Exceptional Exceptions in .NET

Adam Sitnik

About myself

Work:

- Energy Trading
- Energy Production Optimization
- Balance Settlement
- Critical Events Detection

Open Source:

- **BenchmarkDotNet**
- Core CLR
- corefxlab

Have you ever..

- Wondered if finally block are guaranteed to be executed?
- Encountered a silent error?
- Encountered an exception that omitted your catch blocks?
- Failed to find reason for exceptional behaviour?
- Measured performance for *throw exception* vs *return false*?

Single error in logs

```
18:44:38 [Error] The communication object, System.ServiceModel.Channels.
ServiceChannel, cannot be used for communication because it is in the Faulted
state.

Exception rethrown at [0]:
   at System.Runtime.Remoting.Proxies.RealProxy.HandleReturnMessage(..)
   at System.Runtime.Remoting.Proxies.RealProxy.PrivateInvoke(..)
   at System.ServiceModel.ICommunicationObject.Close(TimeSpan timeout)
   at System.ServiceModel.ClientBase`1.System.ServiceModel.(..).Close
   at Samples.Dispose()
```

The code that caused the error

```
var client = new WcfClient();

try
{
    client.Open();

    client.Save(data);
}
finally
{
    client.Dispose();
}
```

What happens if finally block throws an exception?

```
try
{
    try
    {
        throw new Exception("first");
    }
    finally
    {
        throw new Exception("second");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

```
second
```

Reason

C# 4 Language Specification:

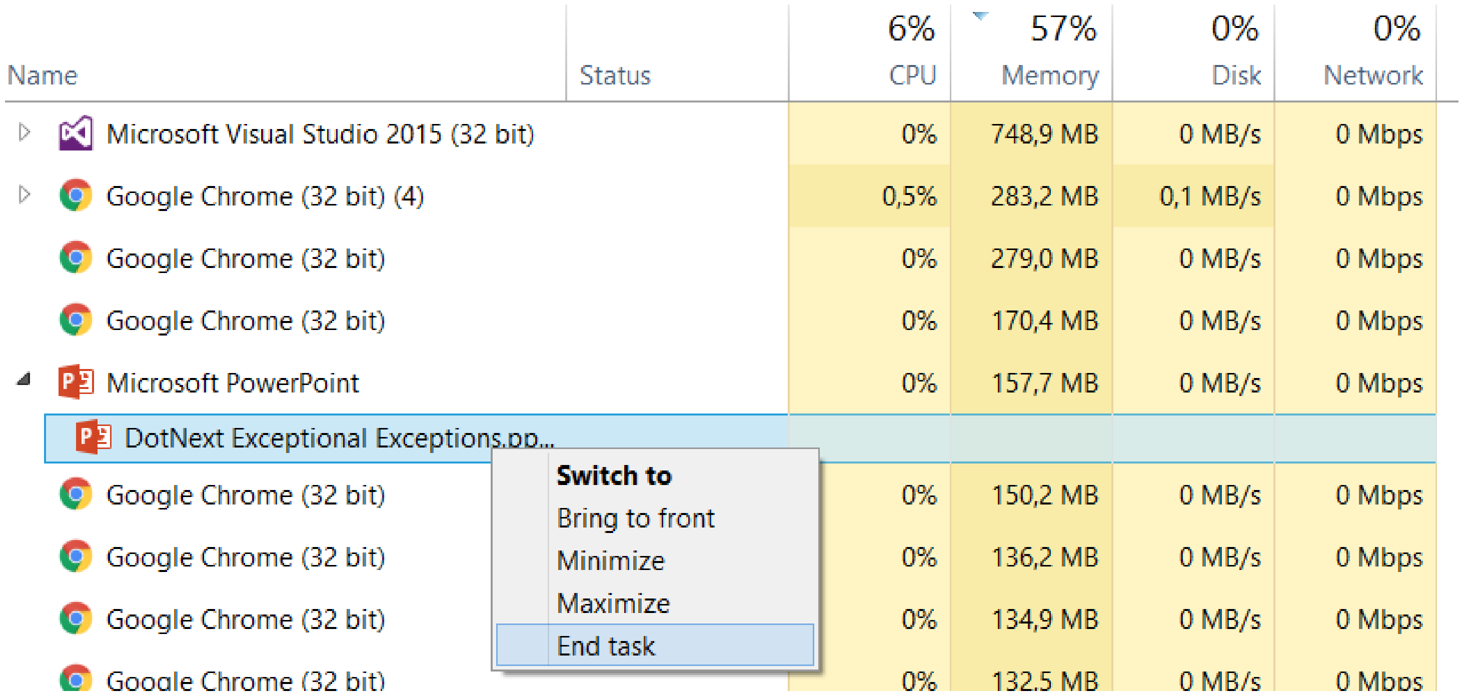
§ 8.9.5: If the finally block throws another exception, processing of the current exception is terminated.

Does the finally block ALWAYS execute?

NO!

- Win32 TerminateThread()
- Win32 TerminateProcess()
- System.Environment.FailFast (*CriticalFinalizerObject)
- Corrupted State Exception*
- Obvious things like pull the plug etc.

What happens to all resources when process gets killed and finally blocks are not executed?



Name	Status	6% CPU	57% Memory	0% Disk	0% Network
Microsoft Visual Studio 2015 (32 bit)		0%	748,9 MB	0 MB/s	0 Mbps
Google Chrome (32 bit) (4)		0,5%	283,2 MB	0,1 MB/s	0 Mbps
Google Chrome (32 bit)		0%	279,0 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	170,4 MB	0 MB/s	0 Mbps
Microsoft PowerPoint		0%	157,7 MB	0 MB/s	0 Mbps
DotNext Exceptional Exceptions.pp...					
Google Chrome (32 bit)		0%	150,2 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	136,2 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	134,9 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	132,5 MB	0 MB/s	0 Mbps

ThreadAbortedException

- Thread.Abort()
- AppDomain.Unload()
- You can catch it, but anyway .NET will rethrow it

Can ThreadAbortedException interrupt finally?

NO!

```
void Execute(Action first, Action second)
{
    try { } // empty on purpose!
    finally
    {
        first();
        // thread abort can't happen here!
        second();
    }
}
```

How to minimalize chance for failure in finally block?

- Keep it as simple as possible: avoid allocations etc
- Use defensive programming

```
if(handle.IsAllocated)
      handle.Free();

stream?.Dispose();
```

- Use Constrained Execution Regions (CER)

Constrained Execution Regions: Sample

```
RuntimeHelpers.PrepareConstrainedRegions();
try
{
    // perform some important operation here
}
finally
{
    // perform cleanup here
}
```

Constrained Execution Regions: What CLR does

Before entering try block:

- load all assemblies
- compile all that code (non-virtual [ReliabillityContract] methods)
- run static constructors
- check if 48 KB of stack space is available

Constrained Execution Regions: Benefits

Elimination of potential exceptions:

- FileLoadException, FileNotFoundException
- BadImageFormatException, InvalidProgramException
- FieldAccessException, MethodAccessException, MissingFieldException, and MissingMethodException
- TypeInitializationException
- StackOverflowException

Constrained Execution Regions: where it throws

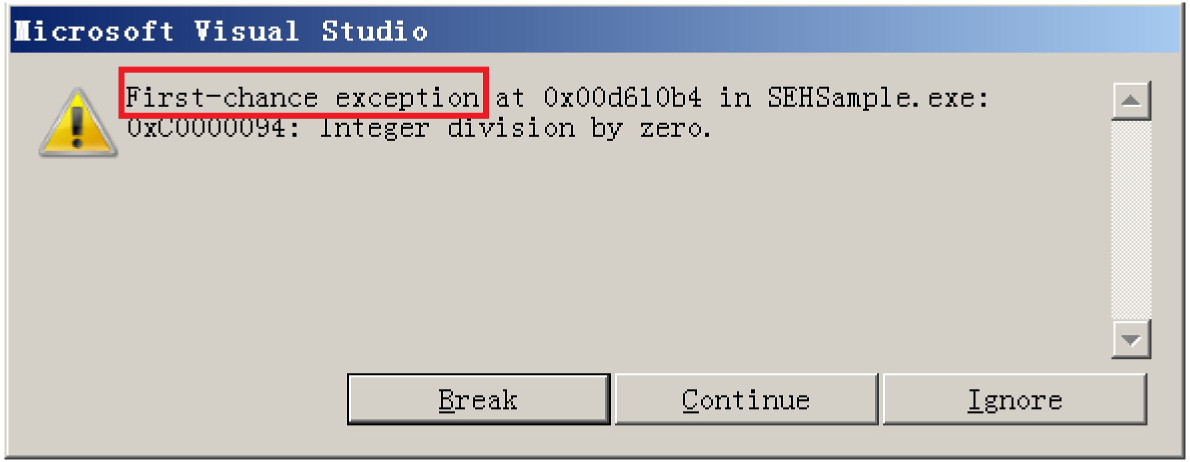
```
void ThrowingCER()
{
    RuntimeHelpers.PrepareConstrainedRegions();
    try
    {
        // will never be executed
    }
    finally
    {
        // static ctor fails with exception
    }
}
```

```
try
{
    ThrowingCER();
}
catch (Exception ex)
{
    // can be handled here!!
}
```

Exceptional Exceptions

Do you know any of these?

FirstChanceException


```
First-chance exception at 0x00d610b4 in SEHSample.exe:
0xC0000094: Integer division by zero.
```

- AppDomain’s **event**
- It occurs **before** CLR starts looking for EH block

It’s not what you think

- CLS: language must support catching & throwing : System.Exception
- CLR allows **any** object to be thrown!
- CLR 2.0 introduced **RuntimeWrappedException**

How to throw anything w/ C# (don’t do this at work!)

```
public static Action<TThrown> BuildThrowingMethod<TThrown>()
{
    var dynamicMethod = new DynamicMethod(
        "Throw",
        returnType: typeof(void),
        parameterTypes: new[] { typeof(TThrown) });

    var cilGenerator = dynamicMethod.GetILGenerator();
    cilGenerator.Emit(OpCodes.Ldarg_0); // load the argument
    cilGenerator.Emit(OpCodes.Throw); // throw whatever it is!

    return (Action<TThrown>)dynamicMethod
                .CreateDelegate(typeof(Action<TThrown>));
}
```

We can throw literally anything but it’s an anti-pattern!!

Catching RWE

```
try {
    Action<string> throwingMethod =
         ThrowAnythingMethodBuilder.BuildThrowingMethod<string>();

    throwingMethod.Invoke("I can throw whatever I want");
}
catch (Exception wrappedException) {
    Console.WriteLine(wrappedException.Message);
}
```

```
An object that does not derive from System.Exception has been wrapped in a
RuntimeWrappedException.
```

TargetInvocationException

```
class Calc {
    static int Sum(int left, int right) => checked(left + right);
}


var method = typeof(Calc).GetMethod("Sum", BindingFlags.Static | BindingFlags.NonPublic);
try {
    var result = method.Invoke(null, new object[] { int.MaxValue, int.MaxValue });
}
catch (OverflowException) {
    Console.WriteLine("Overflow");
}
catch(TargetInvocationException ex) {
    Console.WriteLine("Reflection wraps all exceptions!" + ex.InnerException);
}
```

Does dynamic wraps exceptions too?

```
public class Calc {
    public int Sum(int left, int right) => checked(left + right);
}


dynamic instance = Activator.CreateInstance<Calc>();
try {
    var result = instance.Sum(int.MaxValue, int.MaxValue);
}
catch (OverflowException) {
    Console.WriteLine("Overflow");
}
catch (TargetInvocationException ex) {
    Console.WriteLine("Got wrapped" + ex.InnerException);
}
```

```
Overflow
```

TypeInitializationException

```
class Pool
{
    static byte[] buffer;

    static Pool()
    {
        buffer = new byte[int.MaxValue];
    }

    Span<byte> Acquire(int length)(..)
}
```

```
try {
    Pool.Acquire(100);
}
catch (OutOfMemoryException) {
    Console.WriteLine("OOM");
}
catch (TypeInitializationException ex)
{
    Console.WriteLine("Wrapped!"
        + ex.InnerException);
}
```

Native to Managed translation

Native Exception	Managed Exception
EXCEPTION_STACK_OVERFLOW	System.StackOverflowException
EXCEPTION_ACCESS_VIOLATION	System.AccessViolationException
EXCEPTION_IN_PAGE_ERROR	System.Runtime.InteropServices .SEHException
EXCEPTION_ILLEGAL_INSTRUCTION	
EXCEPTION_INVALID_DISPOSITION	
EXCEPTION_NONCONTINUABLE_EXCEPTION	
EXCEPTION_PRIV_INSTRUCTION	
STATUS_UNWIND_CONSOLIDATE	

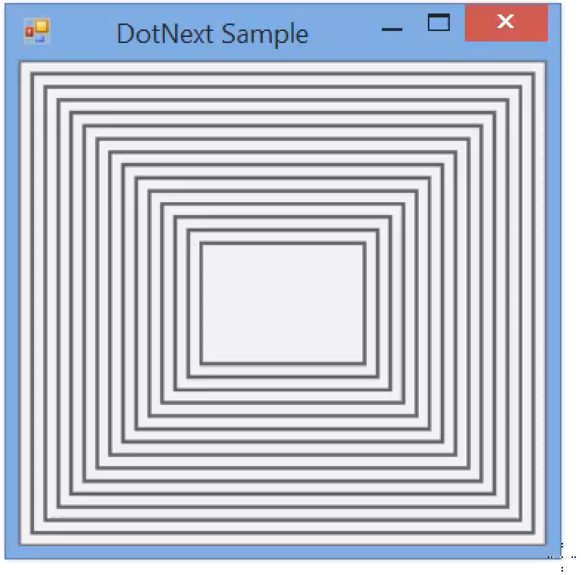
How to catch Corrupted State Exceptions (CSEs)?

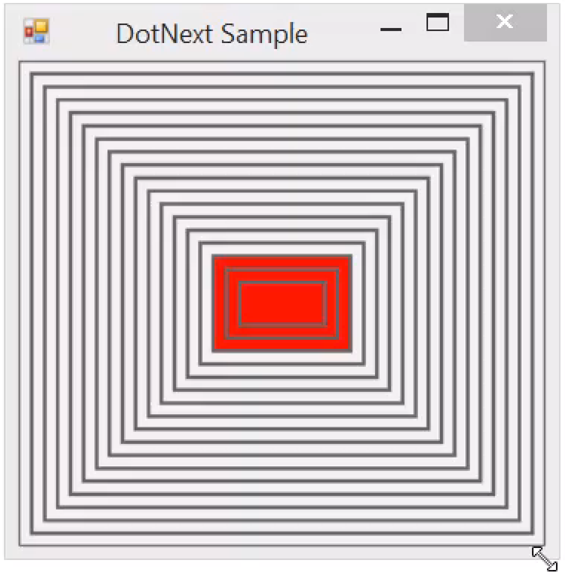
```
[HandleProcessCorruptedStateExceptions] // mandatory
[SecurityCritical] // also mandatory
public void CanCatchCSE()
{
    try
    {
        CallNativeCode();
    }
    catch (Exception ex)
    {
        Handle(ex);
    }
}
```

.NET Core: Breaking changes!

„Unrecoverable exceptions should not be getting caught and will be dealt with on a broad level by a high-level catch-all handler. Therefore, users are not expected to have code that catches these explicit exceptions. The list of unrecoverable exceptions are:

- *StackOverflowException*
- *SEHException*
- *ExecutionEngineException*
- *AccessViolationException*”


DotNext Sample


DotNext Sample

How to avoid StackOverflowException

- Redesign your code to use tail recursion
- Redesign your code to use iterative approach
- Set limits
- Use void RuntimeHelper.EnsureSufficientExecutionStack()
- Use bool TryEnsureSufficientExecutionStack() (.NET Core 1.1)

Can OutOfMemoryException be caught?

It depends on who tried to allocate memory ;)

User:

- Creating new object
- Creating new array
- Boxing
- & more

CLR:

- Loading assemblies
- JITting
- & more

Not enough contiguous memory is available

- Memory leaks
- Heap fragmentation (LOH and/or unmanaged heap)

```
GCSettings.LargeObjectHeapCompactionMode =
GCLargeObjectHeapCompactionMode.CompactOnce;
```

- Hit 32-bit address space limit (2GB by default, can set to 3 GB)
- Tried to allocate array > 2GB, set <gcAllowVeryLargeObjects enabled="true" />
- Other „memory hungry” process took all the available memory from OS
- Reached the configurable limit for the process

ExecutionEngineException

- Thrown by CLR when it detects internal corruption or bug in itself.
- No catch block or finally blocks will be executed after

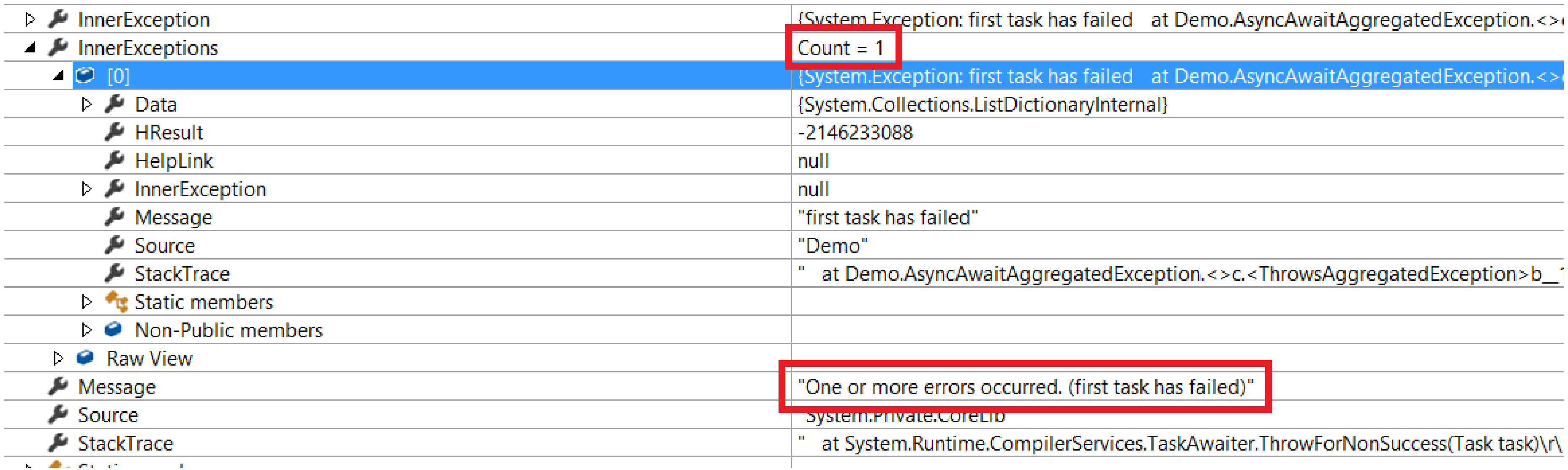
AggregateException

```
Task.Factory.StartNew(() =>
{
    Task.Factory.StartNew(
        () => { throw new Exception("first task has failed"); },
        TaskCreationOptions.AttachedToParent);

    Task.Factory.StartNew(
        () => { throw new Exception("second task has failed"); },
        TaskCreationOptions.AttachedToParent);
});
```

How async/await handles AggregateExceptions?

```
public async Task Demo()
{
    try
    {
        await ThrowsAggregatedExceptionAsync();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Catching awaited AggregatedException



▷ InnerException	{System.Exception: first task has failed at Demo.AsyncAwaitAggregatedException.<>
◢ InnerExceptions	Count = 1
◢ [0]	{System.Exception: first task has failed at Demo.AsyncAwaitAggregatedException.<>
▷ Data	{System.Collections.ListDictionaryInternal}
HResult	-2146233088
HelpLink	null
▷ InnerException	null
Message	"first task has failed"
Source	"Demo"
StackTrace	" at Demo.AsyncAwaitAggregatedException.<>c.<ThrowsAggregatedException>b__
▷ Static members	
▷ Non-Public members	
▷ Raw View	
Message	"One or more errors occurred. (first task has failed)"
Source	System.Private.CoreLib
StackTrace	" at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\

The information about other exceptions has been LOST!

How to handle AggregatedException today

```
async Task DemoAsync() {
    Task firstTask = ThrowsAggregatedExceptionAsync();

    Task errorHandler = firstTask.ContinueWith(previous => Handle(previous.Exception),
        TaskContinuationOptions.OnlyOnFaulted);

    Task processingResults = firstTask.ContinueWith(ProcessResult,
        TaskContinuationOptions.OnlyOnRanToCompletion);

    await Task.WhenAny(errorHandler, processingResults);
}

void Handle(AggregateException ex) {
    foreach (var exception in ex.Flatten().InnerExceptions)
        Console.WriteLine(exception.Message);
}
```

How to deny child task attaching

- `Task.Factory.StartNew(action, TaskCreationOptions.DenyChildAttach);`
- `Task.Run(action);`
- A must have setting for every Task returning method for frameworks

What if Fire&Forget task fails with exception?

```
private void Fail()
{
    throw new Exception("please help me");
}

public void Demo()
{
    Task.Run(() => Fail());
    // the result is not stored or checked anywhere!!
}
```

Unobserved Task Exceptions

- Task-derived objects are finalizable.
- When finalizer thread **eventually** runs the finalizer of failed, unobserved task it raises the UnobservedTaskException event.

```
TaskScheduler.UnobservedTaskException += HandleTaskExceptions;

void HandleTaskExceptions(object sender, UnobservedTaskExceptionEventArgs e) {
    foreach (Exception exception in e.Exception.InnerExceptions)
        Handle(exception);

    e.SetObserved();
}
```

When Task Exception remains unobserved

NET 4.0

The finalizer thread

rethrows the exception.

Which **kills the entire process**!

NET 4.5+

The finalizer thread

swallows the exception.

Silent error!

```
<ThrowUnobservedTaskExceptions enabled="true"/>
```

Unhandled exceptions

- .NET 1.0 – 1.1 silently swallowed for background threads

- .NET 2.0+ - terminates the process

- System.AppDomain.UnhandledException (except Windows Store and .NET Core)
- Windows.UI.Xaml.Application.UnhandledException (Windows Store)
- System.Windows.Application.DispatcherUnhandledException (WPF)
- System.ServiceModel.Dispatcher.ChannelDispatcher.ErrorHandlers (WCF)
- (…)

Performance

So which parts of the exception handling mechanism are taking time?

Throwing? Catching? Executing Finally blocks?

Does it cost anything to have a throw block that is not executed?

Executing finally block when no exception is thrown

```
[MethodImpl(MethodImplOptions.NoInlining)]
void EmptyMethod() { }


[Benchmark]
void NoFinally() => EmptyMethod();
```

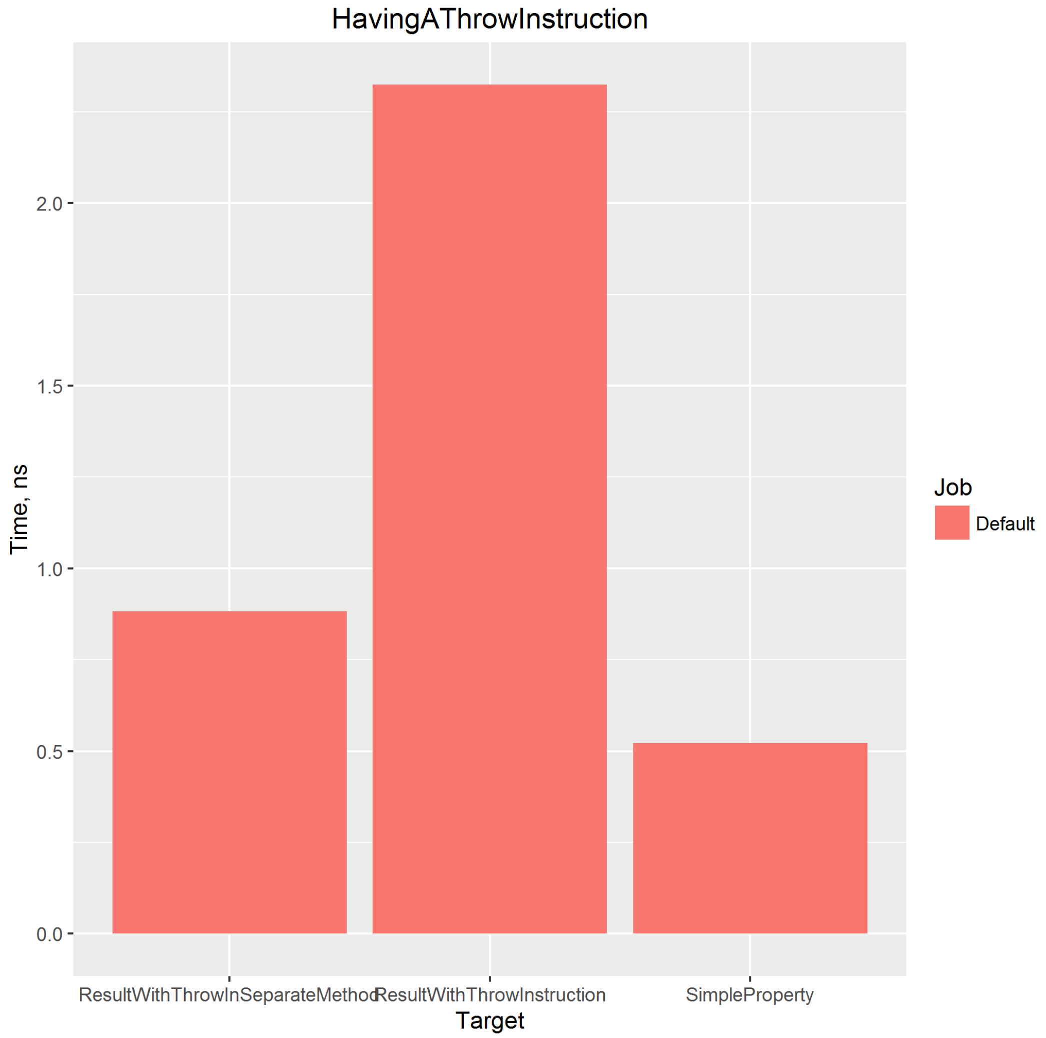
```
[Benchmark]
public void Finally()
{
    try { }
    finally
    {
        EmptyMethod();
    }
}
```

Method	Mean
Finally	3.3245 ns
NoFinally	0.8568 ns

Cost of having a throw instruction inside a method

```
readonly T value;
readonly Exception exception;


T ResultWithThrow()
{
    if (exception != null)
        throw exception;

    return value;
}
```

Effect of inlining



BenchmarkDotNet v0.10.0

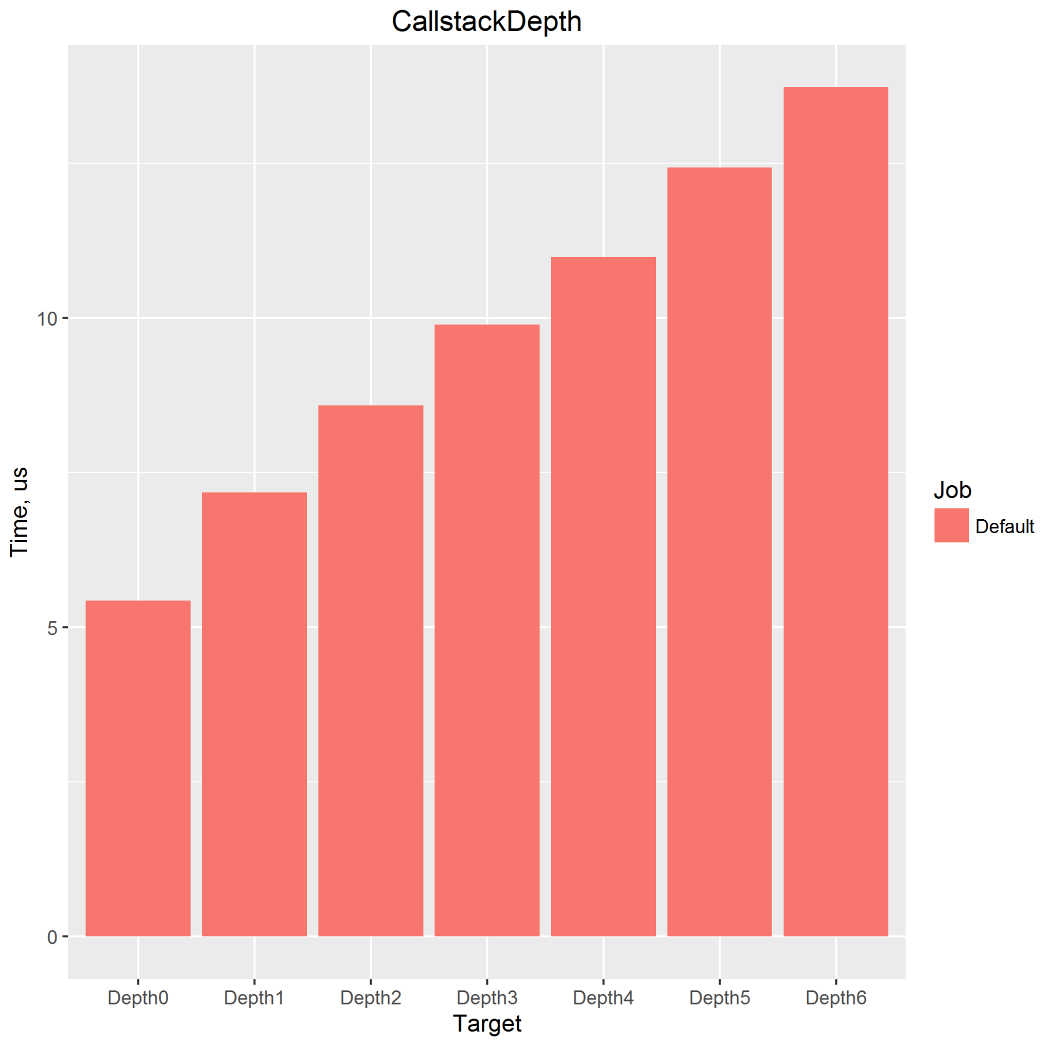
How to make inlining possible

```
T ResultWithThrowInSeparateMethod()
{
    if (exception != null)
        Throw(); // move throw to other method

    return value;
}

void Throw() { throw exception; }
```

Throw & Catch VS Return Failure VS TryOut

Method	**Mean**
ThrowAndCatch	5,533.0462 ns
ReturnFailure	1.792 ns
TryOut	1.779 ns

Depth = distance on Callstack from throw to catch



BenchmarkDotNet v0.10.0

What can we do about the cost of Exception Handling?

Summary

- Finally can overwrite current exception
- You will fail, prepare backup plan for that
- Exceptions gets wrapped (Reflection, static ctors)
- Native exceptions = Corrupted State Exceptions are not catchable by default
- Async/await does not handle AggregatedExceptions well
- Don’t fire and forget the tasks
- Exception handling is very expensive, don’t use it for Flow Control

Sources

Books:

1. CLR via C#
2. .NET IL Assembler
3. Pro Asynchronous Programming with .NET

Websites:

- .NET Core: Breaking Change Rules
- Keep Your Code Running with the Reliability Features of the .NET Framework by Stephen Toub

Questions?

@SitnikAdam

Adam.Sitnik@gmail.com

https://github.com/adamsitnik/ExceptionalExceptions