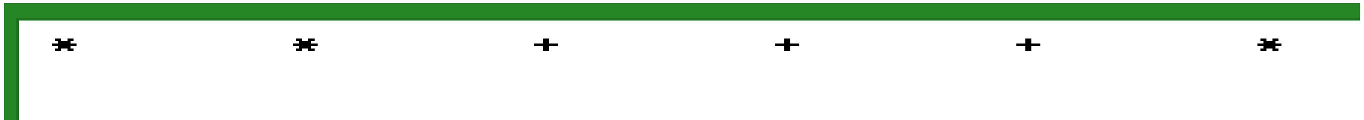
```
public static void Main()
{
    Thread T1 = new Thread(PrintStar);
    T1.Start();

    Thread T2 = new Thread(PrintPlus);
    T2.Start();

    Console.ReadLine();
}
```

The output can be any combination of * and +.

It will surely print characters [*, +], but order is inconsistent

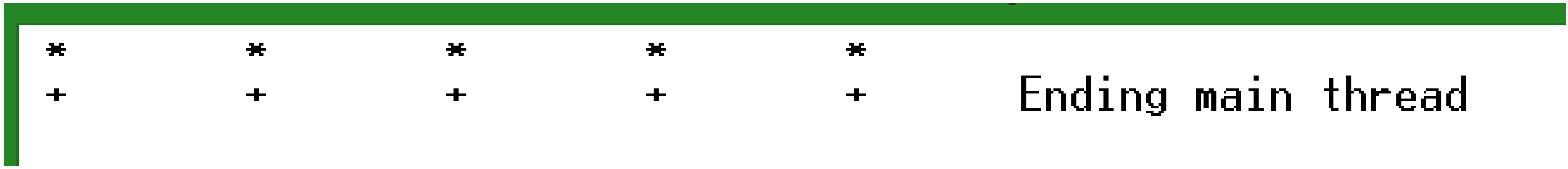


```
*          *          +          +          +          *
```

Thread.Join()

- **Synchronization using Thread.Join()**

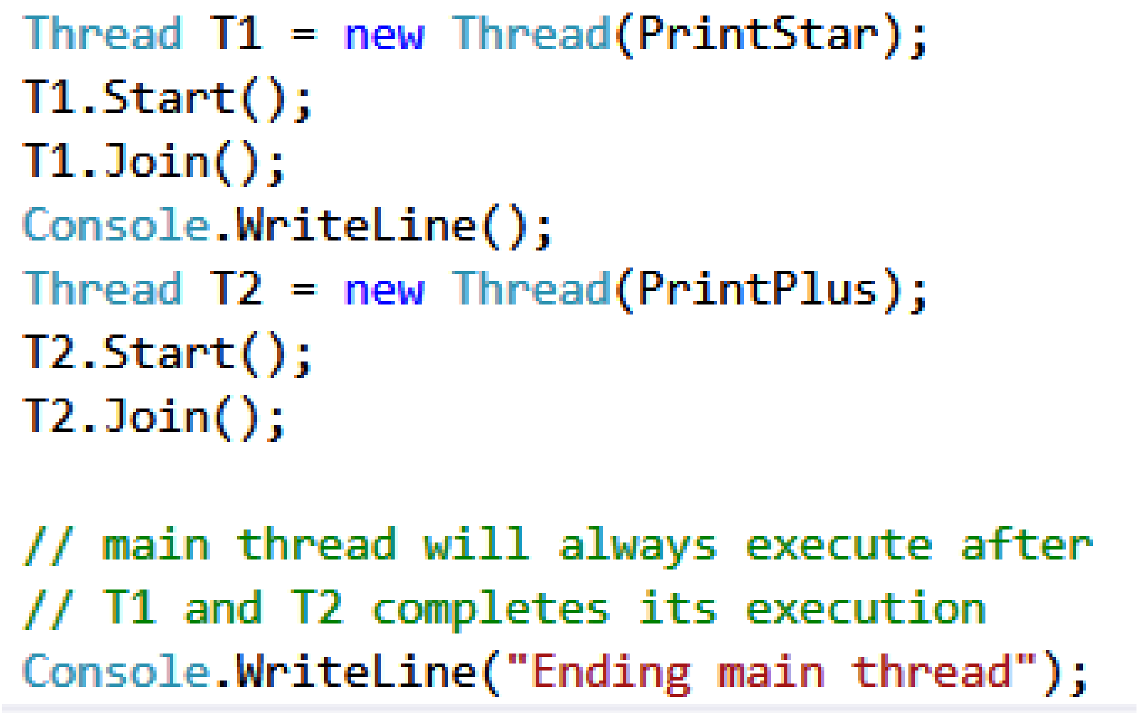
- Change the Main Method
  to get the following output:



```
*       *       *       *       *
+       +       +       +       +     Ending main thread
```

- Join allows one thread to wait
  for the completion of another
  - If t is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until t's thread terminates.

Synchronization mit **Thread.Join()**



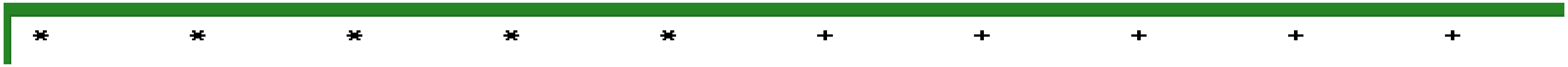
```
Thread T1 = new Thread(PrintStar);
T1.Start();
T1.Join();
Console.WriteLine();
Thread T2 = new Thread(PrintPlus);
T2.Start();
T2.Join();

// main thread will always execute after
// T1 and T2 completes its execution
Console.WriteLine("Ending main thread");
```

Synchronization with **lock(){…}**

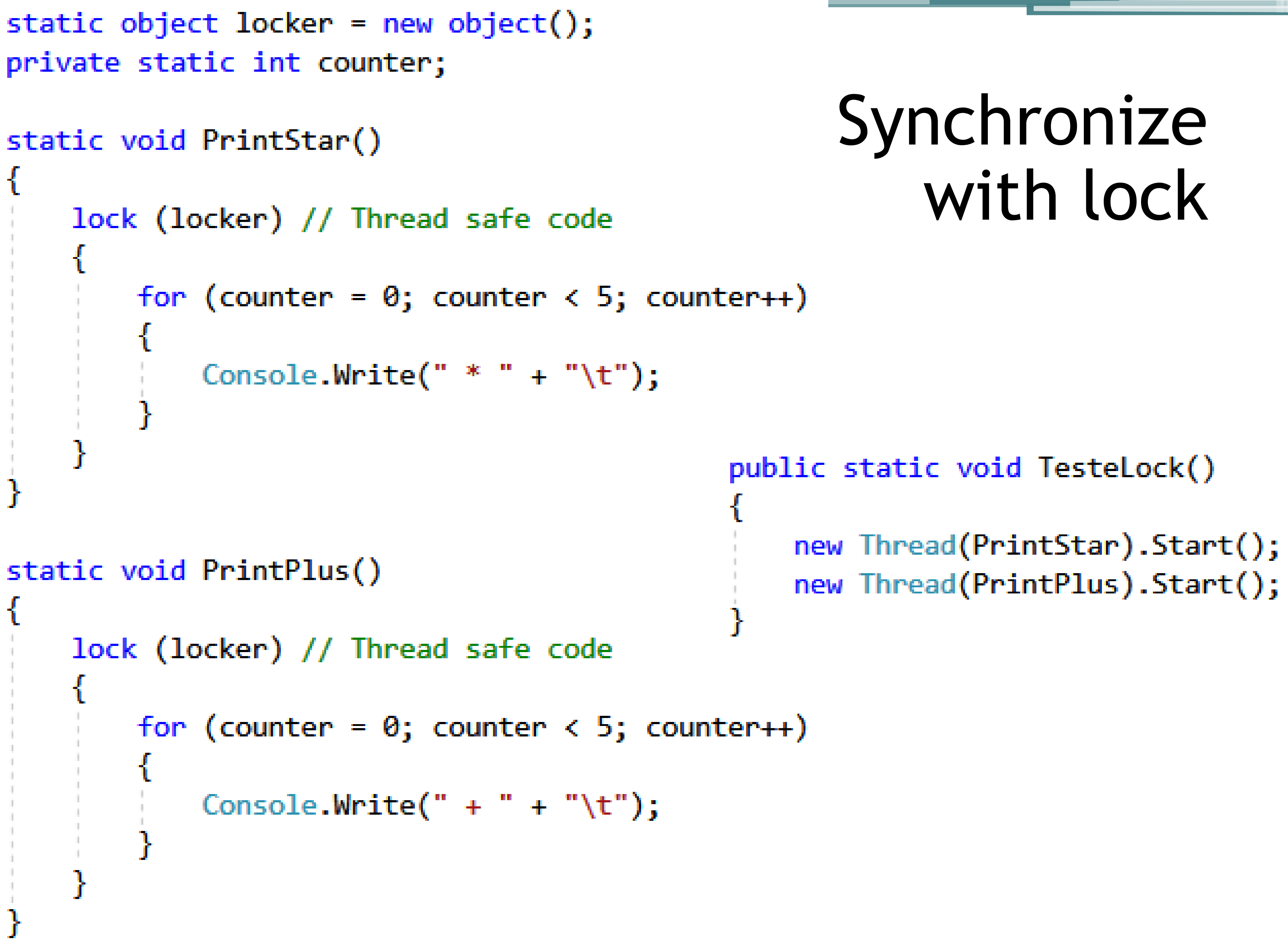
- Lock ensures only one thread
  can be executed at any point of time
- Syntax:
  - `lock(expression) { statement_block }`

- Synchronise the example with „lock“,
  to create the output below



```
*    *    *    *    *    +    +    +    +    +
```

Synchronize with lock



```
static object locker = new object();
private static int counter;

static void PrintStar()
{
    lock (locker) // Thread safe code
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" * " + "\t");
        }
    }
}

static void PrintPlus()
{
    lock (locker) // Thread safe code
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" + " + "\t");
        }
    }
}
```

```
public static void TesteLock()
{
    new Thread(PrintStar).Start();
    new Thread(PrintPlus).Start();
}
```

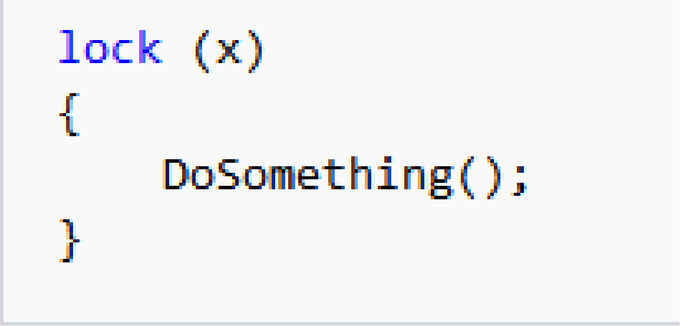
Synchronization with **Monitor**

- monitors prevent blocks of code from simultaneous execution by multiple threads
  - Enter method allows one and only one thread to proceed into the following statements
  - all other threads are blocked
    until the executing thread calls Exit

  - This is just like using the lock keyword.

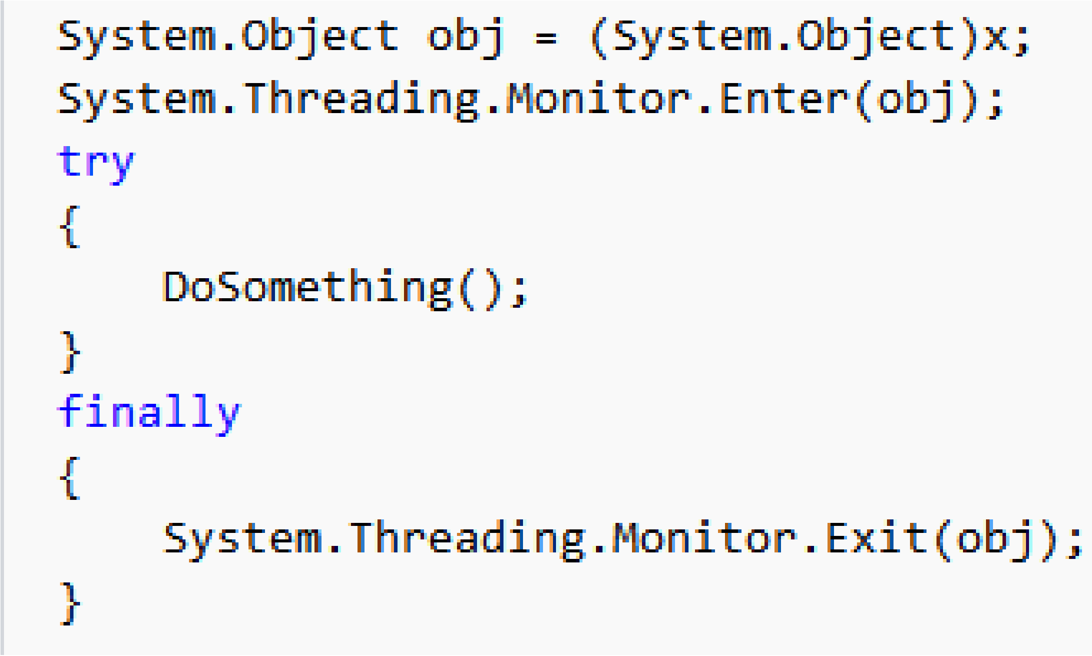
  - Use the Monitor to solve the Race Condition

Lock uses a Monitor



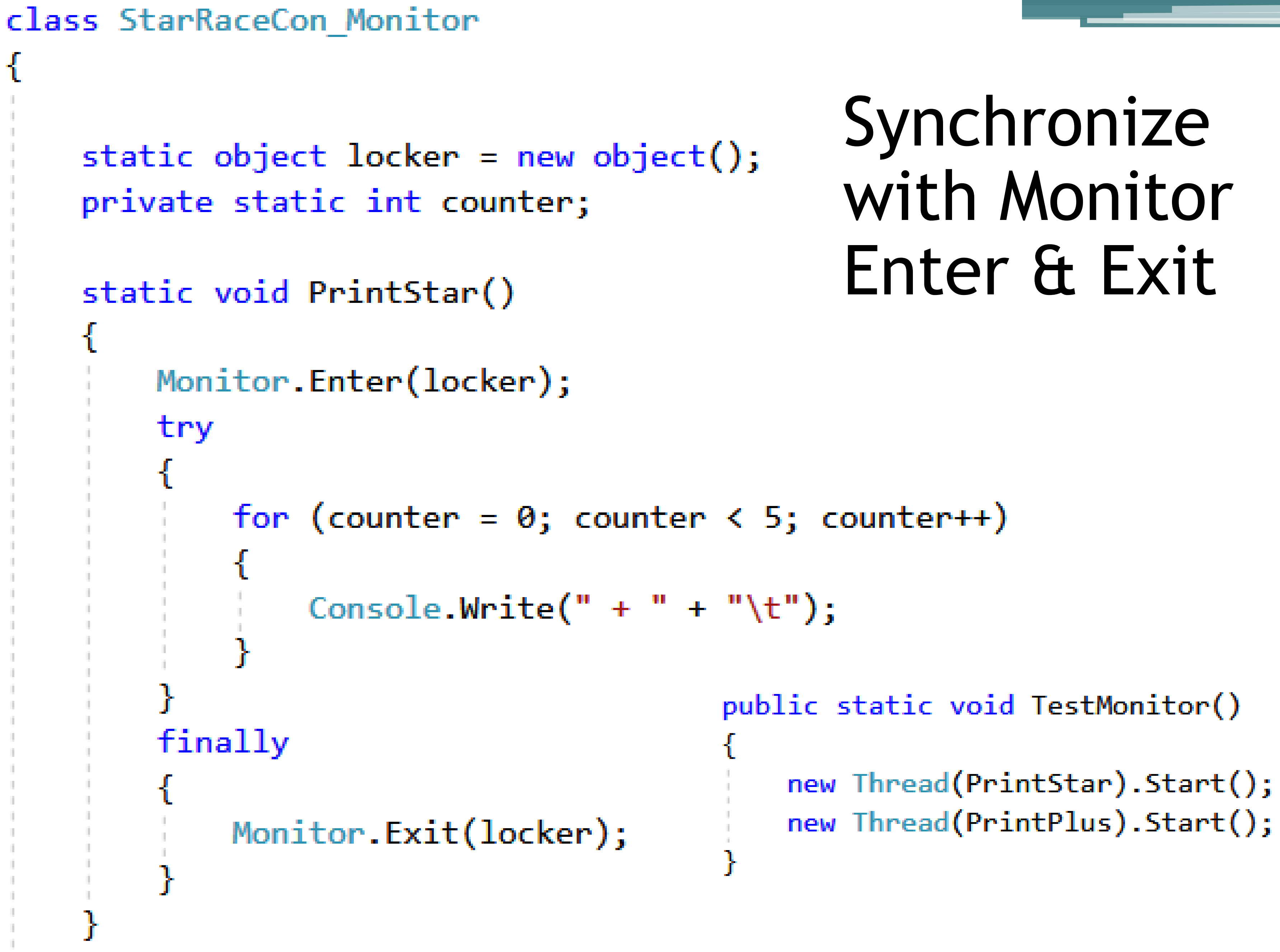
```
lock (x)
{
    DoSomething();
}
```

- This is equivalent to:



```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
    DoSomething();
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```

Synchronize with Monitor Enter & Exit

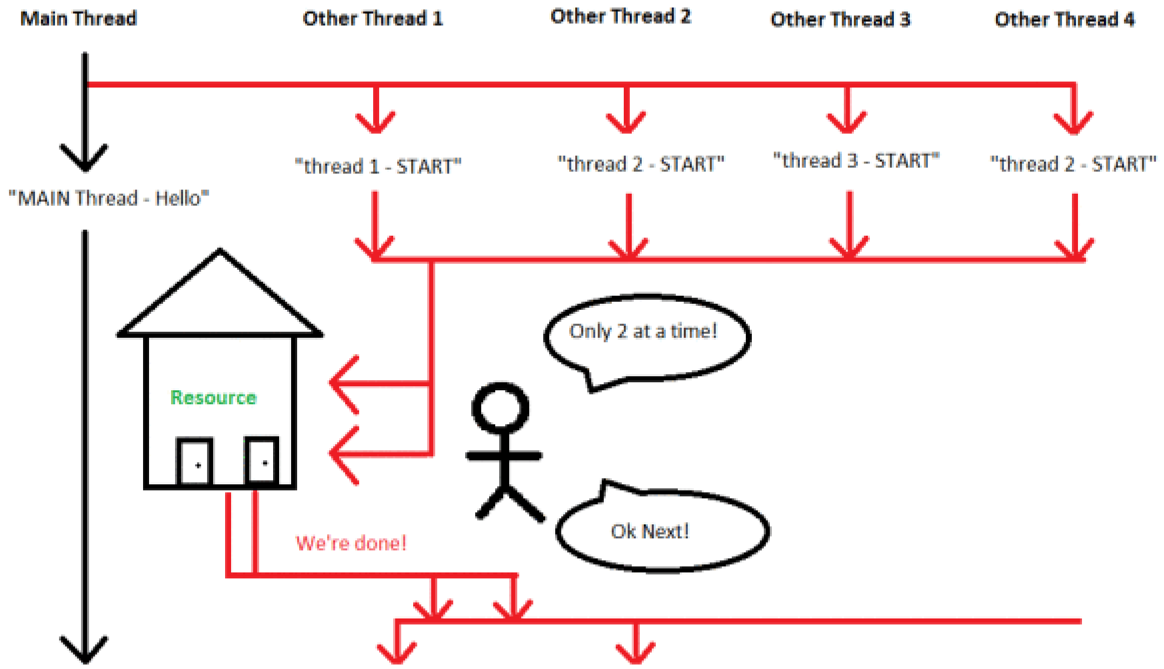


```
class StarRaceCon_Monitor
{

    static object locker = new object();
    private static int counter;

    static void PrintStar()
    {
        Monitor.Enter(locker);
        try
        {
            for (counter = 0; counter < 5; counter++)
            {
                Console.Write(" + " + "\t");
            }
        }
        finally
        {
            Monitor.Exit(locker);
        }
    }
```

```
public static void TestMonitor()
{
    new Thread(PrintStar).Start();
    new Thread(PrintPlus).Start();
}
```

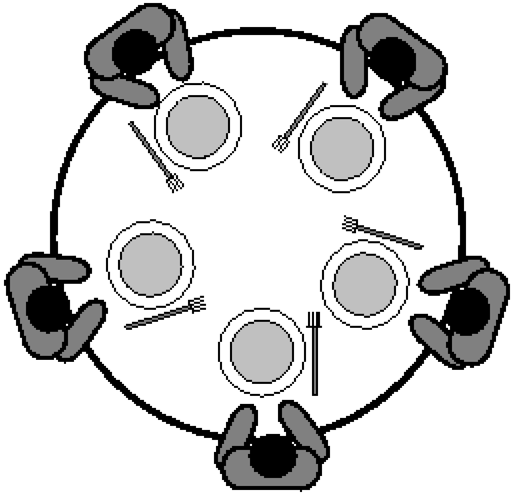


Semaphore

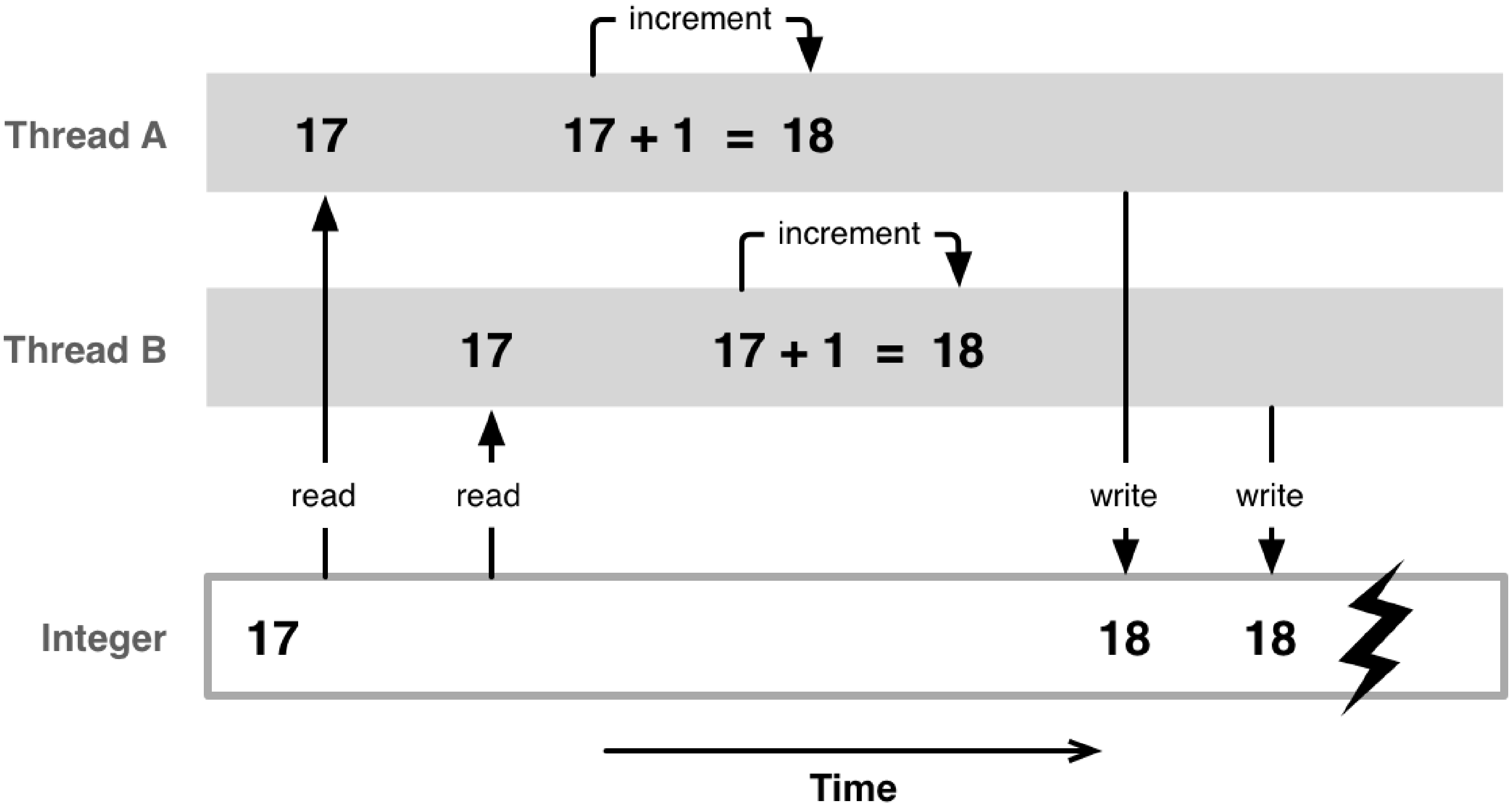
You can set that only a certain number of threads accesses a resource at the same time

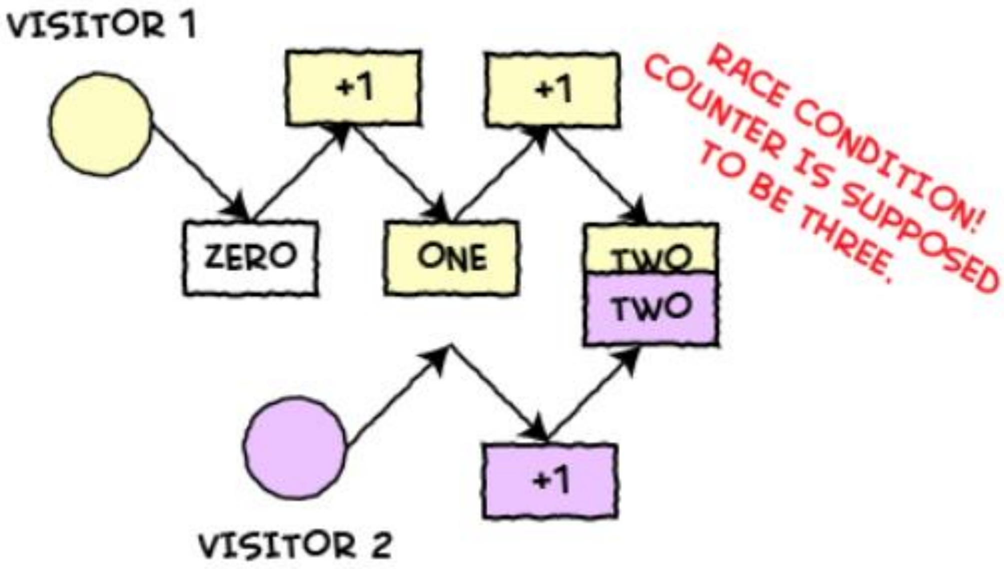
Philosophen Problem

- The dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them



Race Condition

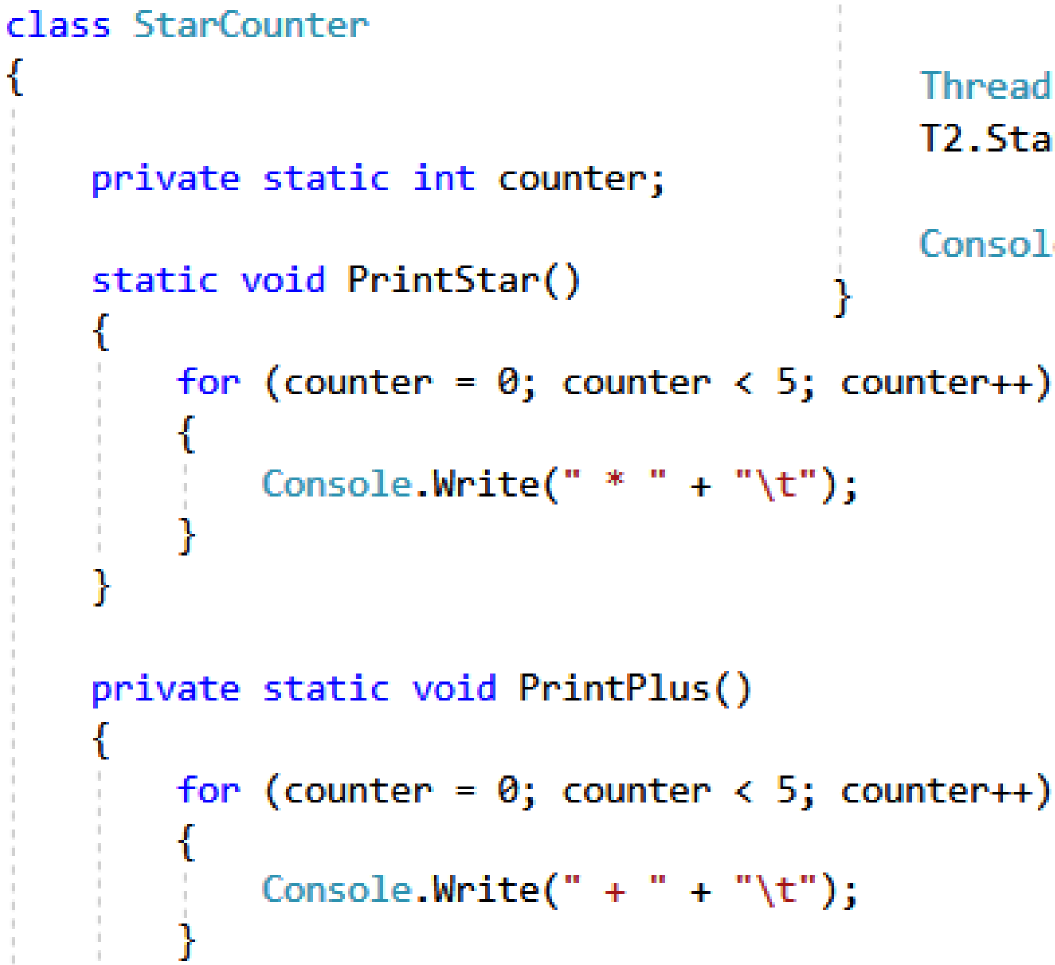




Race Condition

A race condition occurs when two or more threads can access shared data and they try to change it at the same time.

Race Condition Star & Plus

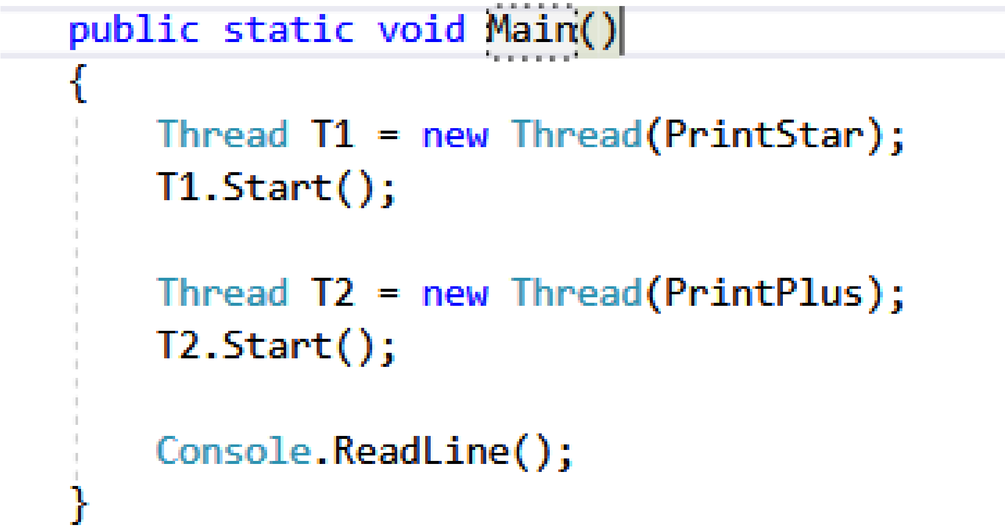


```
class StarCounter
{

    private static int counter;

    static void PrintStar()
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" * " + "\t");
        }
    }

    private static void PrintPlus()
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" + " + "\t");
        }
    }
```



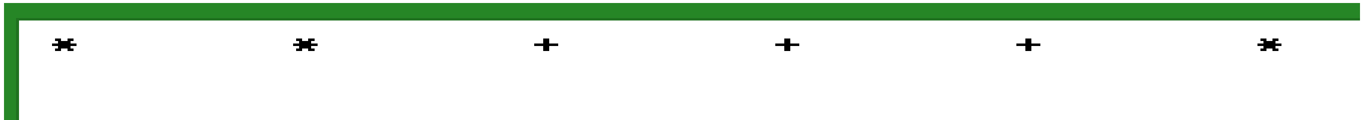
```
public static void Main()
{
    Thread T1 = new Thread(PrintStar);
    T1.Start();

    Thread T2 = new Thread(PrintPlus);
    T2.Start();

    Console.ReadLine();
}
```

The output can be any combination of * and +.

It will surely print characters [*, +], but order is inconsistent

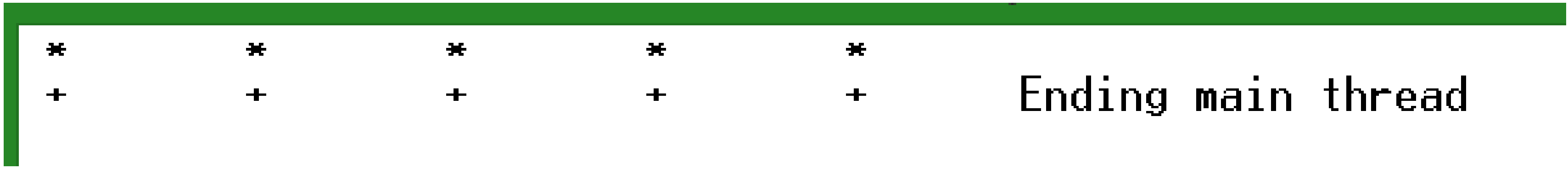


```
*       *       +       +       +       *
```

Thread.Join()

- **Synchronization using Thread.Join()**

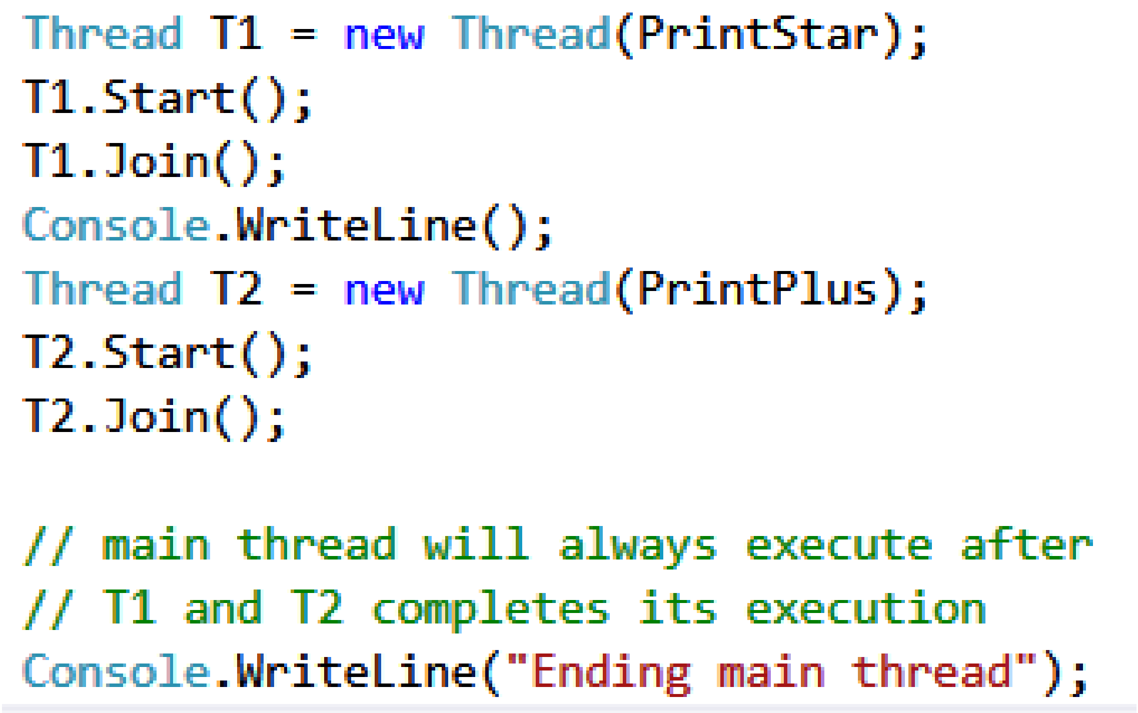
- Change the Main Method
  to get the following output:



```
*       *       *       *       *
+       +       +       +       +       Ending main thread
```

- Join allows one thread to wait
  for the completion of another
  - If t is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until t's thread terminates.

Synchronization mit **Thread.Join()**



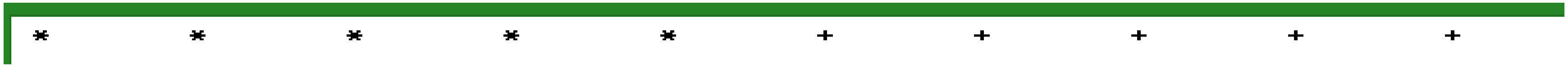
```
Thread T1 = new Thread(PrintStar);
T1.Start();
T1.Join();
Console.WriteLine();
Thread T2 = new Thread(PrintPlus);
T2.Start();
T2.Join();

// main thread will always execute after
// T1 and T2 completes its execution
Console.WriteLine("Ending main thread");
```

Synchronization with **lock(){…}**

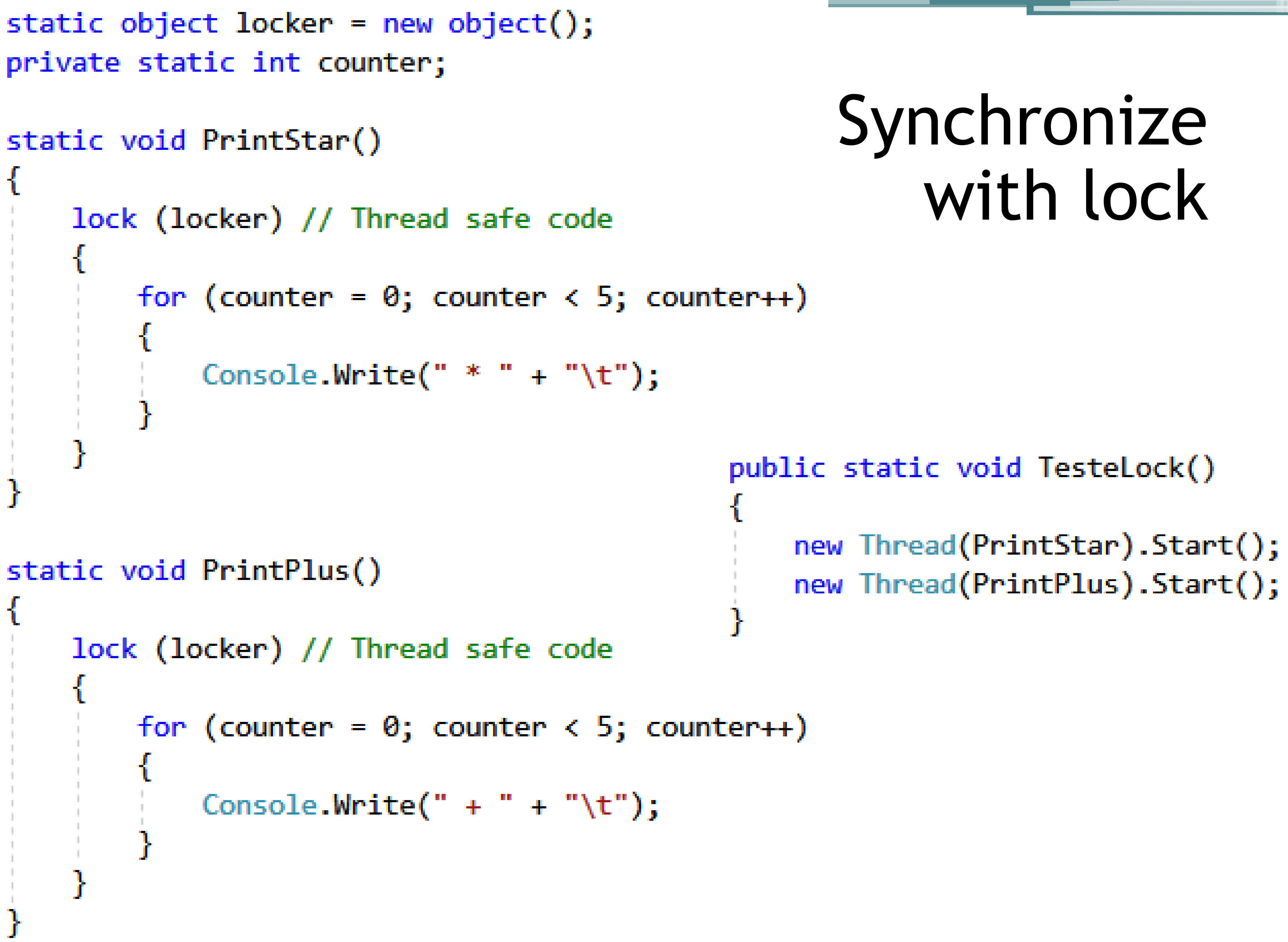
- Lock ensures only one thread can be executed at any point of time
- Syntax:
    - `lock(expression) { statement_block }`

- Synchronise the example with „lock“, to create the output below



```
*    *    *    *    *    +    +    +    +    +
```

Synchronize with lock



```
static object locker = new object();
private static int counter;

static void PrintStar()
{
    lock (locker) // Thread safe code
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" * " + "\t");
        }
    }
}

static void PrintPlus()
{
    lock (locker) // Thread safe code
    {
        for (counter = 0; counter < 5; counter++)
        {
            Console.Write(" + " + "\t");
        }
    }
}
```

```
public static void TesteLock()
{
    new Thread(PrintStar).Start();
    new Thread(PrintPlus).Start();
}
```

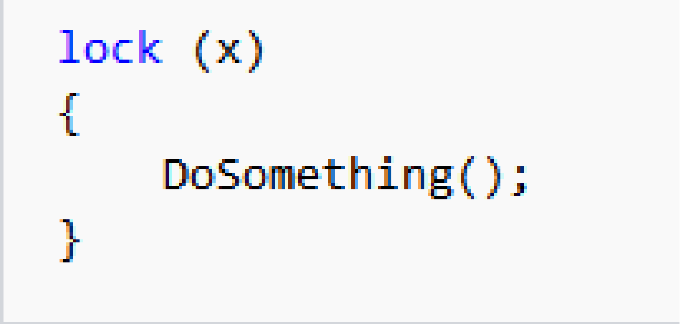
Synchronization with **Monitor**

- monitors prevent blocks of code from simultaneous execution by multiple threads
  - Enter method allows one and only one thread to proceed into the following statements
  - all other threads are blocked
    until the executing thread calls Exit

  - This is just like using the lock keyword.

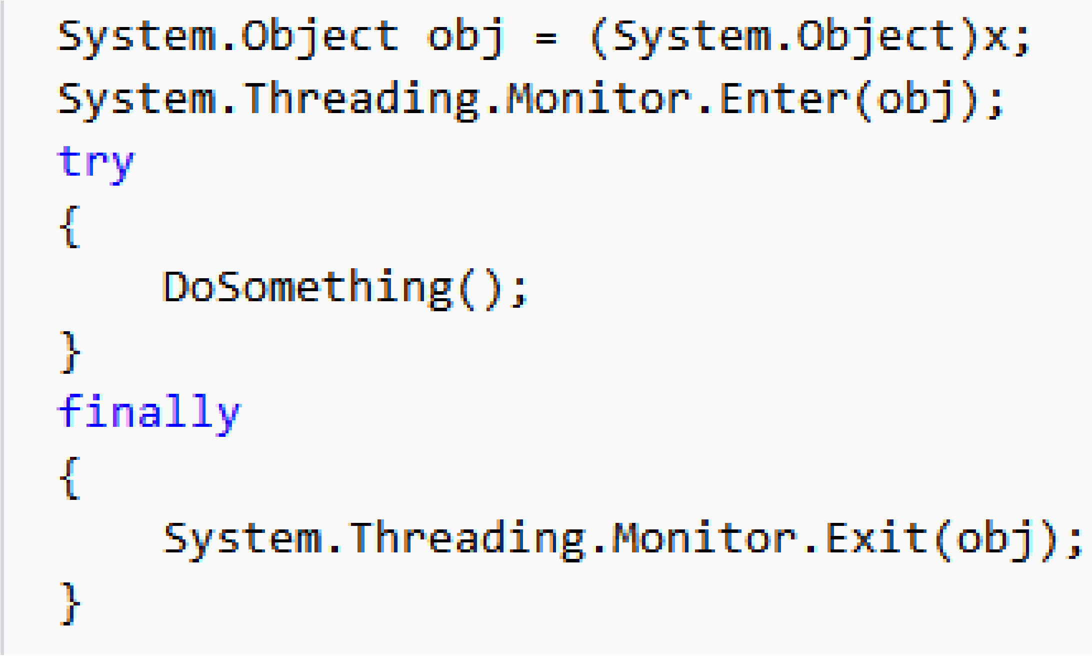
  - Use the Monitor to solve the Race Condition

Lock uses a Monitor



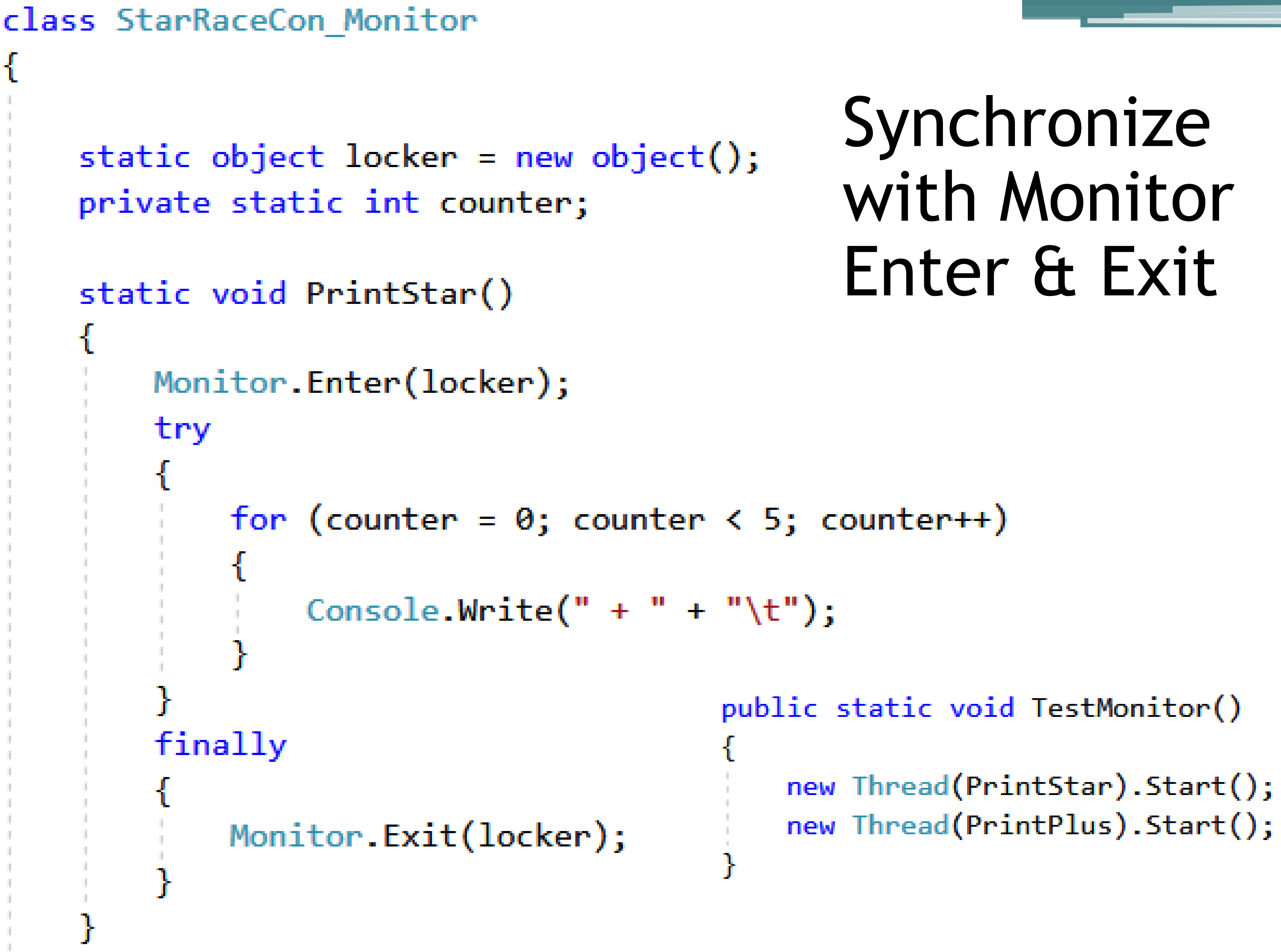
```
lock (x)
{
    DoSomething();
}
```

- This is equivalent to:



```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
    DoSomething();
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```

Synchronize with Monitor Enter & Exit

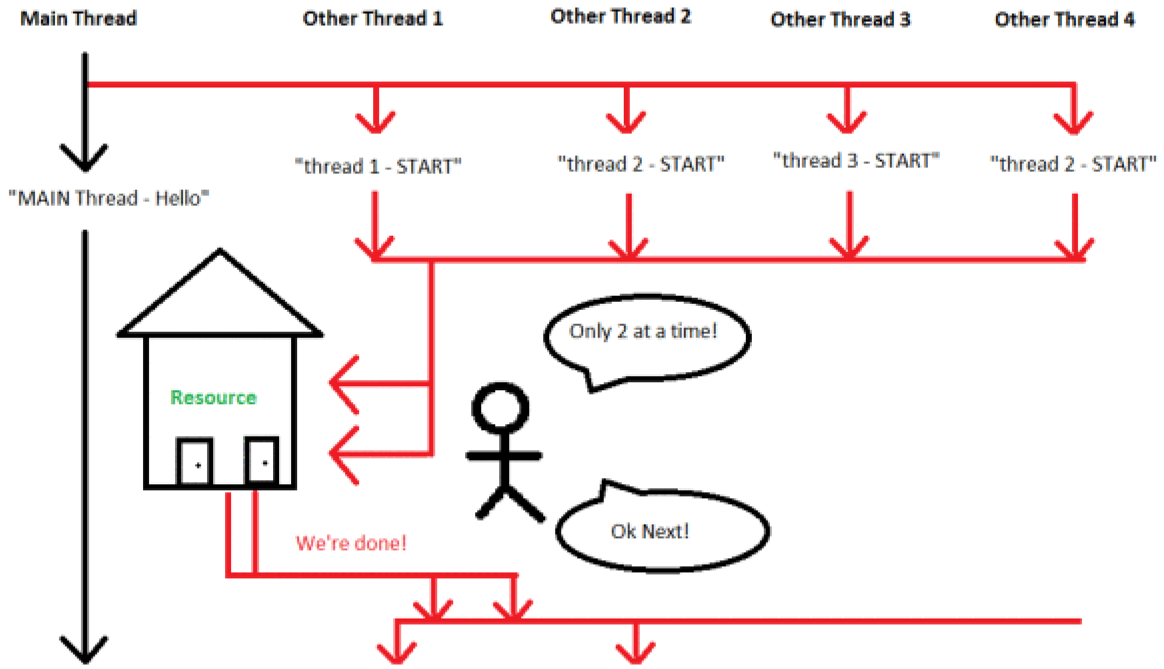


```
class StarRaceCon_Monitor
{

    static object locker = new object();
    private static int counter;

    static void PrintStar()
    {
        Monitor.Enter(locker);
        try
        {
            for (counter = 0; counter < 5; counter++)
            {
                Console.Write(" + " + "\t");
            }
        }
        finally
        {
            Monitor.Exit(locker);
        }
    }
```

```
public static void TestMonitor()
{
    new Thread(PrintStar).Start();
    new Thread(PrintPlus).Start();
}
```



Semaphore

You can set that only a certain number of threads accesses a resource at the same time

Semaphores

- can be useful in limiting concurrency
- preventing too many threads
  from executing a particular piece of code at once

- **Example:**
  Programm a Nightclub, five threads try to enter the nightclub that allows only three threads in at once.

Syntax in C#

- In the namespace System.Threading
- Syntax:

```
Semaphore semaphoreObject = new Semaphore(2, 2);
```

- Intitialized with 2 parameter:
  - Red one defines how many threads can enter the semaphore
  - Green one defines how many threads are in the semaphore

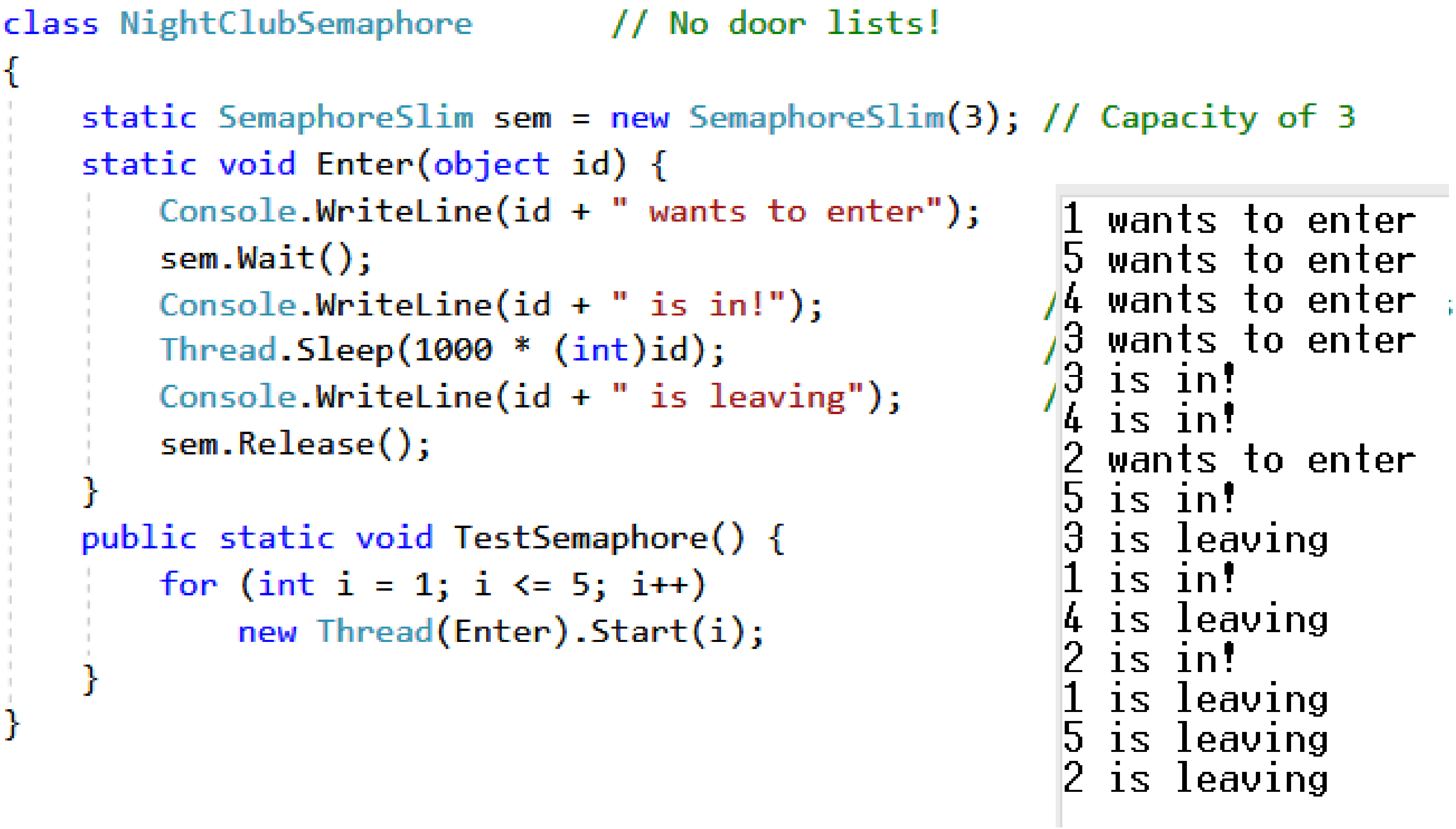
WaitOne Method

- Syntax:
  - semaphoreObject.WaitOne();
- If the sempahore isn´t full, the thread enters
  - Decreases the counter by 1
- Else, the thread waits until he can enter

Release Method

- Syntax:
  - semaphoreObject.Release();
- The calling thread releases
  - Inceaseses the counter by 1
- Other thread can enter semaphore

SemaphoreSlim



```
class NightClubSemaphore          // No door lists!
{
    static SemaphoreSlim sem = new SemaphoreSlim(3); // Capacity of 3
    static void Enter(object id) {
        Console.WriteLine(id + " wants to enter");
        sem.Wait();
        Console.WriteLine(id + " is in!");              /
        Thread.Sleep(1000 * (int)id);                   /
        Console.WriteLine(id + " is leaving");          /
        sem.Release();
    }
    public static void TestSemaphore() {
        for (int i = 1; i <= 5; i++)
            new Thread(Enter).Start(i);
    }
}
```

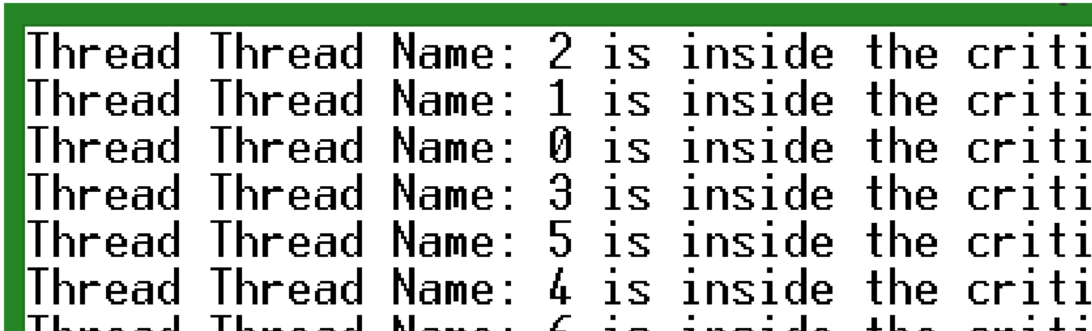
```
1 wants to enter
5 wants to enter
4 wants to enter
3 wants to enter
3 is in!
4 is in!
2 wants to enter
5 is in!
3 is leaving
1 is in!
4 is leaving
2 is in!
1 is leaving
5 is leaving
2 is leaving
```

Semaphore

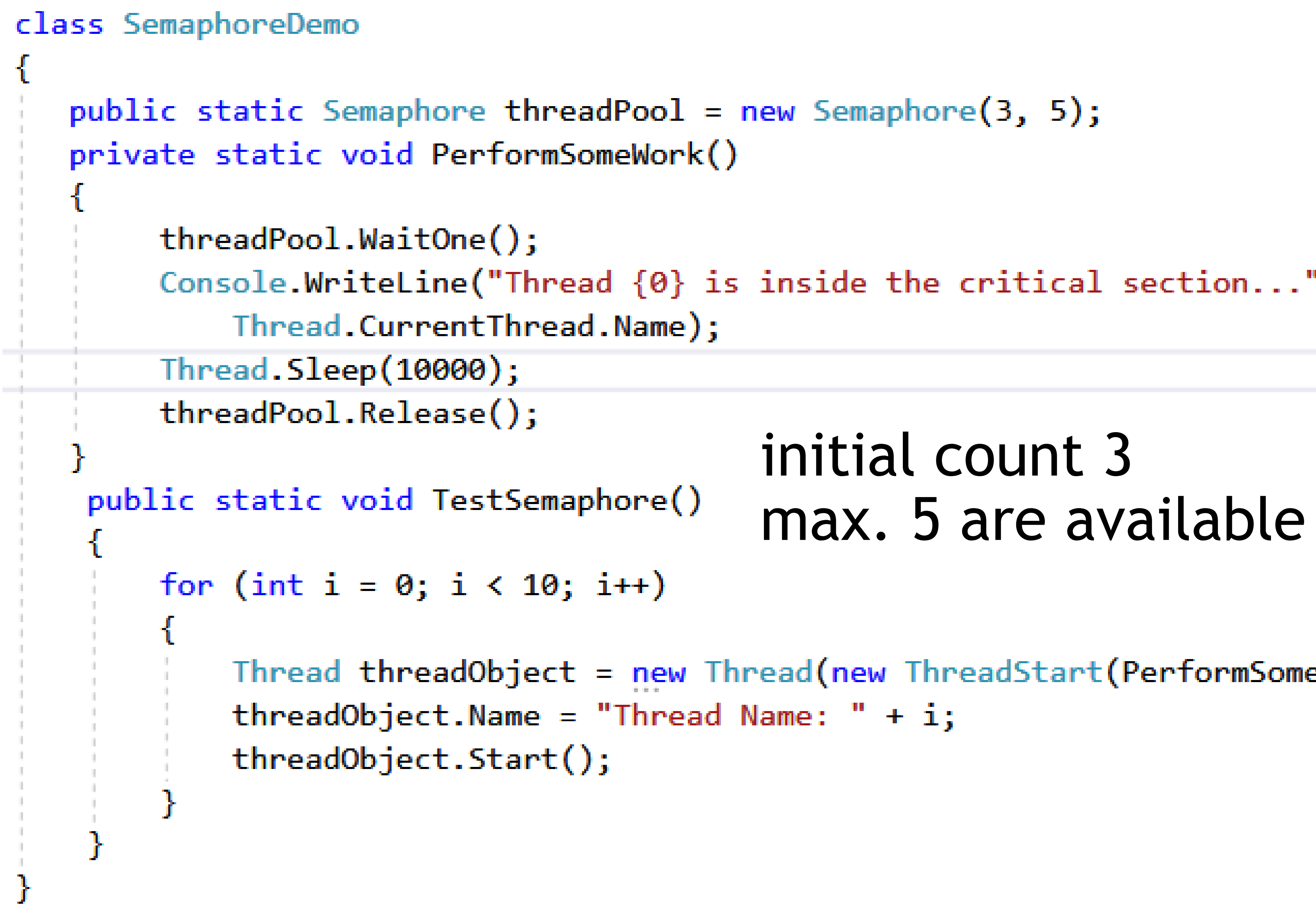
```
public static Semaphore threadPool =
new Semaphore(3, 5);
```

- creates a semaphore object named threadPool
- can support a maximum of 5 concurrent requests
- initial count is set to 3 as indicated in the first parameter to the constructor
- implies that 2 slots are reserved for the current thread and 3 slots are available for other threads

- Write some code!



```
Thread Thread Name: 2 is inside the criti
Thread Thread Name: 1 is inside the criti
Thread Thread Name: 0 is inside the criti
Thread Thread Name: 3 is inside the criti
Thread Thread Name: 5 is inside the criti
Thread Thread Name: 4 is inside the criti
Thread Thread Name: 6 is inside the criti
```



```
class SemaphoreDemo
{
    public static Semaphore threadPool = new Semaphore(3, 5);
    private static void PerformSomeWork()
    {
        threadPool.WaitOne();
        Console.WriteLine("Thread {0} is inside the critical section..."
            Thread.CurrentThread.Name);
        Thread.Sleep(10000);
        threadPool.Release();
    }
     public static void TestSemaphore()
     {
        for (int i = 0; i < 10; i++)
        {
            Thread threadObject = new Thread(new ThreadStart(PerformSome
            threadObject.Name = "Thread Name: " + i;
            threadObject.Start();
        }
     }
}
```

initial count 3
max. 5 are available