Forget about Threads:

Tasks, Asynchronous Methods & Coroutines

Arne Claassen
@sdether

Sequential workflow

- Do *x*, then do *y*, then do *z*
- It's the way we think

Sequential workflow

```
public XDoc GetUnreadAggregatedArticles(int userId) {

    var subscription = GetSubscription(userId);

    var feed = GetFeedFromWeb(subscription.Uri);

    subscription.LastSeenId = FilterFeed(feed);

    SaveSubscription(subscription);

    return feed;
}
```


Sequential, blocking workflow

```
//blocking
public XDoc GetUnreadAggregatedArticles(int userId) {

    // blocking
    var subscription = GetSubscription(userId);

    // blocking
    var feed = GetFeedFromWeb(subscription.Uri);


    subscription.LastSeenId = FilterFeed(feed);

    // blocking
    SaveSubscription(subscription);

    return feed;
}
```


Sequential, non-blocking workflow

- Wait for *x*, then wait for *y*, then wait fo *z*
  - It's *still* the way we think
- Synchronous sequences of work
- Chained together by continuations
- Non-blocking
- Abstracts threading

Continuations: Task<T> & Result<T>

- Functionally equivalent
- Provide a waithandle for a future value
- Can be chained
- Provide exception marshalling
- .NET 3.5 vs .NET 2.0
- Async/Await coroutine coming in C# 5
  vs. Iterator based coroutine in C# 2

Asynchronous Method Signatures

- DReAM Result<T>

```
Result<T> AsyncMethod<T>( ...  , Result<T> result) {
    ...
}
```

- TPL Task<T>

```
Task<T> AsyncMethod<T>( ... ) {
    ...
}
```


Sequential, blocking workflow

```
//blocking
public XDoc GetUnreadAggregatedArticles(int userId) {

    // blocking
    var subscription = GetSubscription(userId);

    // blocking
    var feed = GetFeedFromWeb(subscription.Uri);


    subscription.LastSeenId = FilterFeed(feed);

    // blocking
    SaveSubscription(subscription);

    return feed;
}
```


Sequential, non-blocking workflow – DReAM Result

```
public Result<XDoc> GetUnreadAggregatedArticles(int userId, Result<XDoc> response) {


    GetSubscription(userId, new Result<Subscription>()).WhenDone(rSub => {

        var subscription = rSub.Value;
        GetFeedFromWeb(subscription.Uri, new Result<XDoc>()).WhenDone(rFeed => {

            var feed = rFeed.Value;
            subscription.LastSeenId = FilterFeed(feed);


            SaveSubscription(subscription, new Result()).WhenDone(r => {
                response.Return(feed);
            });
        });
    });
    return response;
}
```


Sequential, non-blocking workflow – TPL Task

```
public Task<XDoc> GetUnreadAggregatedArticles(int userId) {
    var response = new TaskCompletionSource<XDoc>();

    GetSubscription(userId).ContinueWith(tSub => {

        var subscription = tSub.Result;
        GetFeedFromWeb(subscription.Uri).ContinueWith(tFeed => {

            var feed = tFeed.Result;
            subscription.LastSeenId = FilterFeed(feed);


            SaveSubscription(subscription).ContinueWith(t => {
                response.SetResult(feed);
            });
        });
    });
    return response.Task;
}
```


Exit, screen right

```
FirstDoThis().ContinueWith(t1 => {
    ThenDoThat().ContinueWith(t2 => {
        ThenDoThis().ContinueWith(t3 => {
            ThenDoThat().ContinueWith(t4 => {
                ThenDoThis().ContinueWith(t5 => {
                    ThenDoThat().ContinueWith(t6 => {
                        ThenDoThis().ContinueWith(t7 => {
                            ThenDoThat().ContinueWith(t8 => {
```

- Ugly to read
- Hard to reason about

Coroutines

A subroutine that allows multiple entry points for suspending and resuming

- Perfect for asynchronous calls
- Suspend on async call
- Resume on callback
- Preserves sequential flow
- Can use loops, try/catch/finally, using

Coroutine Signatures

- DReAM

```
IEnumerator<IYield> Coroutine<T>( ... , Result<T> result) {
    ...
    yield return AsyncMethod<V>( ... , new Result<V>());
    ...
    result.Return(t);
}
```

- TPL Async CTP

```
async Task<T> Coroutine<T>( ... ) {
    ...
    var x = await AsyncMethod<V>( ... );
    ...
    return t;
}
```


Coroutines – DReAM Result

```
using Yield = IEnumerator<IYield>;
public Yield GetUnreadAggregatedArticles(int userId, Result<XDoc> response) {

    Subscription subscription = null;
    yield return GetSubscription(userId, new Result<Subscription>())
        .Set(x => subscription = x);
    XDoc feed = null;
    yield return GetFeedFromWeb(subscription.Uri, new Result<XDoc>())
        .Set(x => feed = x);

    subscription.LastSeenId = FilterFeed(feed);


    yield return SaveSubscription(subscription, new Result());
    response.Return(feed);
}
```


Coroutines – TPL Task

```
public async Task<XDoc> GetUnreadAggregatedArticles(int userId) {

    var subscription = await GetSubscription(userId);

    var feed = await GetFeedFromWeb(subscription.Uri);

    subscription.LastSeenId = FilterFeed(feed);

    await SaveSubscription(subscription);
    return feed;
}
```


Sequential, blocking workflow

```
public XDoc GetUnreadAggregatedArticles(int userId) {

    var subscription = GetSubscription(userId);

    var feed = GetFeedFromWeb(subscription.Uri);

    subscription.LastSeenId = FilterFeed(feed);

    SaveSubscription(subscription);

    return feed;
}
```


Sharing state asynchronously

- Prevent multi-thread access
- Async method call as queue
- Operate on resource in its own context
- Allows complex changes in atomic fashion

Sharing state asynchronously

```
public interface IAsyncAdapter<T> {

    Task<TValue> Call<TValue>(Func<T, TValue> visitor);

    Task Call(Action<T> visitor);
}
```


Turtles all the way down

- Only async methods can call other async methods without blocking
- Initial entry point must be asynchronous
- Hard to interoperate with standard libraries

What's the cost?

- Coroutines have a lot of overhead
  - 20x - 40x slower than method
- But even the fastest I/O is slower
  - I/O is 200x+ slower
- The goal is throughput not single task speed

When to use Asynchrony

- UI
- Network servers
- File system
- Remote Services
  - Remoting
  - Web Services
- Databases

But wait…
There's *more!*

Cooperative Coroutines

```
Coroutine(Coordinator c) {
        ... do some work ...
        yield to coordinator
        ... do some more work ...
}
```

- Must have the same “shape”
- Coroutines don’t have know about each other
- Can alter workflow dynamically
- Perfect for processing pipeline

Never block again

Arne Claassen
@sdether