Rainer Stropek | software architects gmbh

Parallel & Async in C#

Agenda

Parallele und asynchrone Programmierung sind von einem Nischenthema zu einer wichtigen Herausforderung für jeden .NET-Entwickler geworden. Spätestens am Beispiel der Windows Runtime (WinRT), dem API für Windows Store Apps, sieht man, welche Bedeutung dem Thema beizumessen ist: In WinRT sind nahezu alle Funktionen, die etwas länger dauern könnten, asynchron. Dadurch werden die vorhandenen Ressourcen besser genutzt und das UI friert nie ein.

Rainer Stropek beginnt in seinem Workshop mit Grundlagen über parallele und asynchrone Programmierung mit .NET. Er zeigt TPL und PLINQ in Aktion. Darauf aufbauend geht er auf die neuen C#-Schlüsselwörter async und await ein. Design-Guidelines für moderne, Task-basierende APIs sind der Abschluss des Workshops. Sie erfahren, wie Task Cancellation, Progress Reporting etc. richtig gemacht werden. Rainer stellt zum Workshop ein Slidedeck zum Nachschlagen zur Verfügung, im Workshop selbst stehen aber praktische Beispiele (C#) im Vordergrund.

Introduction

- software architects gmbh
- Rainer Stropek
  - Developer, Speaker, Trainer
  - MVP for Windows Azure since 2010
  - rainer@timecockpit.com
  - twitter FOLLOW ME @rstropek

time cockpit

http://www.timecockpit.com

http://www.timecockpit.com/devblog

Functional Programming Concepts In C#

A Short Recap…

A short recap…

- Lambda expressions in C#
- Anonymous delegates
  - `Func, Func<T>`
  - `Action, Action<T>`
- Concepts of functional programming in C#
  - Use first-class functions to reduce duplication of code
- Application of functional programming in LINQ
- BTW – This has been around since C# 2 and/or 3.x

```
public delegate int MathOperation(int x, int y);

static void Main(string[] args)
{
    MathOperation delegateOperation = Add;

    // Anonymous delegate
    MathOperation anonymousDelegateOperation = delegate(int num1, int num2)
    {
        return num1 + num2;
    };

    // Func<T>
    Func<int, int, int> operationFunction = Add;

    // Simple lambda functions
    operationFunction = (x, y) => x + y;
    delegateOperation = (x, y) => x + y;
}

public static int Add(int num1, int num2)
{
    return num1 + num2;
}
```

```
public delegate string[] GenerateDataOperation(int numberOfElements);

static void Main(string[] args) {
    GenerateDataOperation generateOperation = NameGenerator;

    // Multi-line anonymous delegate
    GenerateDataOperation anonymousGenerateOperation =
        delegate(int numberOfElements) {
        var result = new string[numberOfElements];
        for (int i = 0; i < numberOfElements; i++) {
            result[i] = string.Format("line_{0}", i);
        }

        return result;
    };

    // Multi-line lambda functions
    Func<int, string[]> generatorFunction = (numberOfElements) => {
            var result = new string[numberOfElements];
            for (int i = 0; i < numberOfElements; i++) {
                result[i] = string.Format("line_{0}", n);
            }

            return result;
        };
}
```

Do we need some recap on LINQ?

- `IEnumerable<T>`
  - Covariance
- C# LINQ syntax
- LINQ Support in .NET Framework
- Lambda expressions as data
  - `IQueryable<T>`
  - Expression trees

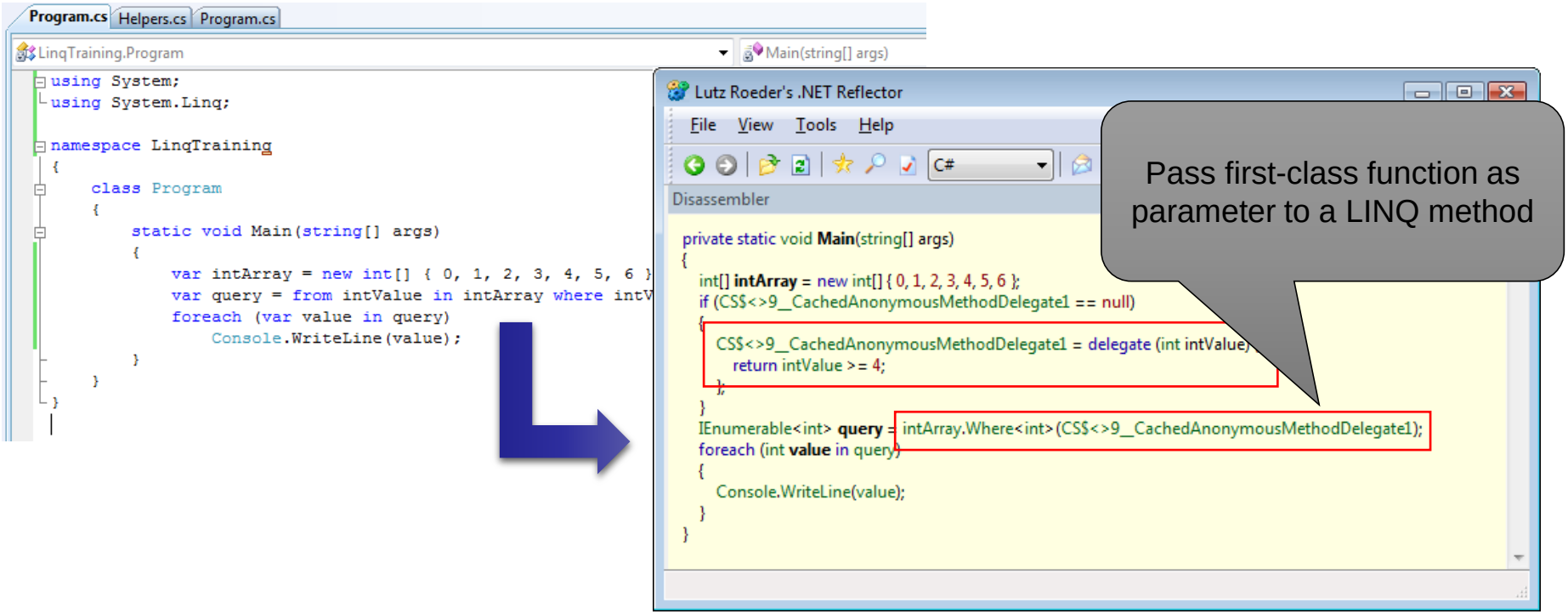
```
// Result of a method
var res5 = from training in demoData
           select training.Attendees.First();

// Result of a property
var res6 = from training in demoData
           select training.Duration;

// Anonymous type
var res7 = from training in demoData
           select new {
             training.Title,
             NumberOfAttendees =
               training.Attendees.Count()
           };

// Instance of existing type
var res8 = from training in demoData
           select new TrainingInfo {
             Title = training.Title,
             NumberOfAttendees =
               training.Attendees.Count()
           };
```

Map part of the famous map/reduce concept

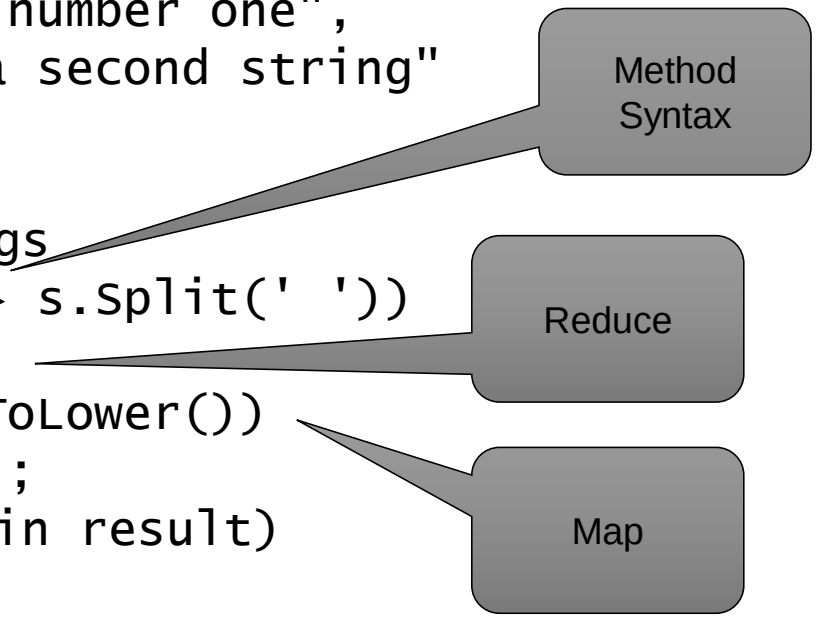
Even if you do not like `var`, here you have to use it

Query vs. Method Syntax

- Linq queries are a concept of the language, not the .NET Framework
- Compiler converts query into method calls



```
public static void SelectManyDemo()
{
   var strings = new []
   {
      "This is string number one",
      "This might be a second string"
   };

   var result = strings
      .SelectMany(s => s.Split(' '))
      .Distinct()
      .Select(s => s.ToLower())
      .OrderBy(s => s);
   foreach (var line in result)
   {
      Console.WriteLine(line);
   }
}
```



```
var res12 = from training in demoData
  group training
    by new { training.Title,
      NumberOfAttendees =
        training.Attendees.Count(
          a => a.CountryOfOrigin == "AT")
    }
    into trainingGroup
  where trainingGroup.Key.NumberOfAttendees > 0
  select new { trainingGroup.Key.Title,
    AttendeesFromAustria =
      trainingGroup.Key.NumberOfAttendees
  };
```

Method syntax embedded in LINQ query

Parallel Programming

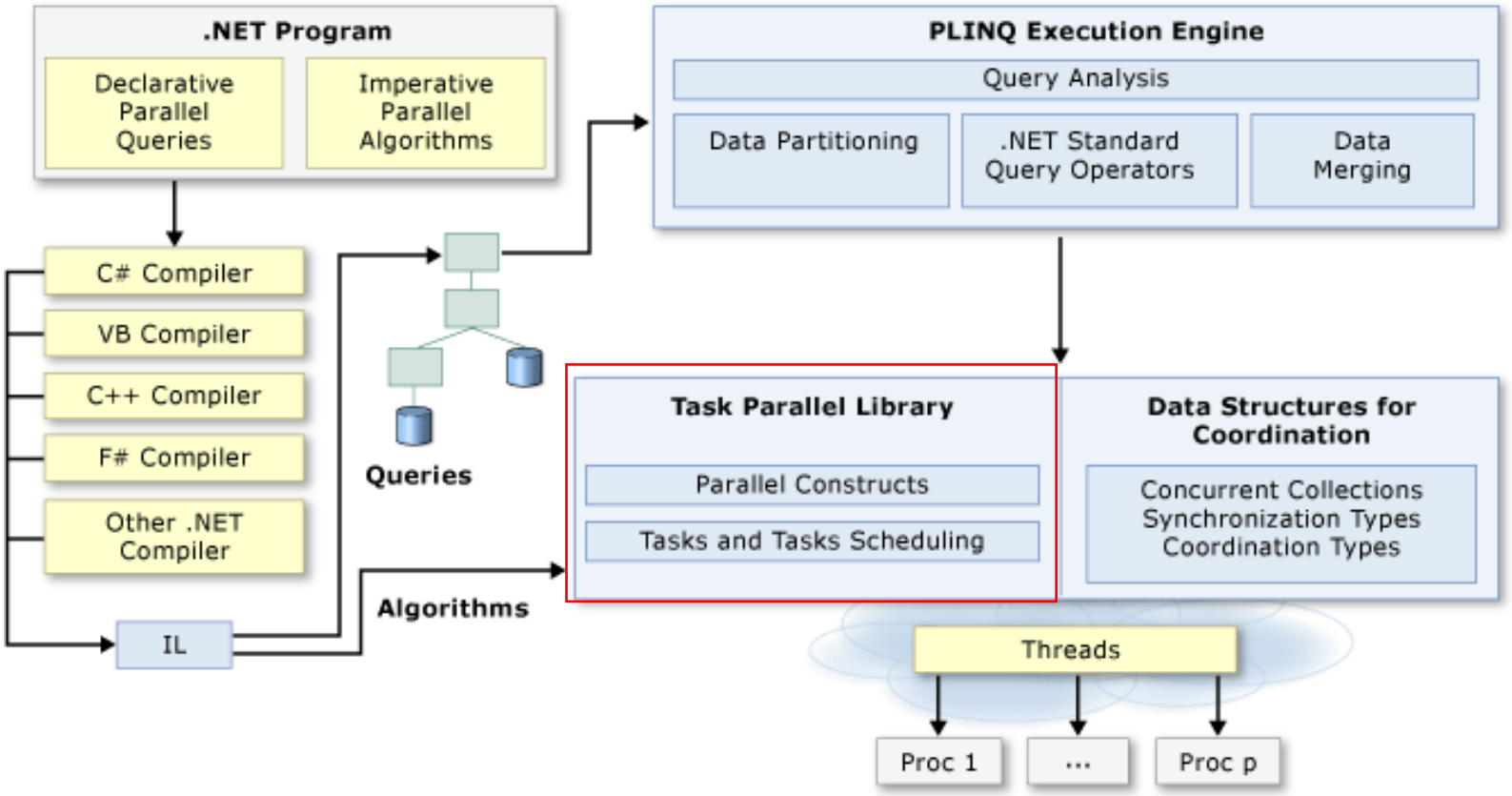
Task Parallel Library

Goals

- Understand Tasks → foundation for `async/await`
- Take a close look at C# 4.5‘s stars `async/await`

Recommended Reading

- Joseph Albahari, Threading in C#
  (from his O'Reilly book C# 4.0 in a Nutshell)
- Patterns of Parallel Programming
- Task-based Asynchronous Pattern
- A technical introduction to the Async CTP
- Using Async for File Access
- Async Performance: Understanding the Costs of Async and Await (MSDN Magazine)



See also [Patterns of Parallel Programming]

Multithreading

Pre .NET 4

- `System.Threading` Namespace
- `Thread` Klasse
- `ThreadPool` Klasse

.NET 4

- `System.Threading.Tasks` Namespace
- `Task` und `Task<TResult>` Klassen
- `TaskFactory` Klasse
- **`Parallel`** Klasse

Kurzer Überblick über Tasks

- Starten
  - `Parallel.Invoke(…)`
  - `Task.Factory.StartNew(…)`
- Warten
  - `myTask.Wait()`
  - `Task.WaitAll`
  - `Task.WaitAny`
  - `Task.Factory.ContinueWhenAll(…)`
  - `Task.Factory.ContinueWhenAny(…)`
- Verknüpfen
  - `Task.Factory.StartNew(…, TaskCreationOptions.AttachedToParent);`
- Abbrechen
  - Cancellation Tokens

Nicht in Silverlight ☹

MSDN

Schleifen - Parallel.For

```
var source = new double[Program.Size];
var destination = new double[Program.Size];

Console.WriteLine(MeasuringTools.Measure(() => {
        for (int i = 0; i < Program.Size; i++) {
            source[i] = (double)i;
        }

        for (int i = 0; i < Program.Size; i++) {
            destination[i] = Math.Pow(source[i], 2);
        }
    }));

Console.WriteLine(MeasuringTools.Measure(() => {
        Parallel.For(0, Program.Size, (i) => source[i] = (double)i);
        Parallel.For(0, Program.Size,
            (i) => destination[i] = Math.Pow(source[i], 2));
    }));
```

Schleifen - `Parallel.For`

- Unterstützung für Exception Handling
- `Break` und `Stop` Operationen
  - `Stop`: Keine weiteren Iterationen
  - `Break`: Keine Iterationen nach dem aktuellen Index mehr
  - Siehe dazu auch `ParallelLoopResult`
- `Int32` und `Int64` Laufvariablen
- Konfigurationsmöglichkeiten (z.B. Anzahl an Threads)
- Schachtelbar
  - Geteilte Threading-Ressourcen
- Effizientes Load Balancing
- U.v.m.

Nicht selbst entwickeln!

Schleifen - `Parallel.ForEach`

```
Console.WriteLine(
    "Serieller Durchlauf mit foreach: {0}",
    MeasuringTools.Measure(() =>
    {
        double sumOfSquares = 0;
        foreach (var square in Enumerable.Range(0, Program.Size).Select(
            i => Math.Pow(i, 2)))
        {
            sumOfSquares += square;
        }
    }));

Console.WriteLine(
    "Paralleler Durchlauf mit foreach: {0}",
    MeasuringTools.Measure(() =>
    {
        double sumOfSquares = 0;
        Parallel.ForEach(Enumerable.Range(0, Program.Size)
            .Select(i => Math.Pow(i, 2)), square => sumOfSquares += square);
    }));
```

Hoher Aufwand für abgesicherten Zugriff auf MoveNext/Current
→ Parallele Version oft langsamer

Von LINQ zu PLINQ

LINQ

```
var result = source
     .Where(…)
     .Select(…)
```

PLINQ

```
var result = source
     .AsParallel()
     .Where(…)
     .Select(…)
```

Aus `IEnumerable` wird `ParallelQuery`

Tipp: `AsOrdered()` erhält die Sortierreihenfolge

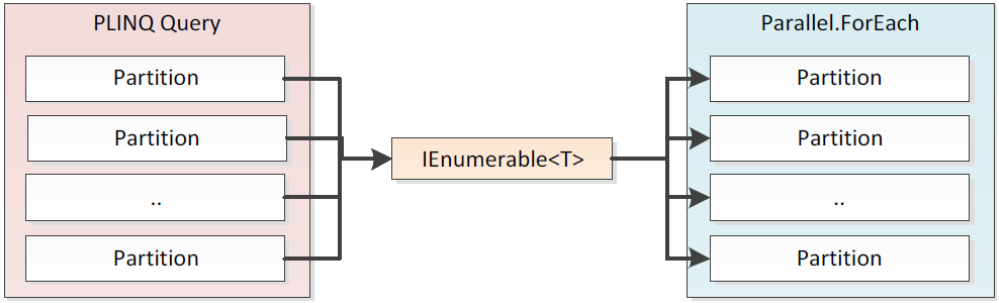
Performancetipps für PLINQ

- Allokieren von Speicher in parallelem Lambdaausdruck vermeiden
  - Sonst kann Speicher + GC zum Engpass werden
  - Wenn am Server: Server GC
- False Sharing vermeiden
- Bei zu kurzen Delegates ist Koordinationsaufwand für Parallelisierung oft höher als Performancegewinn
  - → Expensive Delegates
  - Generell: Auf richtige Granularität der Delegates achten
- `AsParallel()` kann an jeder Stelle im LINQ Query stehen
  - → Teilweise serielle, teilweise parallele Ausführung möglich
- Über `Environment.ProcessorCount` kann Anzahl an Kernen ermittelt werden
- Messen, Messen, Messen!

Was läuft hier falsch? (Code)

```
var result = new List<double>();
Console.WriteLine(
    "Paralleler Durchlauf mit Parallel.ForEach: {0}",
    MeasuringTools.Measure(() =>
    {
        Parallel.ForEach(
            source.AsParallel(),
            i =>
            {
                if (i % 2 == 0)
                {
                    lock (result)
                    {
                        result.Add(i);
                    }
                }
            });
    }));
```

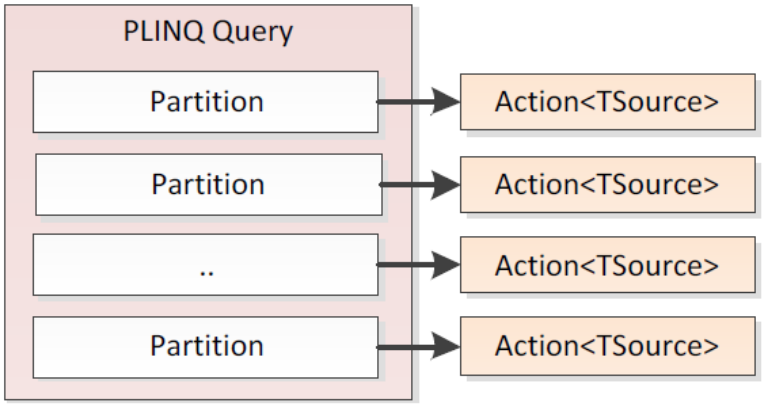
Parallel.ForEach verwendet `IEnumerable<T>` → unnötige Merge-Schritte



Was läuft hier falsch? (Code)

```
Console.WriteLine(
    "Paralleler Durchlauf mit Parallel.ForAll: {0}"
    MeasuringTools.Measure(() =>
    {
        source.AsParallel().ForAll(
            i =>
            {
                if (i % 2 == 0)
                {
                    lock (result)
                    {
                        result.Add(i);
                    }
                }
            });
    }));
```

Lock-free Collection wäre überlegenswert!



Was läuft hier falsch? (Code)

```
Console.WriteLine(
    "Serielles Lesen: {0}",
    MeasuringTools.Measure(() =>
    {
        foreach (var url in urls)
        {
            var request = WebRequest.Create(url);
            using (var response = request.GetResponse())
            {
                using (var stream = response.GetResponseStream())
                {
                    var content = new byte[1024];
                    while (stream.Read(content, 0, 1024) != 0) ;
                }
            }
        }
    }));
```

Optimal für Parallelisierung selbst bei einem Core (IO-Bound Waits)

Was läuft hier falsch? (Code)

```
Console.WriteLine(
    "Paralleles Lesen: {0}",
    MeasuringTools.Measure(() =>
    {
        Parallel.ForEach(urls, url =>
        {
            var request = WebRequest.Create(url);
            using (var response = request.GetResponse())
            {
                using (var stream = response.GetResponseStream())
                {
                    var content = new byte[1024];
                    while (stream.Read(content, 0, 1024) != 0) ;
                }
            }
        });
    }));
```

Anzahl Threads = Anzahl Cores; könnte mehr sein, da IO-Bound waits

```
Parallel.ForEach(
  urls,
  new ParallelOptions() { MaxDegreeOfParallelism = urls.Length },
  url => { … });
```

Was läuft hier falsch? (Code)

```
Console.WriteLine(
    "Paralleles Lesen: {0}",
    MeasuringTools.Measure(() =>
    {
        urls.AsParallel().WithDegreeOfParallelism(urls.Length)
            .Select(url => WebRequest.Create(url))
            .Select(request => request.GetResponse())
            .Select(response => new {
                Response = response,
                Stream = response.GetResponseStream() })
            .ForAll(stream =>
                {
                    var content = new byte[1024];
                    while (stream.Stream.Read(content, 0, 1024) != 0) ;
                    stream.Stream.Dispose();
                    stream.Response.Close();
                });
    }));
```

OK für Client, tödlich für Server!
Wenn Anzahl gleichzeitiger User wichtig ist sind andere Lösungen vorzuziehen.

```
private static void DoSomething()
{
   Action<Action> measure = (body) =>
   {
      var startTime = DateTime.Now;
      body();
      Console.WriteLine("{0} {1}",
         Thread.CurrentThread.ManagedThreadId,
         DateTime.Now - startTime);
   };

   Action calcProcess = () =>
      { for (int i = 0; i < 100000000; i++);};

   measure(() =>
      Task.WaitAll(Enumerable.Range(0, 10)
         .Select(i => Task.Run(() => measure(calcProcess)))
         .ToArray()));
}
```

This process will run in parallel

Note that we use the new `Task.Run` function here; previously you had to use `Task.Factory.StartNew`

```
Action<Action> measure = (body) => {
    var startTime = DateTime.Now;
    body();
    Console.WriteLine("{0} {1}",
      Thread.CurrentThread.ManagedThreadId,
      DateTime.Now - startTime);
};

Action calcProcess = () =>
   { for (int i = 0; i < 350000000; i++);};
Action ioProcess = () =>
   { Thread.Sleep(1000); };

// ThreadPool.SetMinThreads(5, 5);
measure(() =>{
    Task.WaitAll(Enumerable.Range(0, 10)
        .Select(i => Task.Run(() => measure(ioProcess)))
        .ToArray());
});
```

Note that this task is not compute-bound

```
Action<Action> measure = (body) =>{
    var startTime = DateTime.Now;
    body();
    Console.WriteLine("{0} {1}", Thread.CurrentThread.ManagedThreadId,
                        DateTime.Now - startTime);
};

Action calcProcess = () => { for (int i = 0; i < 350000000; i++);};
Action ioProcess = () => { Thread.Sleep(1000); };

ThreadPool.SetMinThreads(5, 5);
measure(() => Enumerable.Range(0, 10)
    .AsParallel()
    .WithDegreeOfParallelism(5)
    .ForAll(i => measure(ioProcess)));
```

Excursus - PLINQ

- Use `.AsParallel` to execute LINQ query in parallel
- Be careful if you care about ordering
  - Use `.AsOrdered` if necessary
- Use `.WithDegreeOfParallelism` in case of IO-bound tasks
- Use `.WithCancellation` to enable cancelling

```
private static void DoSomethingElse()
{
    Func<int, int> longRunningFunc = (prevResult) =>
        {
            Thread.Sleep(1000);
            return prevResult + 42;
        };

    var task = Task.Run(() => longRunningFunc(0))
        .ContinueWith(t => longRunningFunc(t.Result))
        .ContinueWith(t => longRunningFunc(t.Result));
    task.Wait();
    Console.WriteLine(task.Result);
}
```

Concat tasks using `ContinueWith`

Wait for completion of a task.

Exception Handling

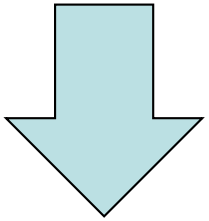
- `AggregateException`
- Remove nested Aggregate Exceptions with `Flatten`
- Use `CancellationToken` for cooperative cancellation
- Use the `Handle` method instead of loop over Aggregate Exceptions
- Use `Task.Exception`

```
var task1 = Task.Factory.StartNew(() =>
{
    throw new MyCustomException("I'm bad, but not too bad!");
});

try
{
    task1.Wait();
}
catch (AggregateException ae)
{
    // Assume we know what's going on with this particular exception.
    // Rethrow anything else. AggregateException.Handle provides
    // another way to express this. See later example.
    foreach (var e in ae.InnerExceptions)
    {
        if (e is MyCustomException)
        {
            Console.WriteLine(e.Message);
        }
        else
        {
            throw;
        }
    }

}
```

```
var task1 = Task.Factory.StartNew(() =>
{
    var child1 = Task.Factory.StartNew(() => {
            var child2 = Task.Factory.StartNew(() => {
                throw new MyCustomException("Attached child2 faulted.");
            },
            TaskCreationOptions.AttachedToParent);

            // Uncomment this line to see the exception rethrown.
            // throw new MyCustomException("Attached child1 faulted.");
        },
        TaskCreationOptions.AttachedToParent);
});

try {
    task1.Wait();
}
catch (AggregateException ae) {

    foreach (var e in ae.Flatten().InnerExceptions)
    {
       …
    }
    // or ...
   // ae.Flatten().Handle((ex) => ex is MyCustomException);

}
```

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var task1 = Task.Factory.StartNew(() =>
{
    CancellationToken ct = token;
    while (someCondition)
    {
        // Do some work...
        Thread.SpinWait(50000);
        ct.ThrowIfCancellationRequested();
    }
},
token);

// No waiting required.
```

```
foreach (var e in ae.InnerExceptions)
{
    if (e is MyCustomException)
    {
        Console.WriteLine(e.Message);
    }
    else
    {
        throw;
    }
}
```



```
ae.Handle((ex) =>
{
    return ex is MyCustomException;
});
```

```
var task1 = Task.Factory.StartNew(() =>
{
    throw new MyCustomException("Task1 faulted.");
})
.ContinueWith((t) =>
    {
        Console.WriteLine("I have observed a {0}",
            t.Exception.InnerException.GetType().Name);
    },
    TaskContinuationOptions.OnlyOnFaulted);
```

Thread Synchronisation

- Use C# `lock` statement to control access to shared variables
  - Under the hoods `Monitor.Enter` and `Monitor.Exit` is used
  - Quite fast, usually fast enough
  - Only care for lock-free algorithms if really necessary
- Note that a thread can lock the same object in a nested fashion

```
// Source: C# 4.0 in a Nutshell, O'Reilly Media
class ThreadSafe
{
  static readonly object _locker = new object();
  static int _val1, _val2;

  static void Go()
  {
    lock (_locker)
    {
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
  }
}

// This is what happens behind the scenes
bool lockTaken = false;
try
{
  Monitor.Enter(_locker, ref lockTaken);
  // Do your stuff...
}
finally
{
   if (lockTaken) Monitor.Exit(_locker);
}
```

```
// Provide a factory for instances of the Random class per thread
var tlr = new ThreadLocal<Random>(
    () => new Random(Guid.NewGuid().GetHashCode()));

var watch = Stopwatch.StartNew();

var tasks =
    // Run 10 tasks in parallel
    Enumerable.Range(0, 10)
        .Select(_ => Task.Run(() =>
            // Create a lot of randoms between 0 and 9 and calculate
            // the sum
            Enumerable.Range(0, 1000000)
                .Select(__ => tlr.Value.Next(10))
                .Sum()))
        .ToArray();
Task.WaitAll(tasks);

// Calculate the total
Console.WriteLine(tasks.Aggregate<Task<int>, int>(
    0, (agg, val) => agg + val.Result));

Console.WriteLine(watch.Elapsed);

watch = Stopwatch.StartNew();
```

Do you think this is a good solution?

```
// Provide a factory for instances of the Random class per thread
var tlr = new ThreadLocal<Random>(
    () => new Random(Guid.NewGuid().GetHashCode()));

var watch = Stopwatch.StartNew();

Console.WriteLine(
    ParallelEnumerable.Range(0, 10000000)
        .Select(_ => tlr.Value.Next(10))
        .Sum());

Console.WriteLine(watch.Elapsed);
```

Prefer PLINQ over TPL because it automatically breaks the workload into packages.

Alternatives For `lock`

- `Mutex`
- `Semaphore(Slim)`
- `ReaderWriterLock(Slim)`
- Not covered here in details

Thread Synchronization

- `AutoResetEvent`
  - Unblocks a thread once when it receives a signal from another thread
- `ManualResetEvent(Slim)`
  - Like a door, opens and closes again
- `CountdownEvent`
  - New in .NET 4
  - Unblocks if a certain number of signals have been received
- `Barrier` class
  - New in .NET 4
  - Not covered here
- `Wait` and `Pulse`
  - Not covered here

Producer/Consumer
Was läuft hier falsch? (Code)

```
var buffer = new Queue<long>();
var cancelTokenSource = new CancellationTokenSource();
var done = false;

var producer = Task.Factory.StartNew((cancelTokenObj) => {
        var counter = 10000000;
        var cancelToken = (CancellationToken)cancelTokenObj;
        try {
            while (!cancelToken.IsCancellationRequested && counter-- > 0) {
                // Here we get some data (e.g. reading it from a file)
                var value = DateTime.Now.Ticks;
                // Write it to buffer with values that have to be processed
                buffer.Enqueue(value);
            }
        }
        finally {
            done = true;
        }
    }, cancelTokenSource.Token);
```

`buffer` wird nicht gelockt

Producer/Consumer
Was läuft hier falsch? (Code)

```
var consumer = Task.Factory.StartNew((cancelTokenObj) =>
    {
        var cancelToken = (CancellationToken)cancelTokenObj;
        while (!cancelToken.IsCancellationRequested && !done)
        {
            // Get the next value to process
            lock (buffer)
            {
                var value = buffer.Dequeue();
            }

            // Here we do some expensive procesing
            Thread.SpinWait(1000);
        }
    }, cancelTokenSource.Token);
```

Prüfung ob leer fehlt

Consumer ist viel langsamer als Producer → Producer überschwemmt Consumer mit Daten

Collections für parallele Programmierung

- `System.Collections.Concurrent` für Thread-Safe Collections
  - `BlockingCollection<T>`
    Blocking und Bounding-Funktionen
  - `ConcurrentDictionary<T>`
  - `ConcurrentQueue<T>`
  - `ConcurrentStack<T>`
  - `ConcurrentBag<T>`
- Optimal zur Umsetzung von Pipelines
  - Datei wird gelesen, gepackt, verschlüsselt, geschrieben

Producer/Consumer
Was läuft hier falsch? (Code)

```
var buffer = new BlockingCollection<long>(10);
var cancelTokenSource = new CancellationTokenSource();

var producer = Task.Factory.StartNew((cancelTokenObj) => {
    var counter = 10000000;
    var cancelToken = (CancellationToken)cancelTokenObj;
    try     {
        while (!cancelToken.IsCancellationRequested && counter-- > 0) {
            // Here we get some data (e.g. reading it from a file)
            var value = DateTime.Now.Ticks;
            // Write it to the buffer with values that have to be processed
            buffer.Add(value);
        }
    }
    finally {
        buffer.CompleteAdding();
    }
}, cancelTokenSource.Token);
```

Producer/Consumer
Was läuft hier falsch? (Code)

```
var consumer = Task.Factory.StartNew((cancelTokenObj) =>
{
    var cancelToken = (CancellationToken)cancelTokenObj;
    foreach (var value in buffer.GetConsumingEnumerable())
    {
        if ( cancelToken.IsCancellationRequested )
        {
            break;
        }

        // Here we do some expensive procesing
        Thread.SpinWait(1000);
    }
}, cancelTokenSource.Token);
```

Async Programming

The big new thing in C# 5

```
private static void DownloadSomeTextSync()
{
   using (var client = new WebClient())
   {
      Console.WriteLine(
         client.DownloadString(new Uri(string.Format(
            "http://{0}",
            (Dns.GetHostAddresses("www.basta.net"))[0]))));
   }
}
```

Synchronous version of the code; would block UI thread

```
private static void DownloadSomeText()
{
    var finishedEvent = new AutoResetEvent(false);

    // Notice the IAsyncResult-pattern here
    Dns.BeginGetHostAddresses("www.basta.net", GetHostEntryFinished,
        finishedEvent);
    finishedEvent.WaitOne();
}

private static void GetHostEntryFinished(IAsyncResult result)
{
    var hostEntry = Dns.EndGetHostAddresses(result);
    using (var client = new WebClient())
    {
        // Notice the Event-based asynchronous pattern here
        client.DownloadStringCompleted += (s, e) =>
        {
            Console.WriteLine(e.Result);
            ((AutoResetEvent)result.AsyncState).Set();
        };
        client.DownloadStringAsync(new Uri(string.Format(
            "http://{0}",
            hostEntry[0].ToString())));
    }
}
```

Notice that control flow is not clear any more.

```
private static void DownloadSomeText()
{
    var finishedEvent = new AutoResetEvent(false);

    // Notice the IAsyncResult-pattern here
    Dns.BeginGetHostAddresses(
        "www.basta.net",
        (result) =>
        {
            var hostEntry = Dns.EndGetHostAddresses(result);
            using (var client = new WebClient())
            {
                // Notice the Event-based asynchronous pattern here
                client.DownloadStringCompleted += (s, e) =>
                {
                    Console.WriteLine(e.Result);
                    ((AutoResetEvent)result.AsyncState).Set();
                };
                client.DownloadStringAsync(new Uri(string.Format(
                    "http://{0}",
                    hostEntry[0].ToString())));
            }
        },
        finishedEvent);
    finishedEvent.WaitOne();
}
```

Notice how lambda expression can make control flow clearer

```
private static void DownloadSomeTextUsingTask()
{
    Dns.GetHostAddressesAsync("www.basta.net")
        .ContinueWith(t =>
        {
            using (var client = new WebClient())
            {
                return client.DownloadStringTaskAsync(new Uri(string.Format(
                       "http://{0}",
                       t.Result[0].ToString())));
            }
        })
        .ContinueWith(t2 => Console.WriteLine(t2.Unwrap().Result))
        .Wait();
}
```

Notice the use of the new Task Async Pattern APIs in .NET 4.5 here

Notice the use of lambda expressions all over the methods

Notice how code has become shorter and more readable

Rules For Async Method Signatures

- Method name ends with `Async`
- Return value
  - `Task` if sync version has return type `void`
  - `Task<T>` if sync version has return type `T`
- Avoid `out` and `ref` parameters
  - Use e.g. `Task<Tuple<T1, T2, …>>` instead

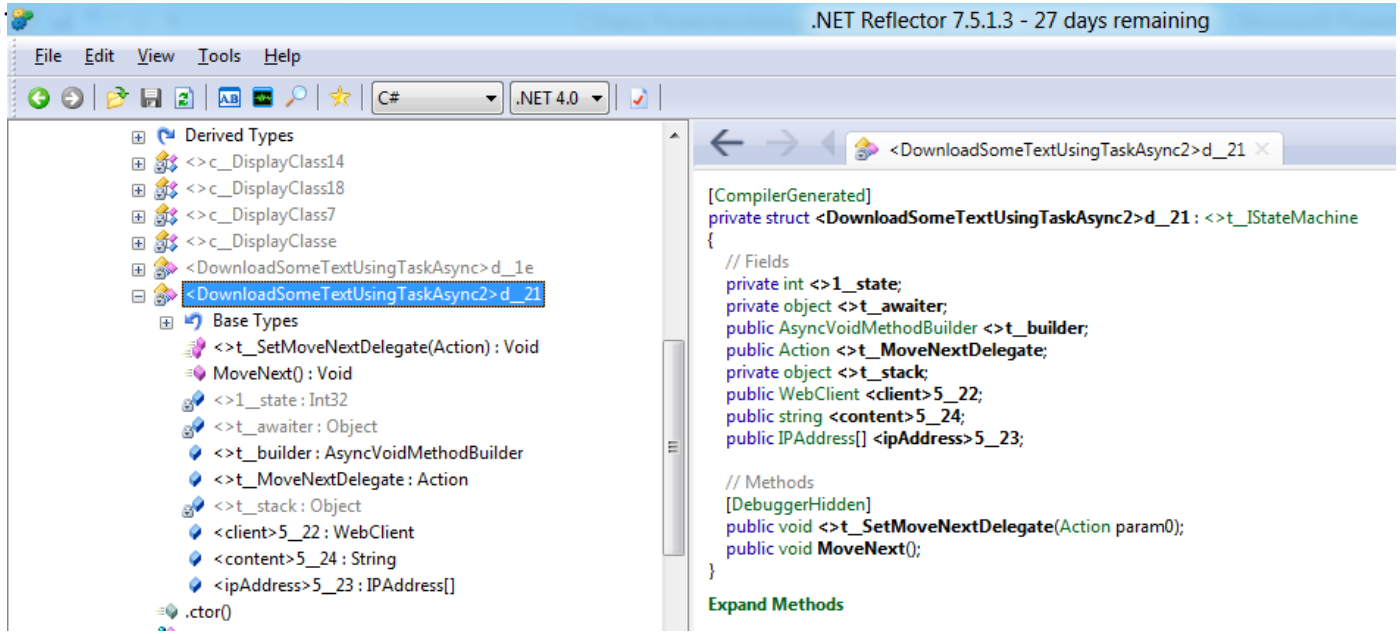
```
// Synchronous version
private static void DownloadSomeTextSync()
{
    using (var client = new WebClient())
    {
        Console.WriteLine(
            client.DownloadString(new Uri(string.Format(
                "http://{0}",
                (Dns.GetHostAddresses("www.basta.net"))[0]))));
    }
}
```

Notice how similar the sync and async versions are!

```
// Asynchronous version
private static async void DownloadSomeTextUsingTaskAsync()
{
    using (var client = new WebClient())
    {
        Console.WriteLine(
            await client.DownloadStringTaskAsync(new Uri(string.Format(
                "http://{0}",
                (await Dns.GetHostAddressesAsync("www.basta.net"))[0]))));
    }
}
```

```
private static async void DownloadSomeTextUsingTaskAsync2()
{
    using (var client = new WebClient())
    {
        try
        {
            var ipAddress = await Dns.GetHostAddressesAsync("www.basta.net");
            var content = await client.DownloadStringTaskAsync(
                new Uri(string.Format("htt://{0}", ipAddress[0])));
            Console.WriteLine(content);
        }
        catch (Exception)
        {
            Conso
        }
    }
}
```

Let's check the generated code and debug the async code



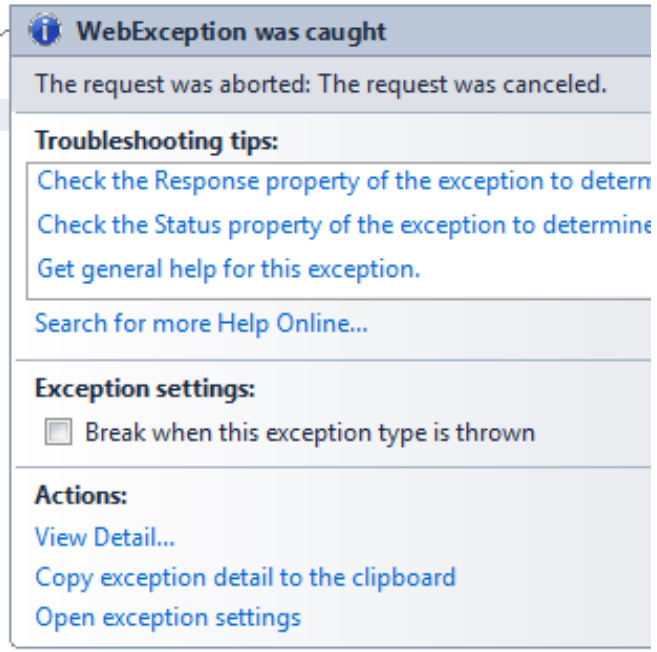
Guidelines for `async/await`

- If `Task` ended in `Canceled` state, `OperationCanceledException` will be thrown

```
private async static void CancelTask()
{
    try
    {
        var cancelSource = new CancellationTokenSource();
        var result = await DoSomethingCancelledAsync(cancelSource.Token);
        Console.WriteLine(result);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Cancelled!");
    }
}

private static Task<int> DoSomethingCancelledAsync(CancellationToken token)
{
    // For demo purposes we ignore token and always return a cancelled task
    var result = new TaskCompletionSource<int>();
    result.SetCanceled();
    return result.Task;
}
```

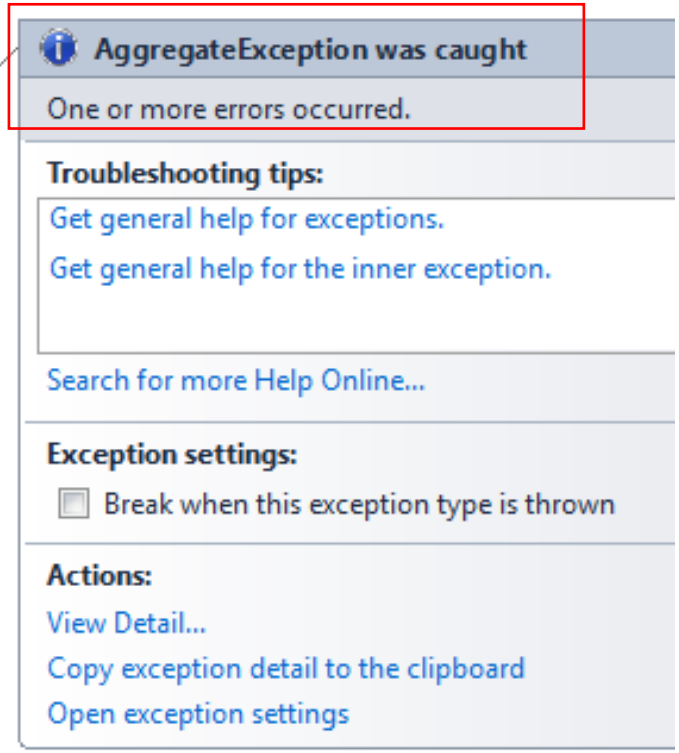
Note usage of `TaskCompletionSource<T>` here

```
private static async void DownloadSomeTextUsingTaskAsync2()
{
    using (var client = new WebClient())
    {
        try
        {
            var ipAddress = await Dns.GetHostAddressesAsync("www.basta.net");
            new Thread(() =>
                {
                    Thread.Sleep(100);
                    client.CancelAsync();
                }).Start();
            var content = await client.DownloadStringTaskAsync(
                new Uri(string.Format("http://{0}", ipAddress[0])));
            Console.WriteLine(content);
        }
        catch (Exception)
        {
            Console.WriteLine("Exception!");
        }
    }
}
```



Note that async API of `WebClient` uses existing cancellation logic instead of `CancellationTokenSource`

```
namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Task.WaitAll(new[] {
                    Task.Run(() =>
                    {
                        Thread.Sleep(1000);
                        throw new ArgumentException();
                    }),
                    Task.Run(() =>
                    {
                        Thread.Sleep(2000);
                        throw new InvalidOperationException();
                    })
                });
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```



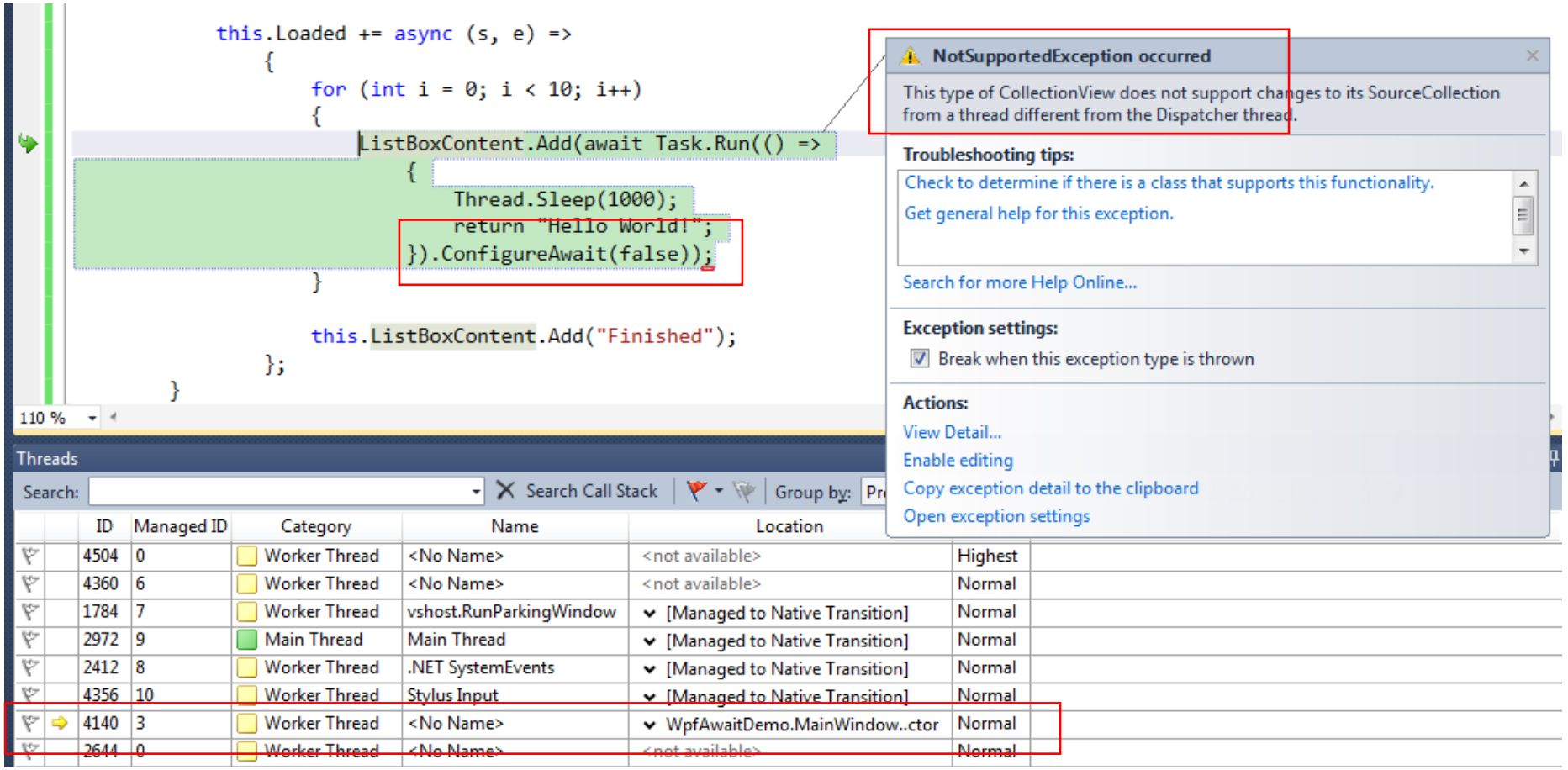
Guidelines for `async/await`

- Caller runs in parallel to awaited methods
- Async methods sometimes do not run async (e.g. if task is already completed when `async` is reached)

Guidelines for `async/await` (UI Layer)

- `async/await` use `SynchronizationContext` to execute the awaiting method → UI thread in case of UI layer
- Use `Task.ConfigureAwait` to disable this behavior
  - E.g. inside library to enhance performance

```
public partial class MainWindow : Window
{
public MainWindow()
{
    this.DataContext = this;
    this.ListBoxContent = new ObservableCollection<string>();
    this.InitializeComponent();
    this.ListBoxContent.Add("Started");

    this.Loaded += async (s, e) =>
        {
            for (int i = 0; i < 10; i++)
            {
                ListBoxContent.Add(await Task.Run(() =>
                    {
                        Thread.Sleep(1000);
                        return "Hello World!";
                    }));
            }

            this.ListBoxContent.Add("Finished");
        };
}

public ObservableCollection<string> ListBoxContent { get; private set; }
```


this.Loaded += async (s, e) =>
{
for (int i = 0; i < 10; i++)
{
ListBoxContent.Add(await Task.Run(() =>
{
Thread.Sleep(1000);
return "Hello World!";
}).ConfigureAwait(false));
}
this.ListBoxContent.Add("Finished");
};
}
110 %
NotSupportedException occurred
This type of CollectionView does not support changes to its SourceCollection from a thread different from the Dispatcher thread.
Troubleshooting tips:
Check to determine if there is a class that supports this functionality.
Get general help for this exception.
Search for more Help Online...
Exception settings:
Break when this exception type is thrown
Actions:
View Detail...
Enable editing
Copy exception detail to the clipboard
Open exception settings
Threads
Search:
Search Call Stack
Group by:
ID Managed ID Category Name Location
4504 0 Worker Thread <No Name> <not available> Highest
4360 6 Worker Thread <No Name> <not available> Normal
1784 7 Worker Thread vshost.RunParkingWindow [Managed to Native Transition] Normal
2972 9 Main Thread Main Thread [Managed to Native Transition] Normal
2412 8 Worker Thread .NET SystemEvents [Managed to Native Transition] Normal
4356 10 Worker Thread Stylus Input [Managed to Native Transition] Normal
4140 3 Worker Thread <No Name> WpfAwaitDemo.MainWindow..ctor Normal
2644 0 Worker Thread <No Name> <not available> Normal

Guidelines For Implementing Methods Ready For `async/await`

- Return `Task/Task<T>`
- Use postfix `Async`
- If method support cancelling, add parameter of type `System.Threading.CancellationToken`
- If method support progress reporting, add `IProgress<T>` parameter
- Only perform very limited work before returning to the caller (e.g. check arguments)
- Directly throw exception only in case of *usage* errors

```
public class Program : IProgress<int>
{
    static void Main(string[] args)
    {
        var finished = new AutoResetEvent(false);
        PerformCalculation(finished);
        finished.WaitOne();
    }

    private static async void PerformCalculation(AutoResetEvent finished)
    {
        Console.WriteLine(await CalculateValueAsync(
            42,
            CancellationToken.None,
            new Program()));
        finished.Set();
    }

    public void Report(int value)
    {
        Console.WriteLine("Progress: {0}", value);
    }
```

```
    private static Task<int> CalculateValueAsync(
        int startingValue,
        CancellationToken cancellationToken,
        IProgress<int> progress)
    {
        if (startingValue < 0)
        {
            // Usage error
            throw new ArgumentOutOfRangeException("startingValue");
        }

        return Task.Run(() =>
            {
                int result = startingValue;
                for (int outer = 0; outer < 10; outer++)
                {
                    cancellationToken.ThrowIfCancellationRequested();

                    // Do some calculation
                    Thread.Sleep(500);
                    result += 42;

                    progress.Report(outer + 1);
                }

                return result;
            });
    }
}
```

Note that this pattern is good for compute-bound jobs

```
private static async void PerformCalculation(AutoResetEvent finished)
{
    try
    {
        var cts = new CancellationTokenSource();
        Task.Run(() =>
            {
                Thread.Sleep(3000);
                cts.Cancel();
            });
        var result = await CalculateValueAsync(
            42,
            cts.Token,
            new Program());
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Cancelled!");
    }

    finished.Set();
}
```

Note cancellation and handling of `OperationCanceledException`.

```
private static Task<int> CalculateValueAsync(
    int startingValue,
    CancellationToken cancellationToken,
    IProgress<int> progress)
{
    if (startingValue < 0)
    {
        // By definition the result has to be 0 if startingValue < 0
        return Task.FromResult(0);
    }

    return Task.Run(() =>
        {
            […]
        });
}
```

Note that you could use `TaskCompletionSource` instead

Note how `Task.FromResult` is used to return a pseudo-task

Was läuft hier falsch? (Code)

```
Console.WriteLine(
    "Paralleles Lesen mit TaskFactory: {0}",
    MeasuringTools.Measure(() =>
        {
            var tasks = new Task[urls.Length];
            for (int i = 0; i < urls.Length; i++)
            {
                tasks[i] = Task.Factory.StartNew(() => ReadUrl(urls[i]));
            }

            Task.WaitAll(tasks);
        }
    ));
…

private static void ReadUrl(object url)
{
    …
}
```

Delegate verwendet Wert von i aus dem Main Thread → `IndexOutOfRangeException`

Was läuft hier falsch? (Code)

```
// Variante 1
…
var tasks = new Task[urls.Length];
for (int i = 0; i < urls.Length; i++)
{
    var tmp = i;
    tasks[i] = Task.Factory.StartNew(() => ReadUrl(urls[tmp]));
}
…
```

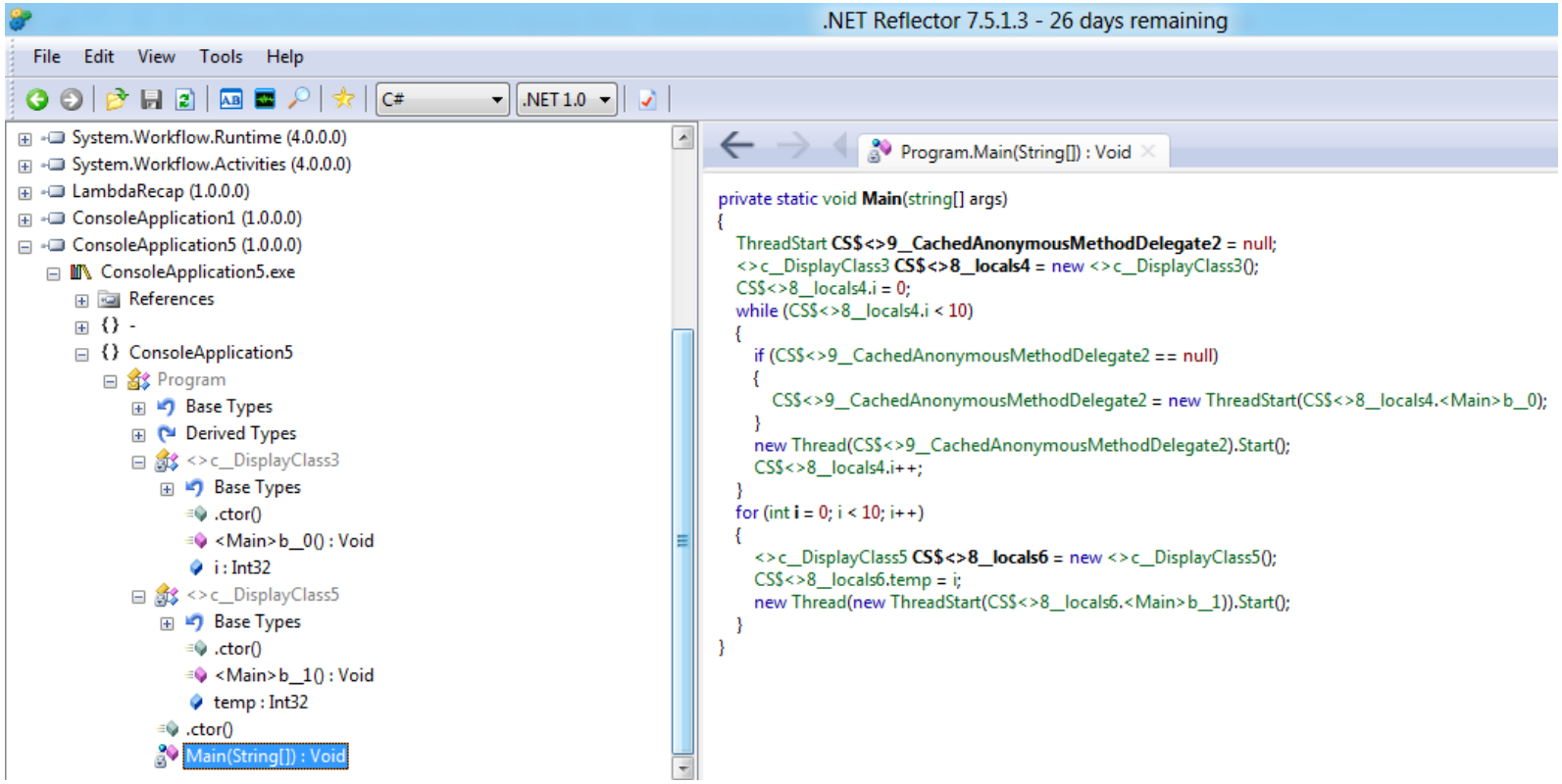
Durch lokale Variable wird delegate unabhängig; mehr zum Thema unter dem Schlagwort *Closures*

```
// Variante 2
var tasks = new Task[urls.Length];
for (int i = 0; i < urls.Length; i++)
{
    tasks[i] = Task.Factory.StartNew(ReadUrl, urls[i]);
}
```

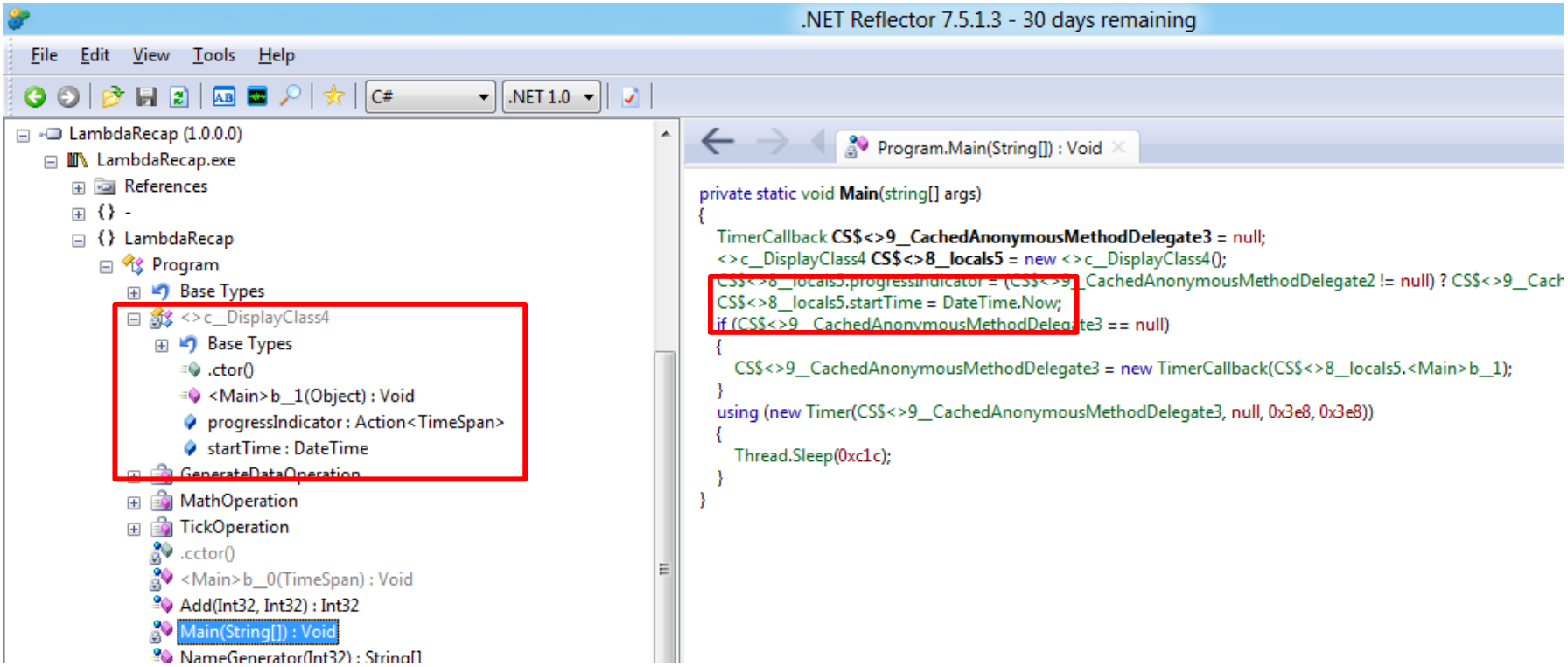
State object wird an delegate übergeben

```
for (var i = 0; i < 10; i++)
{
    new Thread(() => Console.Write(i)).Start();
}

for (var i = 0; i < 10; i++)
{
    var temp = i;
    new Thread(() => Console.Write(temp)).Start();
}
```

You have to be careful with closures and multi-threading.



```
// Setup timer using an action (notice the closure here)
var startTime = DateTime.Now;
using (var timer = new System.Threading.Timer(
    _ => progressIndicator(DateTime.Now - startTime), null, 1000, 1000))
{
    Thread.Sleep(3100);
}
```



F&A

Danke für die Teilnahme!

time cockpit is the leading time tracking solution for knowledge workers. Graphical time tracking calendar, automatic tracking of your work using signal trackers, high level of extensibility and customizability, full support to work offline, and SaaS deployment model make it the optimal choice especially in the IT consulting business.

Try time cockpit for free and without any risk. You can get your trial account at http://www.timecockpit.com. After the trial period you can use time cockpit for only 0,20€ per user and month without a minimal subscription time and without a minimal number of users.

time cockpit ist die führende Projektzeiterfassung für Knowledge Worker. Grafischer Zeitbuchungskalender, automatische Tätigkeitsaufzeichnung über Signal Tracker, umfassende Erweiterbarkeit und Anpassbarkeit, volle Offlinefähigkeit und einfachste Verwendung durch SaaS machen es zur Optimalen Lösung auch speziell im IT-Umfeld.

Probieren Sie time cockpit kostenlos und ohne Risko einfach aus. Einen Testzugang erhalten Sie unter http://www.timecockpit.com. Danach nutzen Sie time cockpit um nur 0,20€ pro Benutzer und Tag ohne Mindestdauer und ohne Mindestbenutzeranzahl.