

[Swift](#) / [AsyncStream](#)

Structure

AsyncStream

An asynchronous sequence generated from a closure that calls a continuation to produce new elements.

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | macOS 10.15+ | tvOS 13.0+ | visionOS 1.0+ | watchOS 6.0+

```
struct AsyncStream<Element>
```

Overview

AsyncStream conforms to AsyncSequence, providing a convenient way to create an asynchronous sequence without manually implementing an asynchronous iterator. In particular, an asynchronous stream is well-suited to adapt callback- or delegation-based APIs to participate with `async-await`.

You initialize an AsyncStream with a closure that receives an `AsyncStream.Continuation`. Produce elements in this closure, then provide them to the stream by calling the continuation's `yield(_:_)` method. When there are no further elements to produce, call the continuation's `finish()` method. This causes the sequence iterator to produce a `nil`, which terminates the sequence. The continuation conforms to `Sendable`, which permits calling it from concurrent contexts external to the iteration of the `AsyncStream`.

An arbitrary source of elements can produce elements faster than they are consumed by a caller iterating over them. Because of this, `AsyncStream` defines a buffering behavior, allowing the stream to buffer a specific number of oldest or newest elements. By default, the buffer limit is `Int.max`, which means the value is unbounded.

Adapting Existing Code to Use Streams

To adapt existing callback code to use `async-await`, use the callbacks to provide values to the stream, by using the continuation's `yield(_ :)` method.

Consider a hypothetical `QuakeMonitor` type that provides callers with `Quake` instances every time it detects an earthquake. To receive callbacks, callers set a custom closure as the value of the monitor's `quakeHandler` property, which the monitor calls back as necessary.

```
class QuakeMonitor {  
    var quakeHandler: ((Quake) -> Void)?  
  
    func startMonitoring() {...}  
    func stopMonitoring() {...}  
}
```

To adapt this to use `async-await`, extend the `QuakeMonitor` to add a `quakes` property, of type `AsyncStream<Quake>`. In the getter for this property, return an `AsyncStream`, whose build closure – called at runtime to create the stream – uses the continuation to perform the following steps:

1. Creates a `QuakeMonitor` instance.
2. Sets the monitor's `quakeHandler` property to a closure that receives each `Quake` instance and forwards it to the stream by calling the continuation's `yield(_ :)` method.
3. Sets the continuation's `onTermination` property to a closure that calls `stopMonitoring()` on the monitor.
4. Calls `startMonitoring` on the `QuakeMonitor`.

```
extension QuakeMonitor {  
  
    static var quakes: AsyncStream<Quake> {  
        AsyncStream { continuation in  
            let monitor = QuakeMonitor()  
            monitor.quakeHandler = { quake in  
                continuation.yield(quake)  
            }  
            continuation.onTermination = { @Sendable _ in  
                monitor.stopMonitoring()  
            }  
            monitor.startMonitoring()  
        }  
    }  
}
```

Because the stream is an `AsyncSequence`, the call point can use the `for-await-in` syntax to process each `Quake` instance as the stream produces it:

```
for await quake in QuakeMonitor.quakes {  
    print("Quake: \(quake.date)")  
}  
print("Stream finished.")
```

Topics

Creating a Continuation-Based Stream

```
init(Element.Type, bufferingPolicy: AsyncStream<Element>.Continuation.  
BufferingPolicy, (AsyncStream<Element>.Continuation) -> Void)
```

Constructs an asynchronous stream for an element type, using the specified buffering policy and element-producing closure.

```
enum BufferingPolicy
```

A strategy that handles exhaustion of a buffer's capacity.

```
struct Continuation
```

A mechanism to interface between synchronous code and an asynchronous stream.

Creating a Stream from an Asynchronous Function

```
init(unfolding: () async -> Element?, onCancel: (() -> Void)?)
```

Constructs an asynchronous stream from a given element-producing closure, with an optional closure to handle cancellation.

Finding Elements

```
func contains(Self.Element) async rethrows -> Bool
```

Returns a Boolean value that indicates whether the asynchronous sequence contains the given element.

```
func contains(where: (Self.Element) async throws -> Bool) async  
rethrows -> Bool
```

Returns a Boolean value that indicates whether the asynchronous sequence contains an element that satisfies the given predicate.

```
func allSatisfy((Self.Element) async throws -> Bool) async rethrows -> Bool
```

Returns a Boolean value that indicates whether all elements produced by the asynchronous sequence satisfy the given predicate.

```
func first(where: (Self.Element) async throws -> Bool) async rethrows -> Self.Element?
```

Returns the first element of the sequence that satisfies the given predicate.

```
func min() async rethrows -> Self.Element?
```

Returns the minimum element in an asynchronous sequence of comparable elements.

```
func min(by: (Self.Element, Self.Element) async throws -> Bool) async rethrows -> Self.Element?
```

Returns the minimum element in the asynchronous sequence, using the given predicate as the comparison between elements.

```
func max() async rethrows -> Self.Element?
```

Returns the maximum element in an asynchronous sequence of comparable elements.

```
func max(by: (Self.Element, Self.Element) async throws -> Bool) async rethrows -> Self.Element?
```

Returns the maximum element in the asynchronous sequence, using the given predicate as the comparison between elements.

Selecting Elements

```
func prefix(Int) -> AsyncPrefixSequence<Self>
```

Returns an asynchronous sequence, up to the specified maximum length, containing the initial elements of the base asynchronous sequence.

```
func prefix(while: (Self.Element) async -> Bool) rethrows -> AsyncPrefixWhileSequence<Self>
```

Returns an asynchronous sequence, containing the initial, consecutive elements of the base sequence that satisfy the given predicate.

Excluding Elements

```
func dropFirst(Int) -> AsyncDropFirstSequence<Self>
```

Omits a specified number of elements from the base asynchronous sequence, then passes through all remaining elements.

```
func drop(while: (Self.Element) async -> Bool) -> AsyncDropWhileSequence<Self>
```

Omits elements from the base asynchronous sequence until a given closure returns false, after which it passes through all remaining elements.

```
func filter((Self.Element) async -> Bool) -> AsyncFilterSequence<Self>
```

Creates an asynchronous sequence that contains, in order, the elements of the base sequence that satisfy the given predicate.

Transforming a Sequence

```
func map<Transformed>((Self.Element) async throws -> Transformed) -> AsyncThrowingMapSequence<Self, Transformed>
```

Creates an asynchronous sequence that maps the given error-throwing closure over the asynchronous sequence's elements.

```
func map<Transformed>((Self.Element) async -> Transformed) -> AsyncMapSequence<Self, Transformed>
```

Creates an asynchronous sequence that maps the given closure over the asynchronous sequence's elements.

```
func compactMap<ElementOfResult>((Self.Element) async -> ElementOfResult?) -> AsyncCompactMapSequence<Self, ElementOfResult>
```

Creates an asynchronous sequence that maps the given closure over the asynchronous sequence's elements, omitting results that don't return a value.

```
func compactMap<ElementOfResult>((Self.Element) async throws -> ElementOfResult?) -> AsyncThrowingCompactMapSequence<Self, ElementOfResult>
```

Creates an asynchronous sequence that maps an error-throwing closure over the base sequence's elements, omitting results that don't return a value.

```
func flatMap<SegmentOfResult>((Self.Element) async throws -> SegmentOfResult) -> AsyncThrowingFlatMapSequence<Self, SegmentOfResult>
```

Creates an asynchronous sequence that concatenates the results of calling the given error-throwing transformation with each element of this sequence.

```
func reduce<Result>(Result, (Result, Self.Element) async throws -> Result) async rethrows -> Result
```

Returns the result of combining the elements of the asynchronous sequence using the given closure.

```
func reduce<Result>(into: Result, (inout Result, Self.Element) async throws -> Void) async rethrows -> Result
```

Returns the result of combining the elements of the asynchronous sequence using the given closure, given a mutable initial value.

Creating an Iterator

```
func makeAsyncIterator() -> AsyncStream<Element>.Iterator
```

Creates the asynchronous iterator that produces elements of this asynchronous sequence.

```
struct Iterator
```

The asynchronous iterator for iterating an asynchronous stream.

Supporting Types

```
typealias AsyncIterator
```

The type of asynchronous iterator that produces elements of this asynchronous sequence.

Type Methods

```
static func makeStream(of: Element.Type, bufferingPolicy: AsyncStream<Element>.Continuation.BufferingPolicy) -> (stream: AsyncStream<Element>, continuation: AsyncStream<Element>.Continuation)
```

Initializes a new [AsyncStream](#) and an [AsyncStream.Continuation](#).

Default Implementations

≡ AsyncSequence Implementations

Relationships

Conforms To

AsyncSequence

Conforms when `Element` conforms to `Copyable` and `Escapable`.

Copyable

Conforms when `Element` conforms to `Copyable` and `Escapable`.

Sendable

Conforms when Element conforms to Copyable, Escapable, and Sendable.

SendableMetatype

Conforms when Element conforms to Copyable, Escapable, and Sendable.

See Also

Asynchronous Sequences

`protocol AsyncSequence`

A type that provides asynchronous, sequential, iterated access to its elements.

`struct AsyncThrowingStream`

An asynchronous sequence generated from an error-throwing closure that calls a continuation to produce new elements.