Async Streams

DotNext Moscow, 2019

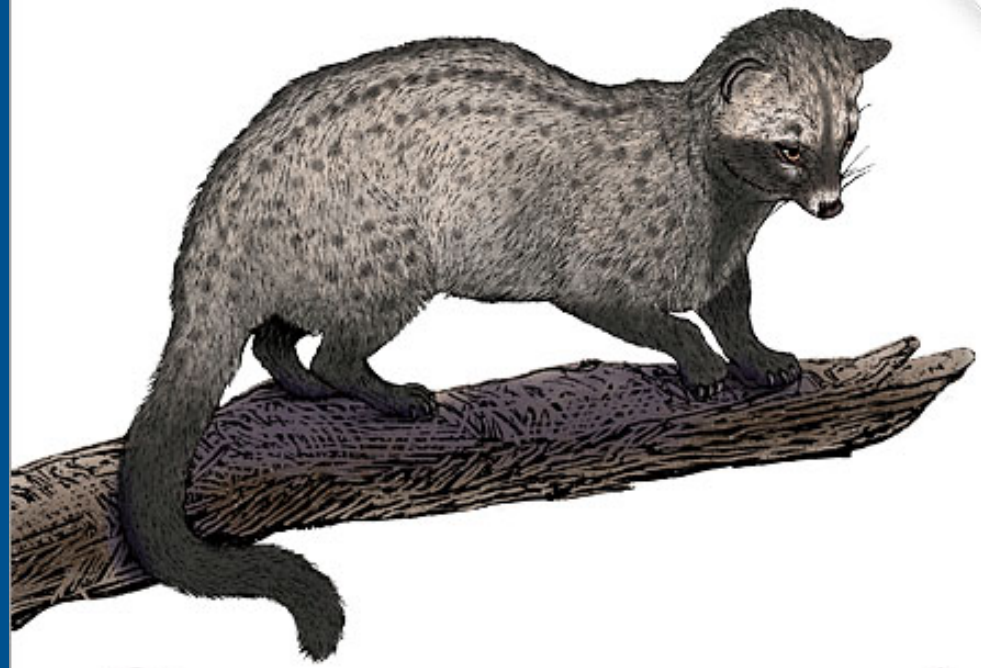
StephenCleary.com

Who is this guy?

StephenCleary.com

O'REILLY®

Second Edition



Concurrency in C# Cookbook

Asynchronous, Parallel, and Multithreaded Programming

Stephen Cleary

The Async Invasion

A Brief History

StephenCleary.com

Async Invasion Timeline

- Async/await (Futures):
  - 2012: C# 5.0
  - 2015: TypeScript 1.6
  - 2015: Python 3.5
  - 2017: JavaScript ES2017
- Async streams (async generators):
  - 2016: Python 3.6
  - 2017: TypeScript 2.3
  - 2018: JavaScript ES2018
  - 2019: C# 8.0 (built-in to .NET Core; also available on .NET Framework via Microsoft.Bcl.AsyncInterfaces)

Why Async Streams?

Why Async?

Real-world benefits from asynchrony

Client	Server
Primary benefit: *Responsiveness*	Primary benefit: *Scalability*
Keep UI thread free	Minimize threads used to serve requests
Better UX	10x-100x scalability (same box)
Required by many app stores	Faster response to bursting traffic

Why Async Streams?

From three perspectives:

- Enumerables
- Tasks
- Observables

Why Async Streams? (vs Enumerable)

What we have: Enumerable

- Enumerables are synchronous.
- Tasks/async/await are also available, and are asynchronous.

What we want:

- We want to do asynchronous work during enumeration.

Why Async Streams? (vs Task)

What we have: Task

- Tasks only produce a result once.
- Enumerables can generate multiple results.

What we want:

- We want to generate multiple results asynchronously.

Why Async Streams? (vs Observable)

What we have: Observable

- Asynchronous and multi-valued.

What we want:

- More natural consumption.
  E.g., `foreach` or `async` subscriptions.
  - Lower training bar.
  - Enumerables and tasks have natural consumption.
  - “Natural consumption” is code for “pull-based”.

Why Async Streams?

Summary of three different perspectives:

Compared to…	which is…	`IAsyncEnumerable<T>` is…
`IEnumerable<T>`	synchronous	asynchronous
`Task<T>`	single value	multiple values
`IObservable<T>`	push-based	pull-based

How Async Streams Fit In

Desired Access	Return Type
Single value, synchronous	`T`
Multiple values, synchronous	`IEnumerable<T>`
Single value, asynchronous (pull)	`Task<T>`
Single value, asynchronous (push)	`IObservable<T>`
Multiple values, asynchronous (pull)	`IAsyncEnumerable<T>`
Multiple values, asynchronous (push)	`IObservable<T>`

Async Streams in One Sentence:

You can use `await` and `yield return` in the same method.

Yield Return (Enumerables) + Async

We end up with *both:*
Deferred execution *and* asynchronous pausing.

Async Streams: The Async Part

StephenCleary.com

Async Streams: More Detail

- Enumerators/Generators/Iterables:
  - "Deferred execution" – generated on demand
  - Pull-based sequence. E.g., `foreach`.

- Async streams:
  - Still deferred execution. Still pull-based.
  - "Get Next Item" is asynchronous.
    - `MoveNext`, `MoveNextAsync` (C#) / `next` (JS) / `__next__`, `__anext__` (Python)
    - Allows asynchronous enumerators/generators/iterables.
    - Requires asynchronous consumers.

Where the Asynchrony Is (C#)

"Get Next Item" (as used by `foreach`) is asynchronous:

```
IEnumerator<out T>
    : IDisposable
{
 T Current { get; }
 bool MoveNext();
}
```

```
IAsyncEnumerator<out T>
    : IAsyncDisposable
{
  T Current { get; }
  ValueTask<bool> MoveNextAsync();
}
```

ValueTask (C#)

`ValueTask<T>` is a more efficient `Task<T>`

- Particularly if the result is commonly synchronous.

Usage restriction: **Only consume once!**

- "Consume" means `await` or `AsTask`

Other properties may behave differently than `Task<T>`

- `Result` is invalid until the value task has completed.
- Same for `GetAwaiter().GetResult()`

foreach

```
foreach (var i in s)
{
  Console.WriteLine(i);
}
```

```
using (var e = s.GetEnumerator())
{
  while (e.MoveNext())
  {
    var i = e.Current;
    Console.WriteLine(i);
  }
}
```

await foreach

```
await foreach (var i in s)
{
  Console.WriteLine(i);
}
```

```
      await using (var e = s.GetAsyncEnumerator())
      {
        while (await e.MoveNextAsync())
        {
          var i = e.Current;
          Console.WriteLine(i);
        }
      }
```

Details: ConfigureAwait(false)

```
await foreach (var i in s.ConfigureAwait(false)) {
  Console.WriteLine(i);
}

  var e = s.GetAsyncEnumerator();
  await using (e.ConfigureAwait(false)) {
    while (await e.MoveNextAsync().ConfigureAwait(false)) {
      var i = e.Current;
      Console.WriteLine(i);
    }
  }
```

Creating Async Streams

Creating Async Streams

C#:

- Return type is `IAsyncEnumerable<T>`
- Use `await` and `yield return` in an `async` method.

Javascript:

- Use `await` and `yield` in an `async function*` function.

Python:

- Use `await` and `yield` in an `async` method.

Consuming Async Streams

StephenCleary.com

Basic Usage

C#:

- Use `await foreach` in an `async` method.

JavaScript:

- Use `foreach await` in an `async` function.

Python:

- Use `async for` in an `async` method.

A word on exceptions

Async Streams Use Case

Use Case: Paging API

Demo

Anti-Use Case: Notification API

E.g.:

- SignalR
- Semi-HTTP multi-response streams (stock quotes)

Anything with a *subscribe + multiple updates + unsubscribe* system is a better fit for observables.

LINQ to Async Streams

StephenCleary.com

Basic LINQ

NuGet: `System.Linq.Async`

- Community project, not Microsoft-supported.

All the standard operators:

- `Where`, `Select`, `SelectMany`, `Join`, etc.

Async LINQ

Demo

Passing Async Lambdas to LINQ

LINQ-to-Streams has overloads for `async` lambdas:

- `WhereAwait`, etc.
- `Await` suffix, since they `await` their delegates.

Return `IAsyncEnumerable<T>` so they chain naturally.

Async Results from LINQ

"Terminal" operators end in `Async` since they return awaitables

- `CountAsync`, etc.

Terminal operators also have overloads for `async` lambdas:

- `CountAwaitAsync`

Async Lambdas: Semantics

All `System.Linq.Async` operators act on data values one at a time.

- No async concurrency or parallelism.

Async LINQ Delegates

Demo

Supercharging Regular LINQ

When you have an ordinary LINQ expression,

and you want to use an `async` lambda (e.g., for `Where`)

Your solution: `ToAsyncEnumerable`

- Then you can use `WhereAwait`, etc.

(This does “lift” to an `IAsyncEnumerable<T>`).

LINQ -> Async Linq

Demo

Cancelling Async Streams

StephenCleary.com

Responding to Cancellation

Take a `CancellationToken`.

Apply the `EnumeratorCancellation` attribute.

- This is new.

Requesting Cancellation (easy)

The simple way:

- Pass `CancellationToken` to the method returning an enumerable.

The complex way:

- Use `WithCancellation` when enumerating.
- Because *enumerators* are cancellable, not *enumerables.*

Async Streams & Cancellation

Demo

Q&A

Go forth and be awesome!

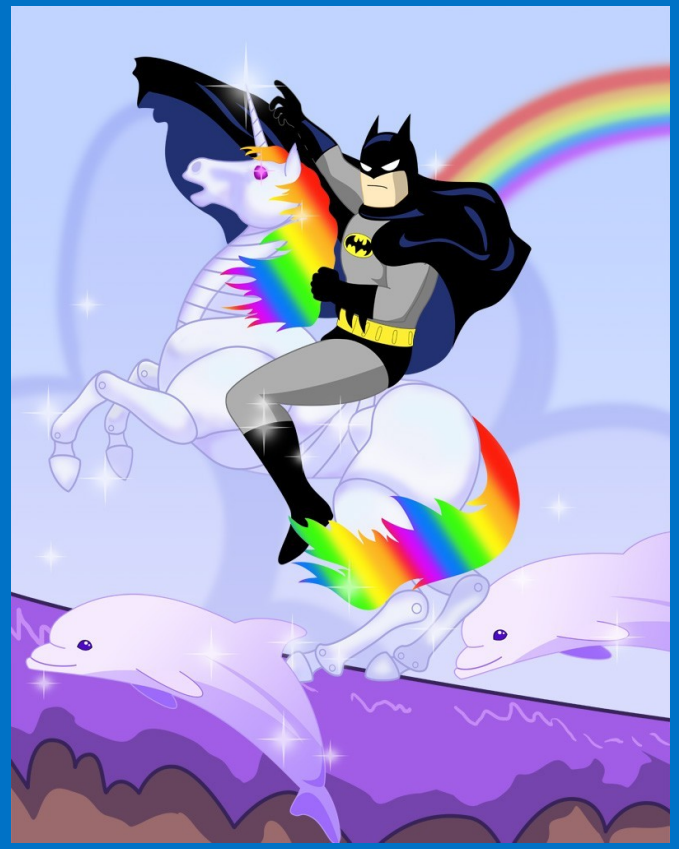


Image from Etsy user Rosewine; used with permission

StephenCleary.com