Thread synchronization pt. 2

AARHUS UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

MICHAEL LOFT
ML@ASE.AU.DK

Agenda

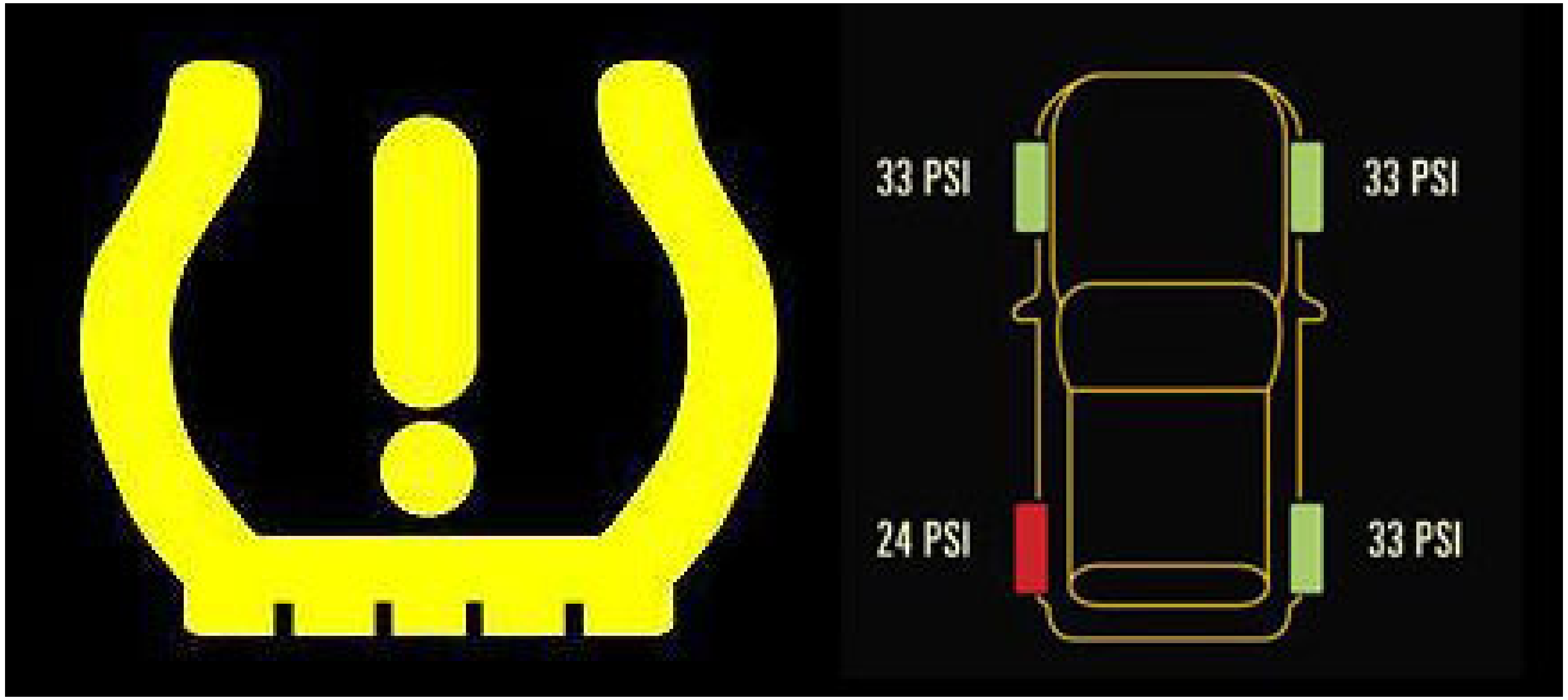
Producer-Consumer

Thread synchronization

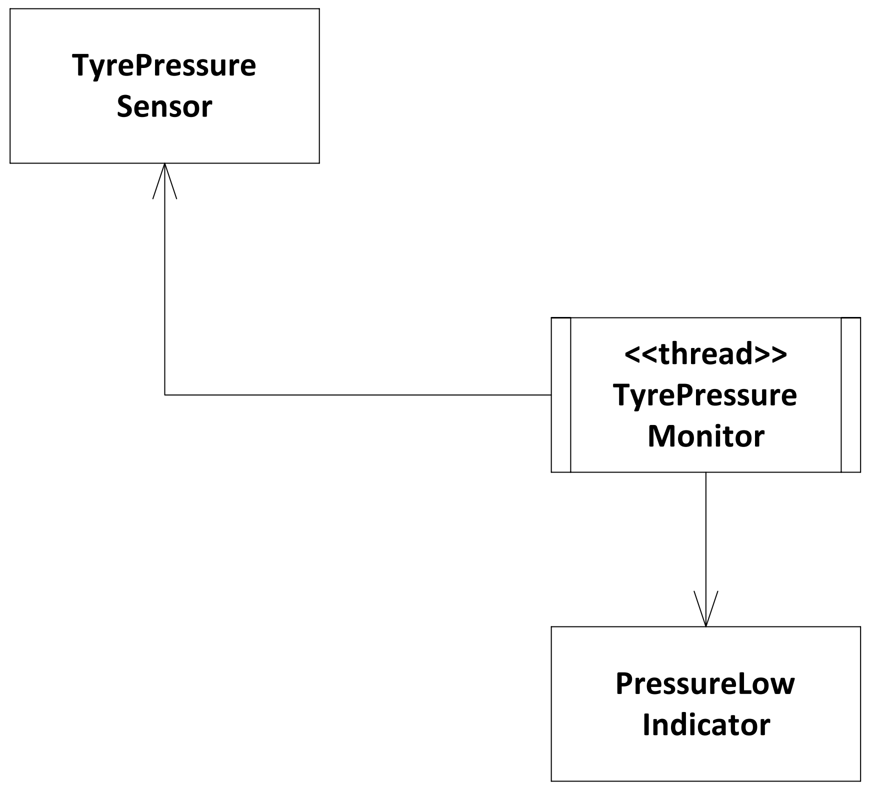
- AutoResetEvent
- ManualResetEvent

Queues and BlockingCollection

Tyre Pressure Monitor System



TPMS design



TyrePressureMonitor has many responsibilities:

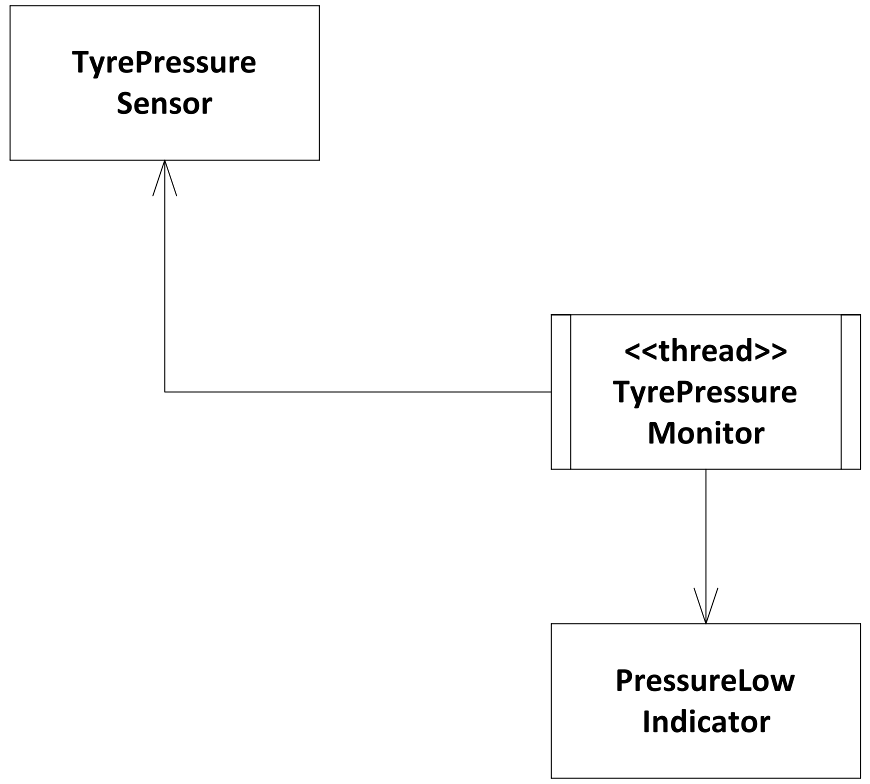
- Read the pressure using the TyrePressureSensor.
- Determine if the pressure is too low.
- Turn on/off the PressureLowIndicator

Design principle: Single Responsibility

THERE SHOULD NEVER BE
MORE THAN ONE REASON
FOR A CLASS TO CHANGE

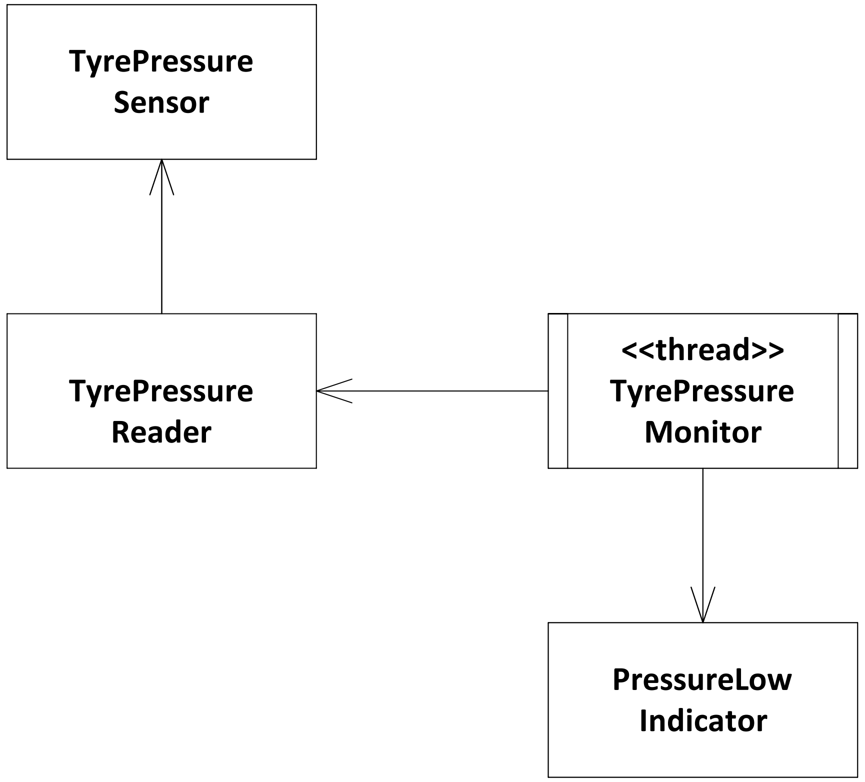
Robert Martin: The Single Responsibility Principle

TPMS design



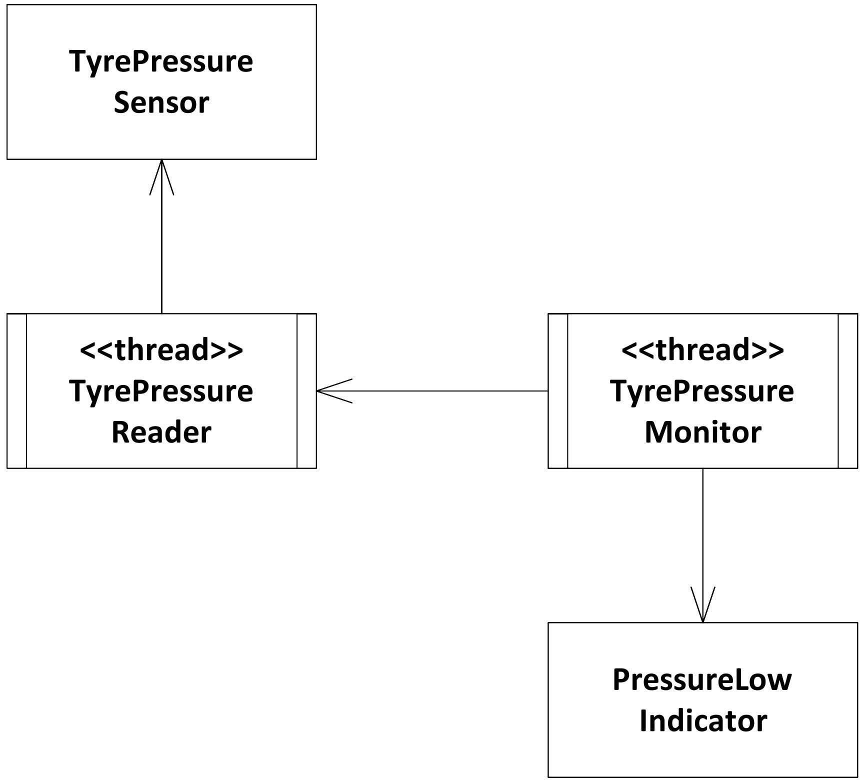
TyrePressureMonitor has many responsibilities:

- Read the pressure using the TyrePressureSensor.
- Determine if the pressure is too low.
- Turn on/off the PressureLowIndicator



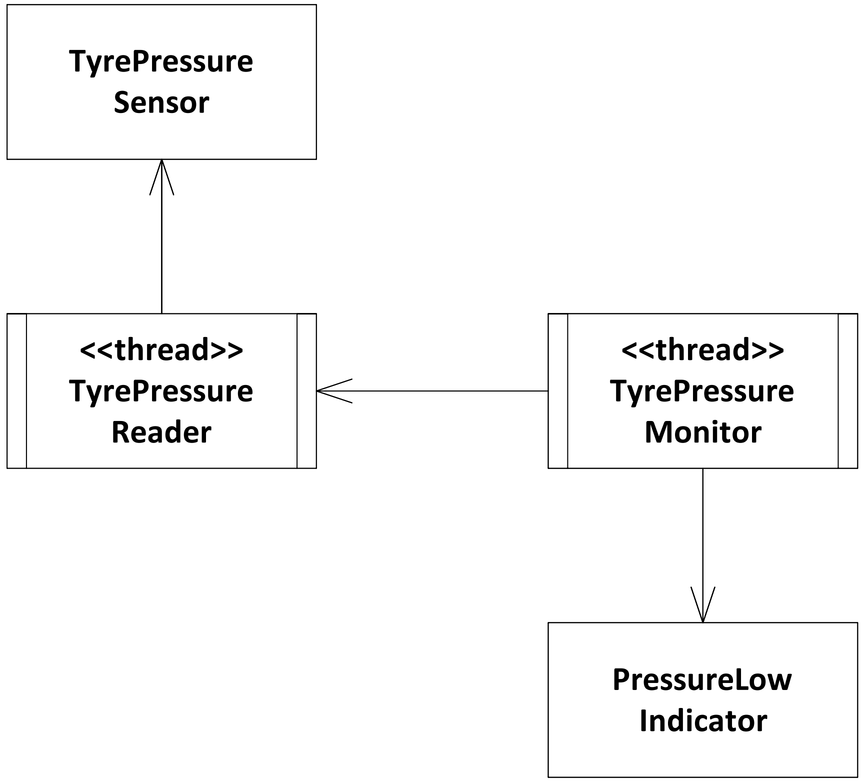
Now, reading the pressure is separated out.

Wouldn’t it be nice, if the pressure monitor did not have to control when the pressure was read?


TyrePressure Sensor
<<thread>> TyrePressure Reader
<<thread>> TyrePressure Monitor
PressureLow Indicator

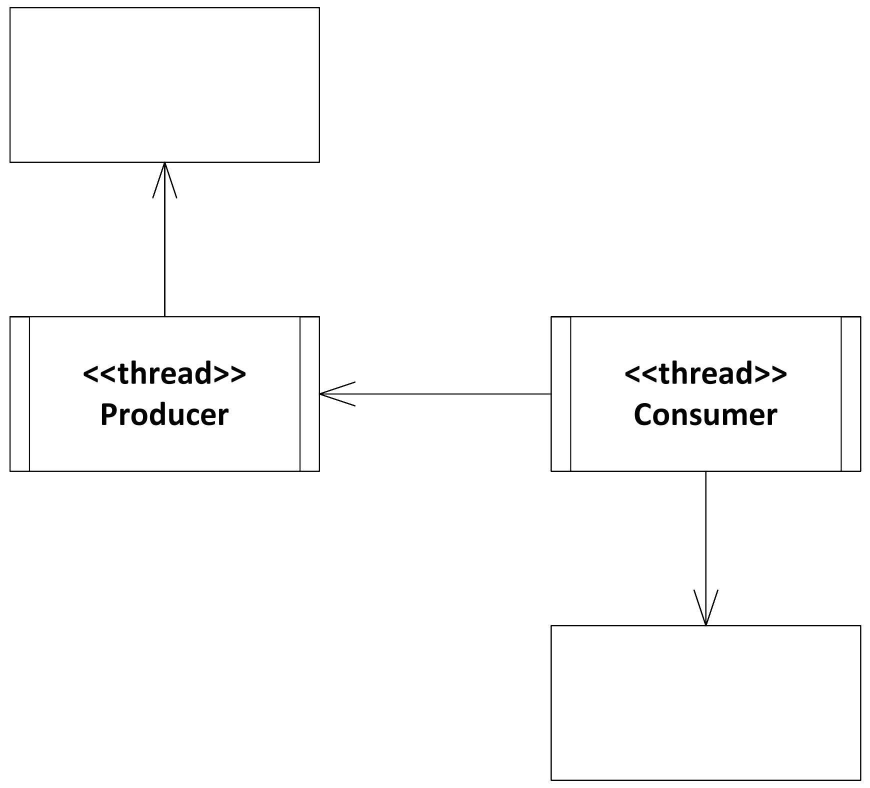
Let’s put the TyrePressureReader on a separate thread.

How does the TyrePressureMonitor know, when a new reading has taken place?


TyrePressure Sensor
<<thread>> TyrePressure Reader
<<thread>> TyrePressure Monitor
PressureLow Indicator

The Monitor consumes **data**, which the Reader provides.

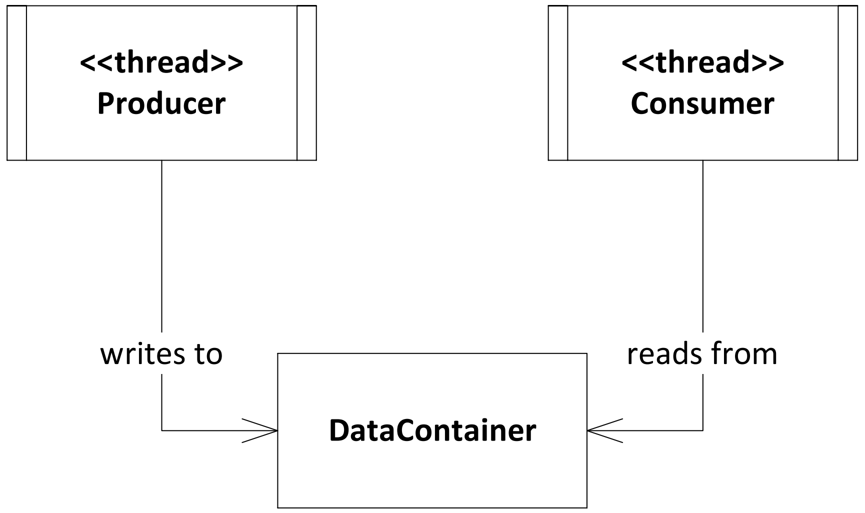
Producer - Consumer



The Monitor consumes **data**, which the Reader provides.

This is a very common design: Producer – Consumer.

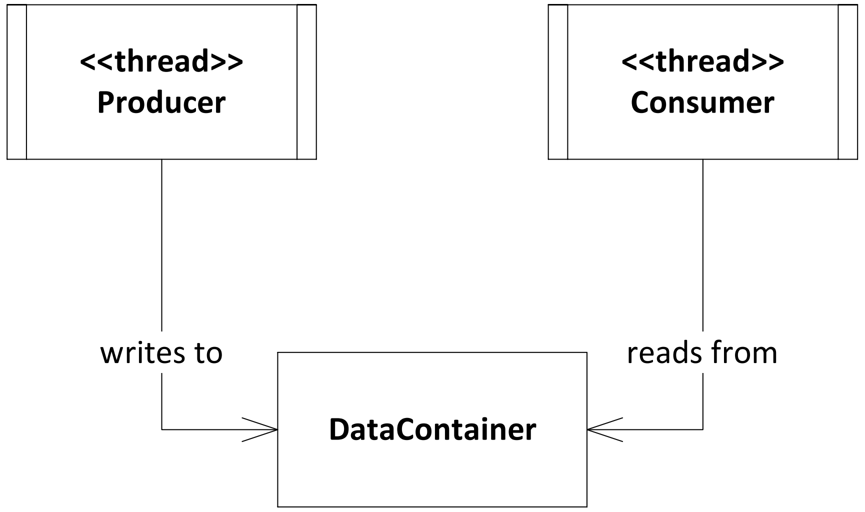
Producer - Consumer



The Consumer consumes **data**, which the Provider provides.

Let's put that data into another object, so the Consumer thread does not have to know the Producer thread.

Producer - Consumer



The Consumer would like to know when new data is available.

The Producer would like to know, if the data has been consumed, so it can provide a new value.

Thread synchronization

AutoResetEvent and ManualResetEvent

Event handles can be used to signal from one thread to another.

AutoResetEvent changes from signaled to unsignaled automatically any time it activates a thread.

ManualResetEvent allows any number of threads to be activated by its signaled state, and will only revert to an unsignaled state when its Reset method is called.

DataContainer

```
class DataContainer
{
    private int tyrePressure;

    public int GetTyrePressure()
    {
        return tyrePressure;
    }

    public void SetTyrePressure(int value)
    {
        tyrePressure = value;
    }
}
```

Objects of the DataContainer class is used to pass data from producer to consumer.

Producer

```
class Producer
{
    private readonly DataContainer _dataContainer;
    private readonly AutoResetEvent _dataConsumedEvent;
    private readonly AutoResetEvent _dataReadyEvent;
    private readonly Random _random = new Random();

    public Producer(DataContainer dataContainer,
                    AutoResetEvent dataConsumedEvent,
                    AutoResetEvent dataReadyEvent)
    {
        _dataContainer = dataContainer;
        _dataConsumedEvent = dataConsumedEvent;
        _dataReadyEvent = dataReadyEvent;
    }

    public void Run()
    {
        while (true)
        {
            _dataConsumedEvent.WaitOne();
            int pressure = _random.Next(0, 50);
            _dataContainer.SetTyrePressure(pressure);
            _dataReadyEvent.Set();
            Thread.Sleep(1000);
        }
    }
}
```

The Consumer would like to know when new data is available.

The Producer would like to know, if the data has been consumed, so it can provide a new value.

Consumer

```
class Consumer
{
    private readonly DataContainer _dataContainer;
    private readonly AutoResetEvent _dataReadyEvent;
    private readonly AutoResetEvent _dataConsumedEvent;

    public Consumer(DataContainer dataContainer,
                    AutoResetEvent dataReadyEvent,
                    AutoResetEvent dataConsumedEvent)
    {
        _dataContainer = dataContainer;
        _dataReadyEvent = dataReadyEvent;
        _dataConsumedEvent = dataConsumedEvent;
    }

    public void Run()
    {
        while (true)
        {
            _dataReadyEvent.WaitOne();
            int pressure = _dataContainer.GetTyrePressure();
            System.Console.WriteLine("Tyre pressure: {0}", pressure);
            _dataConsumedEvent.Set();
        }
    }
}
```

The Consumer would like to know when new data is available.

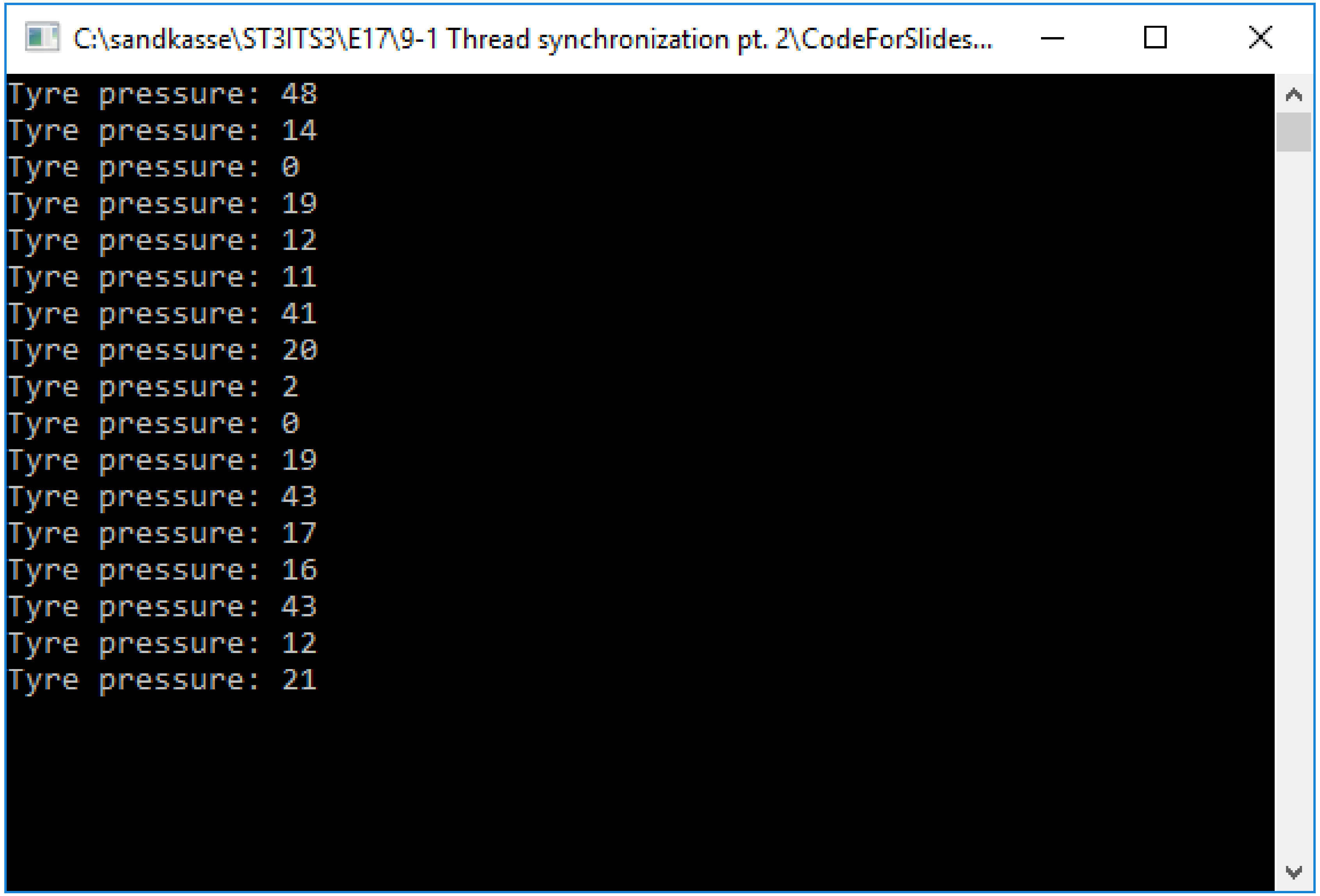
The Producer would like to know, if the data has been consumed, so it can provide a new value.

Producer – Consumer creation

```
class Program
{
    static void Main(string[] args)
    {
        AutoResetEvent dataConsumedEvent = new AutoResetEvent(true);
        AutoResetEvent dataReadyEvent = new AutoResetEvent(false);

        DataContainer dataContainer = new DataContainer();

        Producer producer = new Producer(dataContainer,
                                         dataConsumedEvent,
                                         dataReadyEvent);

        Consumer consumer = new Consumer(dataContainer,
                                         dataReadyEvent,
                                         dataConsumedEvent);

        Thread producerThread = new Thread(producer.Run);
        Thread consumerThread = new Thread(consumer.Run);

        producerThread.Start();
        consumerThread.Start();
    }
}
```

The AutoResetEvent can be initialized to be set (true) or not set (false).

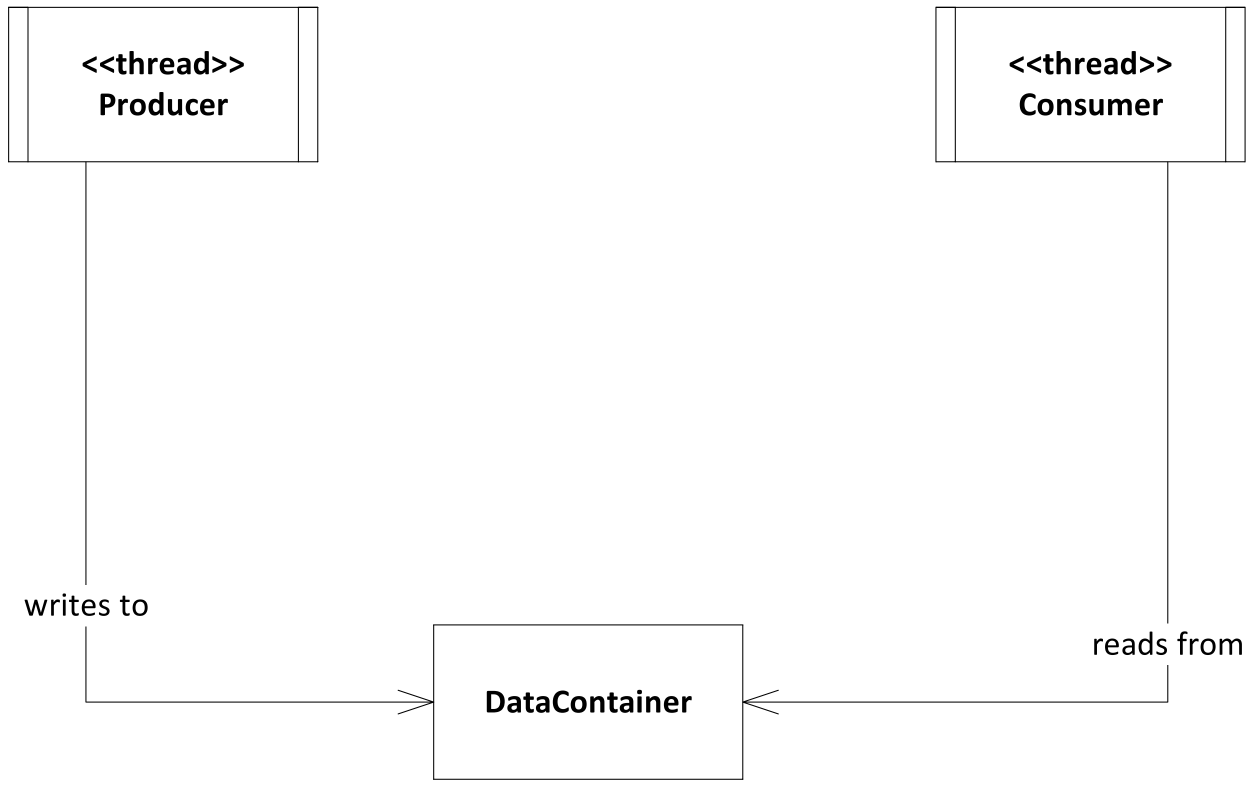
The AutoResetEvent objects are injected into the constructor of both the producer and consumer.


C:\sandkasse\ST3ITS3\E17\9-1 Thread synchronization pt. 2\CodeForSlides...

```
Tyre pressure: 48
Tyre pressure: 14
Tyre pressure: 0
Tyre pressure: 19
Tyre pressure: 12
Tyre pressure: 11
Tyre pressure: 41
Tyre pressure: 20
Tyre pressure: 2
Tyre pressure: 0
Tyre pressure: 19
Tyre pressure: 43
Tyre pressure: 17
Tyre pressure: 16
Tyre pressure: 43
Tyre pressure: 12
Tyre pressure: 21
```

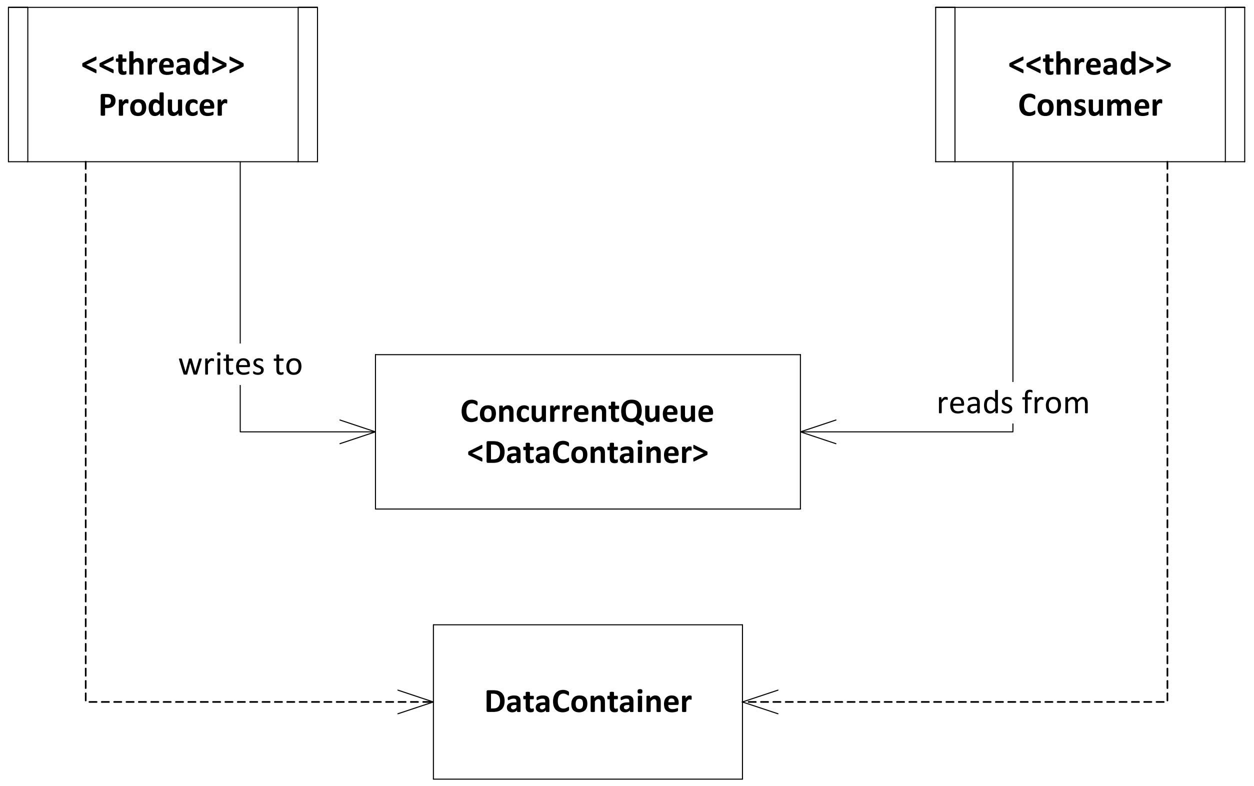
Queues and BlockingCollection

Queues



Right now, the producer and consumer runs in lock-step.

Queues



Right now, the producer and consumer runs in lock-step.

To overcome this we can introduce a queue.

.Net System.Collections.Concurrent

Access to the queue must be thread safe.

We can do this with **locks**, but...

.Net has built in thread safe collections:

- ConcurrentQueue<T>
- ConcurrentStack<T>
- ConcurrentBag<T>

And BlockingCollection<T> which implements the Producer-Consumer pattern.

BlockingCollection<T>

A thread-safe collection class that provides the following features:

An implementation of the Producer-Consumer pattern.

Concurrent adding and taking of items from multiple threads.

Optional maximum capacity.

Insertion and removal operations that block when collection is empty or full.
Insertion and removal "try" operations that do not block or that block up to a specified period of time.

Encapsulates any collection type that implements IProducerConsumerCollection<T>

https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/blockingcollection-overview

BlockingCollection<T> - Producer

```
class Producer
{
    private readonly BlockingCollection<DataContainer> _dataQueue;
    private readonly Random _random = new Random();

    public Producer(BlockingCollection<DataContainer> dataQueue)
    {
        _dataQueue = dataQueue;
    }

    public void Run()
    {
        int cnt = 50;
        while (cnt > 0)
        {
            int pressure = _random.Next(0, 50);
            DataContainer reading = new DataContainer();
            reading.SetTyrePressure(pressure);
            _dataQueue.Add(reading);
            Thread.Sleep(10);
            cnt--;
        }
        _dataQueue.CompleteAdding();
    }
}
```

We'll use a BlockingCollection as the queue.

The BlockingCollection handles all synchronization.

Calling CompleteAdding() signals to the receiver, that it shall expect no more data.

BlockingCollection<T> - Consumer

```
class Consumer
{
    private readonly BlockingCollection<DataContainer> _dataQueue;

    public Consumer(BlockingCollection<DataContainer> dataQueue)
    {
        _dataQueue = dataQueue;
    }

    public void Run()
    {
        while (!_dataQueue.IsCompleted)
        {
            try
            {
                var container = _dataQueue.Take();
                int pressure = container.GetTyrePressure();
                System.Console.WriteLine("Tyre pressure: {0}", pressure);
            }
            catch (InvalidOperationException)
            {
                // IOE means that Take() was called on a completed collection.
            }
            Thread.Sleep(10);
        }
        System.Console.WriteLine("No more data expected");
    }
}
```

The consumer takes data from the queue, until IsCompleted is set to true (by CompleteAdding() by the producer).

Remember try-catch around the Take() invocation. The queue might be marked as completed.

BlockingCollection<T> - Creation

```
static void Main(string[] args)
{
    BlockingCollection<DataContainer> dataQueue = new
                         BlockingCollection<DataContainer>();

    Producer producer = new Producer(dataQueue);
    Consumer consumer = new Consumer(dataQueue);

    Thread producerThread = new Thread(producer.Run);
    Thread consumerThread = new Thread(consumer.Run);

    producerThread.Start();
    consumerThread.Start();

    Console.ReadKey();
}
```

BlockingCollection – Add/Take with timeouts

```
public bool TryAdd (T item, int millisecondsTimeout);
```

```
public bool TryTake (out T item, TimeSpan timeout);
```

If you have something else for the thread to do, you can use timeouts on the Add and Take method.

See code examples on:
https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/how-to-add-and-take-items

Your turn

Solve exercises 1, 2, 3 and 4

(and 5, 6 and 7 if you like or continue with the exercise from last week)

References and image sources

Images:

Printer: https://i5.walmartimages.com/asr/5bf8c70c-c0f4-46c8-8de2-d14417c3dcdb_2.a974142a063bb1f235f672f9a68eeb10.jpeg

TPMS: http://www.rematiptop.com/tpms/img/TPMS-warning-light.jpg

Computer keyboard: http://stockmedia.cc/computing_technology/slides/DSD_8790.jpg

Bonus: http://wjreviews.com/reviews-cta/bonus.png

AARHUS UNIVERSITY