.NET DEVELOPER DAYS

Jiří Činčura | @cincura_net | MVP Microsoft® Most Valuable Professional

Most Common Mistakes in Using Tasks and in Asynchronous Code

Why asynchronous?

Offloading

- I.e. free UI thread
- Not all threads are equal

Concurrency

- Multiple operations at once

Scalability

- (not) wasting resources

Why asynchronous?

Asynchronous operations existed since stone age

- BeginXxx, EndXxx (APM)
- EAP

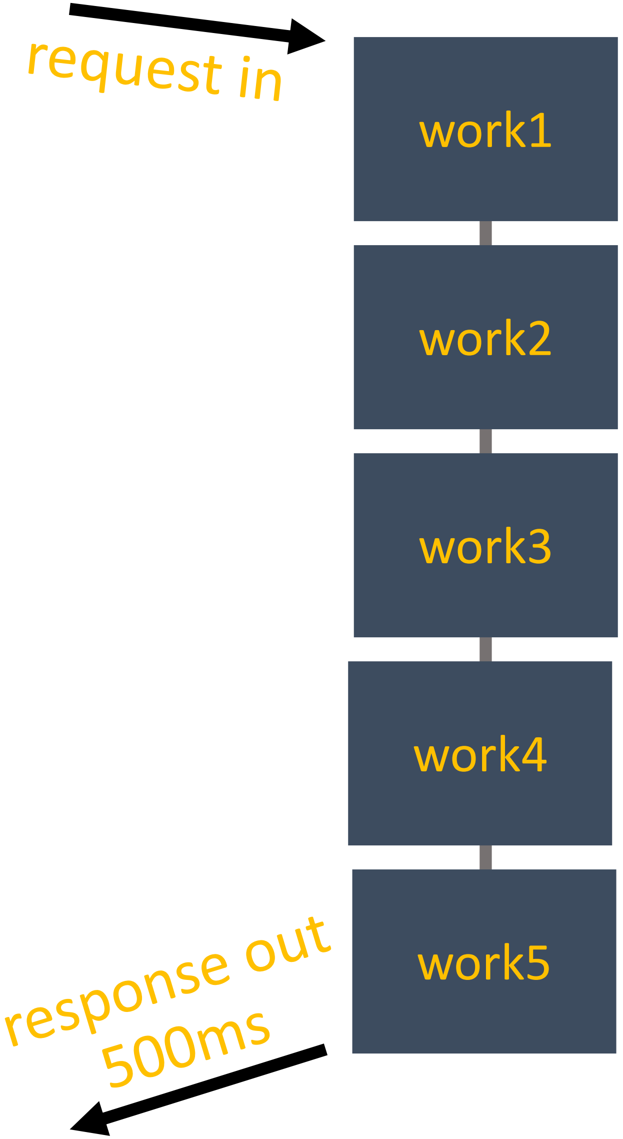
async/*await* is not about creating (from nothing) async methods…

…but a way to compose/consume async methods

CPU bound vs IO bound operations

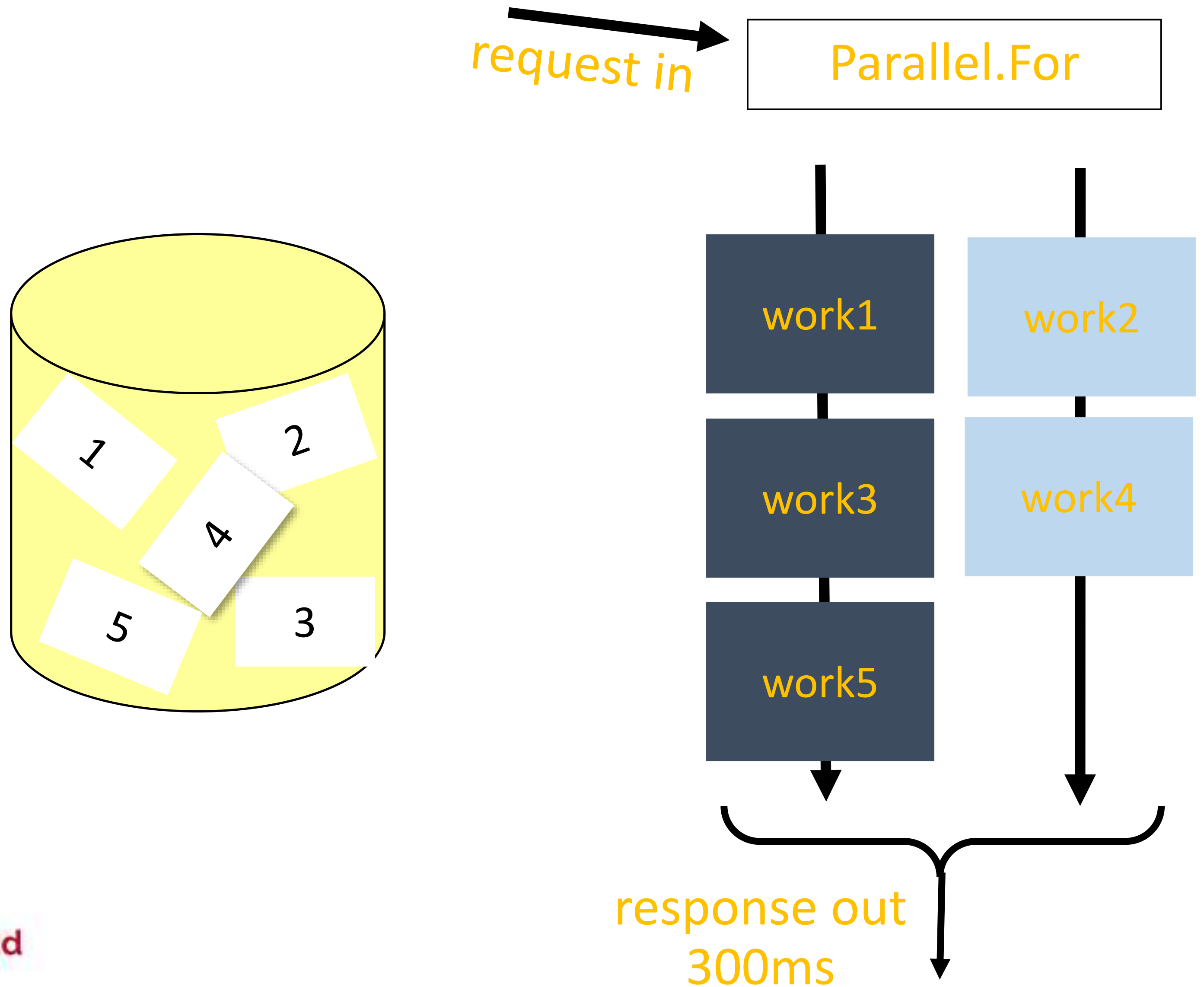
```
public List<Something> LoadSomething()
{
  var result = new List<Something>();
  for (var i = 1; i <= 5; i++)
  {
    var s = Something.LoadFromNetwork(id: i);
    result.Add(s);
  }
  return result;
}
```

CPU bound vs IO bound operations



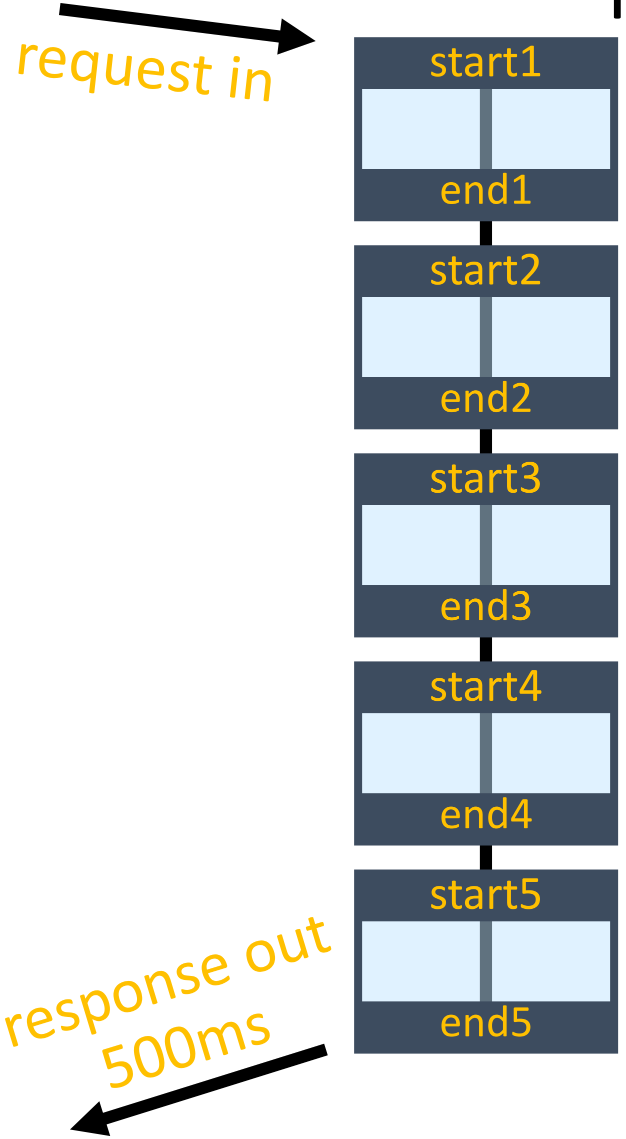
Source: Lucian Wischik

CPU bound vs IO bound operations



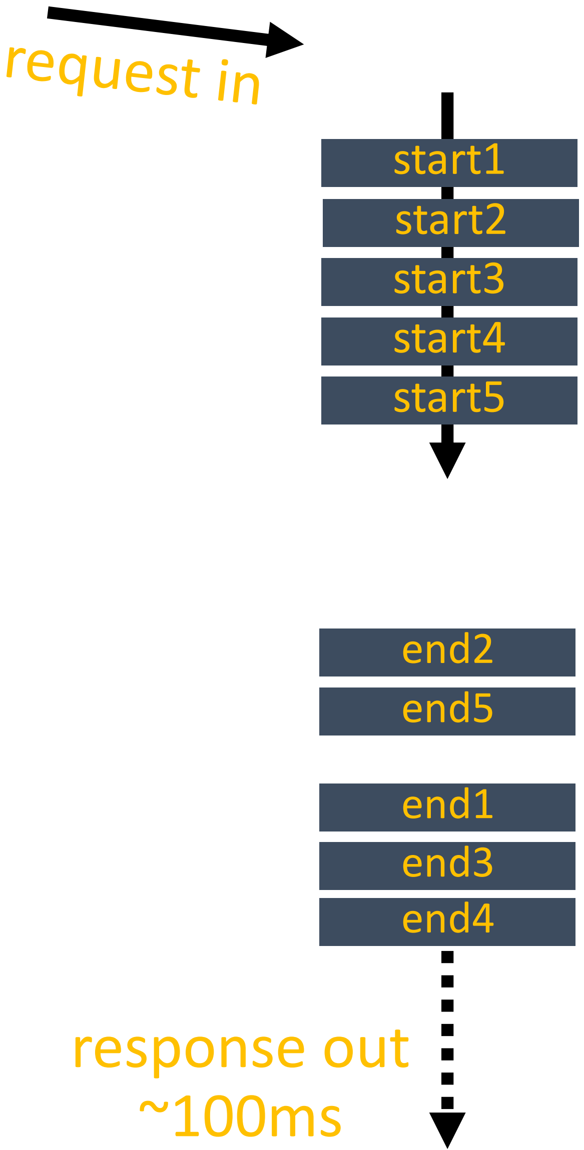
Source: Lucian Wischik

CPU bound vs IO bound operations



Source: Lucian Wischik

CPU bound vs IO bound operations



request in

start1

start2

start3

start4

start5

end2

end5

end1

end3

end4

response out
~100ms

Source: Lucian Wischik

CPU bound vs IO bound operations

CPU bound

Parallel.For, Task.Run, …

IO bound

async (real async code)

Async method runs on...

… thread

- True? False?

… thread pool

- True? False?

False. False.

- Scalability
- Asynchronous operations are performed by HW (then I/O completion port)

Task.FromResult for “known” data

```
public Task<int> AddAsync(int a, int b)
{
  return Task.Run(() => a + b);
}

public Task<int> AddAsync(int a, int b)
{
  return Task.FromResult(a + b);
}
```

Task.FromResult for “known” data

Task is a reference type (=> on heap)

ValueTask<T>

```
public ValueTask<int> AddAsync(int a, int b)
{
  return new ValueTask<int>(a + b);
}
```

Long-running operations

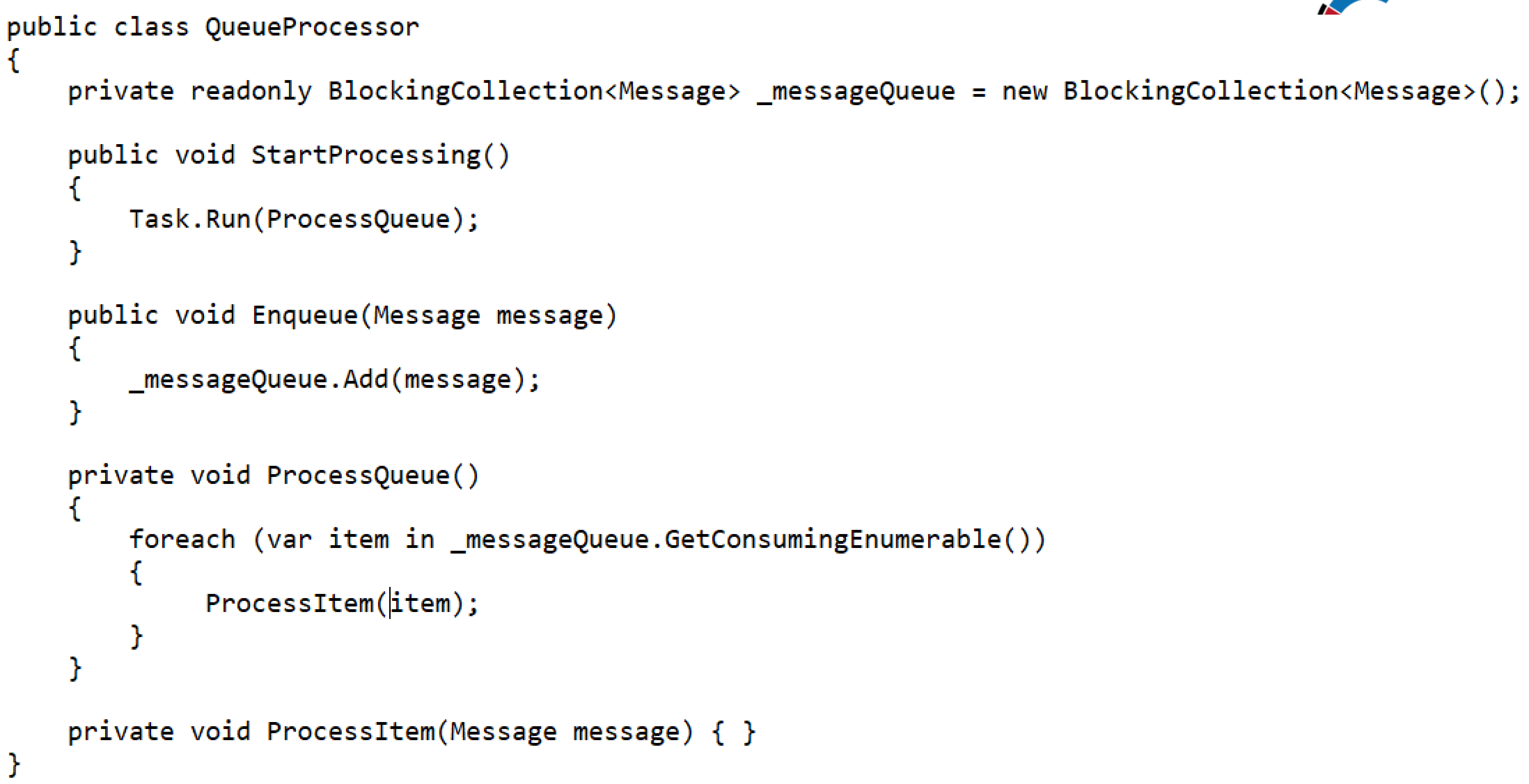
Long-running = background processing, sleep-wake

Thread pool thread blocked

- Injection solves it, but that doesn't make it correct

No `TaskCreationOptions.LongRunning`

- Creates a thread and first `await` destroys is



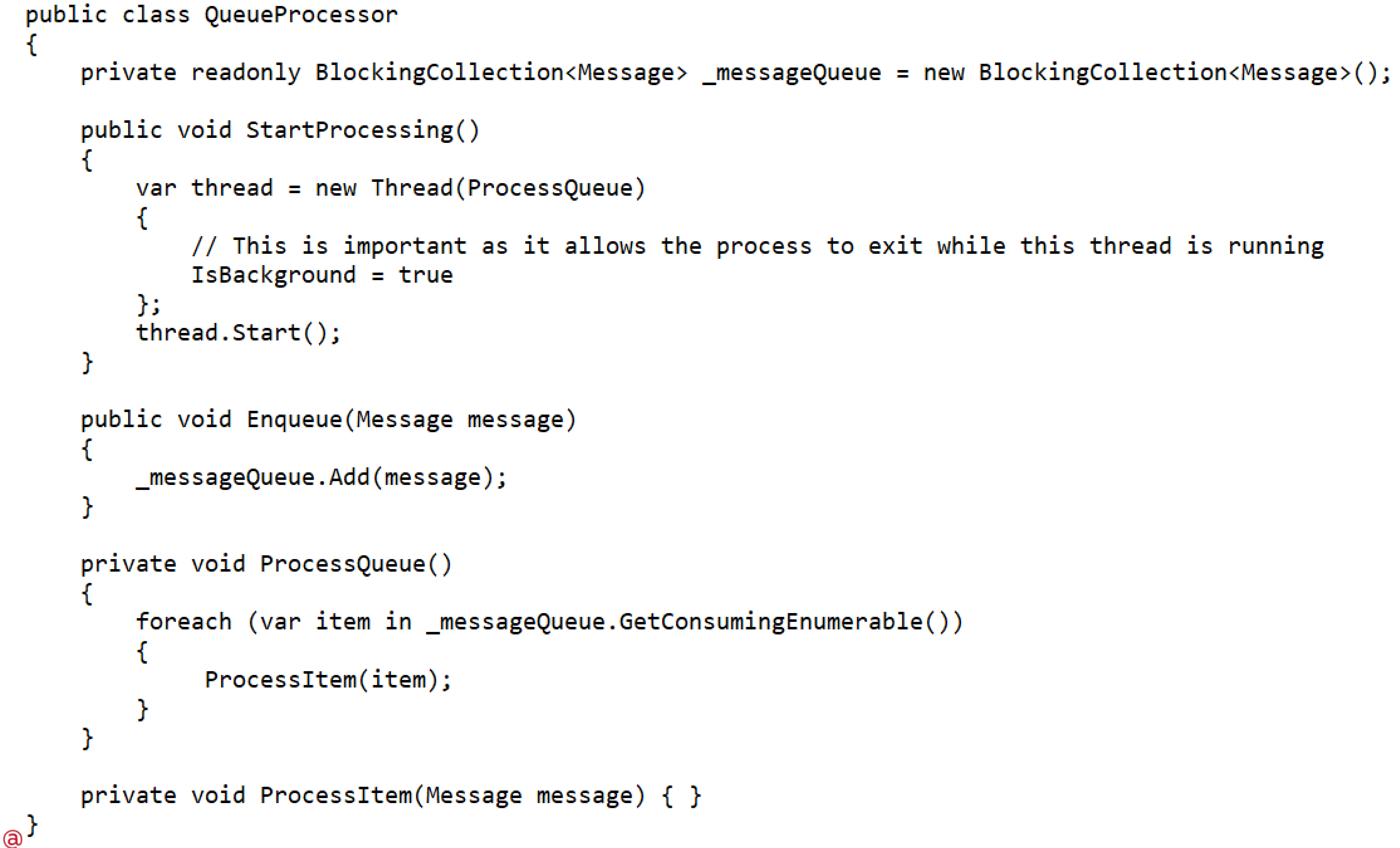
```
public class QueueProcessor
{
    private readonly BlockingCollection<Message> _messageQueue = new BlockingCollection<Message>();

    public void StartProcessing()
    {
        Task.Run(ProcessQueue);
    }

    public void Enqueue(Message message)
    {
        _messageQueue.Add(message);
    }

    private void ProcessQueue()
    {
        foreach (var item in _messageQueue.GetConsumingEnumerable())
        {
            ProcessItem(item);
        }
    }

    private void ProcessItem(Message message) { }
}
```



```
public class QueueProcessor
{
    private readonly BlockingCollection<Message> _messageQueue = new BlockingCollection<Message>();

    public void StartProcessing()
    {
        var thread = new Thread(ProcessQueue)
        {
            // This is important as it allows the process to exit while this thread is running
            IsBackground = true
        };
        thread.Start();
    }

    public void Enqueue(Message message)
    {
        _messageQueue.Add(message);
    }

    private void ProcessQueue()
    {
        foreach (var item in _messageQueue.GetConsumingEnumerable())
        {
             ProcessItem(item);
        }
    }

    private void ProcessItem(Message message) { }
}
```

await task === task.Wait();???

- *.Wait()* is blocking
  - Waiting for completions
- *await* jumps back here as soon as the operation is completed
  - Continuations and coroutines

Task.Result and Task.Wait

Sync over async

Better call synchronous API directly

Uses up to 2 threads

- Blocked + callback
- Thread pool starvation

Deadlocks via `SynchronizationContext`

- Do not invent stuff

Async must be everywhere

```
public int DoSomethingAsync()
{
  var result = CallDependencyAsync().Result;
  return result + 1;
}

public async Task<int> DoSomethingAsync()
{
  var result = await CallDependencyAsync();
  return result + 1;
}
```

Async must be everywhere

```
public string DoOperationBlocking()
{
    return Task.Run(() => DoAsyncOperation()).Result;
}
```

Blocking the thread that enters.
DoAsyncOperation will be scheduled on the default task scheduler, and remove the risk of deadlocking.
In the case of an exception, this method will throw an *AggregateException* wrapping the original exception.

Async must be everywhere

```
public string DoOperationBlocking2()
{
    return Task.Run(() => DoAsyncOperation()).GetAwaiter().GetResult();
}
```

Blocking the thread that enters.
DoAsyncOperation will be scheduled on the default task scheduler, and remove the risk of deadlocking.

Async must be everywhere

```
public string DoOperationBlocking3()
{
    return Task.Run(() => DoAsyncOperation().Result).Result;
}
```

Blocking the thread that enters, and blocking the thead pool thread inside.
In the case of an exception, this method will throw an *AggregateException* containing another *AggregateException*, containing the original exception

Async must be everywhere

```
public string DoOperationBlocking4()
{
    return Task.Run(() =>
DoAsyncOperation().GetAwaiter().GetResult()).GetAwaiter().GetResult();
}
```

Blocking the thread that enters, and blocking the theadpool thread inside.

Async must be everywhere

```
public string DoOperationBlocking5()
{
    return DoAsyncOperation().Result;
}
```

Blocking the thread that enters.
No effort has been made to prevent a present *SynchonizationContext* from becoming deadlocked.
In the case of an exception, this method will throw an *AggregateException* wrapping the original exception.

Async must be everywhere

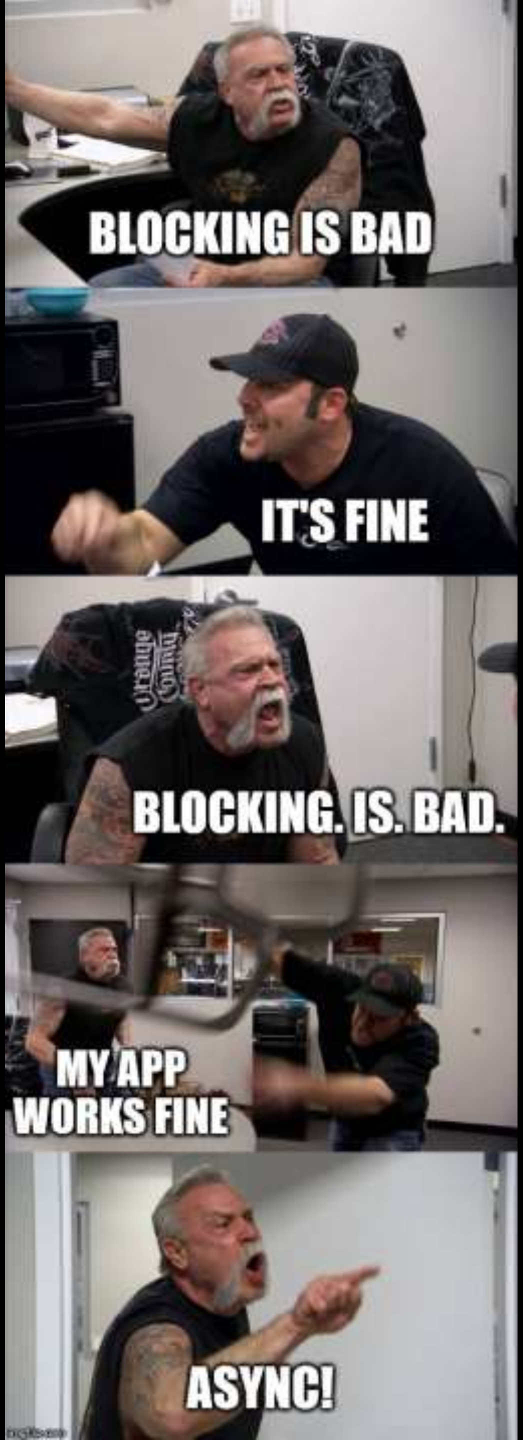
```
public string DoOperationBlocking6()
{
    return DoAsyncOperation().GetAwaiter().GetResult();
}
```

Blocking the thread that enters.
No effort has been made to prevent a present *SynchonizationContext* from becoming deadlocked.

Async must be everywhere

```
public string DoOperationBlocking7()
{
    var task = DoAsyncOperation();
    task.Wait();
    return task.GetAwaiter().GetResult();
}
```

Blocking the thread that enters.
No effort has been made to prevent a present *SynchonizationContext* from becoming deadlocked.



BLOCKING IS BAD
IT'S FINE
BLOCKING. IS. BAD.
MY APP WORKS FINE
ASYNC!

.NET ASYNC!
.NET ASYNC, EVERYWHERE!
makeameme.org

Task.Result and Task.Wait (2)

Constructors

```
public class Service : IService
{
  readonly IRemoteConnection _connection;

  public Service(IRemoteConnectionFactory connectionFactory)
  {
    _connection = connectionFactory.ConnectAsync().Result;
  }
}
```

Factory

```
public static async Task<Service> CreateAsync(IRemoteConnectionFactory
connectionFactory)
{
  return new Service(await connectionFactory.ConnectAsync());
}
```

Await instead of ContinueWith

`ContinueWith` existed before *await*

- I.e. ignores `SynchronizationContext`

```
public Task<int> DoSomethingAsync()
{
  return CallDependencyAsync().ContinueWith(task =>
  {
    return task.Result + 1;
  });
}

public async Task<int> DoSomethingAsync()
{
  var result = await CallDependencyAsync();
  return result + 1;
}
```

TaskCompletionSource<T>

Try/Set(Result/Exception/Canceled) runs inline

Very dangerous

- Re-entrancy, deadlocks, thread pool starvation, broken state, …

`TaskCreationOptions.RunContinuationsAsynchronously`

```
var tcs = new TaskCompletionSource<int>(
  TaskCreationOptions.RunContinuationsAsynchronously);
var operation = new SomeOperation();
operation.Completed += result => { tcs.SetResult(result); };
return tcs.Task;
```

Passing CancellationToken

```
public async Task<string> DoAsyncThing(CancellationToken
cancellationToken = default)
{
  var buffer = new byte[1024];
  var read = await _stream.ReadAsync(buffer, 0, buffer.Length);
  return Encoding.UTF8.GetString(buffer, 0, read);
}

public async Task<string> DoAsyncThing(CancellationToken
cancellationToken = default)
{
  var buffer = new byte[1024];
  var read = await _stream.ReadAsync(buffer, 0, buffer.Length,
cancellationToken);
  return Encoding.UTF8.GetString(buffer, 0, read);
}
```

TimerQueue

TimerQueue per CPU core

Linked list of Timers

Callbacks run on thread pool

TimerQueue uses lock

Disposing Timer removes it from TimerQueue

CancellationTokenSource for timeouts

```
public async Task<Stream> HttpClientAsyncWithCancellationBad() {
  var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
  using (var client = _httpClientFactory.CreateClient()) {
    var response = await client.GetAsync("http://backend/api/1", cts.Token);
    return await response.Content.ReadAsStreamAsync();
  }
}

public async Task<Stream> HttpClientAsyncWithCancellationGood() {
  using (var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10))) {
    using (var client = _httpClientFactory.CreateClient()) {
      var response = await client.GetAsync("http://backend/api/1", cts.Token);
      return await response.Content.ReadAsStreamAsync();
    }
  }
}
```

Timeout Task

```
public static async Task<T> TimeoutAfter<T>(this Task<T> task, TimeSpan timeout) {
  var delayTask = Task.Delay(timeout);
  var resultTask = await Task.WhenAny(task, delayTask);
  if (resultTask == delayTask) {
    throw new OperationCanceledException();
  }
  return await task;
}

public static async Task<T> TimeoutAfter<T>(this Task<T> task, TimeSpan timeout) {
  using (var cts = new CancellationTokenSource()) {
    var delayTask = Task.Delay(timeout, cts.Token);
    var resultTask = await Task.WhenAny(task, delayTask);
    if (resultTask == delayTask) {
      throw new OperationCanceledException();
    } else {
      cts.Cancel();
    }
    return await task;
  }
}
```

FlushAsync for Stream/StreamWriter

```
using (var streamWriter = new StreamWriter(s))
{
  await streamWriter.WriteAsync("Hello World");
}

using (var streamWriter = new StreamWriter(s))
{
  await streamWriter.WriteAsync("Hello World");
  await streamWriter.FlushAsync();
}
```

Timer callbacks

```
public class Pinger
{
  readonly Timer _timer;
  readonly HttpClient _client;

  public Pinger(HttpClient client)
  {
    _client = client;
    _timer = new Timer(Heartbeat, null, 1000, 1000);
  }

  public async void Heartbeat(object state)
  {
    await _client.GetAsync("http://mybackend/api/ping");
  }
}
```

Timer callbacks (2)

```
public class Pinger
{
  readonly Timer _timer;
  readonly HttpClient _client;

  public Pinger(HttpClient client)
  {
    _client = client;
    _timer = new Timer(Heartbeat, null, 1000, 1000);
  }

  public void Heartbeat(object state)
  {
    _ = DoAsyncPing();
  }

  private async Task DoAsyncPing()
  {
    await _client.GetAsync("http://mybackend/api/ping");
  }
}
```

Implicit async void

```
public class BackgroundQueue
{
  public static void FireAndForget(Action action) { }
}

BackgroundQueue.FireAndForget(async () => { await … });

public class BackgroundQueue
{
  public static void FireAndForget(Action action) { }
  public static void FireAndForget(Func<Task> action) { }
}
```

ConcurrentDictionary.GetOrAdd

```
static ConcurrentDictionary<int, Person> _cache;
var person = _cache.GetOrAdd(id, k => db.People.FindAsync(k).Result);

static ConcurrentDictionary<int, Task<Person>> _cache;
var person = _cache.GetOrAdd(id, k => db.People.FindAsync(k));

static ConcurrentDictionary<int, Lazy<Task<Person>>> _cache;
var person = await _cache.GetOrAdd(id, k => new Lazy<Task<Person>>(() =>
db.People.FindAsync(k), …)).Value;
```

Please rate this session using

.NET DeveloperDays
Mobile App

login.developerdays.pl

Session rating booth
(in Expo Hall)