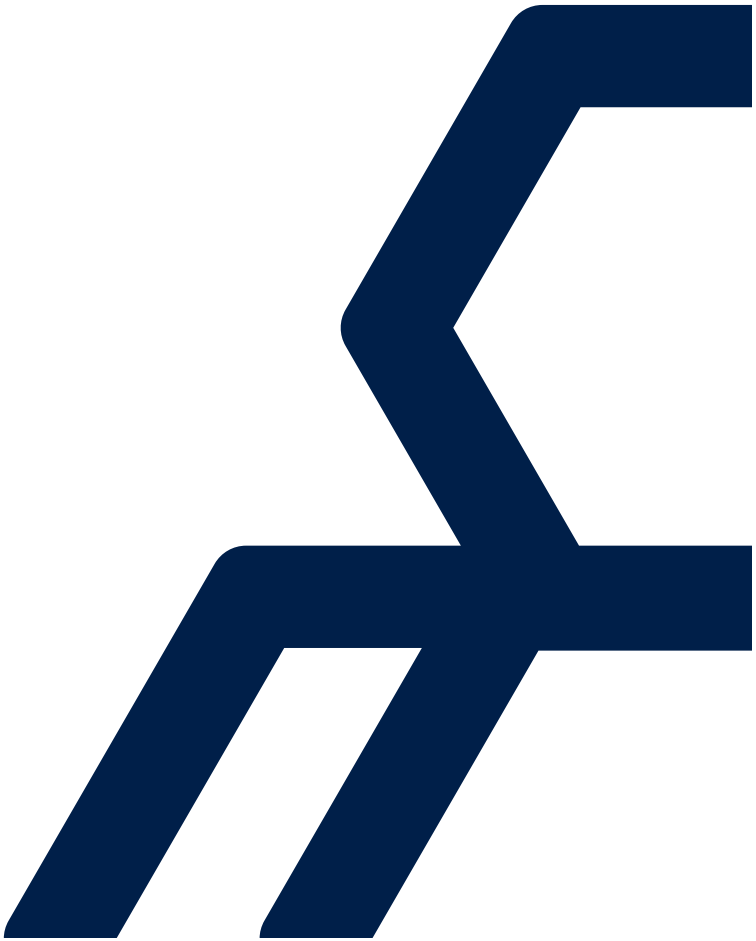
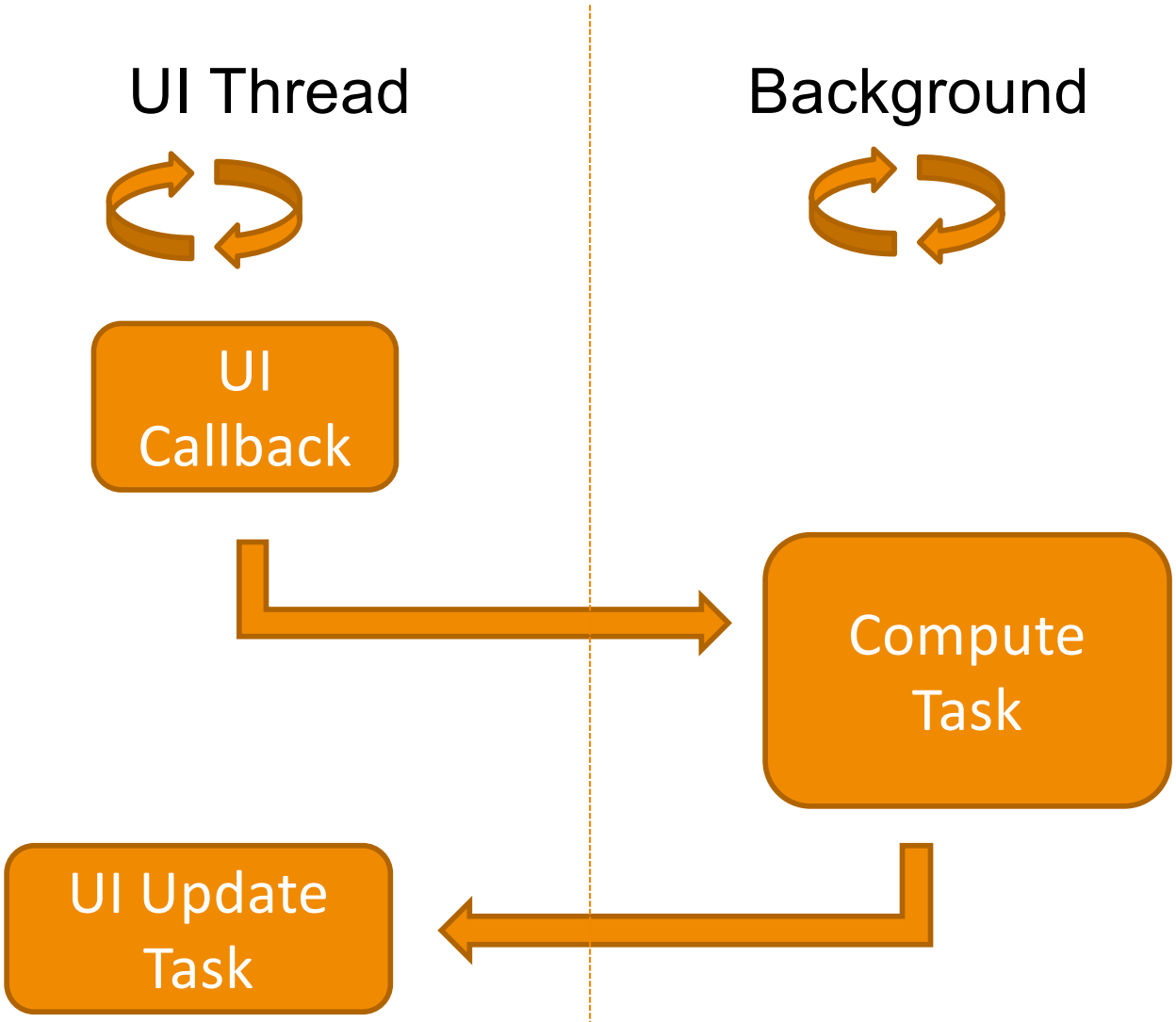
async and await



Objectives

- Why use Continuations
- Simple examples of async/await
- Under the hood
- Gotcha's
- Composition
- Server side async/await
- Asynchronous streams

Continuations



Control flow

- Sequential programming intent is pretty clear.
- Asynchronous programming screws with intent
- Do X Async, Then Do Y , Then Do Z Async
- How to handle errors
  - Where to place the try/catch

async/await keywords

Making async intent as clear as synchronous intent

- Two new keywords introduced in C# 5
- Enables continuations whilst maintaining the readability of sequential code
  - Automatic marshalling back on to the UI thread
- Built around Task, and Task<T>, ValueTask<T>

async and await

Example

- async method must return void or a Task/ValueTask
- async method should include an await
- await <TASK>

```
async void Button_Click(object sender, RoutedEventArgs e) {
  calcButton.IsEnabled = false;
  Task<double> piResult = CalcPiAsync(1000000000);

   // If piResult not ready returns, allowing UI to continue
   // When has completed, returns back to this thread
   // and coerces piResult.Result out
                await piResult;

   calcButton.IsEnabled = true;
   this.pi.Text = pi.ToString();
}
```

async await

Can return Task<T>

- Code returns T, compiler returns Task<T>

```
async Task<byte[]> DownloadDataAsync(Uri source)
{
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

   ProcessData(data);

   return data
}
```


Favour continuations over waiting

- Threads aren't free
- A thread waiting can't be used for anything else.
- Using continuations can reduce the total number of required threads

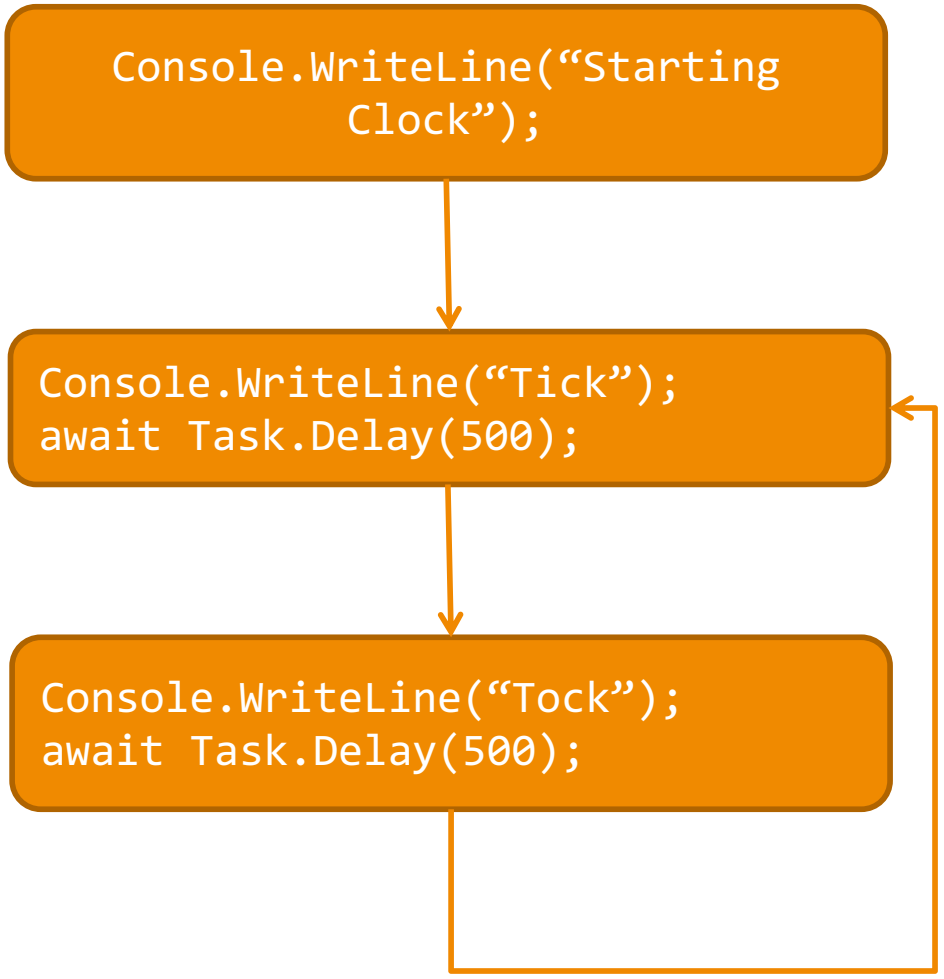
async/await under the hood

Compiler builds state machine

```
private static async void TickTockAsync()
{
 Console.WriteLine("Starting Clock");

while (true)
 {
  Console.WriteLine("Tick");
  await Task.Delay(500);

  Console.WriteLine("Tock");
  await Task.Delay(500);
 }
}
```



async/await

Gotcha

async keyword does not make code run asynchronously

```
async Task DoItAsync()
 {
     // Still on calling thread
       Thread.Sleep(5000);
       Console.WriteLine("done it..");
 }
```

async/await

Gotcha #2

Avoid async methods returning void

```
async void DownloadDataAsync(string uri){
. . .
}
```

- Better to return Task than void
- Allows caller to handle error
- void is there for asynchronous event handlers

```
async Task DownloadData(Uri source)
{
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

   ProcessData(data);
}
```

THINK before using async lambda for Action delegate

```
requests.ForEach(async request =>
{
    var client = new WebClient();
    Console.WriteLine("Downloading {0}", request.Uri);
    request.Content = await
         client.DownloadDataTaskAsync(request.Uri);
});
Console.WriteLine("All done..??");

requests.ForEach(r => Console.WriteLine(r.Content.Length);
```

async/await

Gotcha #4

- await exception handling only delivers first exception
- Tasks can throw many exceptions via an AggregateException
  - Await re-throws only first exception from Aggregate
- Examine Task.Exception property for all errors

```
Task<byte[]> loadDataTask = null;
try {
  loadDataTask = LoadAsync();
  byte[] data = await loadDataTask;
  ProcessData(data
} catch(Exception firstError) ){
     loadDataTask.Exception.Flatten().Handle( MyErrorHandler );
}
```

ConfigureAwait

Possibly the worst API ever conceived

- Not all await's need to make use of SynchronizationContext

```
public static async Task DownloadData(Uri source, string destination)
{
  WebClient client =new WebClient();
  byte[] data = await client.DownloadDataTaskAsync(source);

// DON'T NEED TO BE ON UI THREAD HERE…
  ProcessData(data);

  using (Stream downloadStream = File.OpenWrite(destination))
  {
    await downloadStream.WriteAsync(data, 0, data.Length);
  }
  // Must be back on UI thread
  UpdateUI("Download
```

ConfigureAwait

Possibly the worst API ever conceived

- First attempt, but wrong

```
 static async Task DownloadData(Uri source, string destination)
{
  WebClient client =new WebClient();
                 await client
                        .DownloadDataTaskAsync(source)
                        .ConfigureAwait(false);

   // Will continue  not on UI thread
         Stream downloadStream = File.OpenWrite(destination)) {
    await downloadStream.WriteAsync(data, 0, data.Length);
  }

  // Hmmm…Need to be back on UI thread here

  UpdateUI("All downloaded");
```

ConfigureAwait

Effective use of ConfigureAwait with composition

- Get compiler to create Task per context

```
static async Task DownloadData(Uri source, string destination){
   await DownloadAsync(source, destination);
   // on UI thread
   UpdateUI("All downloaded");
}

 private static async Task DownloadAsync(Uri source, string destination) {
    WebClient client = new WebClient();
    byte[] data = await client
                          .DownloadDataTaskAsync(source)
                          .ConfigureAwait(continueOnCapturedContext:false);

    using (Stream downloadStream = File.OpenWrite(destination)) {
       await downloadStream.WriteAsync(data, 0, data.Length);
    }
 }
```

Await and Monitors

- Compile will not allow an await inside a lock block
  - Lock only works if the entire block is executed on the same thread

Won't compile

```
lock (account)
{
    await UpdateAccount().ConfigureAwait(false);
}
```

Will compile

```
Monitor.Enter(account)
try
{
    await UpdateAccount().ConfigureAwait(false);
}
finally{ Monitor.Exit(account); }
```

Utilise Semaphore slim

Await compatible locking

- Semaphore with count of 1 has similar behavior to that of Mutex
- Semapore can be aquired and released around a await
- Can await on a semaphore, for non blocking synchronization
- Wrap in using pattern to maintain programming model

Awaiting with timeout

- Waiting for ever, is bad
- Awaiting for ever possibly less bad
- What if awaiting task has no cancellation?
  - await keyword has no time out
- Consider using Task.WhenAny
- .NET 6 introduces Task.WaitAsync()

Async on the server

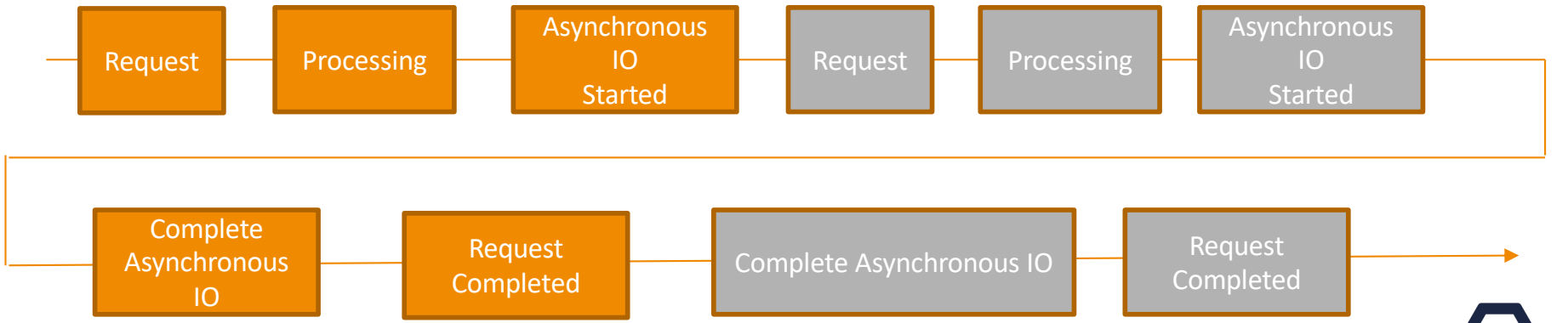
Not just client side technology

- MVC and WebAPI both understand
  - Task<T>
  - ValueTask<T>
- Blocking on a thread is harmful to your application
  - New requests may force a new thread to be created
  - New threads take time to start and consume resource
- Server threads shouldn't block
  - Release thread until they need it again
  - Allow N concurrent requests to share a single thread of execution

Async on the server

Thread re-use

- Thread starts processing a given request
- Initiates some asynchronous IO, and frees itself to perform another request
- Second request initiates asynchronous IO, and frees it self it complete the previous request



AsyncLocal<T>

- Applications sometimes need to flow ambient state
  - HttpContext
  - LoggingContext
- When using async/await can't use thread local storage
- AsyncLocal<T> used to flow ambient state across await boundaries

Async Enumerable aka Async Streams

- Why the need for asynchronous iteration
- Foreach async
- Disposable Async

DIY async iteration

- **Asynchronous iteration awkward for creation and consumption**

```
public static  IEnumerable<Task<string[]>> LoadCsv(string filename){
  using (var reader = new StreamReader(filename)) {
   while (!reader.EndOfStream){
    yield return LoadAndSplit(reader);
   }
  }
}

private static async Task<string[]> LoadAndSplit(StreamReader reader) {
  return (await reader.ReadLineAsync()).Split(',');
}
```

IAsyncEnumerable<T>

- Async version of IEnumerable,IEnumerator

```
public interface IAsyncEnumerable<out T> {
  IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken ct);
}

public interface IAsyncEnumerator<out T> :IAsyncDisposable {

  T Current {get; }
  ValueTask<bool> MoveNextAsync();
}
```

Yield return async enumerable

- Compiler creates an async enumerable
- Methods must by marked async
- Methods must return IAsyncEnumerable<T> or IAsyncEnumerator<T>
- Yield return used as per iterator methods

```
static async IAsyncEnumerable<string[]> LoadCsv(string filename) {
   using (var reader = new StreamReader(filename))
   {
     while (!reader.EndOfStream) {
      string row = await reader.ReadLineAsync();
      yield return row.Split(',');
    }
   }
}
```

Foreach async

- Iterates through async enumerable
- Delivers each item, not Task<T>

```
IAsyncEnumerable<string[]> rows = LoadCsv(@"stockData.csv");

await foreach (string[] row in rows)
{
    Console.WriteLine(row[1]);
}
```


Async LINQ

- Nuget package System.Linq.Async
- IAsyncEnumerable<T> extension methods defined in AsyncEnumerable

```
var tradingDays = LoadCsv(@"stockData.csv")
                  .Skip(1)
                  .Select(row => new
                  {
                      When = DateTime.Parse(row[0]),
                      Open = decimal.Parse(row[1]),
                      Close = decimal.Parse(row[4])
                  });

await foreach (var tradingDay in tradingDays) {
  Console.WriteLine(tradingDay);
}
```

Foreach async cancellation

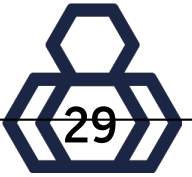
- Iterator method can take cancellation token
- Problem: this is scoped for the IAsyncEnumerable not the IAsyncEnumerator

```
 var cts = new CancellationTokenSource();
 var rows = LoadCsv("stockData.csv" , cts.Token);

await foreach (string[] row in rows) {
  Console.WriteLine(row[1]);
  cts.Cancel();
}

await foreach (string[] row in rows) {
  Console.WriteLine(row[1]); // NEVER EXECUTES
}

public static async IAsyncEnumerable<string[]> LoadCsv(
                string filename , CancellationToken ct)
```



Enumeration cancellation

- Cancellation can be scoped to the enumerator
- Iterator method parameter marked to accept cancellation token

```
 var cts = new CancellationTokenSource();
 var rows = LoadCsv("stockData.csv" , CancellationToken.None);

await foreach (string[] row in rows.WithCancellation(ct.Token) {
  Console.WriteLine(row[1]);
  cts.Cancel();
}

public static async IAsyncEnumerable<string[]> LoadCsv(
                string filename ,
                [EnumeratorCancellation]CancellationToken ct)
```

Async Disposable

- New interface IAsyncDisposable
- Using statement prefix with await uses DisposeAsync

```
public interface IAsyncDisposable  {
  ValueTask DisposeAsync();
}
```

```
await using (FileStream s = File.OpenRead("stockData.csv")){

}
```

Async iteration on the server

- MVC and WebAPI both understand IAsyncEnumerable<T>

```
[Route("TradingDays")]
[HttpGet]
public IAsyncEnumerable<TradingDayDTO> GetStocks()
{
  var rows = Context.TradingDays;
  return rows.Select(r => new TradingDayDTO()
    {
      When = r.When.ToShortDateString(),
      Close = r.Close,
      Open = r.Open,
      Volume = r.Volume
    }).AsAsyncEnumerable();
}
```

Summary

- Utilise async/await to
  - Simplify continuations
  - Reduce number of threads
- Use Semaphore as an await safe locking primitive
- Use ConfigureAwait to reduce work on UI thread
- Use asynchronous apis for greater scalability and performance
- Use IAsyncEnumable for streaming asynchronous results