Thread synchronization pt. 2

AARHUS UNIVERSITY

AARHUS UNIVERSITY SCHOOL OF ENGINEERING

MICHAEL LOFT
ML@ASE.AU.DK

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS

Agenda

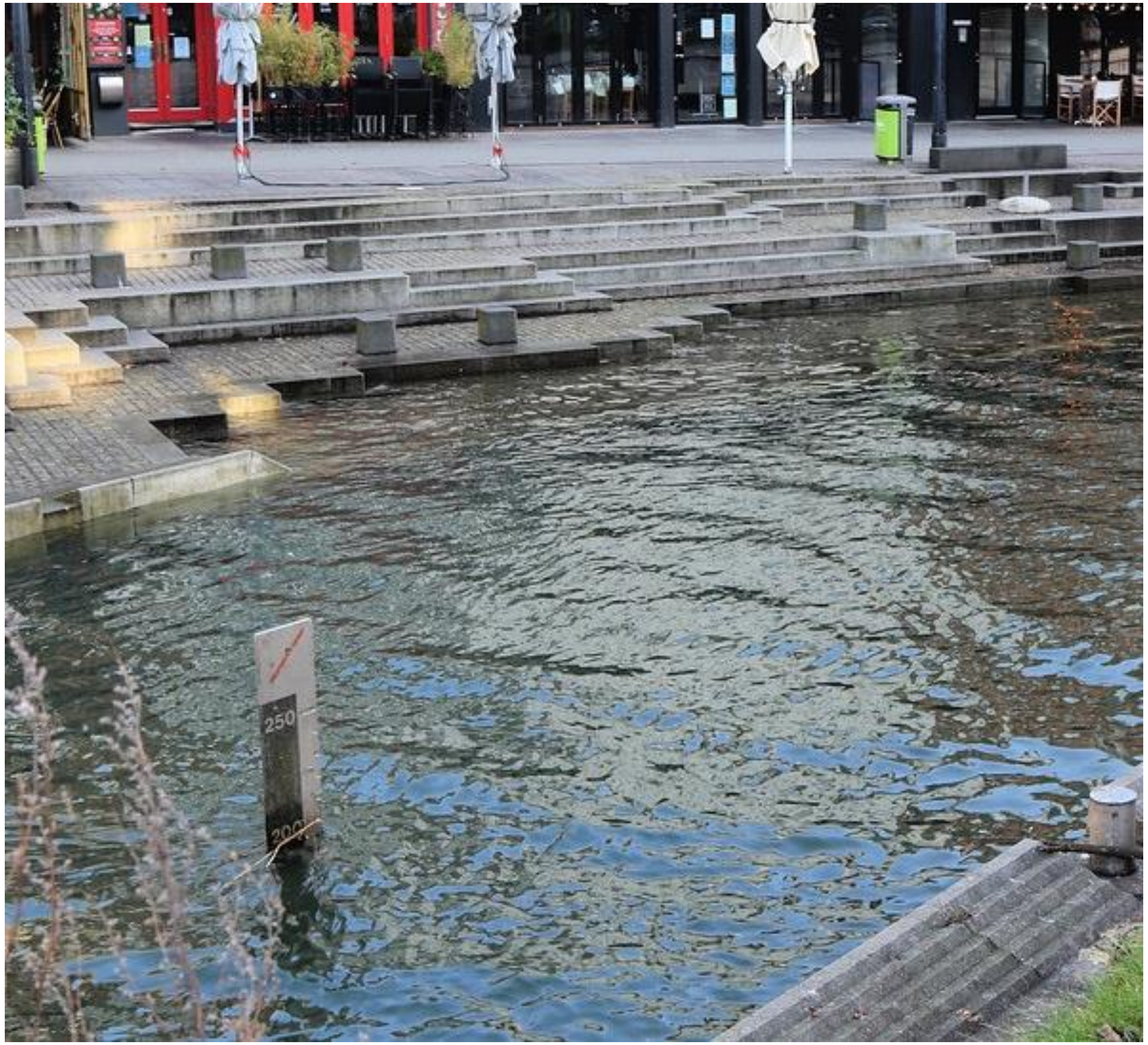
Producer-Consumer

Thread synchronization

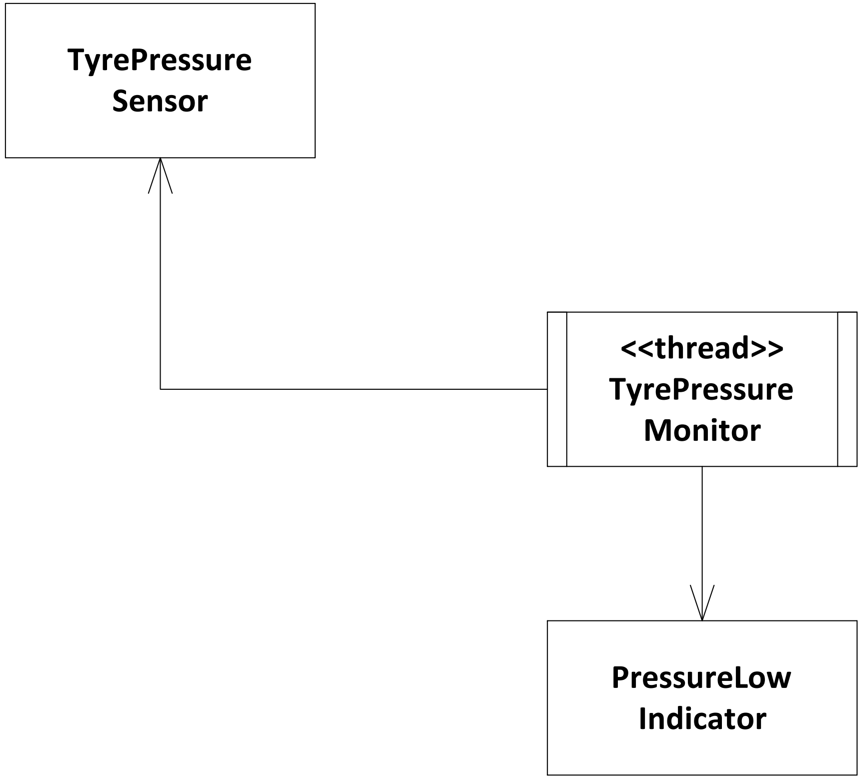
- AutoResetEvent
- ManualResetEvent

Queues and BlockingCollection

Water level Monitor System

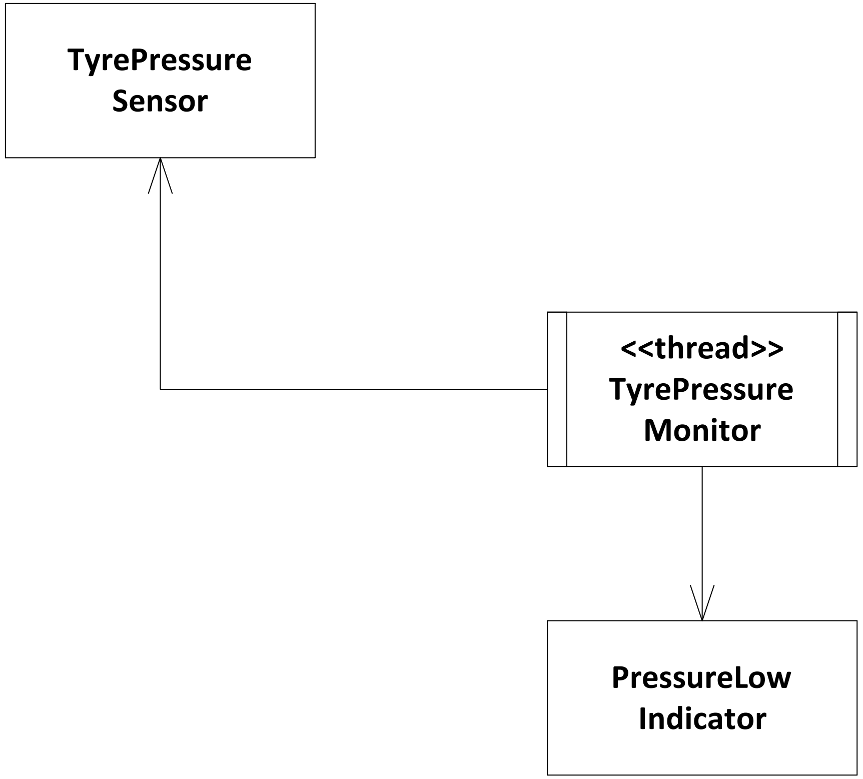


TPMS design



TyrePressureMonitor has many responsibilities:

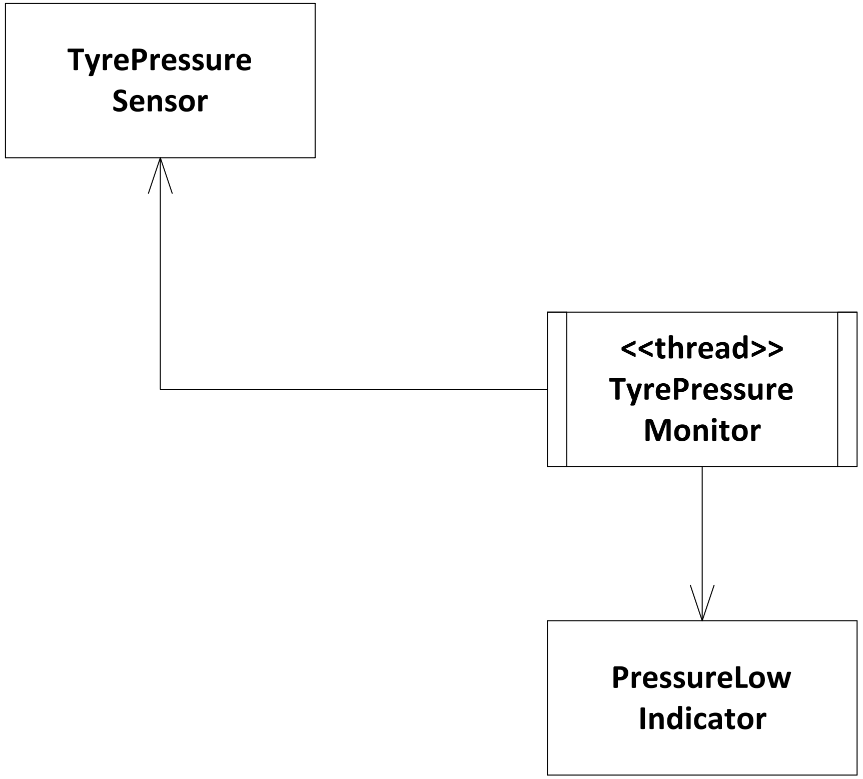
TPMS design



TyrePressureMonitor has many responsibilities:

- Read the pressure using the TyrePressureSensor.

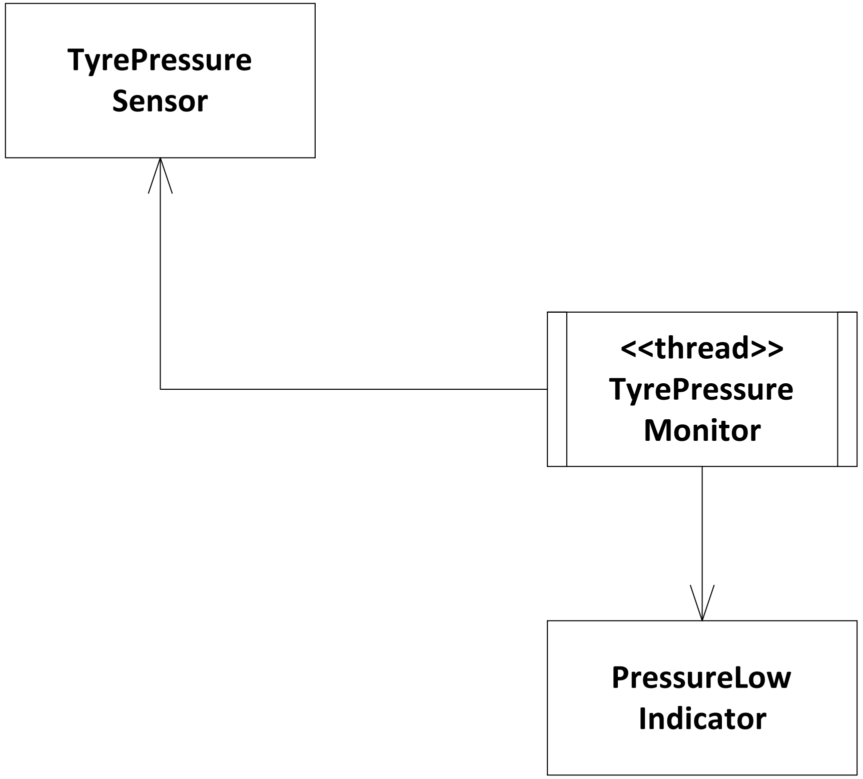
TPMS design



TyrePressureMonitor has many responsibilities:

- Read the pressure using the TyrePressureSensor.
- Determine if the pressure is too low.

TPMS design



TyrePressureMonitor has many responsibilities:

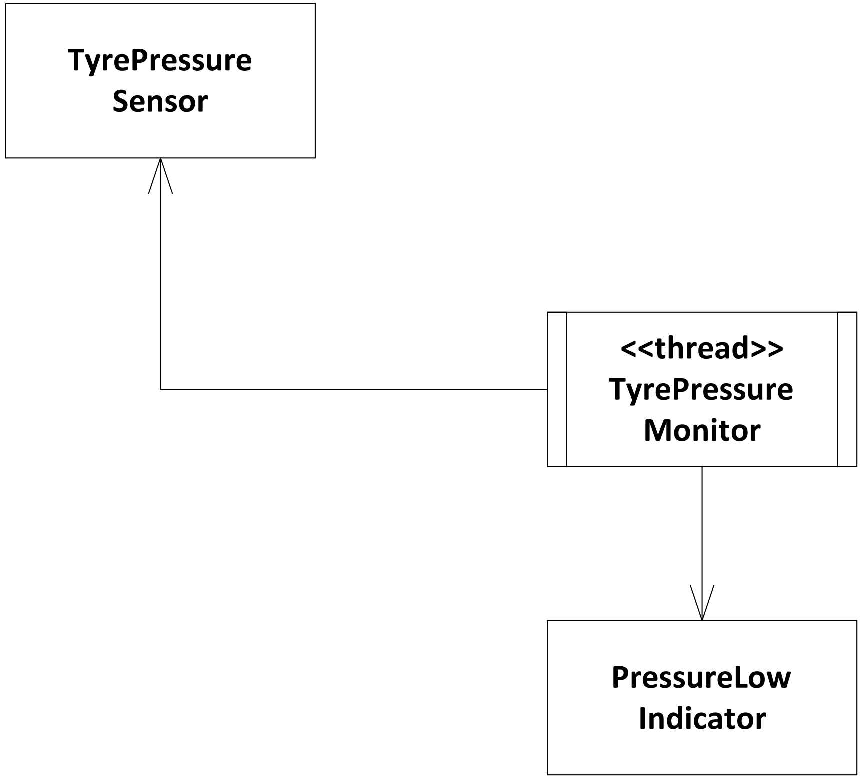
- Read the pressure using the TyrePressureSensor.
- Determine if the pressure is too low.
- Turn on/off the PressureLowIndicator

Design principle: Single Responsibility

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

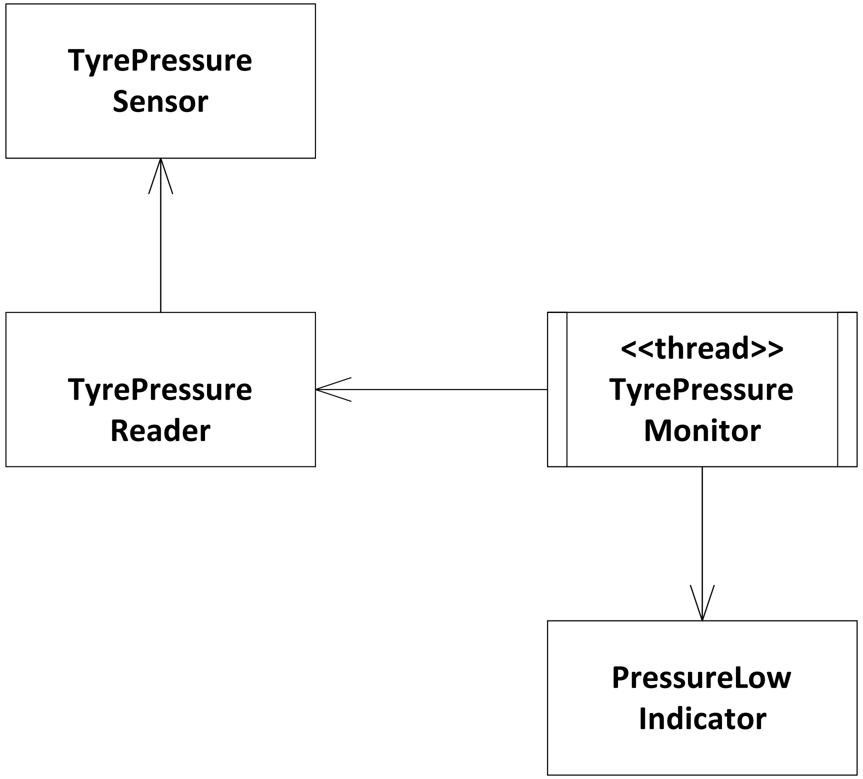
Robert Martin: The Single Responsibility Principle

TPMS design

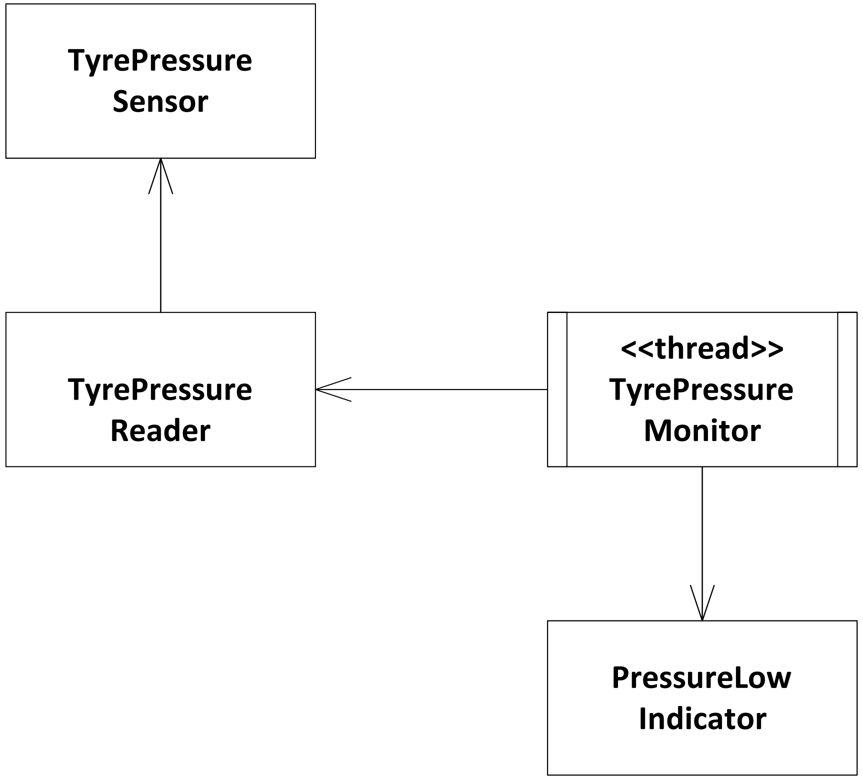


TyrePressureMonitor has many responsibilities:

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

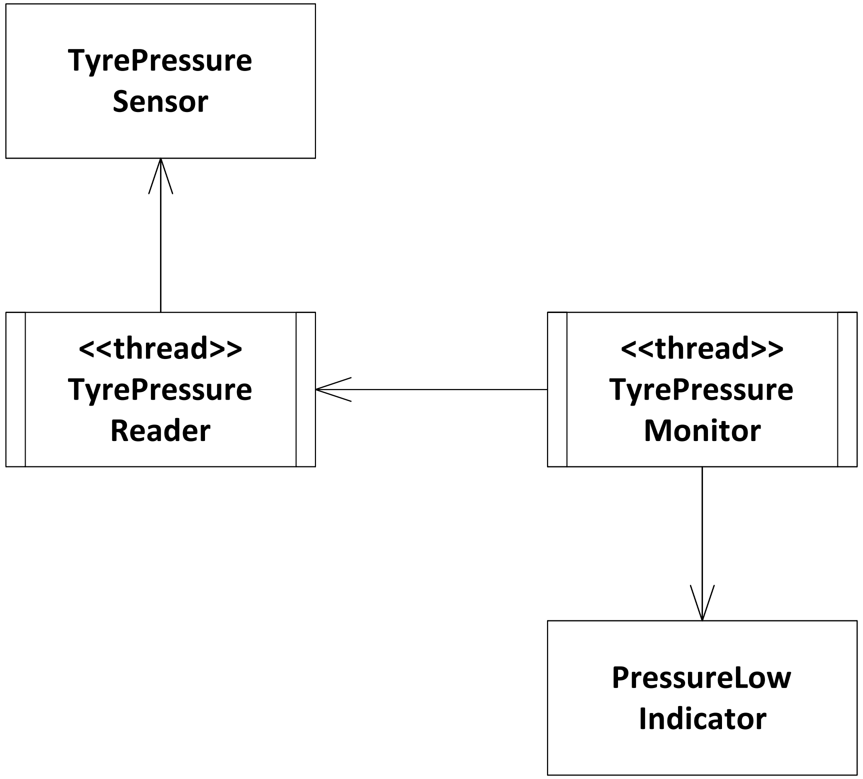


Now, reading the pressure is separated out.

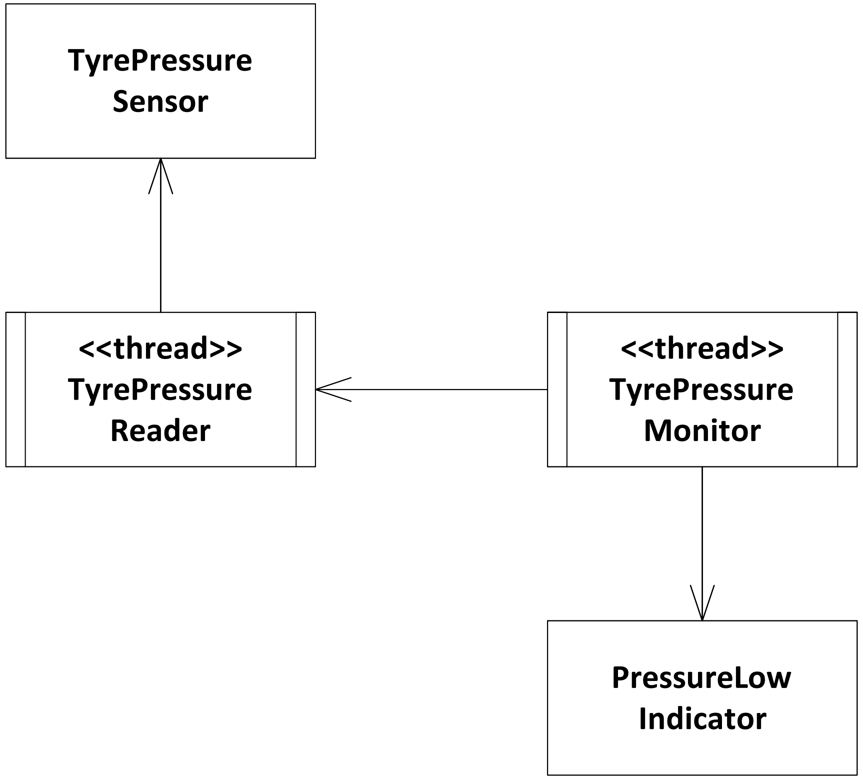


Now, reading the pressure is separated out.

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

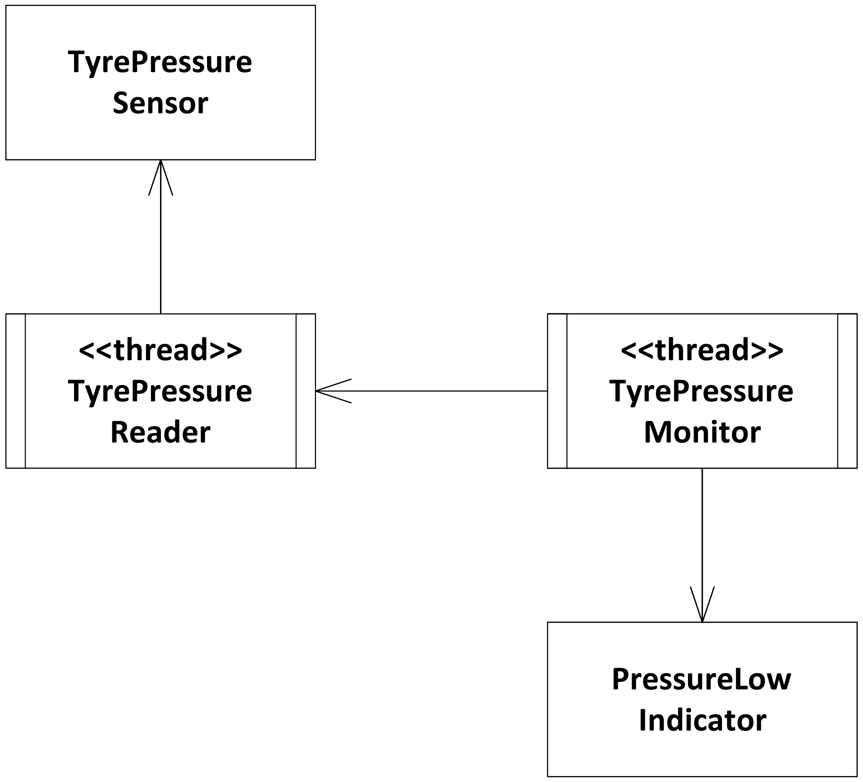

TyrePressure Sensor
<<thread>> TyrePressure Reader
<<thread>> TyrePressure Monitor
PressureLow Indicator

Let’s put the TyrePressureReader on a separate thread.


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

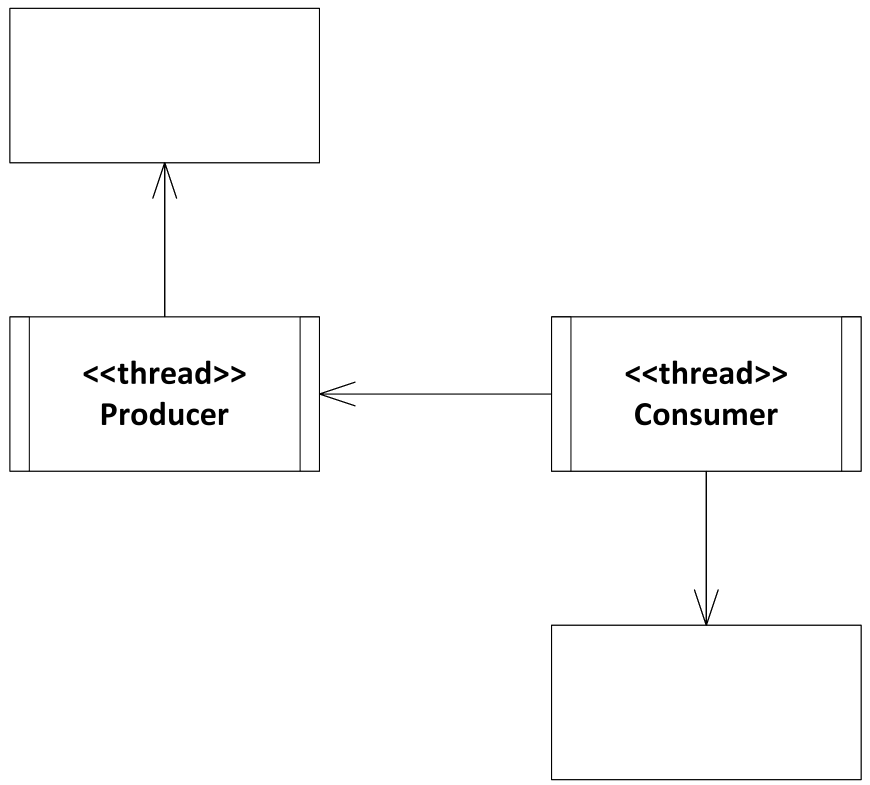
Let’s put the TyrePressureReader on a separate thread.

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

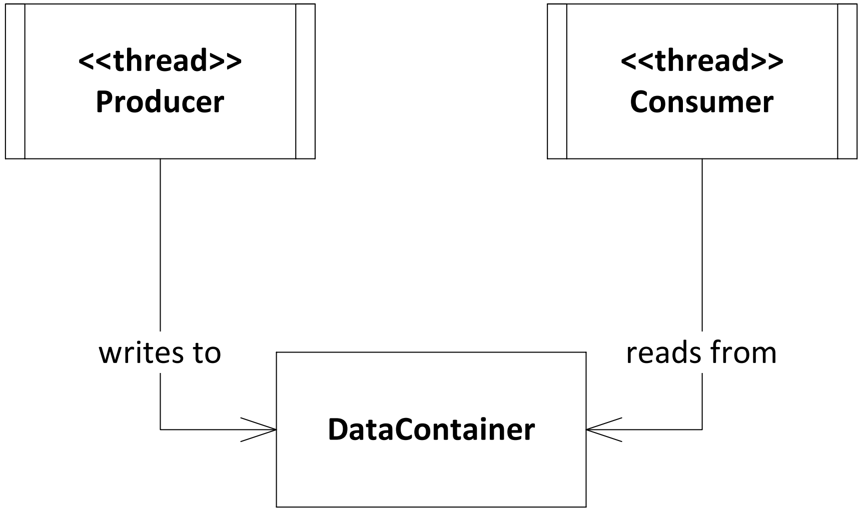


The Monitor consumes **data**, which the Reader provides.

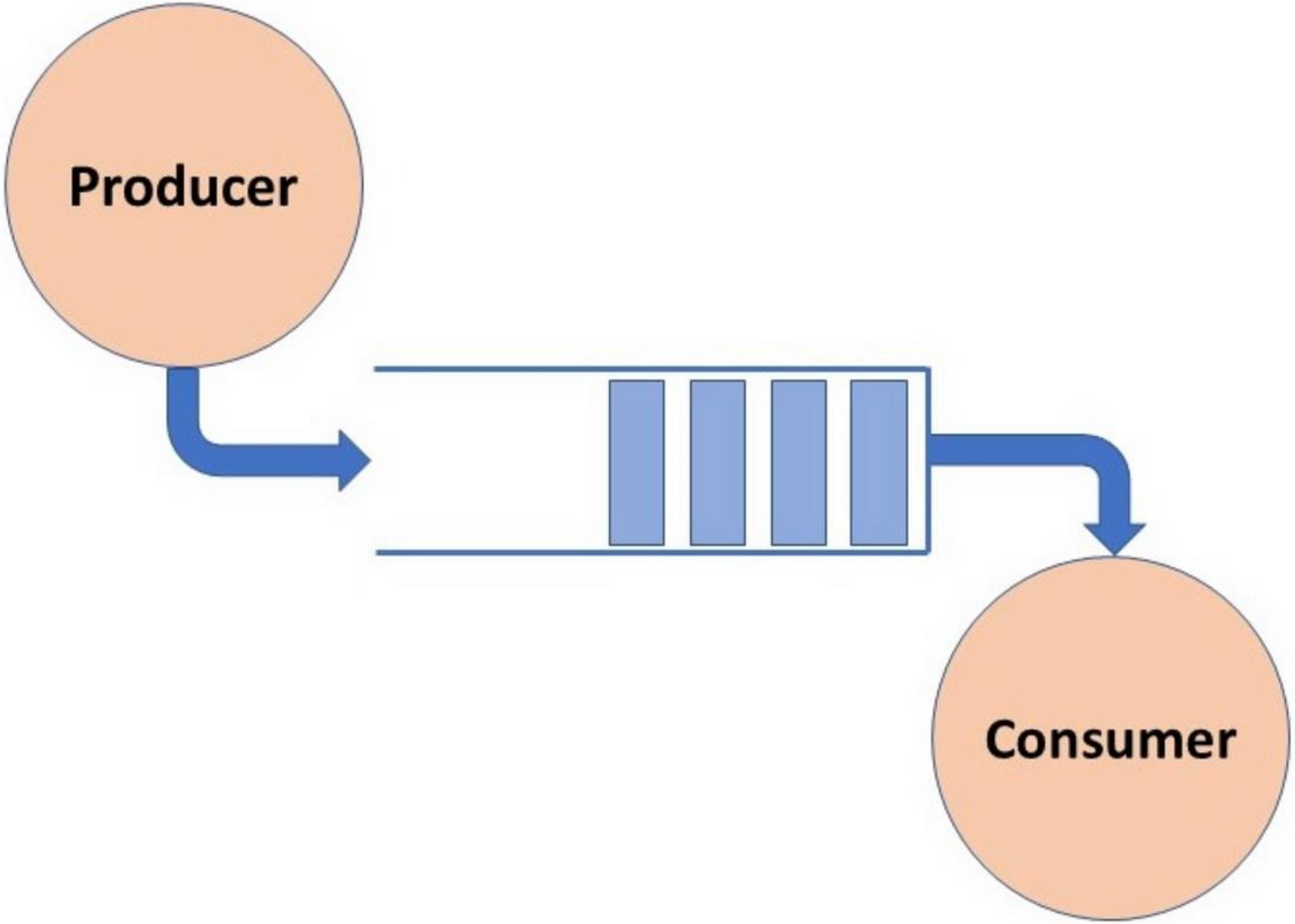
This is a very common design: Producer – Consumer.

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

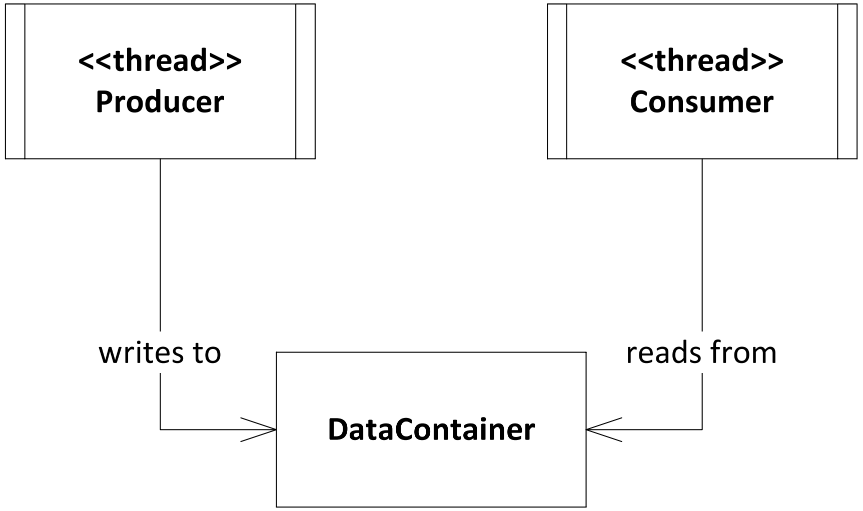
DataContainer

```
class DataContainer
{
    private int tyrePressure;

    public int GetTyrePressure()
    {
        return tyrePressure;
    }

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

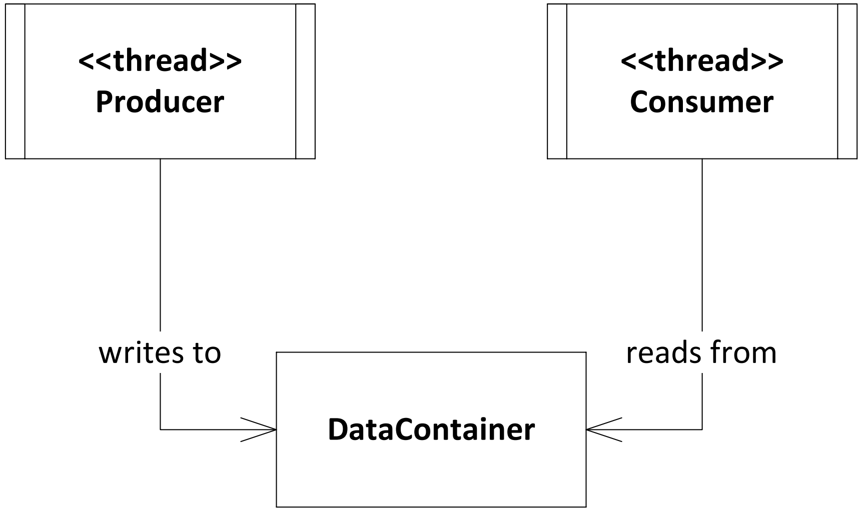
Objects of the DataContainer class is used to pass data from producer to consumer.

Producer - Consumer

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



Producer - Consumer

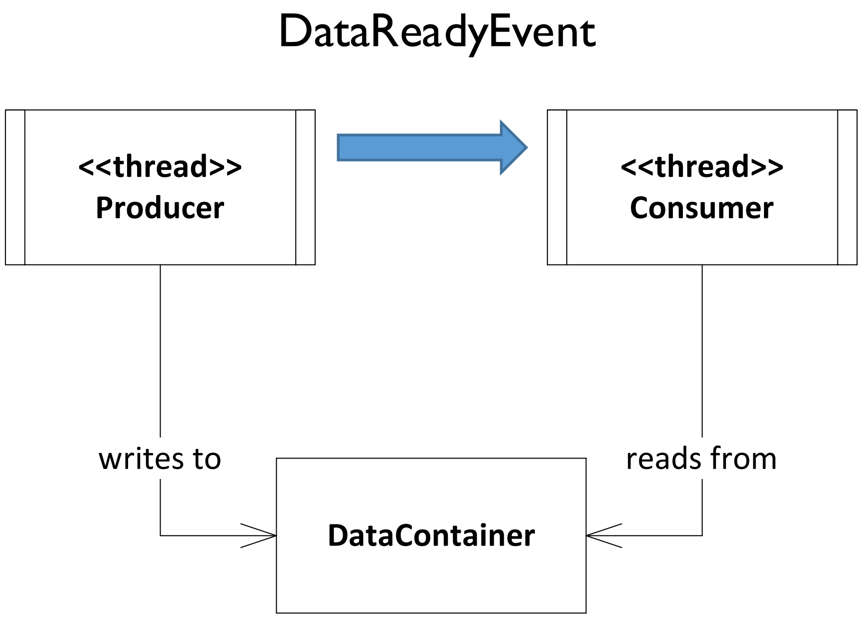


The Consumer would like to know when new data is available.

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

Thread synchronization

Producer - Consumer

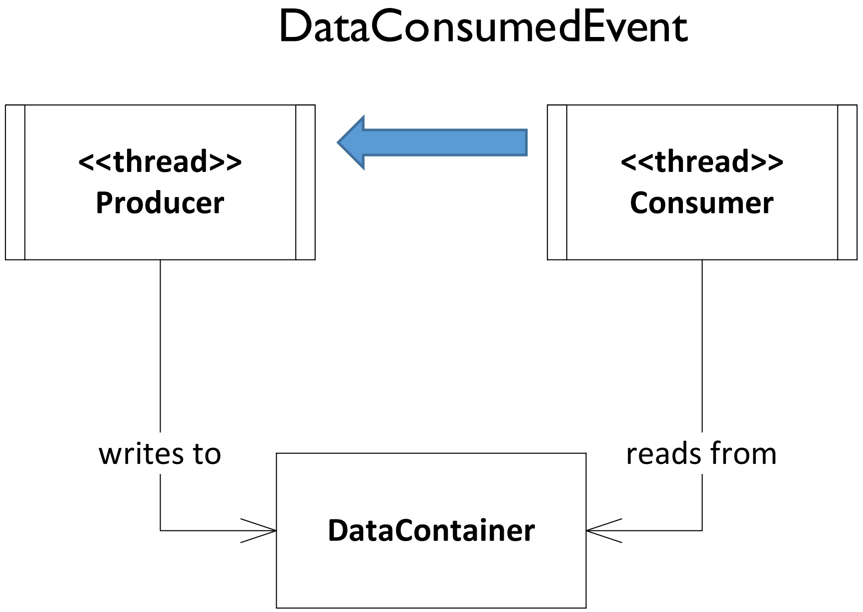


The Consumer would like to know when new data is available.

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

We can signal this between threads with Events

Producer - Consumer

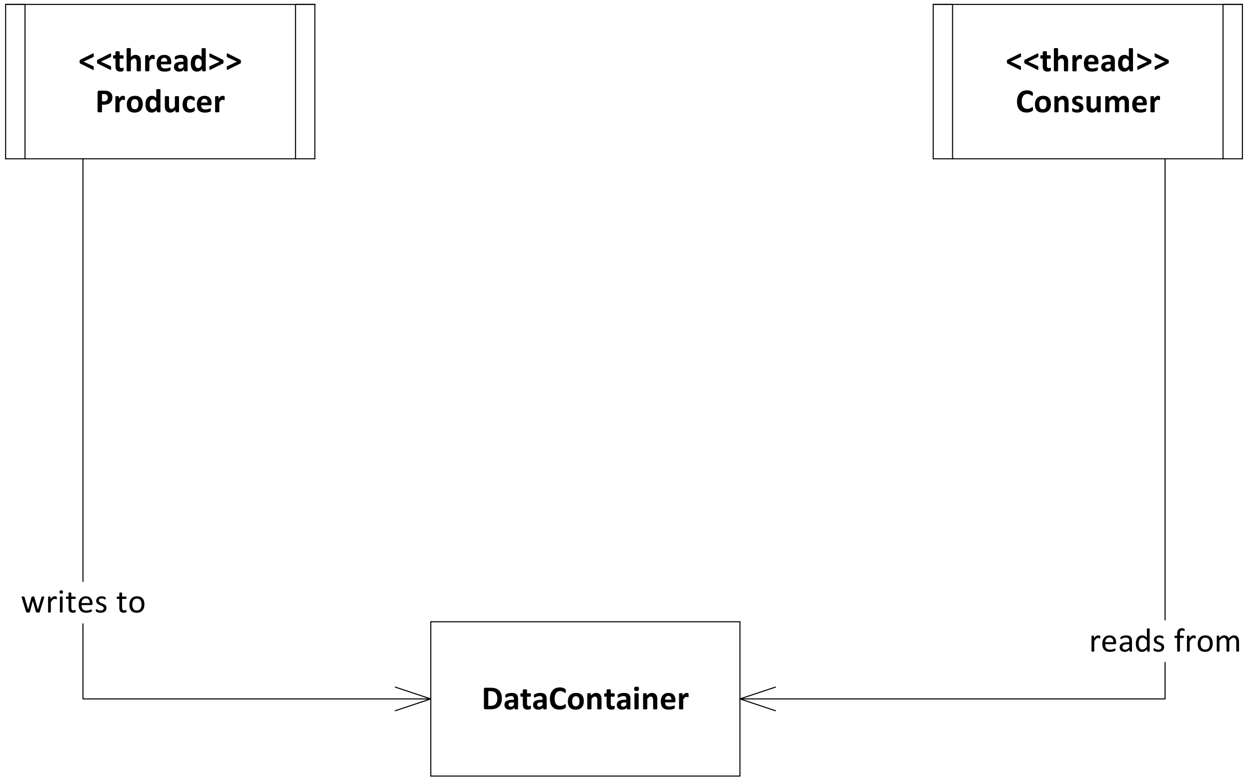


The Consumer would like to know when new data is available.

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

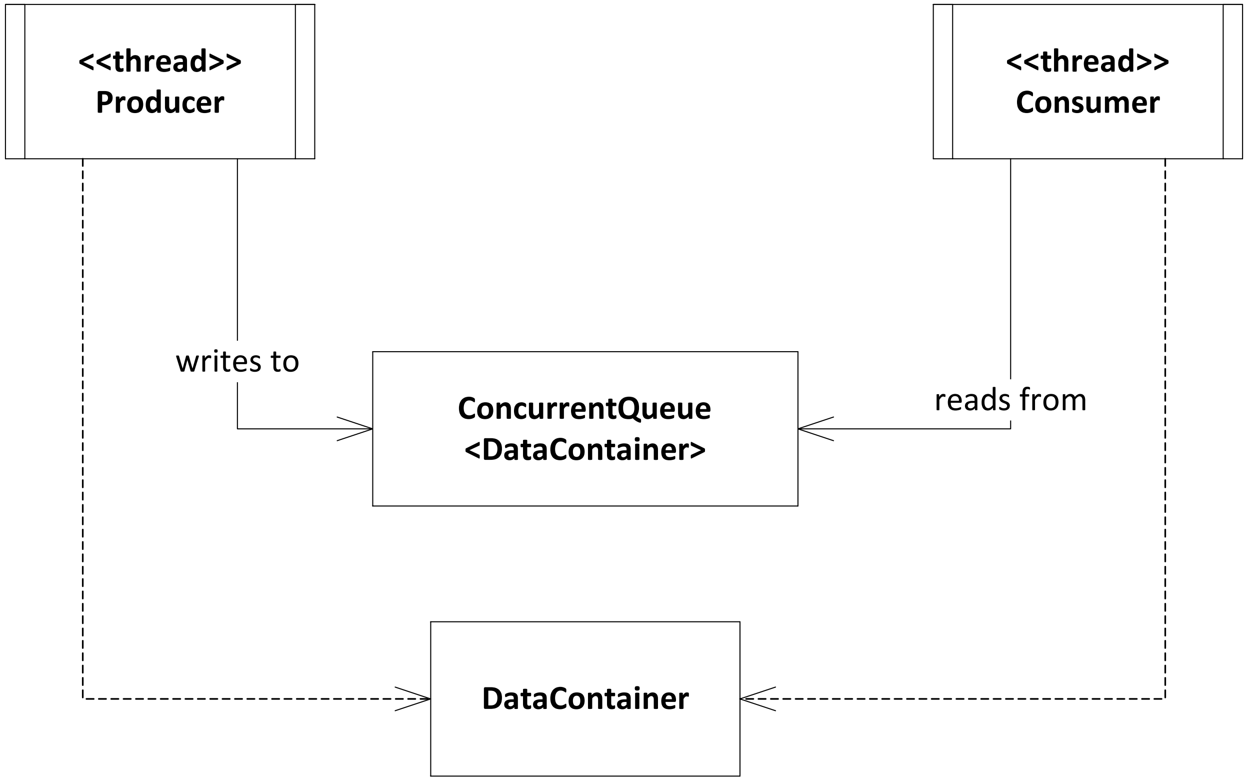
We can signal this between threads with Events

Queues



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

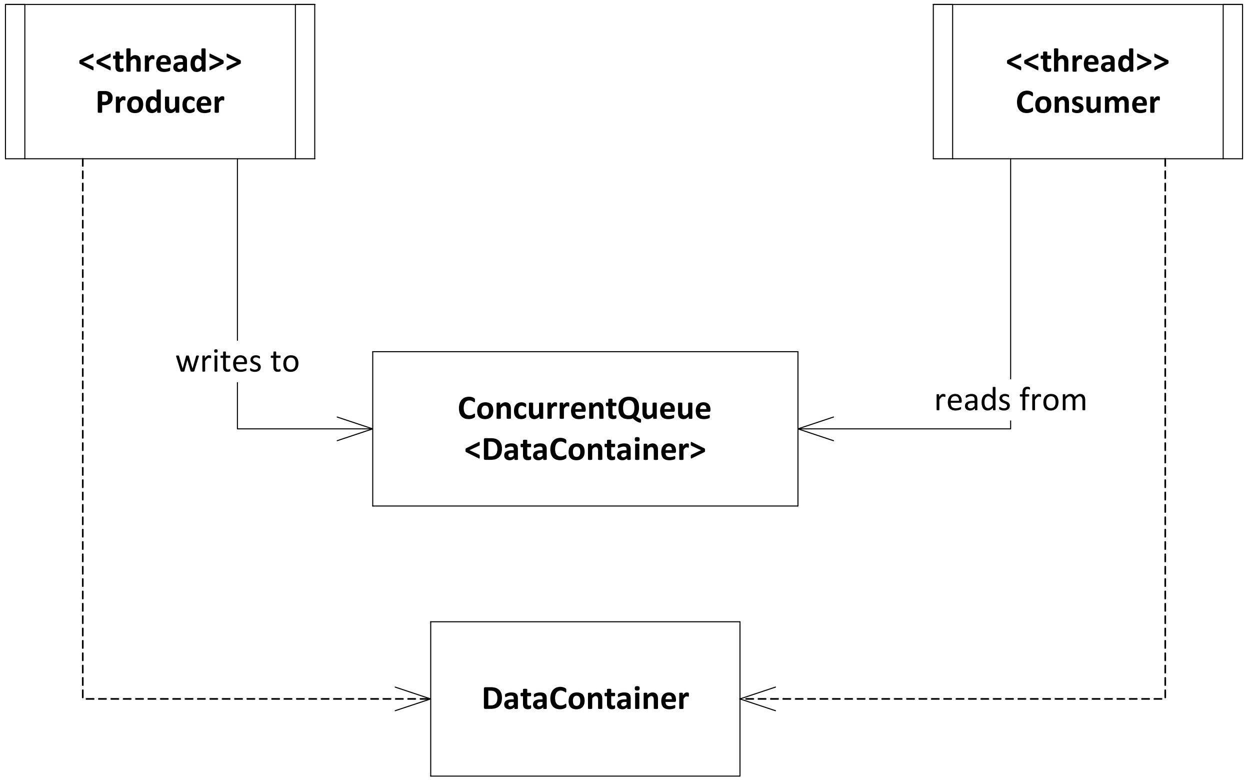
Queues



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

To overcome this we can introduce a queue.

Queues



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

To overcome this we can introduce a queue.

Question then – which queue to use?

Queues and BlockingCollection

.Net System.Collections.Concurrent

Access to the queue must be thread safe.

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

.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>

.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:

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

BlockingCollection<T>

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

An implementation of the Producer-Consumer pattern.

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

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.

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

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.

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

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.

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

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.

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).

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

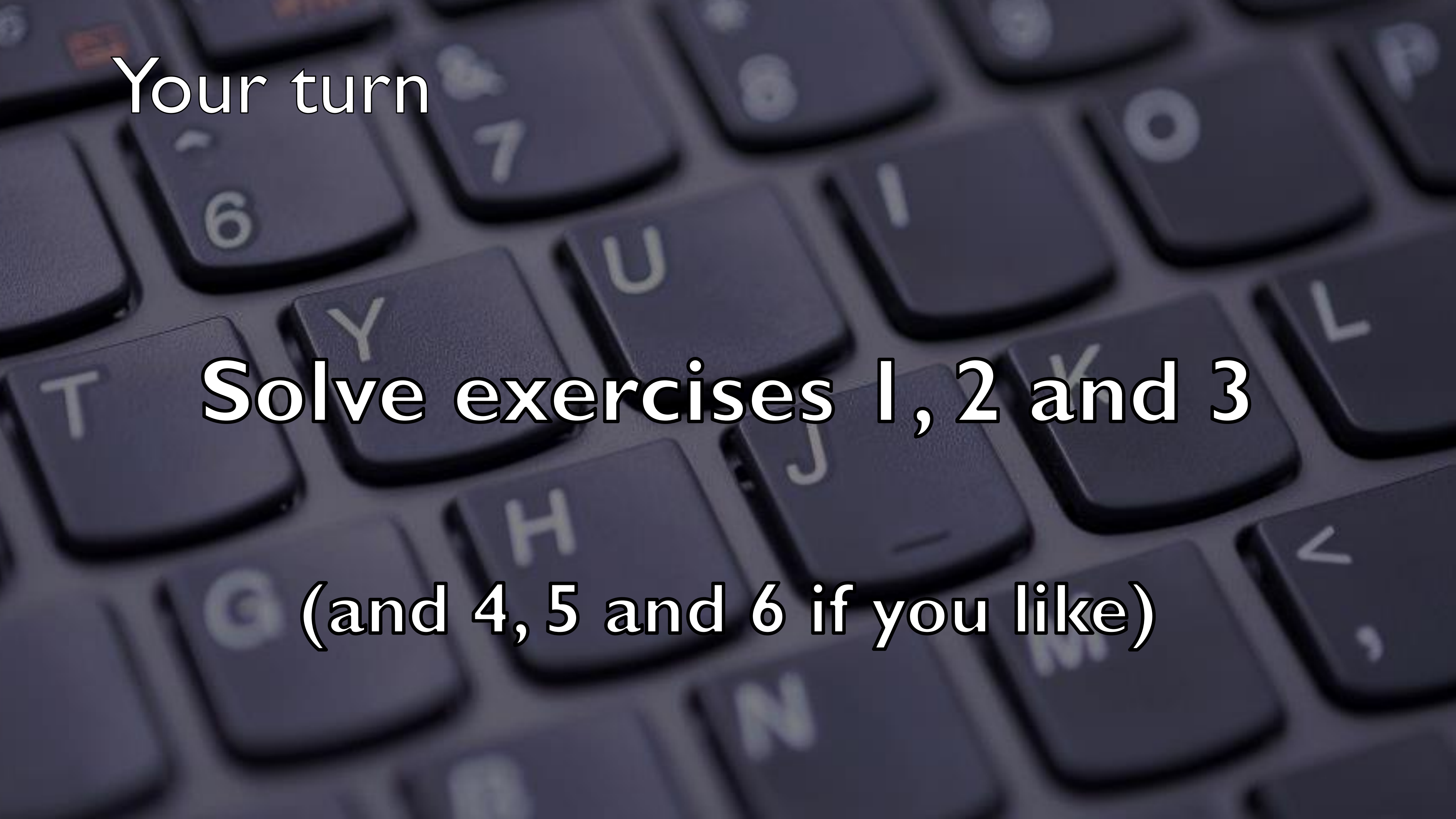
```
public bool TryAdd (T item, int millisecondsTimeout);
```

```
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

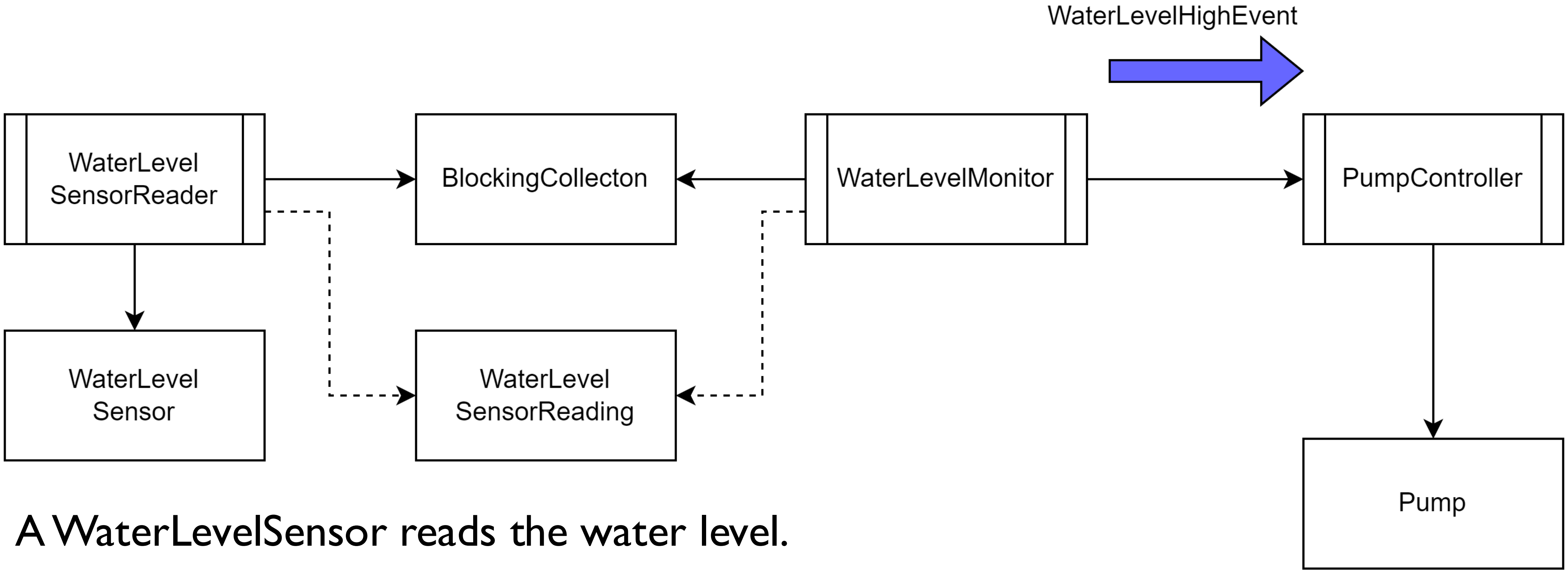
Your turn



Solve exercises 1, 2 and 3

(and 4, 5 and 6 if you like)

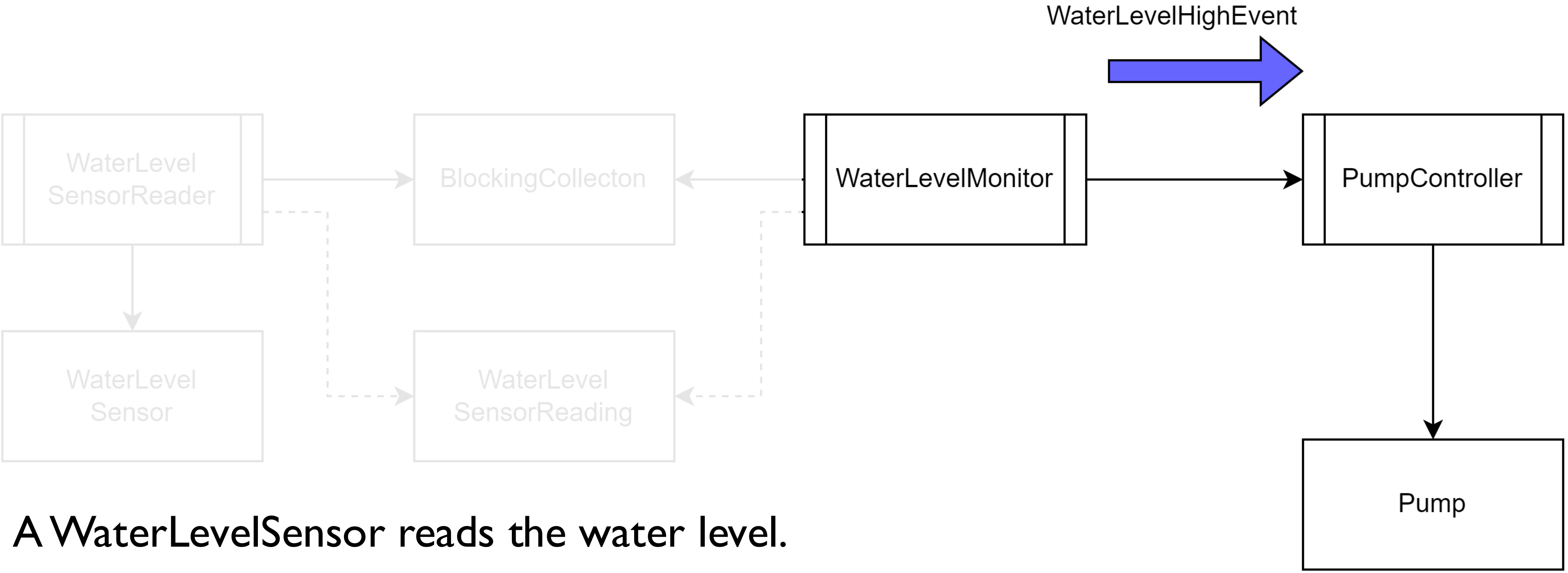
Thread communication with Events



A WaterLevelSensor reads the water level.

The reading is sent to a WaterLevelMonitor.

If the water level is too high, it sends an event to a PumpController, which runs a pump for a given time.



A WaterLevelSensor reads the water level.

The reading is sent to a WaterLevelMonitor.

If the water level is too high, it sends an event to a PumpController, which runs a pump for a given time.

AutoResetEvent and ManualResetEvent

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

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.

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.

WaterLevelMonitor

```
public class WaterLevelMonitor
{
   private readonly AutoResetEvent _waterLevelHighAutoResetEvent;
   private readonly Random _random = new Random();

   public WaterLevelMonitor(AutoResetEvent waterLevelHighAutoResetEvent)
   {
      _waterLevelHighAutoResetEvent = waterLevelHighAutoResetEvent;
   }

   public void Run()
   {
      for (int i = 0; i < 10; i++)
      {
         int randomValue = _random.Next(0, 2);
         Console.WriteLine("Random value was: {0}", randomValue);
         if (randomValue > 0)
         {
            _waterLevelHighAutoResetEvent.Set();
         }
         Thread.Sleep(1000);
      }
   }
}
```

The WaterLevelMonitor and PumpController share the same AutoResetEvent.

WaterLevelMonitor set the event.

PumpController

```
public class PumpController
{
  private readonly AutoResetEvent _waterLevelHighAutoResetEvent;

  public PumpController(AutoResetEvent waterLevelHighAutoResetEvent)
  {
   _waterLevelHighAutoResetEvent = waterLevelHighAutoResetEvent;
  }

  public void Run()
  {
    while (!ShallStop)
    {
     bool wasSet = _waterLevelHighAutoResetEvent.WaitOne(5000);
     if (wasSet)
     {
      Console.WriteLine("Event was set - Water level high.");
      Console.WriteLine("Running pump for 2 seconds.");
     }
     else
     {
      Console.WriteLine("Waiting timed out");
     }
    }
  }

  public bool ShallStop { get; set; }
}
```

The WaterLevelMonitor and PumpController share the same AutoResetEvent.

The PumpController waits for the event.

A timeout is used to allow the thread to shut down properly and not wait forever, if the other thread stops sending events.

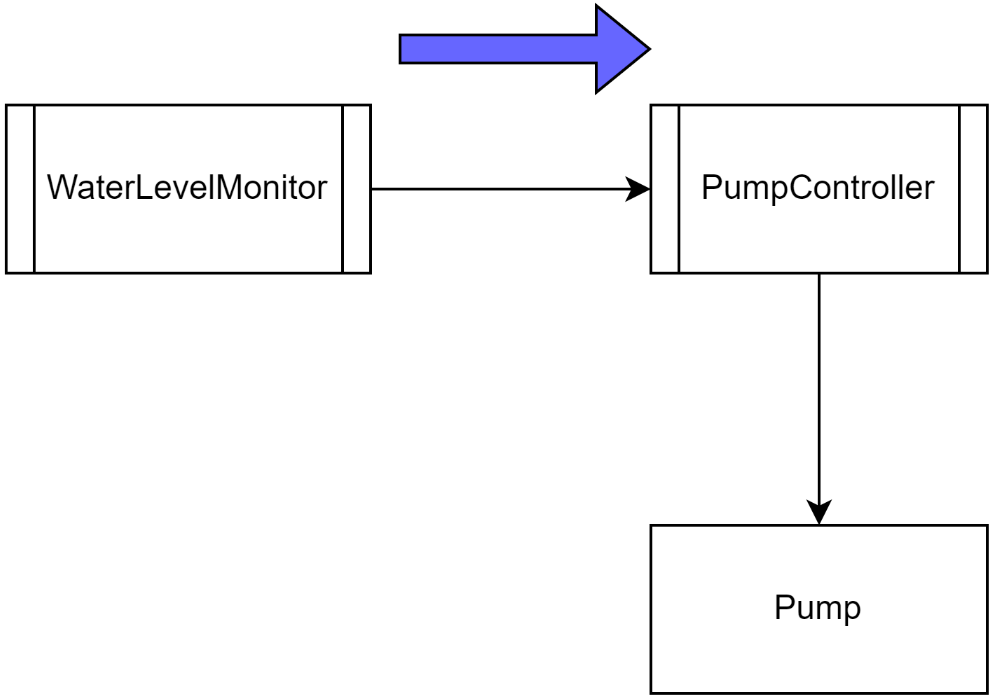
Program

```
namespace ResetEvents
{
  internal class Program
  {
   static void Main(string[] args)
   {
    AutoResetEvent dataReadyAutoResetEvent = new AutoResetEvent(false);

    WaterLevelMonitor waterLevelMonitor =
      new WaterLevelMonitor(dataReadyAutoResetEvent);

    PumpController pumpController =
      new PumpController(dataReadyAutoResetEvent);

    Thread producerThread = new Thread(waterLevelMonitor.Run);
    Thread consumerThread = new Thread(pumpController.Run);

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

    producerThread.Join();

    pumpController.ShallStop = true;
    consumerThread.Join();
   }
  }
}
```

Main creates the WaterLevelMonitor, the PumpController and the shared AutoResetEvent.

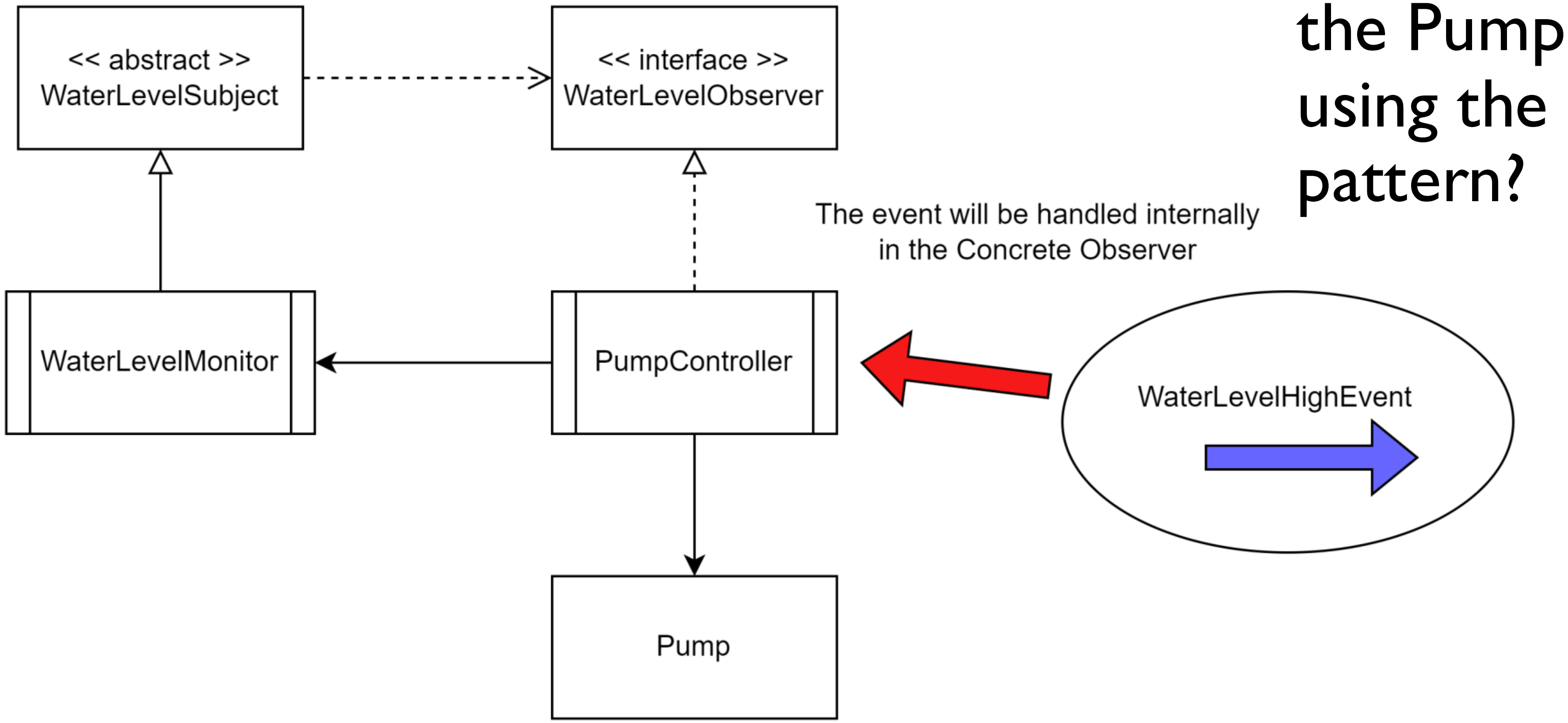
The AutoResetEvent is 'not set' when created.

GoF Observer with threads



What to do, if we want to decouple the WaterLevelMonitor from the PumpController using the GoF Observer pattern?

What to do, if we want to decouple the WaterLevelMonitor from the PumpController using the GoF Observer pattern?



The event will be handled internally in the Concrete Observer

PumpController as Observer

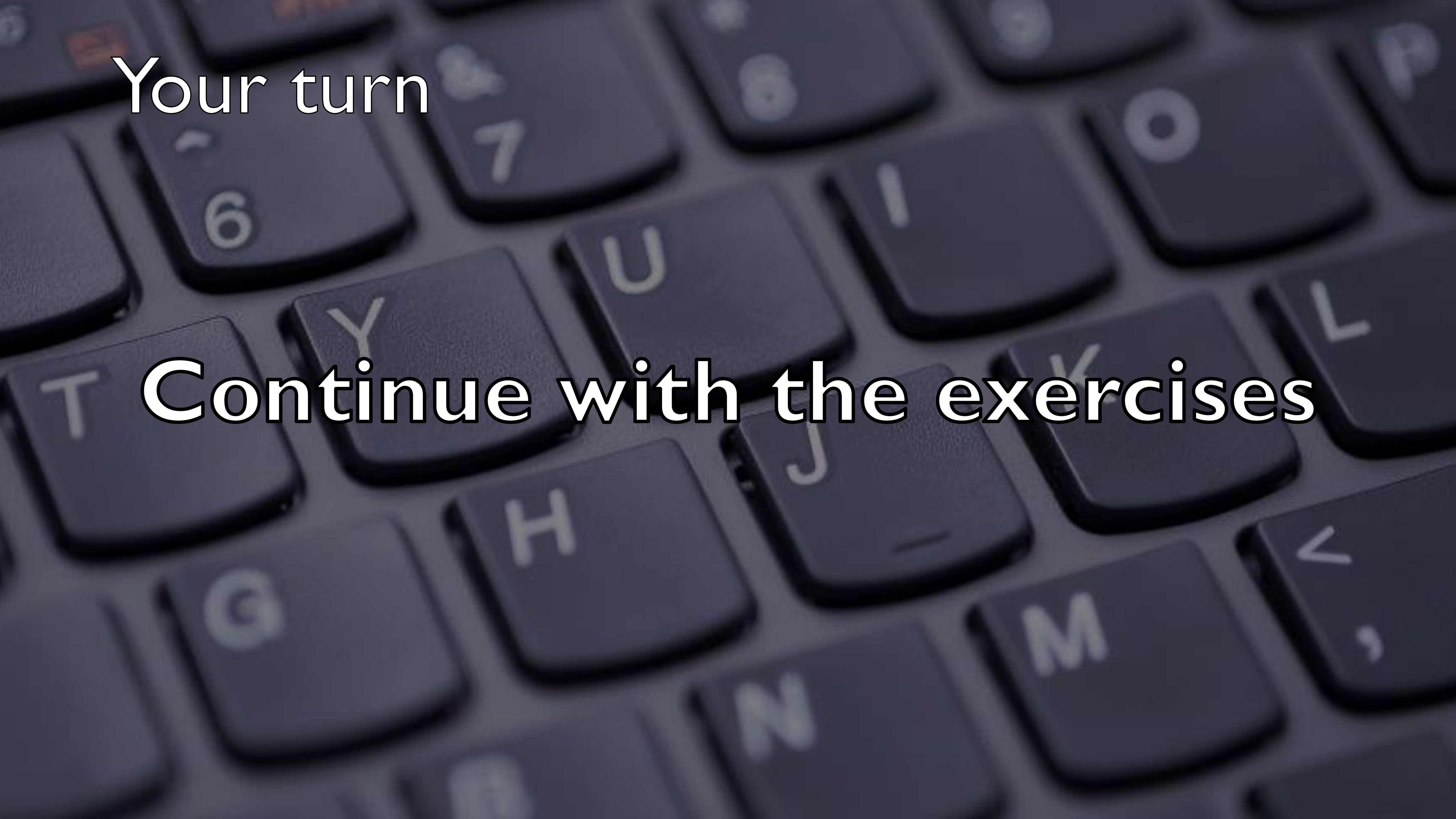
```
public class PumpController : IWaterLevelObserver
{
   private readonly AutoResetEvent _waterLevelHighAutoResetEvent = new
AutoResetEvent(false);
   private int _waterLevel;
   private readonly object _waterLevelLockObject =
      new object();

   public PumpController(WaterLevelSubject subject)
   {
      subject.Attach(this);
   }

   public void Update(int waterLevel)
   {
      WaterLevel = waterLevel;
      _waterLevelHighAutoResetEvent.Set();
   }
```

```
public void Run()
{
 while (!ShallStop)
 {
    bool wasSet = _waterLevelHighAutoResetEvent.WaitOne(5000);
    if (wasSet)
    {
       Console.WriteLine("Event was set - Water level: " + WaterLevel);
       Console.WriteLine("Running pump for 2 seconds.");
    }
    else
    {
       Console.WriteLine("Waiting timed out");
    }
 }
}

 public bool ShallStop { get; set; }
```

PumpController as Observer

```
public int WaterLevel
    {
        get
        {
            lock (_waterLevelLockObject)
            {
                return _waterLevel;
            }
        }
        set
        {
            lock (_waterLevelLockObject)
            {
                _waterLevel = value;
            }
        }
    }
```

Your turn



Continue with the exercises

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