```
import Darwin
import _Concurrency
import _StringProcessing
import _SwiftConcurrencyShims

/// A type-erasing cancellable object
that executes a provided closure when
canceled.
///
/// Subscriber implementations can use
this type to provide a “cancellation
token” that makes it possible for a
caller to cancel a publisher, but not to
use the ``Subscription`` object to
request items.
///
/// An ``AnyCancellable`` instance
automatically calls
``Cancellable/cancel()`` when
deinitialized.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
final public class AnyCancellable :
Cancellable, Hashable {

    /// Initializes the cancellable
object with the given cancel-time
closure.
    ///
    /// - Parameter cancel: A closure
that the `cancel()` method executes.
    public init(_ cancel: @escaping () ->
Void)
```

```
    public init<C>(_ canceller: C) where
C : Cancellable

    /// Cancel the activity.
    ///
    /// When implementing ``Cancellable``
in support of a custom publisher,
implement `cancel()` to request that your
publisher stop calling its downstream
subscribers. Combine doesn't require that
the publisher stop immediately, but the
`cancel()` call should take effect
quickly. Canceling should also eliminate
any strong references it currently holds.
    ///
    /// After you receive one call to
`cancel()`, subsequent calls shouldn't do
anything. Additionally, your
implementation must be thread-safe, and
it shouldn't block the caller.
    ///
    /// > Tip: Keep in mind that your
`cancel()` may execute concurrently with
another call to `cancel()` --- including
the scenario where an ``AnyCancellable``
is deallocating --- or to
``Subscription/request(_:)``.
    final public func cancel()

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
```

```
    ///
    /// Implement this method to conform
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    final public func hash(into hasher:
inout Hasher)

    /// Returns a Boolean value that
indicates whether two instances are
equal, as determined by comparing whether
their references point to the same
instance.
    /// - Parameters:
    ///   - lhs: An `AnyCancellable`
instance to compare.
    ///   - rhs: Another `AnyCancellable`
```

```
instance to compare.
    /// - Returns: A Boolean value that
indicates whether two instances are
equal, as determined by comparing whether
their references point to the same
instance.
    public static func == (lhs:
AnyCancellable, rhs: AnyCancellable) ->
Bool

    /// The hash value.
    ///
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    final public var hashValue: Int { get
}
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension AnyCancellable {
```

```
    /// Stores this type-erasing
cancellable instance in the specified
collection.
    ///
    /// - Parameter collection: The
collection in which to store this
``AnyCancellable``.
    @available(macOS 10.15, iOS 13.0,
tvOS 13.0, watchOS 6.0, *)
    final public func store<C>(in
collection: inout C) where C :
RangeReplaceableCollection, C.Element ==
AnyCancellable

    /// Stores this type-erasing
cancellable instance in the specified
set.
    ///
    /// - Parameter set: The set in which
to store this ``AnyCancellable``.
    @available(macOS 10.15, iOS 13.0,
tvOS 13.0, watchOS 6.0, *)
    final public func store(in set: inout
Set<AnyCancellable>)
}

/// A publisher that performs type
erasure by wrapping another publisher.
///
/// ``AnyPublisher`` is a concrete
implementation of ``Publisher`` that has
no significant properties of its own, and
passes through elements and completion
```

```
values from its upstream publisher.
///
/// Use ``AnyPublisher`` to wrap a
publisher whose type has details you
don’t want to expose across API
boundaries, such as different modules.
Wrapping a ``Subject`` with
``AnyPublisher`` also prevents callers
from accessing its ``Subject/send(_:)``
method. When you use type erasure this
way, you can change the underlying
publisher implementation over time
without affecting existing clients.
///
/// You can use Combine’s
``Publisher/eraseToAnyPublisher()``
operator to wrap a publisher with
``AnyPublisher``.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
@frozen public struct
AnyPublisher<Output, Failure> :
CustomStringConvertible,
CustomPlaygroundDisplayConvertible where
Failure : Error {

    /// A textual representation of this
instance.
    ///
    /// Calling this property directly is
discouraged. Instead, convert an
    /// instance of any type to a string
by using the `String(describing:)`
```

```
    /// initializer. This initializer
works with any type, and uses the custom
    /// `description` property for types
that conform to
    /// `CustomStringConvertible`:
    ///
    ///     struct Point:
CustomStringConvertible {
    ///         let x: Int, y: Int
    ///
    ///         var description: String {
    ///             return "(\(x), \(y))"
    ///         }
    ///     }
    ///
    ///     let p = Point(x: 21, y: 30)
    ///     let s = String(describing: p)
    ///     print(s)
    ///     // Prints "(21, 30)"
    ///
    /// The conversion of `p` to a string
in the assignment to `s` uses the
    /// `Point` type's `description`
property.
    public var description: String {
get }

    /// A custom playground description
for this instance.
    public var playgroundDescription: Any
{ get }

    /// Creates a type-erasing publisher
```

```
to wrap the provided publisher.
    ///
    /// - Parameter publisher: A
publisher to wrap with a type-eraser.
    @inlinable public init<P>(_
publisher: P) where Output == P.Output,
Failure == P.Failure, P : Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension AnyPublisher : Publisher {

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    @inlinable public func
receive<S>(subscriber: S) where Output ==
S.Input, Failure == S.Failure, S :
Subscriber
}
```

```
/// A type-erasing subscriber.
///
/// Use an ``AnySubscriber`` to wrap an
existing subscriber whose details you
don’t want to expose. You can also use
``AnySubscriber`` to create a custom
subscriber by providing closures for the
methods defined in ``Subscriber``, rather
than implementing ``Subscriber``
directly.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
@frozen public struct
AnySubscriber<Input, Failure> :
Subscriber, CustomStringConvertible,
CustomReflectable,
CustomPlaygroundDisplayConvertible where
Failure : Error {

    /// A unique identifier for
identifying publisher streams.
    public let combineIdentifier:
CombineIdentifier

    /// A textual representation of this
instance.
    ///
    /// Calling this property directly is
discouraged. Instead, convert an
    /// instance of any type to a string
by using the `String(describing:)`
    /// initializer. This initializer
works with any type, and uses the custom
```

```
    /// `description` property for types
that conform to
    /// `CustomStringConvertible`:
    ///
    ///     struct Point:
CustomStringConvertible {
    ///         let x: Int, y: Int
    ///
    ///         var description: String {
    ///             return "(\(x), \(y))"
    ///         }
    ///     }
    ///
    ///     let p = Point(x: 21, y: 30)
    ///     let s = String(describing: p)
    ///     print(s)
    ///     // Prints "(21, 30)"
    ///
    /// The conversion of `p` to a string
in the assignment to `s` uses the
    /// `Point` type's `description`
property.
    public var description: String {
get }

    /// The custom mirror for this
instance.
    ///
    /// If this type has value semantics,
the mirror should be unaffected by
    /// subsequent mutations of the
instance.
    public var customMirror: Mirror { get
```

```
}

    /// A custom playground description
for this instance.
    public var playgroundDescription: Any
{ get }

    /// Creates a type-erasing subscriber
to wrap an existing subscriber.
    ///
    /// - Parameter s: The subscriber to
type-erase.
    @inlinable public init<S>(_ s: S)
where Input == S.Input, Failure ==
S.Failure, S : Subscriber

    public init<S>(_ s: S) where Input ==
S.Output, Failure == S.Failure, S :
Subject

    /// Creates a type-erasing subscriber
that executes the provided closures.
    ///
    /// - Parameters:
    ///   - receiveSubscription: A
closure to execute when the subscriber
receives the initial subscription from
the publisher.
    ///   - receiveValue: A closure to
execute when the subscriber receives a
value from the publisher.
    ///   - receiveCompletion: A closure
to execute when the subscriber receives a
```

```
completion callback from the publisher.
    @inlinable public
init(receiveSubscription: ((any
Subscription) -> Void)? = nil,
receiveValue: ((Input) ->
Subscribers.Demand)? = nil,
receiveCompletion:
((Subscribers.Completion<Failure>) ->
Void)? = nil)

    /// Tells the subscriber that it has
successfully subscribed to the publisher
and may request items.
    ///
    /// Use the received ``Subscription``
to request items from the publisher.
    /// - Parameter subscription: A
subscription that represents the
connection between publisher and
subscriber.
    @inlinable public func
receive(subscription: any Subscription)

    /// Tells the subscriber that the
publisher has produced an element.
    ///
    /// - Parameter input: The published
element.
    /// - Returns: A `Subscribers.Demand`
instance indicating how many more
elements the subscriber expects to
receive.
    @inlinable public func receive(_
```

```
value: Input) -> Subscribers.Demand

    /// Tells the subscriber that the
publisher has completed publishing,
either normally or with an error.
    ///
    /// - Parameter completion: A
``Subscribers/Completion`` case
indicating whether publishing completed
normally or with an error.
    @inlinable public func
receive(completion:
Subscribers.Completion<Failure>)
}

/// A publisher that exposes its elements
as an asynchronous sequence.
///
/// `AsyncPublisher` conforms to
<doc://com.apple.documentation/documentat
ion/Swift/AsyncSequence>, which allows
callers to receive values with the `for`-
`await`-`in` syntax, rather than
attaching a ``Subscriber``.
///
/// Use the
``Combine/Publisher/values-1dm9r``
property of the ``Combine/Publisher``
protocol to wrap an existing publisher
with an instance of this type.
@available(macOS 12.0, iOS 15.0, tvOS
15.0, watchOS 8.0, *)
public struct AsyncPublisher<P> :
```

```
AsyncSequence where P : Publisher,
P.Failure == Never {

    /// The type of element produced by
this asynchronous sequence.
    public typealias Element = P.Output

    /// The iterator that produces
elements of the asynchronous publisher
sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the prefix sequence.
        ///
        /// - Returns: The next published
element, or nil if the publisher finishes
normally.
        public mutating func next() async
-> P.Output?

        @available(iOS 15.0, tvOS 15.0,
watchOS 8.0, macOS 12.0, *)
        public typealias Element =
P.Output
    }

    /// Creates a publisher that exposes
elements received from an upstream
publisher as an asynchronous sequence.
    ///
    /// - Parameter publisher: An
```

```
upstream publisher. The asynchronous
publisher converts elements received from
this publisher into an asynchronous
sequence.
    public init(_ publisher: P)

    /// Creates the asynchronous iterator
that produces elements of this
asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
elements of the asynchronous sequence.
    public func makeAsyncIterator() ->
AsyncPublisher<P>.Iterator

    /// The type of asynchronous iterator
that produces elements of this
    /// asynchronous sequence.
    @available(iOS 15.0, tvOS 15.0,
watchOS 8.0, macOS 12.0, *)
    public typealias AsyncIterator =
AsyncPublisher<P>.Iterator
}

/// A publisher that exposes its elements
as a throwing asynchronous sequence.
///
/// `AsyncThrowingPublisher` conforms to
<doc://com.apple.documentation/documentat
ion/Swift/AsyncSequence>, which allows
callers to receive values with the `for`-
`await`-`in` syntax, rather than
```

```
attaching a ``Subscriber``. If the
upstream publisher terminates with an
error, `AsyncThrowingPublisher` throws
the error to the awaiting caller.
///
/// Use the ``Combine/Publisher/values-
v7nz`` property of the
``Combine/Publisher`` protocol to wrap an
existing publisher with an instance of
this type.
@available(macOS 12.0, iOS 15.0, tvOS
15.0, watchOS 8.0, *)
public struct AsyncThrowingPublisher<P> :
AsyncSequence where P : Publisher {

    /// The type of element produced by
this asynchronous sequence.
    public typealias Element = P.Output

    /// The iterator that produces
elements of the asynchronous publisher
sequence.
    public struct Iterator :
AsyncIteratorProtocol {

        /// Produces the next element in
the prefix sequence.
        ///
        /// - Returns: The next published
element, or nil if the publisher finishes
normally. If the publisher terminates
with an error, the call point receives
the error as a `throw`.
```

```
        public mutating func next() async
throws -> P.Output?

        @available(iOS 15.0, tvOS 15.0,
watchOS 8.0, macOS 12.0, *)
        public typealias Element =
P.Output
    }

    /// Creates a publisher that exposes
elements received from an upstream
publisher as a throwing asynchronous
sequence.
    /// - Parameter publisher: An
upstream publisher. The asynchronous
publisher converts elements received from
this publisher into an asynchronous
sequence.
    public init(_ publisher: P)

    /// Creates the asynchronous iterator
that produces elements of this
asynchronous sequence.
    ///
    /// - Returns: An instance of the
`AsyncIterator` type used to produce
elements of the asynchronous sequence.
    public func makeAsyncIterator() ->
AsyncThrowingPublisher<P>.Iterator

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

```
    @available(iOS 15.0, tvOS 15.0,
watchOS 8.0, macOS 12.0, *)
    public typealias AsyncIterator =
AsyncThrowingPublisher<P>.Iterator
}

/// A protocol indicating that an
activity or action supports cancellation.
///
/// Calling ``Cancellable/cancel()``
frees up any allocated resources. It also
stops side effects such as timers,
network access, or disk I/O.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol Cancellable {

    /// Cancel the activity.
    ///
    /// When implementing ``Cancellable``
in support of a custom publisher,
implement `cancel()` to request that your
publisher stop calling its downstream
subscribers. Combine doesn't require that
the publisher stop immediately, but the
`cancel()` call should take effect
quickly. Canceling should also eliminate
any strong references it currently holds.
    ///
    /// After you receive one call to
`cancel()`, subsequent calls shouldn't do
anything. Additionally, your
implementation must be thread-safe, and
```

```
it shouldn't block the caller.
    ///
    /// > Tip: Keep in mind that your
`cancel()` may execute concurrently with
another call to `cancel()` --- including
the scenario where an ``AnyCancellable``
is deallocating --- or to
``Subscription/request(_:)``.
    func cancel()
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Cancellable {

    /// Stores this cancellable instance
in the specified collection.
    ///
    /// - Parameter collection: The
collection in which to store this
``Cancellable``.
    @available(macOS 10.15, iOS 13.0,
tvOS 13.0, watchOS 6.0, *)
    public func store<C>(in collection:
inout C) where C :
RangeReplaceableCollection, C.Element ==
AnyCancellable

    /// Stores this cancellable instance
in the specified set.
    ///
    /// - Parameter set: The set in which
to store this ``Cancellable``.
```

```
    @available(macOS 10.15, iOS 13.0,
tvOS 13.0, watchOS 6.0, *)
    public func store(in set: inout
Set<AnyCancellable>)
}

/// A unique identifier for identifying
publisher streams.
///
/// To conform to
``CustomCombineIdentifierConvertible`` in
a
/// ``Subscription`` or ``Subject`` that
you implement as a structure, create an
instance of ``CombineIdentifier`` as
follows:
///
///     let combineIdentifier =
CombineIdentifier()
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct CombineIdentifier :
Hashable, CustomStringConvertible {

    /// Creates a unique Combine
identifier.
    public init()

    /// Creates a Combine identifier,
using the bit pattern of the provided
object.
    public init(_ obj: AnyObject)
```

```
    /// A textual representation of this
instance.
    public var description: String {
get }

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// Implement this method to conform
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    public func hash(into hasher: inout
Hasher)

    /// Returns a Boolean value
```

```
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (a:
CombineIdentifier, b: CombineIdentifier)
-> Bool

    /// The hash value.
    ///
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    public var hashValue: Int { get }
}
```

```
/// A publisher that provides an explicit
means of connecting and canceling
publication.
///
/// Use a ``ConnectablePublisher`` when
you need to perform additional
configuration or setup prior to producing
any elements.
///
/// This publisher doesn’t produce any
elements until you call its
``ConnectablePublisher/connect()``
method.
///
/// Use ``Publisher/makeConnectable()``
to create a ``ConnectablePublisher`` from
any publisher whose failure type is
<doc://com.apple.documentation/documentat
ion/Swift/Never>.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol
ConnectablePublisher<Output, Failure> :
Publisher {

    /// Connects to the publisher,
allowing it to produce elements, and
returns an instance with which to cancel
publishing.
    ///
    /// - Returns: A ``Cancellable``
instance that you use to cancel
publishing.
```

```
    func connect() -> any Cancellable
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension ConnectablePublisher {

    /// Automates the process of
connecting or disconnecting from this
connectable publisher.
    ///
    /// Use
``ConnectablePublisher/autoconnect()`` to
simplify working with
``ConnectablePublisher`` instances, such
as
<doc://com.apple.documentation/documentat
ion/Foundation/Timer/TimerPublisher> in
the Foundation framework.
    ///
    /// In the following example, the
<doc://com.apple.documentation/documentat
ion/Foundation/Timer/3329589-publish>
operator creates a
<doc://com.apple.documentation/documentat
ion/Foundation/Timer/TimerPublisher>,
which is a ``ConnectablePublisher``. As a
result, subscribers don’t receive any
values until after a call to
``ConnectablePublisher/connect()``.
    /// For convenience when working with
a single subscriber, the
``ConnectablePublisher/autoconnect()``
```

```
operator performs the
``ConnectablePublisher/connect()`` call
when attached to by the subscriber.
    ///
    ///     cancellable =
Timer.publish(every: 1, on: .main,
in: .default)
    ///         .autoconnect()
    ///         .sink { date in
    ///             print ("Date now: \
(date)")
    ///         }
    /// - Returns: A publisher which
automatically connects to its upstream
connectable publisher.
    public func autoconnect() ->
Publishers.Autoconnect<Self>
}

/// A subject that wraps a single value
and publishes a new element whenever the
value changes.
///
/// Unlike ``PassthroughSubject``,
``CurrentValueSubject`` maintains a
buffer of the most recently published
element.
///
/// Calling
``CurrentValueSubject/send(_:)`` on a
``CurrentValueSubject`` also updates the
current value, making it equivalent to
updating the
```

```
``CurrentValueSubject/value`` directly.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
final public class
CurrentValueSubject<Output, Failure> :
Subject where Failure : Error {

    /// The value wrapped by this
subject, published as a new element
whenever it changes.
    final public var value: Output

    /// Creates a current value subject
with the given initial value.
    ///
    /// - Parameter value: The initial
value to publish.
    public init(_ value: Output)

    /// Sends a subscription to the
subscriber.
    ///
    /// This call provides the
``Subject`` an opportunity to establish
demand for any new upstream
subscriptions.
    ///
    /// - Parameter subscription: The
subscription instance through which the
subscriber can request elements.
    final public func send(subscription:
any Subscription)
```

```
    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    final public func
receive<S>(subscriber: S) where Output ==
S.Input, Failure == S.Failure, S :
Subscriber

    /// Sends a value to the subscriber.
    ///
    /// - Parameter value: The value to
send.
    final public func send(_ input:
Output)

    /// Sends a completion signal to the
subscriber.
    ///
    /// - Parameter completion: A
`Completion` instance which indicates
whether publishing has finished normally
or failed with an error.
```

```
    final public func send(completion:
Subscribers.Completion<Failure>)
}

/// A protocol for uniquely identifying
publisher streams.
///
/// If you create a custom
``Subscription`` or ``Subscriber`` type,
implement this protocol so that
development tools can uniquely identify
publisher chains in your app.
/// If your type is a class, Combine
provides an implementation of
``CustomCombineIdentifierConvertible/comb
ineIdentifier-1frze`` for you.
/// If your type is a structure, set up
the identifier as follows:
///
///     let combineIdentifier =
CombineIdentifier()
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol
CustomCombineIdentifierConvertible {

    /// A unique identifier for
identifying publisher streams.
    var combineIdentifier:
CombineIdentifier { get }
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension
CustomCombineIdentifierConvertible where
Self : AnyObject {

    /// A unique identifier for
identifying publisher streams.
    public var combineIdentifier:
CombineIdentifier { get }
}

/// A publisher that awaits subscription
before running the supplied closure to
create a publisher for the new
subscriber.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct Deferred<DeferredPublisher>
: Publisher where DeferredPublisher :
Publisher {

    /// The kind of values published by
this publisher.
    public typealias Output =
DeferredPublisher.Output

    /// The kind of errors this publisher
might publish.
    public typealias Failure =
DeferredPublisher.Failure

    /// The closure to execute when this
deferred publisher receives a
```

```
subscription.
    ///
    /// The publisher returned by this
closure immediately receives the incoming
subscription.
    public let createPublisher: () ->
DeferredPublisher

    /// Creates a deferred publisher.
    ///
    /// - Parameter createPublisher: The
closure to execute when calling
`subscribe(_:)`.
    public init(createPublisher:
@escaping () -> DeferredPublisher)

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    public func receive<S>(subscriber: S)
where S : Subscriber,
DeferredPublisher.Failure == S.Failure,
```

```
DeferredPublisher.Output == S.Input
}

/// A publisher that never publishes any
values, and optionally finishes
immediately.
///
/// You can create a ”Never” publisher —
one which never sends values and never
finishes or fails — with the initializer
`Empty(completeImmediately: false)`.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct Empty<Output, Failure> :
Publisher, Equatable where Failure :
Error {

    /// Creates an empty publisher.
    ///
    /// - Parameter completeImmediately:
A Boolean value that indicates whether
the publisher should immediately finish.
    public init(completeImmediately: Bool
= true)

    /// Creates an empty publisher with
the given completion behavior and output
and failure types.
    ///
    /// Use this initializer to connect
the empty publisher to subscribers or
other publishers that have specific
output and failure types.
```

```
    ///
    /// - Parameters:
    ///   - completeImmediately: A
Boolean value that indicates whether the
publisher should immediately finish.
    ///   - outputType: The output type
exposed by this publisher.
    ///   - failureType: The failure type
exposed by this publisher.
    public init(completeImmediately: Bool
= true, outputType: Output.Type,
failureType: Failure.Type)

    /// A Boolean value that indicates
whether the publisher immediately sends a
completion.
    ///
    /// If `true`, the publisher finishes
immediately after sending a subscription
to the subscriber. If `false`, it never
completes.
    public let completeImmediately: Bool

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
```

```
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    public func receive<S>(subscriber: S)
where Output == S.Input, Failure ==
S.Failure, S : Subscriber

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: An `Empty` instance to
compare.
    ///   - rhs: Another `Empty` instance
to compare.
    /// - Returns: `true` if the two
publishers have equal
`completeImmediately` properties;
otherwise `false`.
    public static func == (lhs:
Empty<Output, Failure>, rhs:
Empty<Output, Failure>) -> Bool
}

/// A publisher that immediately
terminates with the specified error.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct Fail<Output, Failure> :
Publisher where Failure : Error {

    /// Creates a publisher that
```

```
immediately terminates with the specified
failure.
    ///
    /// - Parameter error: The failure to
send when terminating the publisher.
    public init(error: Failure)

    /// Creates publisher with the given
output type, that immediately terminates
with the specified failure.
    ///
    /// Use this initializer to create a
`Fail` publisher that can work with
subscribers or publishers that expect a
given output type.
    ///
    /// - Parameters:
    ///   - outputType: The output type
exposed by this publisher.
    ///   - failure: The failure to send
when terminating the publisher.
    public init(outputType: Output.Type,
failure: Failure)

    /// The failure to send when
terminating the publisher.
    public let error: Failure

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
```

```
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    public func receive<S>(subscriber: S)
where Output == S.Input, Failure ==
S.Failure, S : Subscriber
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Fail : Equatable where
Failure : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `Fail` publisher to
compare for equality.
    ///   - rhs: Another `Fail` publisher
to compare for equality.
    /// - Returns: `true` if the
publishers have equal `error` properties;
otherwise `false`.
    public static func == (lhs:
Fail<Output, Failure>, rhs: Fail<Output,
Failure>) -> Bool
```

```
}

/// A publisher that eventually produces
a single value and then finishes or
fails.
///
/// Use a future to perform some work and
then asynchronously publish a single
element. You initialize the future with a
closure that takes a
``Combine/Future/Promise``; the closure
calls the promise with a
<doc://com.apple.documentation/documentat
ion/Swift/Result> that indicates either
success or failure. In the success case,
the future's downstream subscriber
receives the element prior to the
publishing stream finishing normally. If
the result is an error, publishing
terminates with that error.
///
/// The following example shows a method
that uses a future to asynchronously
publish a random number after a brief
delay:
///
///     func
generateAsyncRandomNumberFromFuture() ->
Future <Int, Never> {
///         return Future() { promise in
///
DispatchQueue.main.asyncAfter(deadline: .
now() + 2) {
```

```
///                 let number =
Int.random(in: 1...10)
///
promise(Result.success(number))
///             }
///         }
///     }
///
/// To receive the published value, you
use any Combine subscriber, such as a
``Combine/Subscribers/Sink``, like this:
///
///     cancellable =
generateAsyncRandomNumberFromFuture()
///         .sink { number in print("Got
random number \(number).") }
///
/// ### Integrating with Swift
Concurrency
///
/// To integrate with the `async`-`await`
syntax in Swift 5.5, `Future` can provide
its value to an awaiting caller. This is
particularly useful because unlike other
types that conform to ``Publisher`` and
potentially publish many elements, a
`Future` only publishes one element (or
fails). By using the
``Combine/Future/value-9iwjz`` property,
the above call point looks like this:
///
///     let number = await
generateAsyncRandomNumberFromFuture().val
```

```
ue
///     print("Got random number \
(number).")
///
/// ### Alternatives to Futures
///
/// The `async`-`await` syntax in Swift
can also replace the use of a future
entirely, for the case where you want to
perform some operation after an
asynchronous task completes.
///
/// You do this with the function
<doc://com.apple.documentation/documentat
ion/swift/withCheckedContinuation(functio
n:_:)> and its throwing equivalent,
<doc://com.apple.documentation/documentat
ion/swift/swift/withCheckedThrowingContin
uation(function:_:)>. The following
example performs the same asynchronous
random number generation as the `Future`
example above, but as an `async` method:
///
///     func
generateAsyncRandomNumberFromContinuation
() async -> Int {
///         return await
withCheckedContinuation { continuation in
///
DispatchQueue.main.asyncAfter(deadline: .
now() + 2) {
///                 let number =
Int.random(in: 1...10)
```

```
///
continuation.resume(returning: number)
///             }
///         }
///     }
///
/// The call point for this method
doesn't use a closure like the future's
sink subscriber does; it simply awaits
and assigns the result:
///
///     let asyncRandom = await
generateAsyncRandomNumberFromContinuation
()
///
/// For more information on
continuations, see the
<doc://com.apple.documentation/documentat
ion/swift/concurrency> topic in the Swift
standard library.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
final public class Future<Output,
Failure> : Publisher where Failure :
Error {

    /// A type that represents a closure
to invoke in the future, when an element
or error is available.
    ///
    /// The promise closure receives one
parameter: a `Result` that contains
either a single element published by a
```

```
``Future``, or an error.
    public typealias Promise =
(Result<Output, Failure>) -> Void

    /// Creates a publisher that invokes
a promise closure when the publisher
emits an element.
    ///
    /// - Parameter attemptToFulfill: A
``Future/Promise`` that the publisher
invokes when the publisher emits an
element or terminates with an error.
    public init(_ attemptToFulfill:
@escaping (@escaping Future<Output,
Failure>.Promise) -> Void)

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    final public func
receive<S>(subscriber: S) where Output ==
S.Input, Failure == S.Failure, S :
```

```
Subscriber
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Future where Failure == Never {

    /// The published value of the
future, delivered asynchronously.
    ///
    /// This property subscribes to the
`Future` and delivers the value
asynchronously when the `Future`
publishes it. Use this property when you
want to use the `async`-`await` syntax
with a `Future`.
    @available(macOS 12.0, iOS 15.0, tvOS
15.0, watchOS 8.0, *)
    final public var value: Output { get
async }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Future {

    /// The published value of the future
or an error, delivered asynchronously.
    ///
    /// This property subscribes to the
`Future` and delivers the value
asynchronously when the `Future`
publishes it. If the `Future` terminates
```

```
with an error, the awaiting caller
receives the error instead. Use this
property when you want to the `async`-
`await` syntax with a `Future` whose
``Publisher/Failure`` type is not
<doc://com.apple.documentation/documentat
ion/Swift/Never>.
    @available(macOS 12.0, iOS 15.0, tvOS
15.0, watchOS 8.0, *)
    final public var value: Output { get
async throws }
}

/// A scheduler for performing
synchronous actions.
///
/// You can only use this scheduler for
immediate actions. If you attempt to
schedule actions after a specific date,
this scheduler ignores the date and
performs them immediately.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct ImmediateScheduler :
Scheduler {

    /// The time type used by the
immediate scheduler.
    public struct SchedulerTimeType :
Strideable {

        /// Returns the distance to
another immediate scheduler time; this
```

```
distance is always `0` in the context of
an immediate scheduler.
        ///
        /// - Parameter other: The other
scheduler time.
        /// - Returns: `0`, as a
`Stride`.
        public func distance(to other:
ImmediateScheduler.SchedulerTimeType) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

        /// Advances the time by the
specified amount; this is meaningless in
the context of an immediate scheduler.
        ///
        /// - Parameter n: The amount to
advance by. The `ImmediateScheduler`
ignores this value.
        /// - Returns: An empty
`SchedulerTimeType`.
        public func advanced(by n:
ImmediateScheduler.SchedulerTimeType.Stri
de) ->
ImmediateScheduler.SchedulerTimeType

        /// The increment by which the
immediate scheduler counts time.
        public struct Stride :
ExpressibleByFloatLiteral, Comparable,
SignedNumeric, Codable,
SchedulerTimeIntervalConvertible {
```

```
            /// The type used when
evaluating floating-point literals.
            public typealias
FloatLiteralType = Double

            /// The type used when
evaluating integer literals.
            public typealias
IntegerLiteralType = Int

            /// The type used for
expressing the stride's magnitude.
            public typealias Magnitude =
Int

            /// The value of this time
interval in seconds.
            public var magnitude: Int

            /// Creates an immediate
scheduler time interval from the given
time interval.
            public init(_ value: Int)

            /// Creates an immediate
scheduler time interval from an integer
seconds value.
            public init(integerLiteral
value: Int)

            /// Creates an immediate
scheduler time interval from a floating-
point seconds value.
```

```
            public init(floatLiteral
value: Double)

            /// Creates an immediate
scheduler time interval from a binary
integer type.
            ///
            /// If `exactly` can't
convert to an `Int`, the resulting time
interval is `nil`.
            public init?<T>(exactly
source: T) where T : BinaryInteger

            /// Returns a Boolean value
indicating whether the value of the first
            /// argument is less than
that of the second argument.
            ///
            /// This function is the only
requirement of the `Comparable` protocol.
The
            /// remainder of the
relational operator functions are
implemented by the
            /// standard library for any
type that conforms to `Comparable`.
            ///
            /// - Parameters:
            ///   - lhs: A value to
compare.
            ///   - rhs: Another value to
compare.
            public static func < (lhs:
```

```
ImmediateScheduler.SchedulerTimeType.Stride, rhs:
ImmediateScheduler.SchedulerTimeType.Stride) -> Bool

            /// Multiplies two values and produces their product.
            ///
            /// The multiplication operator (`*`) calculates the product of its two
            /// arguments. For example:
            ///
            ///     2 * 3
// 6
            ///     100 * 21
// 2100
            ///     -10 * 15
// -150
            ///     3.5 * 2.25
// 7.875
            ///
            /// You cannot use `*` with arguments of different types. To multiply values
            /// of different types, convert one of the values to the other value's type.
            ///
            ///     let x: Int8 = 21
            ///     let y: Int = 1000000
            ///     Int(x) * y
// 21000000
```

```
            ///
            /// - Parameters:
            ///   - lhs: The first value
to multiply.
            ///   - rhs: The second value
to multiply.
            public static func * (lhs:
ImmediateScheduler.SchedulerTimeType.Stri
de, rhs:
ImmediateScheduler.SchedulerTimeType.Stri
de) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Adds two values and
produces their sum.
            ///
            /// The addition operator
(`+`) calculates the sum of its two
arguments. For
            /// example:
            ///
            ///     1 + 2
// 3
            ///     -10 + 15
// 5
            ///     -15 + -5
// -20
            ///     21.5 + 3.25
// 24.75
            ///
            /// You cannot use `+` with
arguments of different types. To add
```

```
values of
            /// different types, convert
one of the values to the other value's
type.
            ///
            ///     let x: Int8 = 21
            ///     let y: Int = 1000000
            ///     Int(x) + y
// 1000021
            ///
            /// - Parameters:
            ///   - lhs: The first value
to add.
            ///   - rhs: The second value
to add.
            public static func + (lhs:
ImmediateScheduler.SchedulerTimeType.Stri
de, rhs:
ImmediateScheduler.SchedulerTimeType.Stri
de) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Subtracts one value from
another and produces their difference.
            ///
            /// The subtraction operator
(`-`) calculates the difference of its
two
            /// arguments. For example:
            ///
            ///     8 - 3
// 5
```

```
            ///     -10 - 5
// -15
            ///     100 - -5
// 105
            ///     10.5 - 100.0
// -89.5
            ///
            /// You cannot use `-` with
arguments of different types. To subtract
values
            /// of different types,
convert one of the values to the other
value's type.
            ///
            ///     let x: UInt8 = 21
            ///     let y: UInt = 1000000
            ///     y - UInt(x)
// 999979
            ///
            /// - Parameters:
            ///   - lhs: A numeric value.
            ///   - rhs: The value to
subtract from `lhs`.
            public static func - (lhs:
ImmediateScheduler.SchedulerTimeType.Stri
de, rhs:
ImmediateScheduler.SchedulerTimeType.Stri
de) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Subtracts the second
value from the first and stores the
```

```
difference in the
            /// left-hand-side variable.
            ///
            /// - Parameters:
            ///   - lhs: A numeric value.
            ///   - rhs: The value to
subtract from `lhs`.
            public static func -= (lhs:
inout
ImmediateScheduler.SchedulerTimeType.Stri
de, rhs:
ImmediateScheduler.SchedulerTimeType.Stri
de)

            /// Multiplies two values and
stores the result in the left-hand-side
            /// variable.
            ///
            /// - Parameters:
            ///   - lhs: The first value
to multiply.
            ///   - rhs: The second value
to multiply.
            public static func *= (lhs:
inout
ImmediateScheduler.SchedulerTimeType.Stri
de, rhs:
ImmediateScheduler.SchedulerTimeType.Stri
de)

            /// Adds two values and
stores the result in the left-hand-side
variable.
```

```
            ///
            /// - Parameters:
            ///   - lhs: The first value
to add.
            ///   - rhs: The second value
to add.
            public static func += (lhs:
inout
ImmediateScheduler.SchedulerTimeType.Stri
de, rhs:
ImmediateScheduler.SchedulerTimeType.Stri
de)

            /// Converts the specified
number of seconds into an instance of
this scheduler time type.
            public static func seconds(_
s: Int) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Converts the specified
number of seconds, as a floating-point
value, into an instance of this scheduler
time type.
            public static func seconds(_
s: Double) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Converts the specified
number of milliseconds into an instance
of this scheduler time type.
```

```
            public static func
milliseconds(_ ms: Int) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Converts the specified
number of microseconds into an instance
of this scheduler time type.
            public static func
microseconds(_ us: Int) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Converts the specified
number of nanoseconds into an instance of
this scheduler time type.
            public static func
nanoseconds(_ ns: Int) ->
ImmediateScheduler.SchedulerTimeType.Stri
de

            /// Returns a Boolean value
indicating whether two values are equal.
            ///
            /// Equality is the inverse
of inequality. For any values `a` and
`b`,
            /// `a == b` implies that
`a != b` is `false`.
            ///
            /// - Parameters:
            ///   - lhs: A value to
compare.
```

```
            ///   - rhs: Another value to
compare.
            public static func == (a:
ImmediateScheduler.SchedulerTimeType.Stri
de, b:
ImmediateScheduler.SchedulerTimeType.Stri
de) -> Bool

            /// Encodes this value into
the given encoder.
            ///
            /// If the value fails to
encode anything, `encoder` will encode an
empty
            /// keyed container in its
place.
            ///
            /// This function throws an
error if any values are invalid for the
given
            /// encoder's format.
            ///
            /// - Parameter encoder: The
encoder to write data to.
            public func encode(to
encoder: any Encoder) throws

            /// Creates a new instance by
decoding from the given decoder.
            ///
            /// This initializer throws
an error if reading from the decoder
fails, or
```

```
            /// if the data read is
corrupted or otherwise invalid.
            ///
            /// - Parameter decoder: The
decoder to read data from.
            public init(from decoder: any
Decoder) throws
        }
    }

    /// A type that defines options
accepted by the immediate scheduler.
    public typealias SchedulerOptions =
Never

    /// The shared instance of the
immediate scheduler.
    ///
    /// You cannot create instances of
the immediate scheduler yourself. Use
only the shared instance.
    public static let shared:
ImmediateScheduler

    /// Performs the action at the next
possible opportunity.
    public func schedule(options:
ImmediateScheduler.SchedulerOptions?, _
action: @escaping () -> Void)

    /// The immediate scheduler's
definition of the current moment in time.
    public var now:
```

```
ImmediateScheduler.SchedulerTimeType {
get }

    /// The minimum tolerance allowed by
the immediate scheduler.
    public var minimumTolerance:
ImmediateScheduler.SchedulerTimeType.Stri
de { get }

    /// Performs the action at some time
after the specified date.
    ///
    /// The immediate scheduler ignores
`date` and performs the action
immediately.
    public func schedule(after date:
ImmediateScheduler.SchedulerTimeType,
tolerance:
ImmediateScheduler.SchedulerTimeType.Stri
de, options:
ImmediateScheduler.SchedulerOptions?, _
action: @escaping () -> Void)

    /// Performs the action at some time
after the specified date, at the
specified frequency, optionally taking
into account tolerance if possible.
    ///
    /// The immediate scheduler ignores
`date` and performs the action
immediately.
    @discardableResult
    public func schedule(after date:
```

```
ImmediateScheduler.SchedulerTimeType,
interval:
ImmediateScheduler.SchedulerTimeType.Stri
de, tolerance:
ImmediateScheduler.SchedulerTimeType.Stri
de, options:
ImmediateScheduler.SchedulerOptions?, _
action: @escaping () -> Void) -> any
Cancellable
}

/// A publisher that emits an output to
each subscriber just once, and then
finishes.
///
/// You can use a ``Just`` publisher to
start a chain of publishers. A ``Just``
publisher is also useful when replacing a
value with ``Publishers/Catch``.
///
/// In contrast with
<doc://com.apple.documentation/documentat
ion/Swift/Result/publisher-swift.struct>,
a ``Just`` publisher can't fail with an
error. And unlike
<doc://com.apple.documentation/documentat
ion/Swift/Optional/publisher-
swift.struct>, a ``Just`` publisher
always produces a value.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct Just<Output> : Publisher {
```

```
    /// The kind of errors this publisher
might publish.
    ///
    /// Use `Never` if this `Publisher`
does not publish errors.
    public typealias Failure = Never

    /// The one element that the
publisher emits.
    public let output: Output

    /// Initializes a publisher that
emits the specified output just once.
    ///
    /// - Parameter output: The one
element that the publisher emits.
    public init(_ output: Output)

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    public func receive<S>(subscriber: S)
```

```
where Output == S.Input, S : Subscriber,
S.Failure == Never
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Just : Equatable where Output :
Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `Just` publisher to
compare for equality.
    ///   - rhs: Another `Just` publisher
to compare for equality.
    /// - Returns: `true` if the
publishers have equal `output`
properties; otherwise `false`.
    public static func == (lhs:
Just<Output>, rhs: Just<Output>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Just where Output : Comparable
{

    public func min() -> Just<Output>

    public func max() -> Just<Output>
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Just where Output : Equatable {

    public func contains(_ output:
Output) -> Just<Bool>

    public func removeDuplicates() ->
Just<Output>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Just {

    public func allSatisfy(_ predicate:
(Output) -> Bool) -> Just<Bool>

    public func tryAllSatisfy(_
predicate: (Output) throws -> Bool) ->
Result<Bool, any Error>.Publisher

    public func collect() ->
Just<[Output]>

    public func compactMap<T>(_
transform: (Output) -> T?) ->
Optional<T>.Publisher

    public func min(by
areInIncreasingOrder: (Output, Output) ->
Bool) -> Just<Output>
```

```
    public func max(by
areInIncreasingOrder: (Output, Output) ->
Bool) -> Just<Output>

    public func prepend(_ elements:
Output...) ->
Publishers.Sequence<[Output],
Just<Output>.Failure>

    public func prepend<S>(_ elements: S)
-> Publishers.Sequence<[Output],
Just<Output>.Failure> where Output ==
S.Element, S : Sequence

    public func append(_ elements:
Output...) ->
Publishers.Sequence<[Output],
Just<Output>.Failure>

    public func append<S>(_ elements: S)
-> Publishers.Sequence<[Output],
Just<Output>.Failure> where Output ==
S.Element, S : Sequence

    public func contains(where predicate:
(Output) -> Bool) -> Just<Bool>

    public func tryContains(where
predicate: (Output) throws -> Bool) ->
Result<Bool, any Error>.Publisher

    public func count() -> Just<Int>
```

```
    public func dropFirst(_ count: Int =
1) -> Optional<Output>.Publisher

    public func drop(while predicate:
(Output) -> Bool) ->
Optional<Output>.Publisher

    public func first() -> Just<Output>

    public func first(where predicate:
(Output) -> Bool) ->
Optional<Output>.Publisher

    public func last() -> Just<Output>

    public func last(where predicate:
(Output) -> Bool) ->
Optional<Output>.Publisher

    public func filter(_ isIncluded:
(Output) -> Bool) ->
Optional<Output>.Publisher

    public func ignoreOutput() ->
Empty<Output, Just<Output>.Failure>

    public func map<T>(_ transform:
(Output) -> T) -> Just<T>

    public func tryMap<T>(_ transform:
(Output) throws -> T) -> Result<T, any
Error>.Publisher
```

```
    public func mapError<E>(_ transform:
(Just<Output>.Failure) -> E) ->
Result<Output, E>.Publisher where E :
Error

    public func output(at index: Int) ->
Optional<Output>.Publisher

    public func output<R>(in range: R) ->
Optional<Output>.Publisher where R :
RangeExpression, R.Bound == Int

    public func prefix(_ maxLength: Int)
-> Optional<Output>.Publisher

    public func prefix(while predicate:
(Output) -> Bool) ->
Optional<Output>.Publisher

    public func reduce<T>(_
initialResult: T, _ nextPartialResult:
(T, Output) -> T) -> Result<T,
Just<Output>.Failure>.Publisher

    public func tryReduce<T>(_
initialResult: T, _ nextPartialResult:
(T, Output) throws -> T) -> Result<T, any
Error>.Publisher

    public func removeDuplicates(by
predicate: (Output, Output) -> Bool) ->
Just<Output>
```

```
    public func tryRemoveDuplicates(by
predicate: (Output, Output) throws ->
Bool) -> Result<Output, any
Error>.Publisher

    public func replaceError(with output:
Output) -> Just<Output>

    public func replaceEmpty(with output:
Output) -> Just<Output>

    public func retry(_ times: Int) ->
Just<Output>

    public func scan<T>(_ initialResult:
T, _ nextPartialResult: (T, Output) -> T)
-> Result<T,
Just<Output>.Failure>.Publisher

    public func tryScan<T>(_
initialResult: T, _ nextPartialResult:
(T, Output) throws -> T) -> Result<T, any
Error>.Publisher

    public func setFailureType<E>(to
failureType: E.Type) -> Result<Output,
E>.Publisher where E : Error
}

/// A type of object with a publisher
that emits before the object has changed.
///
```

```
/// By default an ``ObservableObject`` synthesizes an ``ObservableObject/objectWillChange-2oa5v`` publisher that emits the changed value before any of its `@Published` properties changes.
///
///     class Contact: ObservableObject {
///         @Published var name: String
///         @Published var age: Int
///
///         init(name: String, age: Int) {
///             self.name = name
///             self.age = age
///         }
///
///         func haveBirthday() -> Int {
///             age += 1
///             return age
///         }
///     }
///
///     let john = Contact(name: "John Appleseed", age: 24)
///     cancellable = john.objectWillChange
///         .sink { _ in
///             print("\(john.age) will change")
///     }
///     print(john.haveBirthday())
///     // Prints "24 will change"
```

```
///     // Prints "25"
@available(iOS 13.0, macOS 10.15, tvOS
13.0, watchOS 6.0, *)
public protocol ObservableObject :
AnyObject {

    /// The type of publisher that emits
before the object has changed.
    associatedtype
ObjectWillChangePublisher : Publisher =
ObservableObjectPublisher where
Self.ObjectWillChangePublisher.Failure ==
Never

    /// A publisher that emits before the
object has changed.
    var objectWillChange:
Self.ObjectWillChangePublisher { get }
}

@available(iOS 13.0, macOS 10.15, tvOS
13.0, watchOS 6.0, *)
extension ObservableObject where
Self.ObjectWillChangePublisher ==
ObservableObjectPublisher {

    /// A publisher that emits before the
object has changed.
    public var objectWillChange:
ObservableObjectPublisher { get }
}

/// A publisher that publishes changes
```

```
from observable objects.
@available(iOS 13.0, macOS 10.15, tvOS
13.0, watchOS 6.0, *)
final public class
ObservableObjectPublisher : Publisher {

    /// The kind of values published by
this publisher.
    public typealias Output = Void

    /// The kind of errors this publisher
might publish.
    ///
    /// Use `Never` if this `Publisher`
does not publish errors.
    public typealias Failure = Never

    /// Creates an observable object
publisher instance.
    public init()

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
```

```
``Publisher``, after which it can receive
values.
    final public func
receive<S>(subscriber: S) where S :
Subscriber, S.Failure == Never, S.Input
== ()

    /// Sends the changed value to the
downstream subscriber.
    final public func send()
}

/// A subject that broadcasts elements to
downstream subscribers.
///
/// As a concrete implementation of
``Subject``, the ``PassthroughSubject``
provides a convenient way to adapt
existing imperative code to the Combine
model.
///
/// Unlike ``CurrentValueSubject``, a
``PassthroughSubject`` doesn't have an
initial value or a buffer of the most
recently-published element.
/// A ``PassthroughSubject`` drops values
if there are no subscribers, or its
current demand is zero.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
final public class
PassthroughSubject<Output, Failure> :
Subject where Failure : Error {
```

```
    public init()

    /// Sends a subscription to the
subscriber.
    ///
    /// This call provides the
``Subject`` an opportunity to establish
demand for any new upstream
subscriptions.
    ///
    /// - Parameter subscription: The
subscription instance through which the
subscriber can request elements.
    final public func send(subscription:
any Subscription)

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    final public func
receive<S>(subscriber: S) where Output ==
```

```
S.Input, Failure == S.Failure, S :
Subscriber

    /// Sends a value to the subscriber.
    ///
    /// - Parameter value: The value to
send.
    final public func send(_ input:
Output)

    /// Sends a completion signal to the
subscriber.
    ///
    /// - Parameter completion: A
`Completion` instance which indicates
whether publishing has finished normally
or failed with an error.
    final public func send(completion:
Subscribers.Completion<Failure>)
}

/// A type that publishes a property
marked with an attribute.
///
/// Publishing a property with the
`@Published` attribute creates a
publisher of this type. You access the
publisher with the `$` operator, as shown
here:
///
///     class Weather {
///         @Published var temperature:
Double
```

```
///         init(temperature: Double) {
///             self.temperature =
temperature
///         }
///     }
///
///     let weather =
Weather(temperature: 20)
///     cancellable = weather.
$temperature
///         .sink() {
///             print ("Temperature now:
\($0)")
///     }
///     weather.temperature = 25
///
///     // Prints:
///     // Temperature now: 20.0
///     // Temperature now: 25.0
///
/// When the property changes, publishing
occurs in the property's `willSet` block,
meaning subscribers receive the new value
before it's actually set on the property.
In the above example, the second time the
sink executes its closure, it receives
the parameter value `25`. However, if the
closure evaluated `weather.temperature`,
the value returned would be `20`.
///
/// > Important: The `@Published`
attribute is class constrained. Use it
with properties of classes, not with non-
```

```
class types like structures.
///
/// ### See Also
///
/// - ``Combine/Publisher/assign(to:)``
@available(iOS 13.0, macOS 10.15, tvOS
13.0, watchOS 6.0, *)
@propertyWrapper public struct
Published<Value> {

    /// Creates the published instance
with an initial wrapped value.
    ///
    /// Don't use this initializer
directly. Instead, create a property with
the `@Published` attribute, as shown
here:
    ///
    ///     @Published var lastUpdated:
Date = Date()
    ///
    /// - Parameter wrappedValue: The
publisher's initial value.
    public init(wrappedValue: Value)

    /// Creates the published instance
with an initial value.
    ///
    /// Don't use this initializer
directly. Instead, create a property with
the `@Published` attribute, as shown
here:
    ///
```

```
    ///     @Published var lastUpdated:
Date = Date()
    ///
    /// - Parameter initialValue: The
publisher's initial value.
    public init(initialValue: Value)

    /// A publisher for properties marked
with the `@Published` attribute.
    public struct Publisher : Publisher {

        /// The kind of values published
by this publisher.
        public typealias Output = Value

        /// The kind of errors this
publisher might publish.
        ///
        /// Use `Never` if this
`Publisher` does not publish errors.
        public typealias Failure = Never

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
```

```
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Value ==
S.Input, S : Subscriber, S.Failure ==
Never
    }

    /// The property for which this
instance exposes a publisher.
    ///
    /// The ``Published/projectedValue``
is the property accessed with the `$`
operator.
    public var projectedValue:
Published<Value>.Publisher { mutating get
set }
}

/// Declares that a type can transmit a
sequence of values over time.
///
/// A publisher delivers elements to one
or more ``Subscriber`` instances.
/// The subscriber's ``Subscriber/Input``
and ``Subscriber/Failure`` associated
types must match the ``Publisher/Output``
and ``Publisher/Failure`` types declared
by the publisher.
/// The publisher implements the
``Publisher/receive(subscriber:)``method
to accept a subscriber.
```

```
///
/// After this, the publisher can call
the following methods on the subscriber:
/// -
``Subscriber/receive(subscription:)``:
Acknowledges the subscribe request and
returns a ``Subscription`` instance. The
subscriber uses the subscription to
demand elements from the publisher and
can use it to cancel publishing.
/// - ``Subscriber/receive(_:)``:
Delivers one element from the publisher
to the subscriber.
/// -
``Subscriber/receive(completion:)``: Info
rms the subscriber that publishing has
ended, either normally or with an error.
///
/// Every `Publisher` must adhere to this
contract for downstream subscribers to
function correctly.
///
/// Extensions on `Publisher` define a
wide variety of _operators_ that you
compose to create sophisticated event-
processing chains.
/// Each operator returns a type that
implements the ``Publisher`` protocol
/// Most of these types exist as
extensions on the ``Publishers``
enumeration.
/// For example, the
``Publisher/map(_:)-99evh`` operator
```

```
returns an instance of
``Publishers/Map``.
///
/// > Tip: A Combine publisher fills a
role similar to, but distinct from, the
///
<doc://com.apple.documentation/documentat
ion/Swift/AsyncSequence> in the
/// Swift standard library. A `Publisher`
and an
/// `AsyncSequence` both produce elements
over time. However, the pull model in
Combine
/// uses a ``Combine/Subscriber`` to
request elements from a publisher, while
Swift
/// concurrency uses the `for`-`await`-
`in` syntax to iterate over elements
/// published by an `AsyncSequence`. Both
APIs offer methods to modify the sequence
/// by mapping or filtering elements,
while only Combine provides time-based
/// operations like
///
``Publisher/debounce(for:scheduler:option
s:)`` and
///
``Publisher/throttle(for:scheduler:latest
:)``, and combining operations like
/// ``Publisher/merge(with:)-7fk3a`` and
``Publisher/combineLatest(_:_:)-1n30g``.
/// To bridge the two approaches, the
property ``Publisher/values-1dm9r``
```

```
exposes
/// a publisher's elements as an
`AsyncSequence`, allowing you to iterate
over
/// them with `for`-`await`-`in` rather
than attaching a ``Subscriber``.
///
/// # Creating Your Own Publishers
///
/// Rather than implementing the
`Publisher` protocol yourself, you can
create your own publisher by using one of
several types provided by the Combine
framework:
///
/// - Use a concrete subclass of
``Subject``, such as
``PassthroughSubject``, to publish values
on-demand by calling its
``Subject/send(_:)`` method.
/// - Use a ``CurrentValueSubject`` to
publish whenever you update the subject's
underlying value.
/// - Add the `@Published` annotation to
a property of one of your own types. In
doing so, the property gains a publisher
that emits an event whenever the
property's value changes. See the
``Published`` type for an example of this
approach.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol Publisher<Output,
```

```
Failure> {

    /// The kind of values published by
this publisher.
    associatedtype Output

    /// The kind of errors this publisher
might publish.
    ///
    /// Use `Never` if this `Publisher`
does not publish errors.
    associatedtype Failure : Error

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
    func receive<S>(subscriber: S) where
S : Subscriber, Self.Failure ==
S.Failure, Self.Output == S.Input
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Publisher {

    /// Applies a closure to create a
subject that delivers elements to
subscribers.
    ///
    /// Use a multicast publisher when
you have multiple downstream subscribers,
but you want upstream publishers to only
process one ``Subscriber/receive(_:)``
call per event. This is useful when
upstream publishers are doing expensive
work you don’t want to duplicate, like
performing network requests.
    ///
    /// In contrast with
``Publisher/multicast(subject:)``, this
method produces a publisher that creates
a separate ``Subject`` for each
subscriber.
    ///
    /// The following example uses a
sequence publisher as a counter to
publish three random numbers, generated
by a ``Publisher/map(_:)-99evh``
operator.
    /// It uses a
``Publisher/multicast(_:)`` operator
whose closure creates a
``PassthroughSubject`` to share the same
random number to each of two subscribers.
Because the multicast publisher is a
```

```
``ConnectablePublisher``, publishing only
begins after a call to
``ConnectablePublisher/connect()``.
    ///
    ///     let pub = ["First", "Second",
"Third"].publisher
    ///         .map( { return ($0,
Int.random(in: 0...100)) } )
    ///         .print("Random")
    ///         .multicast
{ PassthroughSubject<(String, Int),
Never>() }
    ///
    ///     cancellable1 = pub
    ///        .sink { print ("Stream 1
received: \($0)")}
    ///     cancellable2 = pub
    ///        .sink { print ("Stream 2
received: \($0)")}
    ///     pub.connect()
    ///
    ///     // Prints:
    ///     // Random: receive value:
(("First", 9))
    ///     // Stream 2 received:
("First", 9)
    ///     // Stream 1 received:
("First", 9)
    ///     // Random: receive value:
(("Second", 46))
    ///     // Stream 2 received:
("Second", 46)
    ///     // Stream 1 received:
```

```
("Second", 46)
    ///     // Random: receive value:
(("Third", 26))
    ///     // Stream 2 received:
("Third", 26)
    ///     // Stream 1 received:
("Third", 26)
    ///
    /// In this example, the output shows
that the ``Publisher/print(_:to:)``
operator receives each random value only
one time, and then sends the value to
both subscribers.
    ///
    /// - Parameter createSubject: A
closure to create a new ``Subject`` each
time a subscriber attaches to the
multicast publisher.
    public func multicast<S>(_
createSubject: @escaping () -> S) ->
Publishers.Multicast<Self, S> where S :
Subject, Self.Failure == S.Failure,
Self.Output == S.Output

    /// Provides a subject to deliver
elements to multiple subscribers.
    ///
    /// Use a multicast publisher when
you have multiple downstream subscribers,
but you want upstream publishers to only
process one ``Subscriber/receive(_:)``
call per event. This is useful when
upstream publishers are doing expensive
```

```
work you don’t want to duplicate, like
performing network requests.
    ///
    /// In contrast with
``Publisher/multicast(_:)``, this method
produces a publisher that shares the
provided ``Subject`` among all the
downstream subscribers.
    ///
    /// The following example uses a
sequence publisher as a counter to
publish three random numbers, generated
by a ``Publisher/map(_:)-99evh``
operator.
    /// It uses a
``Publisher/multicast(subject:)``
operator with a ``PassthroughSubject`` to
share the same random number to each of
two subscribers. Because the multicast
publisher is a ``ConnectablePublisher``,
publishing only begins after a call to
``ConnectablePublisher/connect()``.
    ///
    ///     let pub = ["First", "Second",
"Third"].publisher
    ///         .map( { return ($0,
Int.random(in: 0...100)) } )
    ///         .print("Random")
    ///         .multicast(subject:
PassthroughSubject<(String, Int),
Never>())
    ///
    ///     cancellable1 = pub
```

```
    ///          .sink { print ("Stream 1
received: \($0)")}
    ///      cancellable2 = pub
    ///          .sink { print ("Stream 2
received: \($0)")}
    ///      pub.connect()
    ///
    ///      // Prints:
    ///      // Random: receive value:
(("First", 78))
    ///      // Stream 2 received:
("First", 78)
    ///      // Stream 1 received:
("First", 78)
    ///      // Random: receive value:
(("Second", 98))
    ///      // Stream 2 received:
("Second", 98)
    ///      // Stream 1 received:
("Second", 98)
    ///      // Random: receive value:
(("Third", 61))
    ///      // Stream 2 received:
("Third", 61)
    ///      // Stream 1 received:
("Third", 61)
    ///
    /// In this example, the output shows
that the ``Publisher/print(_:to:)``
operator receives each random value only
one time, and then sends the value to
both subscribers.
    ///
```

```
    /// - Parameter subject: A subject to
deliver elements to downstream
subscribers.
    public func multicast<S>(subject: S)
-> Publishers.Multicast<Self, S> where
S : Subject, Self.Failure == S.Failure,
Self.Output == S.Output
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Specifies the scheduler on which
to perform subscribe, cancel, and request
operations.
    ///
    /// In contrast with
``Publisher/receive(on:options:)``, which
affects downstream messages,
``Publisher/subscribe(on:options:)``
changes the execution context of upstream
messages.
    ///
    /// In the following example, the
``Publisher/subscribe(on:options:)``
operator causes `ioPerformingPublisher`
to receive requests on `backgroundQueue`,
while the
``Publisher/receive(on:options:)`` causes
`uiUpdatingSubscriber` to receive
elements and completion on
`RunLoop.main`.
```

```
    ///
    ///     let ioPerformingPublisher
== // Some publisher.
    ///     let uiUpdatingSubscriber
== // Some subscriber that updates the
UI.
    ///
    ///     ioPerformingPublisher
    ///         .subscribe(on:
backgroundQueue)
    ///         .receive(on:
RunLoop.main)
    ///         .subscribe(uiUpdatingSubs
criber)
    ///
    ///
    /// Using
``Publisher/subscribe(on:options:)`` also
causes the upstream publisher to perform
``Cancellable/cancel()`` using the
specfied scheduler.
    ///
    /// - Parameters:
    ///   - scheduler: The scheduler used
to send messages to upstream publishers.
    ///   - options: Options that
customize the delivery of elements.
    /// - Returns: A publisher which
performs upstream operations on the
specified scheduler.
    public func subscribe<S>(on
scheduler: S, options:
S.SchedulerOptions? = nil) ->
```

```
Publishers.SubscribeOn<Self, S> where S :
Scheduler
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Measures and emits the time
interval between events received from an
upstream publisher.
    ///
    /// Use
``Publisher/measureInterval(using:options
:)`` to measure the time between events
delivered from an upstream publisher.
    ///
    /// In the example below, a 1-second
<doc://com.apple.documentation/documentat
ion/Foundation/Timer> is used as the data
source for an event publisher; the
``Publisher/measureInterval(using:options
:)`` operator reports the elapsed time
between the reception of events on the
main run loop:
    ///
    ///     cancellable =
Timer.publish(every: 1, on: .main,
in: .default)
    ///         .autoconnect()
    ///         .measureInterval(using:
RunLoop.main)
    ///         .sink { print("\($0)",
```

```
terminator: "\n") }
    ///
    ///     // Prints:
    ///     //      Stride(magnitude:
1.001361012458801 3)
    ///     //      Stride(magnitude:
0.999276041984558 1)
    ///
    /// The output type of the returned
publisher is the time interval of the
provided scheduler.
    ///
    /// This operator uses the provided
scheduler's ``Scheduler/now`` property to
measure intervals between events.
    ///
    /// - Parameters:
    ///   - scheduler: A scheduler to use
for tracking the timing of events.
    ///   - options: Options that
customize the delivery of elements.
    /// - Returns: A publisher that emits
elements representing the time interval
between the elements it receives.
    public func measureInterval<S>(using
scheduler: S, options:
S.SchedulerOptions? = nil) ->
Publishers.MeasureInterval<Self, S> where
S : Scheduler
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
```

```
extension Publisher {

    /// Omits elements from the upstream
publisher until a given closure returns
false, before republishing all remaining
elements.
    ///
    /// Use ``Publisher/drop(while:)`` to
omit elements from an upstream publisher
until the element received meets a
condition you specify.
    ///
    /// In the example below, the
operator omits all elements in the stream
until the first element arrives that’s a
positive integer, after which the
operator publishes all remaining
elements:
    ///
    ///     let numbers = [-62, -1, 0,
10, 0, 22, 41, -1, 5]
    ///     cancellable =
numbers.publisher
    ///         .drop { $0 <= 0 }
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "10 0, 22 41 -1 5"
    ///
    ///
    /// - Parameter predicate: A closure
that takes an element as a parameter and
returns a Boolean value indicating
whether to drop the element from the
```

```
publisher's output.
    /// - Returns: A publisher that skips
over elements until the provided closure
returns `false`.
    public func drop(while predicate:
@escaping (Self.Output) -> Bool) ->
Publishers.DropWhile<Self>

    /// Omits elements from the upstream
publisher until an error-throwing closure
returns false, before republishing all
remaining elements.
    ///
    /// Use ``Publisher/tryDrop(while:)``
to omit elements from an upstream until
an error-throwing closure you provide
returns false, after which the remaining
items in the stream are published. If the
closure throws, no elements are emitted
and the publisher fails with an error.
    ///
    /// In the example below, elements
are ignored until `-1` is encountered in
the stream and the closure returns
`false`. The publisher then republishes
the remaining elements and finishes
normally. Conversely, if the `guard`
value in the closure had been
encountered, the closure would throw and
the publisher would fail with an error.
    ///
    ///     struct RangeError: Error {}
    ///     var numbers = [1, 2, 3, 4, 5,
```

```
6, -1, 7, 8, 9, 10]
    ///     let range:
CountableClosedRange<Int> = (1...100)
    ///     cancellable =
numbers.publisher
    ///         .tryDrop {
    ///             guard $0 != 0 else
{ throw RangeError() }
    ///             return
range.contains($0)
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)") },
    ///             receiveValue: { print
("value: \($0)") }
    ///         )
    ///
    ///     // Prints: "-1 7 8 9 10
completion: finished"
    ///     // If instead numbers was [1,
2, 3, 4, 5, 6, 0, -1, 7, 8, 9, 10],
tryDrop(while:) would fail with a
RangeError.
    ///
    /// - Parameter predicate: A closure
that takes an element as a parameter and
returns a Boolean value indicating
whether to drop the element from the
publisher’s output.
    /// - Returns: A publisher that skips
over elements until the provided closure
returns `false`, and then republishes all
```

```
remaining elements. If the predicate
closure throws, the publisher fails with
an error.
    public func tryDrop(while predicate:
@escaping (Self.Output) throws -> Bool)
-> Publishers.TryDropWhile<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Republishes all elements that
match a provided closure.
    ///
    /// Combine's
``Publisher/filter(_:)`` operator
performs an operation similar to that of
<doc://com.apple.documentation/documentat
ion/Swift/Sequence/filter(_:)-5y9d2> in
the Swift Standard Library: it uses a
closure to test each element to determine
whether to republish the element to the
downstream subscriber.
    ///
    /// The following example, uses a
filter operation that receives an `Int`
and only republishes a value if it's
even.
    ///
    ///     let numbers: [Int] = [1, 2,
3, 4, 5]
    ///     cancellable =
```

```
numbers.publisher
    ///          .filter { $0 % 2 == 0 }
    ///          .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "2 4"
    ///
    /// - Parameter isIncluded: A closure
that takes one element and returns a
Boolean value indicating whether to
republish the element.
    /// - Returns: A publisher that
republishes all elements that satisfy the
closure.
    public func filter(_ isIncluded:
@escaping (Self.Output) -> Bool) ->
Publishers.Filter<Self>

    /// Republishes all elements that
match a provided error-throwing closure.
    ///
    /// Use ``Publisher/tryFilter(_:)``
to filter elements evaluated in an error-
throwing closure. If the `isIncluded`
closure throws an error, the publisher
fails with that error.
    ///
    /// In the example below,
``Publisher/tryFilter(_:)`` checks to see
if the element provided by the publisher
is zero, and throws a `ZeroError` before
terminating the publisher with the thrown
error. Otherwise, it republishes the
```

```
element only if it's even:
    ///
    ///     struct ZeroError: Error {}
    ///
    ///     let numbers: [Int] = [1, 2,
3, 4, 0, 5]
    ///     cancellable =
numbers.publisher
    ///           .tryFilter{
    ///               if $0 == 0 {
    ///                   throw ZeroError()
    ///               } else {
    ///                   return $0 % 2 ==
0
    ///               }
    ///           }
    ///           .sink(
    ///               receiveCompletion:
{ print ("\($0)") },
    ///               receiveValue: { print
("\($0)", terminator: " ") }
    ///            )
    ///
    ///     // Prints: "2 4
failure(DivisionByZeroError())".
    ///
    /// - Parameter isIncluded: A closure
that takes one element and returns a
Boolean value that indicated whether to
republish the element or throws an error.
    /// - Returns: A publisher that
republishes all elements that satisfy the
closure.
```

```
    public func tryFilter(_ isIncluded:
@escaping (Self.Output) throws -> Bool)
-> Publishers.TryFilter<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Raises a debugger signal when a
provided closure needs to stop the
process in the debugger.
    ///
    /// Use
``Publisher/breakpoint(receiveSubscriptio
n:receiveOutput:receiveCompletion:)`` to
examine one or more stages of the
subscribe/publish/completion process and
stop in the debugger, based on conditions
you specify. When any of the provided
closures returns `true`, this operator
raises the `SIGTRAP` signal to stop the
process in the debugger. Otherwise, this
publisher passes through values and
completions as-is.
    ///
    /// In the example below, a
``PassthroughSubject`` publishes strings
to a breakpoint republisher. When the
breakpoint receives the string
“`DEBUGGER`”, it returns `true`, which
stops the app in the debugger.
    ///
```

```
    ///     let publisher =
PassthroughSubject<String?, Never>()
    ///     cancellable = publisher
    ///         .breakpoint(
    ///             receiveOutput:
{ value in return value == "DEBUGGER" }
    ///         )
    ///         .sink { print("\
(String(describing: $0))" , terminator: "
") }
    ///
    ///     publisher.send("DEBUGGER")
    ///
    ///     // Prints: "error: Execution
was interrupted, reason: signal SIGTRAP."
    ///     // Depending on your specific
environment, the console messages may
    ///     // also include stack trace
information, which is not shown here.
    ///
    /// - Parameters:
    ///   - receiveSubscription: A
closure that executes when the publisher
receives a subscription. Return `true`
from this closure to raise `SIGTRAP`, or
false to continue.
    ///   - receiveOutput: A closure that
executes when the publisher receives a
value. Return `true` from this closure to
raise `SIGTRAP`, or false to continue.
    ///   - receiveCompletion: A closure
that executes when the publisher receives
a completion. Return `true` from this
```

```
closure to raise `SIGTRAP`, or false to
continue.
    /// - Returns: A publisher that
raises a debugger signal when one of the
provided closures returns `true`.
    public func
breakpoint(receiveSubscription: ((any
Subscription) -> Bool)? = nil,
receiveOutput: ((Self.Output) -> Bool)? =
nil, receiveCompletion:
((Subscribers.Completion<Self.Failure>)
-> Bool)? = nil) ->
Publishers.Breakpoint<Self>

    /// Raises a debugger signal upon
receiving a failure.
    ///
    /// When the upstream publisher fails
with an error, this publisher raises the
`SIGTRAP` signal, which stops the process
in the debugger. Otherwise, this
publisher passes through values and
completions as-is.
    ///
    /// In this example a
``PassthroughSubject`` publishes strings,
but its downstream
``Publisher/tryMap(_:)`` operator throws
an error. This sends the error downstream
as a
``Subscribers/Completion/failure(_:)``.
The ``Publisher/breakpointOnError()``
operator receives this completion and
```

```
stops the app in the debugger.
    ///
    ///      struct CustomError : Error
{}
    ///      let publisher =
PassthroughSubject<String?, Error>()
    ///      cancellable = publisher
    ///          .tryMap { stringValue in
    ///              throw CustomError()
    ///          }
    ///          .breakpointOnError()
    ///          .sink(
    ///              receiveCompletion: {
completion in print("Completion: \
(String(describing: completion))") },
    ///              receiveValue:
{ aValue in print("Result: \
(String(describing: aValue))") }
    ///          )
    ///
    ///      publisher.send("TEST DATA")
    ///
    ///      // Prints: "error: Execution
was interrupted, reason: signal SIGTRAP."
    ///      // Depending on your
specific environment, the console
messages may
    ///      // also include stack trace
information, which is not shown here.
    ///
    /// - Returns: A publisher that
raises a debugger signal upon receiving a
failure.
```

```
    public func breakpointOnError() ->
Publishers.Breakpoint<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes a single Boolean value
that indicates whether all received
elements pass a given predicate.
    ///
    /// Use the
``Publisher/allSatisfy(_:)`` operator to
determine if all elements in a stream
satisfy a criteria you provide. When this
publisher receives an element, it runs
the predicate against the element. If the
predicate returns `false`, the publisher
produces a `false` value and finishes. If
the upstream publisher finishes normally,
this publisher produces a `true` value
and finishes.
    ///
    /// In the example below, the
``Publisher/allSatisfy(_:)`` operator
tests if each an integer array
publisher's elements fall into the
`targetRange`:
    ///
    ///     let targetRange = (-1...100)
    ///     let numbers = [-1, 0, 10, 5]
    ///     numbers.publisher
```

```
    ///         .allSatisfy
{ targetRange.contains($0) }
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "true"
    ///
    /// With operators similar to
``Publisher/reduce(_:_:)``, this
publisher produces at most one value.
    ///
    ///  > Note: Upon receiving any
request greater than zero, this publisher
requests unlimited elements from the
upstream publisher.
    ///
    /// - Parameter predicate: A closure
that evaluates each received element.
Return `true` to continue, or `false` to
cancel the upstream and complete.
    /// - Returns: A publisher that
publishes a Boolean value that indicates
whether all received elements pass a
given predicate.
    public func allSatisfy(_ predicate:
@escaping (Self.Output) -> Bool) ->
Publishers.AllSatisfy<Self>

    /// Publishes a single Boolean value
that indicates whether all received
elements pass a given error-throwing
predicate.
    ///
    /// Use the
```

```
``Publisher/tryAllSatisfy(_:)`` operator
to determine if all elements in a stream
satisfy a criteria in an error-throwing
predicate you provide. When this
publisher receives an element, it runs
the predicate against the element. If the
predicate returns `false`, the publisher
produces a `false` value and finishes. If
the upstream publisher finishes normally,
this publisher produces a `true` value
and finishes. If the predicate throws an
error, the publisher fails and passes the
error to its downstream subscriber.
    ///
    /// In the example below, an error-
throwing predicate tests if each of an
integer array publisher’s elements fall
into the `targetRange`; the predicate
throws an error if an element is zero and
terminates the stream.
    ///
    ///     let targetRange = (-1...100)
    ///     let numbers = [-1, 10, 5, 0]
    ///
    ///     numbers.publisher
    ///         .tryAllSatisfy { anInt in
    ///             guard anInt != 0 else
{ throw RangeError() }
    ///             return
targetRange.contains(anInt)
    ///         }
    ///         .sink(
    ///             receiveCompletion:
```

```
{ print ("completion: \($0)") },
    ///             receiveValue: { print
("value: \($0)") }
    ///         )
    ///
    ///     // Prints: "completion:
failure(RangeError())"
    ///
    /// With operators similar to
``Publisher/reduce(_:_:)``, this
publisher produces at most one value.
    ///
    ///  > Note: Upon receiving any
request greater than zero, this publisher
requests unlimited elements from the
upstream publisher.
    ///
    /// - Parameter predicate: A closure
that evaluates each received element.
Return `true` to continue, or `false` to
cancel the upstream and complete. The
closure may throw an error, in which case
the publisher cancels the upstream
publisher and fails with the thrown
error.
    /// - Returns: A publisher that
publishes a Boolean value that indicates
whether all received elements pass a
given predicate.
    public func tryAllSatisfy(_
predicate: @escaping (Self.Output) throws
-> Bool) ->
Publishers.TryAllSatisfy<Self>
```

```
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Attaches a subscriber with
closure-based behavior.
    ///
    /// Use
``Publisher/sink(receiveCompletion:receiv
eValue:)`` to observe values received by
the publisher and process them using a
closure you specify.
    ///
    /// In this example, a
<doc://com.apple.documentation/documentat
ion/Swift/Range> publisher publishes
integers to a
``Publisher/sink(receiveCompletion:receiv
eValue:)`` operator's `receiveValue`
closure that prints them to the console.
Upon completion the
``Publisher/sink(receiveCompletion:receiv
eValue:)`` operator's `receiveCompletion`
closure indicates the successful
termination of the stream.
    ///
    ///     let myRange = (0...3)
    ///     cancellable =
myRange.publisher
    ///         .sink(receiveCompletion:
{ print ("completion: \($0)") },
```

```
    ///                 receiveValue:
{ print ("value: \($0)") })
    ///
    ///     // Prints:
    ///     //  value: 0
    ///     //  value: 1
    ///     //  value: 2
    ///     //  value: 3
    ///     //  completion: finished
    ///
    /// This method creates the
subscriber and immediately requests an
unlimited number of values, prior to
returning the subscriber.
    /// The return value should be held,
otherwise the stream will be canceled.
    ///
    /// - parameter receiveComplete: The
closure to execute on completion.
    /// - parameter receiveValue: The
closure to execute on receipt of a value.
    /// - Returns: A cancellable
instance, which you use when you end
assignment of the received value.
Deallocation of the result will tear down
the subscription stream.
    public func sink(receiveCompletion:
@escaping
((Subscribers.Completion<Self.Failure>)
-> Void), receiveValue: @escaping
((Self.Output) -> Void)) ->
AnyCancellable
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Failure ==
Never {

    /// Attaches a subscriber with
closure-based behavior to a publisher
that never fails.
    ///
    /// Use
``Publisher/sink(receiveValue:)`` to
observe values received by the publisher
and print them to the console. This
operator can only be used when the stream
doesn't fail, that is, when the
publisher's ``Publisher/Failure`` type is
<doc://com.apple.documentation/documentat
ion/Swift/Never>.
    ///
    /// In this example, a
<doc://com.apple.documentation/documentat
ion/Swift/Range> publisher publishes
integers to a
``Publisher/sink(receiveValue:)``
operator's
    /// `receiveValue` closure that
prints them to the console:
    ///
    ///     let integers = (0...3)
    ///     integers.publisher
    ///         .sink { print("Received \
($0)") }
```

```
    ///
    ///     // Prints:
    ///     //  Received 0
    ///     //  Received 1
    ///     //  Received 2
    ///     //  Received 3
    ///
    /// This method creates the
subscriber and immediately requests an
unlimited number of values, prior to
returning the subscriber.
    /// The return value should be held,
otherwise the stream will be canceled.
    ///
    /// - parameter receiveValue: The
closure to execute on receipt of a value.
    /// - Returns: A cancellable
instance, which you use when you end
assignment of the received value.
Deallocation of the result will tear down
the subscription stream.
    public func sink(receiveValue:
@escaping ((Self.Output) -> Void)) ->
AnyCancellable
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Output :
Equatable {

    /// Publishes only elements that
don’t match the previous element.
```

```
    ///
    /// Use
``Publisher/removeDuplicates()`` to
remove repeating elements from an
upstream publisher. This operator has a
two-element memory: the operator uses the
current and previously published elements
as the basis for its comparison.
    ///
    /// In the example below,
``Publisher/removeDuplicates()`` triggers
on the doubled, tripled, and quadrupled
occurrences of `1`, `3`, and `4`
respectively. Because the two-element
memory considers only the current element
and the previous element, the operator
prints the final `0` in the example data
since its immediate predecessor is `4`.
    ///
    ///     let numbers = [0, 1, 2, 2, 3,
3, 3, 4, 4, 4, 4, 0]
    ///     cancellable =
numbers.publisher
    ///         .removeDuplicates()
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 2 3 4 0"
    ///
    /// - Returns: A publisher that
consumes — rather than publishes
— duplicate elements.
    public func removeDuplicates() ->
```

```
Publishers.RemoveDuplicates<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes only elements that
don’t match the previous element, as
evaluated by a provided closure.
    ///
    /// Use
``Publisher/removeDuplicates(by:)`` to
remove repeating elements from an
upstream publisher based upon the
evaluation of the current and previously
published elements using a closure you
provide.
    ///
    /// Use the
``Publisher/removeDuplicates(by:)``
operator when comparing types that don’t
themselves implement `Equatable`, or if
you need to compare values differently
than the type’s `Equatable`
implementation.
    ///
    /// In the example below, the
``Publisher/removeDuplicates(by:)``
functionality triggers when the `x`
property of the current and previous
elements are equal, otherwise the
operator publishes the current `Point` to
```

```
the downstream subscriber:
    ///
    ///     struct Point {
    ///         let x: Int
    ///         let y: Int
    ///     }
    ///
    ///     let points = [Point(x: 0, y:
0), Point(x: 0, y: 1),
    ///                   Point(x: 1, y:
1), Point(x: 2, y: 1)]
    ///     cancellable =
points.publisher
    ///         .removeDuplicates { prev,
current in
    ///             // Considers points
to be duplicate if the x coordinate
    ///             // is equal, and
ignores the y coordinate
    ///             prev.x == current.x
    ///         }
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: Point(x: 0, y: 0)
Point(x: 1, y: 1) Point(x: 2, y: 1)
    ///
    /// - Parameter predicate: A closure
to evaluate whether two elements are
equivalent, for purposes of filtering.
Return `true` from this closure to
indicate that the second element is a
duplicate of the first.
```

```
    /// - Returns: A publisher that
consumes — rather than publishes
— duplicate elements.
    public func removeDuplicates(by
predicate: @escaping (Self.Output,
Self.Output) -> Bool) ->
Publishers.RemoveDuplicates<Self>

    /// Publishes only elements that
don’t match the previous element, as
evaluated by a provided error-throwing
closure.
    ///
    /// Use
``Publisher/tryRemoveDuplicates(by:)`` to
remove repeating elements from an
upstream publisher based upon the
evaluation of elements using an error-
throwing closure you provide. If your
closure throws an error, the publisher
terminates with the error.
    ///
    /// In the example below, the closure
provided to
``Publisher/tryRemoveDuplicates(by:)``
returns `true` when two consecutive
elements are equal, thereby filtering out
`0`,
    /// `1`, `2`, and `3`. However, the
closure throws an error when it
encounters `4`. The publisher then
terminates with this error.
    ///
```

```
    ///     struct BadValuesError: Error
{}
    ///     let numbers = [0, 0, 0, 0, 1,
2, 2, 3, 3, 3, 4, 4, 4, 4]
    ///     cancellable =
numbers.publisher
    ///         .tryRemoveDuplicates
{ first, second -> Bool in
    ///             if (first == 4 &&
second == 4) {
    ///                 throw
BadValuesError()
    ///             }
    ///             return first ==
second
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("\($0)") },
    ///             receiveValue: { print
("\($0)", terminator: " ") }
    ///          )
    ///
    ///      // Prints: "0 1 2 3 4
failure(BadValuesError()"
    ///
    /// - Parameter predicate: A closure
to evaluate whether two elements are
equivalent, for purposes of filtering.
Return `true` from this closure to
indicate that the second element is a
duplicate of the first. If this closure
throws an error, the publisher terminates
```

```
with the thrown error.
    /// - Returns: A publisher that
consumes — rather than publishes
— duplicate elements.
    public func tryRemoveDuplicates(by
predicate: @escaping (Self.Output,
Self.Output) throws -> Bool) ->
Publishers.TryRemoveDuplicates<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Decodes the output from the
upstream using a specified decoder.
    ///
    /// Use
``Publisher/decode(type:decoder:)`` with
a
<doc://com.apple.documentation/documentat
ion/Foundation/JSONDecoder> (or a
<doc://com.apple.documentation/documentat
ion/Foundation/PropertyListDecoder> for
property lists) to decode data received
from a
<doc://com.apple.documentation/documentat
ion/Foundation/URLSession/
DataTaskPublisher> or other data source
using the
<doc://com.apple.documentation/documentat
ion/Swift/Decodable> protocol.
    ///
```

```
    /// In this example, a
``PassthroughSubject`` publishes a JSON
string. The JSON decoder parses the
string, converting its fields according
to the
<doc://com.apple.documentation/documentat
ion/Swift/Decodable> protocol implemented
by `Article`, and successfully populating
a new `Article`. The
``Publishers/Decode`` publisher then
publishes the `Article` to the
downstream. If a decoding operation
fails, which happens in the case of
missing or malformed data in the source
JSON string, the stream terminates and
passes the error to the downstream
subscriber.
    ///
    ///     struct Article: Codable {
    ///         let title: String
    ///         let author: String
    ///         let pubDate: Date
    ///     }
    ///
    ///     let dataProvider =
PassthroughSubject<Data, Never>()
    ///     cancellable = dataProvider
    ///         .decode(type:
Article.self, decoder: JSONDecoder())
    ///         .sink(receiveCompletion:
{ print ("Completion: \($0)")},
    ///               receiveValue:
{ print ("value: \($0)") })
```

```
    ///
    ///
dataProvider.send(Data("{\"pubDate\":1574
273638.575666, \"title\" : \"My First
Article\", \"author\" : \"Gita
Kumar\" }".utf8))
    ///
    ///     // Prints: ".sink() data
received Article(title: "My First
Article", author: "Gita Kumar", pubDate:
2050-11-20 18:13:58 +0000)"
    ///
    /// - Parameters:
    ///   - type: The encoded data to
decode into a struct that conforms to the
<doc://com.apple.documentation/documentat
ion/Swift/Decodable> protocol.
    ///   - decoder:  A decoder that
implements the ``TopLevelDecoder``
protocol.
    /// - Returns: A publisher that
decodes a given type using a specified
decoder and publishes the result.
    public func decode<Item, Coder>(type:
Item.Type, decoder: Coder) ->
Publishers.Decode<Self, Item, Coder>
where Item : Decodable, Coder :
TopLevelDecoder, Self.Output ==
Coder.Input
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
```

```
extension Publisher where Self.Output :
Encodable {

    /// Encodes the output from upstream
using a specified encoder.
    ///
    /// Use
``Publisher/encode(encoder:)`` with a
<doc://com.apple.documentation/documentat
ion/Foundation/JSONDecoder> (or a
<doc://com.apple.documentation/documentat
ion/Foundation/PropertyListDecoder> for
property lists) to encode an
<doc://com.apple.documentation/documentat
ion/Swift/Encodable> struct into
<doc://com.apple.documentation/documentat
ion/Foundation/Data> that could be used
to make a JSON string (or written to disk
as a binary plist in the case of property
lists).
    ///
    /// In this example, a
``PassthroughSubject`` publishes an
`Article`. The
``Publisher/encode(encoder:)`` operator
encodes the properties of the `Article`
struct into a new JSON string according
to the
<doc://com.apple.documentation/documentat
ion/Swift/Codable> protocol adopted by
`Article`. The operator publishes the
resulting JSON string to the downstream
subscriber. If the encoding operation
```

```
fails, which can happen in the case of
complex properties that can’t be directly
transformed into JSON, the stream
terminates and the error is passed to the
downstream subscriber.
    ///
    ///     struct Article: Codable {
    ///         let title: String
    ///         let author: String
    ///         let pubDate: Date
    ///     }
    ///
    ///     let dataProvider =
PassthroughSubject<Article, Never>()
    ///     let cancellable =
dataProvider
    ///         .encode(encoder:
JSONEncoder())
    ///         .sink(receiveCompletion:
{ print ("Completion: \($0)") },
    ///               receiveValue:
{  data in
    ///                 guard let
stringRepresentation = String(data: data,
encoding: .utf8) else { return }
    ///                 print("Data
received \(data) string representation: \
(stringRepresentation)")
    ///         })
    ///
    ///
dataProvider.send(Article(title: "My
First Article", author: "Gita Kumar",
```

```
pubDate: Date()))
    ///
    ///     // Prints: "Data received 86
bytes string representation: {"title":"My
First Article","author":"Gita
Kumar","pubDate":606211803.279603}"
    ///
    /// - Parameter encoder: An encoder
that implements the ``TopLevelEncoder``
protocol.
    /// - Returns: A publisher that
encodes received elements using a
specified encoder, and publishes the
resulting data.
    public func encode<Coder>(encoder:
Coder) -> Publishers.Encode<Self, Coder>
where Coder : TopLevelEncoder
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Output :
Equatable {

    /// Publishes a Boolean value upon
receiving an element equal to the
argument.
    ///
    /// Use ``Publisher/contains(_:)`` to
find the first element in an upstream
that’s equal to the supplied argument.
The contains publisher consumes all
received elements until the upstream
```

```
publisher produces a matching element.
Upon finding the first match, it emits
`true` and finishes normally. If the
upstream finishes normally without
producing a matching element, this
publisher emits `false` and finishes.
    ///
    /// In the example below, the
``Publisher/contains(_:)`` operator emits
`true` the first time it receives the
value `5` from the `numbers.publisher`,
and then finishes normally.
    ///
    ///     let numbers = [-1, 5, 10, 5]
    ///     numbers.publisher
    ///         .contains(5)
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "true"
    ///
    /// - Parameter output: An element to
match against.
    /// - Returns: A publisher that emits
the Boolean value `true` when the
upstream publisher emits a matching
value.
    public func contains(_ output:
Self.Output) -> Publishers.Contains<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {
```

```
    /// Subscribes to an additional
publisher and publishes a tuple upon
receiving output from either publisher.
    ///
    /// Use
``Publisher/combineLatest(_:)`` when you
want the downstream subscriber to receive
a tuple of the most-recent element from
multiple publishers when any of them emit
a value. To pair elements from multiple
publishers, use ``Publisher/zip(_:)``
instead. To receive just the most-recent
element from multiple publishers rather
than tuples, use
``Publisher/merge(with:)-7qt71``.
    ///
    /// > Tip: The combined publisher
doesn't produce elements until each of
its upstream publishers publishes at
least one element.
    ///
    /// The combined publisher passes
through any requests to *all* upstream
publishers. However, it still obeys the
demand-fulfilling rule of only sending
the request amount downstream. If the
demand isn't
``Subscribers/Demand/unlimited``, it
drops values from upstream publishers. It
implements this by using a buffer size of
1 for each upstream, and holds the most-
recent value in each buffer.
```

```
    ///
    /// In this example, ``PassthroughSubject`` `pub1` and also `pub2` emit values; as ``Publisher/combineLatest(_:)`` receives input from either upstream publisher, it combines the latest value from each publisher into a tuple and publishes it.
    ///
    ///     let pub1 = PassthroughSubject<Int, Never>()
    ///     let pub2 = PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pub1
    ///         .combineLatest(pub2)
    ///         .sink { print("Result: \($0).") }
    ///
    ///     pub1.send(1)
    ///     pub1.send(2)
    ///     pub2.send(2)
    ///     pub1.send(3)
    ///     pub1.send(45)
    ///     pub2.send(22)
    ///
    ///     // Prints:
    ///     //    Result: (2, 2).    // pub1 latest = 2, pub2 latest = 2
    ///     //    Result: (3, 2).    // pub1 latest = 3, pub2 latest = 2
    ///     //    Result: (45, 2).   // pub1 latest = 45, pub2 latest = 2
```

```
    ///     //    Result: (45, 22).  //
pub1 latest = 45, pub2 latest = 22
    ///
    /// When all upstream publishers
finish, this publisher finishes. If an
upstream publisher never publishes a
value, this publisher never finishes.
    ///
    /// - Parameter other: Another
publisher to combine with this one.
    /// - Returns: A publisher that
receives and combines elements from this
and another publisher.
    public func combineLatest<P>(_ other:
P) -> Publishers.CombineLatest<Self, P>
where P : Publisher, Self.Failure ==
P.Failure

    /// Subscribes to an additional
publisher and invokes a closure upon
receiving output from either publisher.
    ///
    /// Use `combineLatest<P,T>(_:)` to
combine the current and one additional
publisher and transform them using a
closure you specify to publish a new
value to the downstream.
    ///
    /// > Tip: The combined publisher
doesn't produce elements until each of
its upstream publishers publishes at
least one element.
    ///
```

```
    /// The combined publisher passes
through any requests to *all* upstream
publishers. However, it still obeys the
demand-fulfilling rule of only sending
the request amount downstream. If the
demand isn't `.unlimited`, it drops
values from upstream publishers. It
implements this by using a buffer size of
1 for each upstream, and holds the most-
recent value in each buffer.
    ///
    /// In the example below,
`combineLatest()` receives the most-
recent values published by the two
publishers, it multiplies them together,
and republishes the result:
    ///
    ///     let pub1 =
PassthroughSubject<Int, Never>()
    ///     let pub2 =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pub1
    ///         .combineLatest(pub2)
{ (first, second) in
    ///             return first * second
    ///         }
    ///         .sink { print("Result: \
($0)."") }
    ///
    ///     pub1.send(1)
    ///     pub1.send(2)
    ///     pub2.send(2)
```

```
    ///     pub1.send(9)
    ///     pub1.send(3)
    ///     pub2.send(12)
    ///     pub1.send(13)
    ///     //
    ///     // Prints:
    ///     //Result: 4.    (pub1 latest
= 2, pub2 latest = 2)
    ///     //Result: 18.   (pub1 latest
= 9, pub2 latest = 2)
    ///     //Result: 6.    (pub1 latest
= 3, pub2 latest = 2)
    ///     //Result: 36.   (pub1 latest
= 3, pub2 latest = 12)
    ///     //Result: 156.  (pub1 latest
= 13, pub2 latest = 12)
    ///
    /// All upstream publishers need to
finish for this publisher to finish. If
an upstream publisher never publishes a
value, this publisher never finishes.
    /// If any of the combined publishers
terminates with a failure, this publisher
also fails.
    ///
    /// - Parameters:
    ///   - other: Another publisher to
combine with this one.
    ///   - transform: A closure that
receives the most-recent value from each
publisher and returns a new value to
publish.
    /// - Returns: A publisher that
```

```
receives and combines elements from this
and another publisher.
    public func combineLatest<P, T>(_
other: P, _ transform: @escaping
(Self.Output, P.Output) -> T) ->
Publishers.Map<Publishers.CombineLatest<S
elf, P>, T> where P : Publisher,
Self.Failure == P.Failure

    /// Subscribes to two additional
publishers and publishes a tuple upon
receiving output from any of the
publishers.
    ///
    /// Use
``Publisher/combineLatest(_:_:)-5crqg``
when you want the downstream subscriber
to receive a tuple of the most-recent
element from multiple publishers when any
of them emit a value. To combine elements
from multiple publishers, use
``Publisher/zip(_:_:)-8d7k7`` instead. To
receive just the most-recent element from
multiple publishers rather than tuples,
use ``Publisher/merge(with:_:)``.
    ///
    /// > Tip: The combined publisher
doesn't produce elements until each of
its upstream publishers publishes at
least one element.
    ///
    /// The combined publisher passes
through any requests to *all* upstream
```

```
publishers. However, it still obeys the
demand-fulfilling rule of only sending
the request amount downstream. If the
demand isn't
``Subscribers/Demand/unlimited``, it
drops values from upstream publishers. It
implements this by using a buffer size of
1 for each upstream, and holds the most-
recent value in each buffer.
    ///
    /// All upstream publishers need to
finish for this publisher to finish. If
an upstream publisher never publishes a
value, this publisher never finishes.
    ///
    /// In this example, three instances
of ``PassthroughSubject`` emit values; as
``Publisher/combineLatest(_:_:)-5crqg``
receives input from any of the upstream
publishers, it combines the latest value
from each publisher into a tuple and
publishes it:
    ///
    ///     let pub =
PassthroughSubject<Int, Never>()
    ///     let pub2 =
PassthroughSubject<Int, Never>()
    ///     let pub3 =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pub
    ///         .combineLatest(pub2,
pub3)
```

```
    ///         .sink { print("Result: \
($0).") }
    ///
    ///     pub.send(1)
    ///     pub.send(2)
    ///     pub2.send(2)
    ///     pub3.send(9)
    ///
    ///     pub.send(3)
    ///     pub2.send(12)
    ///     pub.send(13)
    ///     pub3.send(19)
    ///
    ///     // Prints:
    ///     //  Result: (2, 2, 9).
    ///     //  Result: (3, 2, 9).
    ///     //  Result: (3, 12, 9).
    ///     //  Result: (13, 12, 9).
    ///     //  Result: (13, 12, 19).
    ///
    /// If any of the combined publishers
terminates with a failure, this publisher
also fails.
    /// - Parameters:
    ///   - publisher1: A second
publisher to combine with the first
publisher.
    ///   - publisher2: A third publisher
to combine with the first publisher.
    /// - Returns: A publisher that
receives and combines elements from this
publisher and two other publishers.
    public func combineLatest<P, Q>(_
```

```
publisher1: P, _ publisher2: Q) ->
Publishers.CombineLatest3<Self, P, Q>
where P : Publisher, Q : Publisher,
Self.Failure == P.Failure, P.Failure ==
Q.Failure

    /// Subscribes to two additional
publishers and invokes a closure upon
receiving output from any of the
publishers.
    ///
    /// Use `combineLatest<P, Q>(_:,_:)`
to combine the current and two additional
publishers and transform them using a
closure you specify to publish a new
value to the downstream.
    ///
    /// > Tip: The combined publisher
doesn't produce elements until each of
its upstream publishers publishes at
least one element.
    ///
    /// The combined publisher passes
through any requests to *all* upstream
publishers. However, it still obeys the
demand-fulfilling rule of only sending
the request amount downstream. If the
demand isn't `.unlimited`, it drops
values from upstream publishers. It
implements this by using a buffer size of
1 for each upstream, and holds the most-
recent value in each buffer.
    /// All upstream publishers need to
```

```
finish for this publisher to finish. If
an upstream publisher never publishes a
value, this publisher never finishes.
    /// If any of the combined publishers
terminates with a failure, this publisher
also fails.
    ///
    /// In the example below,
`combineLatest()` receives the most-
recent values published by three
publishers, multiplies them together, and
republishes the result:
    ///
    ///     let pub =
PassthroughSubject<Int, Never>()
    ///     let pub2 =
PassthroughSubject<Int, Never>()
    ///     let pub3 =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pub
    ///         .combineLatest(pub2,
pub3) { firstValue, secondValue,
thirdValue in
    ///             return firstValue *
secondValue * thirdValue
    ///         }
    ///         .sink { print("Result: \
($0).") }
    ///
    ///     pub.send(1)
    ///     pub.send(2)
    ///     pub2.send(2)
```

```
    ///     pub3.send(10)
    ///
    ///     pub.send(9)
    ///     pub3.send(4)
    ///     pub2.send(12)
    ///
    ///     // Prints:
    ///     //  Result: 40.     // pub =
2, pub2 = 2, pub3 = 10
    ///     //  Result: 180.    // pub =
9, pub2 = 2, pub3 = 10
    ///     //  Result: 72.     // pub =
9, pub2 = 2, pub3 = 4
    ///     //  Result: 432.    // pub =
9, pub2 = 12, pub3 = 4
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher to combine with the first
publisher.
    ///   - publisher2: A third publisher
to combine with the first publisher.
    ///   - transform: A closure that
receives the most-recent value from each
publisher and returns a new value to
publish.
    /// - Returns: A publisher that
receives and combines elements from this
publisher and two other publishers.
    public func combineLatest<P, Q, T>(_
publisher1: P, _ publisher2: Q, _
transform: @escaping (Self.Output,
P.Output, Q.Output) -> T) ->
```

```
Publishers.Map<Publishers.CombineLatest3<
Self, P, Q>, T> where P : Publisher, Q :
Publisher, Self.Failure == P.Failure,
P.Failure == Q.Failure

    /// Subscribes to three additional
publishers and publishes a tuple upon
receiving output from any of the
publishers.
    ///
    /// Use
``Publisher/combineLatest(_:_:_:)-48buc``
when you want the downstream subscriber
to receive a tuple of the most-recent
element from multiple publishers when any
of them emit a value. To combine elements
from multiple publishers, use
``Publisher/zip(_:_:_:)-16rcy`` instead.
To receive just the most-recent element
from multiple publishers rather than
tuples, use
``Publisher/merge(with:_:_:)``.
    ///
    /// > Tip: The combined publisher
doesn't produce elements until each of
its upstream publishers publishes at
least one element.
    ///
    /// The combined publisher passes
through any requests to *all* upstream
publishers. However, it still obeys the
demand-fulfilling rule of only sending
the request amount downstream. If the
```

```
demand isn’t
``Subscribers/Demand/unlimited``, it
drops values from upstream publishers. It
implements this by using a buffer size of
1 for each upstream, and holds the most-
recent value in each buffer.
    ///
    /// All upstream publishers need to
finish for this publisher to finish. If
an upstream publisher never publishes a
value, this publisher never finishes.
    ///
    /// In the example below,
``Publisher/combineLatest(_:_:_:)-48buc``
receives input from any of the
publishers, combines the latest value
from each publisher into a tuple and
publishes it:
    ///
    ///     let pub =
PassthroughSubject<Int, Never>()
    ///     let pub2 =
PassthroughSubject<Int, Never>()
    ///     let pub3 =
PassthroughSubject<Int, Never>()
    ///     let pub4 =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pub
    ///         .combineLatest(pub2,
pub3, pub4)
    ///         .sink { print("Result: \
($0).") }
```

```
    ///
    ///     pub.send(1)
    ///     pub.send(2)
    ///     pub2.send(2)
    ///     pub3.send(9)
    ///     pub4.send(1)
    ///
    ///     pub.send(3)
    ///     pub2.send(12)
    ///     pub.send(13)
    ///     pub3.send(19)
    ///     //
    ///     // Prints:
    ///     //  Result: (2, 2, 9, 1).
    ///     //  Result: (3, 2, 9, 1).
    ///     //  Result: (3, 12, 9, 1).
    ///     //  Result: (13, 12, 9, 1).
    ///     //  Result: (13, 12, 19, 1).
    ///
    /// If any individual publisher of
the combined set terminates with a
failure, this publisher also fails.
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher to combine with the first
publisher.
    ///   - publisher2: A third publisher
to combine with the first publisher.
    ///   - publisher3: A fourth
publisher to combine with the first
publisher.
    /// - Returns: A publisher that
```

```
receives and combines elements from this
publisher and three other publishers.
    public func combineLatest<P, Q, R>(_
publisher1: P, _ publisher2: Q, _
publisher3: R) ->
Publishers.CombineLatest4<Self, P, Q, R>
where P : Publisher, Q : Publisher, R :
Publisher, Self.Failure == P.Failure,
P.Failure == Q.Failure, Q.Failure ==
R.Failure

    /// Subscribes to three additional
publishers and invokes a closure upon
receiving output from any of the
publishers.
    ///
    /// Use
``Publisher/combineLatest(_:_:_:)``
when you need to combine the current and
3 additional publishers and transform the
values using a closure in which you
specify the published elements, to
publish a new element.
    ///
    /// > Tip: The combined publisher
doesn't produce elements until each of
its upstream publishers publishes at
least one element.
    ///
    /// The combined publisher passes
through any requests to *all* upstream
publishers. However, it still obeys the
demand-fulfilling rule of only sending
```

```
the request amount downstream. If the
demand isn’t
``Subscribers/Demand/unlimited``, it
drops values from upstream publishers. It
implements this by using a buffer size of
1 for each upstream, and holds the most-
recent value in each buffer.
    ///
    /// All upstream publishers need to
finish for this publisher to finish. If
an upstream publisher never publishes a
value, this publisher never finishes.
    ///
    /// In the example below, as
``Publisher/combineLatest(_:_:_:_:)``
receives the most-recent values published
by four publishers, multiplies them
together, and republishes the result:
    ///
    ///     let pub =
PassthroughSubject<Int, Never>()
    ///     let pub2 =
PassthroughSubject<Int, Never>()
    ///     let pub3 =
PassthroughSubject<Int, Never>()
    ///     let pub4 =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pub
    ///         .combineLatest(pub2,
pub3, pub4) { firstValue, secondValue,
thirdValue, fourthValue in
    ///             return firstValue *
```

```
secondValue * thirdValue * fourthValue
    ///         }
    ///         .sink { print("Result: \
($0).") }
    ///
    ///     pub.send(1)
    ///     pub.send(2)
    ///     pub2.send(2)
    ///     pub3.send(9)
    ///     pub4.send(1)
    ///
    ///     pub.send(3)
    ///     pub2.send(12)
    ///     pub.send(13)
    ///     pub3.send(19)
    ///
    ///     // Prints:
    ///     //  Result: 36.     // pub =
2,  pub2 = 2,   pub3 = 9,  pub4 = 1
    ///     //  Result: 54.     // pub =
3,  pub2 = 2,   pub3 = 9,  pub4 = 1
    ///     //  Result: 324.    // pub =
3,  pub2 = 12,  pub3 = 9,  pub4 = 1
    ///     //  Result: 1404.   // pub =
13, pub2 = 12,  pub3 = 9,  pub4 = 1
    ///     //  Result: 2964.   // pub =
13, pub2 = 12,  pub3 = 19, pub4 = 1
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher to combine with the first
publisher.
    ///   - publisher2: A third publisher
```

```
to combine with the first publisher.
    ///   - publisher3: A fourth
publisher to combine with the first
publisher.
    ///   - transform: A closure that
receives the most-recent value from each
publisher and returns a new value to
publish.
    /// - Returns: A publisher that
receives and combines elements from this
publisher and three other publishers.
    public func combineLatest<P, Q, R,
T>(_ publisher1: P, _ publisher2: Q, _
publisher3: R, _ transform: @escaping
(Self.Output, P.Output, Q.Output,
R.Output) -> T) ->
Publishers.Map<Publishers.CombineLatest4<
Self, P, Q, R>, T> where P : Publisher, Q
: Publisher, R : Publisher, Self.Failure
== P.Failure, P.Failure == Q.Failure,
Q.Failure == R.Failure
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Republishes elements up to the
specified maximum count.
    ///
    /// Use ``Publisher/prefix(_:)`` to
limit the number of elements republished
to the downstream subscriber.
```

```
    ///
    /// In the example below, the
``Publisher/prefix(_:)`` operator limits
its output to the first two elements
before finishing normally:
    ///
    ///     let numbers = (0...10)
    ///     cancellable =
numbers.publisher
    ///         .prefix(2)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1"
    ///
    /// - Parameter maxLength: The
maximum number of elements to republish.
    /// - Returns: A publisher that
publishes up to the specified number of
elements.
    public func prefix(_ maxLength: Int)
-> Publishers.Output<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Prints log messages for all
publishing events.
    ///
    /// Use ``Publisher/print(_:to:)`` to
log messages the console.
```

```
    ///
    /// In the example below, log
messages are printed on the console:
    ///
    ///     let integers = (1...2)
    ///     cancellable =
integers.publisher
    ///         .print("Logged a message",
to: nil)
    ///         .sink { _ in }
    ///
    ///     // Prints:
    ///     //  Logged a message: receive
subscription: (1..<2)
    ///     //  Logged a message: request
unlimited
    ///     //  Logged a message: receive
value: (1)
    ///     //  Logged a message: receive
finished
    ///
    /// - Parameters:
    ///   - prefix: A string -- which
defaults to empty -- with which to prefix
all log messages.
    ///   - stream: A stream for text
output that receives messages, and which
directs output to the console by default.
A custom stream can be used to log
messages to other destinations.
    /// - Returns: A publisher that
prints log messages for all publishing
events.
```

```
    public func print(_ prefix: String =
"", to stream: (any TextOutputStream)? =
nil) -> Publishers.Print<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Republishes elements while a
predicate closure indicates publishing
should continue.
    ///
    /// Use ``Publisher/prefix(while:)``
to emit values while elements from the
upstream publisher meet a condition you
specify. The publisher finishes when the
closure returns `false`.
    ///
    /// In the example below, the
``Publisher/prefix(while:)`` operator
emits values while the element it
receives is less than five:
    ///
    ///     let numbers = (0...10)
    ///     numbers.publisher
    ///         .prefix { $0 < 5 }
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 2 3 4"
    ///
    /// - Parameter predicate: A closure
```

```
that takes an element as its parameter
and returns a Boolean value that
indicates whether publishing should
continue.
    /// - Returns: A publisher that
passes through elements until the
predicate indicates publishing should
finish.
    public func prefix(while predicate:
@escaping (Self.Output) -> Bool) ->
Publishers.PrefixWhile<Self>

    /// Republishes elements while an
error-throwing predicate closure
indicates publishing should continue.
    ///
    /// Use
``Publisher/tryPrefix(while:)`` to emit
values from the upstream publisher that
meet a condition you specify in an error-
throwing closure.
    /// The publisher finishes when the
closure returns `false`. If the closure
throws an error, the publisher fails with
that error.
    ///
    ///     struct OutOfRangeError: Error
{}
    ///
    ///     let numbers =
(0...10).reversed()
    ///     cancellable =
numbers.publisher
```

```
    ///          .tryPrefix {
    ///              guard $0 != 0 else
{throw OutOfRangeError()}
    ///              return $0 <=
numbers.max()!
    ///          }
    ///          .sink(
    ///              receiveCompletion:
{ print ("completion: \($0)", terminator:
" ") },
    ///              receiveValue: { print
("\($0)", terminator: " ") }
    ///          )
    ///
    ///      // Prints: "10 9 8 7 6 5 4 3
2 1 completion:
failure(OutOfRangeError()) "
    ///
    /// - Parameter predicate: A closure
that takes an element as its parameter
and returns a Boolean value indicating
whether publishing should continue.
    /// - Returns: A publisher that
passes through elements until the
predicate throws or indicates publishing
should finish.
    public func tryPrefix(while
predicate: @escaping (Self.Output) throws
-> Bool) ->
Publishers.TryPrefixWhile<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Publisher where Self.Failure ==
Never {

    /// Changes the failure type declared
by the upstream publisher.
    ///
    /// Use
``Publisher/setFailureType(to:)`` when
you need set the error type of a
publisher that cannot fail.
    ///
    /// Conversely, if the upstream can
fail, you would use
``Publisher/mapError(_:)`` to provide
instructions on converting the error
types to needed by the downstream
publisher's inputs.
    ///
    /// The following example has two
publishers with mismatched error types:
`pub1`'s error type is
<doc://com.apple.documentation/documentat
ion/Swift/Never>, and `pub2`'s error type
is
<doc://com.apple.documentation/documentat
ion/Swift/Error>. Because of the
mismatch, the
``Publisher/combineLatest(_:)`` operator
requires that `pub1` use
``Publisher/setFailureType(to:)`` to make
it appear that `pub1` can produce the
<doc://com.apple.documentation/documentat
```

```
ion/Swift/Error> type, like `pub2` can.
    ///
    ///     let pub1 = [0, 1, 2, 3, 4,
5].publisher
    ///     let pub2 =
CurrentValueSubject<Int, Error>(0)
    ///     let cancellable = pub1
    ///         .setFailureType(to:
Error.self)
    ///         .combineLatest(pub2)
    ///         .sink(
    ///             receiveCompletion:
{ print ("completed: \($0)") },
    ///             receiveValue: { print
("value: \($0)")}
    ///          )
    ///
    ///     // Prints: "value: (5, 0)".
    ///
    /// - Parameter failureType: The
`Failure` type presented by this
publisher.
    /// - Returns: A publisher that
appears to send the specified failure
type.
    public func setFailureType<E>(to
failureType: E.Type) ->
Publishers.SetFailureType<Self, E> where
E : Error
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
```

```
extension Publisher {

    /// Publishes a Boolean value upon
receiving an element that satisfies the
predicate closure.
    ///
    /// Use
``Publisher/contains(where:)`` to find
the first element in an upstream that
satisfies the closure you provide. This
operator consumes elements produced from
the upstream publisher until the upstream
publisher produces a matching element.
    ///
    /// This operator is useful when the
upstream publisher produces elements that
don’t conform to `Equatable`.
    ///
    /// In the example below, the
``Publisher/contains(where:)`` operator
tests elements against the supplied
closure and emits `true` for the first
elements that’s greater than `4`, and
then finishes normally.
    ///
    ///     let numbers = [-1, 0, 10, 5]
    ///     numbers.publisher
    ///         .contains {$0 > 4}
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "true"
    ///
    /// - Parameter predicate: A closure
```

```
that takes an element as its parameter
and returns a Boolean value that
indicates whether the element satisfies
the closure's comparison logic.
    /// - Returns: A publisher that emits
the Boolean value `true` when the
upstream  publisher emits a matching
value.
    public func contains(where predicate:
@escaping (Self.Output) -> Bool) ->
Publishers.ContainsWhere<Self>

    /// Publishes a Boolean value upon
receiving an element that satisfies the
throwing predicate closure.
    ///
    /// Use
``Publisher/tryContains(where:)`` to find
the first element in an upstream that
satisfies the error-throwing closure you
provide.
    ///
    /// This operator consumes elements
produced from the upstream publisher
until the upstream publisher either:
    ///
    /// - Produces a matching element,
after which it emits `true` and the
publisher finishes normally.
    /// - Emits `false` if no matching
element is found and the publisher
finishes normally.
    ///
```

```
    /// If the predicate throws an error,
the publisher fails, passing the error to
its downstream.
    ///
    /// In the example below, the
``Publisher/tryContains(where:)``
operator tests values to find an element
less than `10`; when the closure finds an
odd number, like `3`, the publisher
terminates with an `IllegalValueError`.
    ///
    ///     struct IllegalValueError:
Error {}
    ///
    ///     let numbers = [3, 2, 10, 5,
0, 9]
    ///     numbers.publisher
    ///         .tryContains {
    ///             if ($0 % 2 != 0) {
    ///                 throw
IllegalValueError()
    ///             }
    ///             return $0 < 10
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)") },
    ///             receiveValue: { print
("value: \($0)") }
    ///         )
    ///
    ///     // Prints: "completion:
failure(IllegalValueError())"
```

```
    ///
    /// - Parameter predicate: A closure
that takes an element as its parameter
and returns a Boolean value that
indicates whether the element satisfies
the closure's comparison logic.
    /// - Returns: A publisher that emits
the Boolean value `true` when the
upstream publisher emits a matching
value.
    public func tryContains(where
predicate: @escaping (Self.Output) throws
-> Bool) ->
Publishers.TryContainsWhere<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Always call this function instead
of ``Publisher/receive(subscriber:)``.
    /// Adopters of ``Publisher`` must
implement
``Publisher/receive(subscriber:)``. The
implementation of
``Publisher/subscribe(_:)-4u8kn``
provided by ``Publisher`` calls through
to ``Publisher/receive(subscriber:)``.
    ///
```

```
    /// - Parameter subscriber: The
subscriber to attach to this publisher.
After attaching, the subscriber can start
to receive values.
    public func subscribe<S>(_
subscriber: S) where S : Subscriber,
Self.Failure == S.Failure, Self.Output ==
S.Input
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Failure ==
Never {

    /// Republishes elements received
from a publisher, by assigning them to a
property marked as a publisher.
    ///
    /// Use this operator when you want
to receive elements from a publisher and
republish them through a property marked
with the `@Published` attribute. The
`assign(to:)` operator manages the life
cycle of the subscription, canceling the
subscription automatically when the
``Published`` instance deinitializes.
Because of this, the `assign(to:)`
operator doesn't return an
``AnyCancellable`` that you're
responsible for like ``assign(to:on:)``
does.
    ///
```

```
    /// The example below shows a model class that receives elements from an internal <doc://com.apple.documentation/documentation/Foundation/Timer/TimerPublisher>, and assigns them to a `@Published` property called `lastUpdated`. Because the `to` parameter has the `inout` keyword, you need to use the `&` operator when calling this method.
    ///
    ///     class MyModel: ObservableObject {
    ///         @Published var lastUpdated: Date = Date()
    ///         init() {
    ///             Timer.publish(every: 1.0, on: .main, in: .common)
    ///                 .autoconnect()
    ///                 .assign(to: &$lastUpdated)
    ///         }
    ///     }
    ///
    /// If you instead implemented `MyModel` with `assign(to: lastUpdated, on: self)`, storing the returned ``AnyCancellable`` instance could cause a reference cycle, because the ``Subscribers/Assign`` subscriber would hold a strong reference to `self`. Using `assign(to:)` solves this problem.
    ///
```

```
    /// While the `to` parameter uses the
`inout` keyword, this method doesn't
replace a reference type passed to it.
Instead, this notation indicates that the
operator may modify members of the
assigned object, as seen in the following
example:
    ///
    ///          class MyModel2:
ObservableObject {
    ///              @Published var id:
Int = 0
    ///          }
    ///          let model2 = MyModel2()
    ///          Just(100).assign(to:
&model2.$id)
    ///
    /// - Parameter published: A property
marked with the `@Published` attribute,
which receives and republishes all
elements received from the upstream
publisher.
    @available(iOS 14.0, macOS 11.0, tvOS
14.0, watchOS 7.0, *)
    public func assign(to published:
inout Published<Self.Output>.Publisher)
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Failure ==
Never {
```

```
    /// Creates a connectable wrapper
around the publisher.
    ///
    /// In the following example,
``Publisher/makeConnectable()`` wraps its
upstream publisher (an instance of
``Publishers/Share``) with a
``ConnectablePublisher``. Without this,
the first sink subscriber would receive
all the elements from the sequence
publisher and cause it to complete before
the second subscriber attaches. By making
the publisher connectable, the publisher
doesn’t produce any elements until after
the ``ConnectablePublisher/connect()``
call.
    ///
    ///      let subject =
Just<String>("Sent")
    ///      let pub = subject
    ///          .share()
    ///          .makeConnectable()
    ///      cancellable1 = pub.sink
{ print ("Stream 1 received: \($0)")  }
    ///
    ///      // For example purposes, use
DispatchQueue to add a second subscriber
    ///      // a second later, and then
connect to the publisher a second after
that.
    ///
DispatchQueue.main.asyncAfter(deadline: .
now() + 1) {
```

```
    ///           self.cancellable2 =
pub.sink { print ("Stream 2 received: \
($0)") }
    ///       }
    ///
DispatchQueue.main.asyncAfter(deadline: .
now() + 2) {
    ///           self.connectable =
pub.connect()
    ///       }
    ///       // Prints:
    ///       // Stream 2 received: Sent
    ///       // Stream 1 received: Sent
    ///
    ///  > Note: The
``ConnectablePublisher/connect()``
operator returns a ``Cancellable``
instance that you must retain. You can
also use this instance to cancel
publishing.
    ///
    /// - Returns: A
``ConnectablePublisher`` wrapping this
publisher.
    public func makeConnectable() ->
Publishers.MakeConnectable<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Collects all received elements,
```

```
and emits a single array of the
collection when the upstream publisher
finishes.
    ///
    /// Use ``Publisher/collect()`` to
gather elements into an array that the
operator emits after the upstream
publisher finishes.
    ///
    /// If the upstream publisher fails
with an error, this publisher forwards
the error to the downstream receiver
instead of sending its output.
    ///
    /// This publisher requests an
unlimited number of elements from the
upstream publisher and uses an unbounded
amount of memory to store the received
values. The publisher may exert memory
pressure on the system for very large
sets of elements.
    ///
    /// The ``Publisher/collect()``
operator only sends the collected array
to its downstream receiver after a
request whose demand is greater than 0
items. Otherwise, ``Publisher/collect()``
waits until it receives a non-zero
request.
    ///
    /// In the example below, an Integer
range is a publisher that emits an array
of integers:
```

```
    ///
    ///     let numbers = (0...10)
    ///     cancellable =
numbers.publisher
    ///         .collect()
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "[0, 1, 2, 3, 4,
5, 6, 7, 8, 9, 10]"
    ///
    /// - Returns: A publisher that
collects all received items and returns
them as an array upon completion.
    public func collect() ->
Publishers.Collect<Self>

    /// Collects up to the specified
number of elements, and then emits a
single array of the collection.
    ///
    /// Use ``Publisher/collect(_:)`` to
emit arrays of at most `count` elements
from an upstream publisher. If the
upstream publisher finishes before
collecting the specified number of
elements, the publisher sends an array of
only the items it received. This may be
fewer than `count` elements.
    ///
    /// If the upstream publisher fails
with an error, this publisher forwards
the error to the downstream receiver
instead of sending its output.
```

```
    ///
    /// In the example below, the
``Publisher/collect(_:)`` operator emits
one partial and two full arrays based on
the requested collection size of `5`:
    ///
    ///     let numbers = (0...10)
    ///     cancellable =
numbers.publisher
    ///         .collect(5)
    ///         .sink { print("\($0),
terminator: " "") }
    ///
    ///     // Prints "[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9] [10] "
    ///
    /// > Note: When this publisher
receives a request for `.max(n)`
elements, it requests `.max(count * n)`
from the upstream publisher.
    ///
    /// - Parameter count: The maximum
number of received elements to buffer
before publishing.
    /// - Returns: A publisher that
collects up to the specified number of
elements, and then publishes them as an
array.
    public func collect(_ count: Int) ->
Publishers.CollectByCount<Self>

    /// Collects elements by a given
time-grouping strategy, and emits a
```

```
single array of the collection.
    ///
    /// Use
``Publisher/collect(_:options:)`` to emit
arrays of elements on a schedule
specified by a ``Scheduler`` and `Stride`
that you provide. At the end of each
scheduled interval, the publisher sends
an array that contains the items it
collected. If the upstream publisher
finishes before filling the buffer, the
publisher sends an array that contains
items it received. This may be fewer than
the number of elements specified in the
requested `Stride`.
    ///
    /// If the upstream publisher fails
with an error, this publisher forwards
the error to the downstream receiver
instead of sending its output.
    ///
    /// The example above collects
timestamps generated on a one-second
<doc://com.apple.documentation/documentat
ion/Foundation/Timer> in groups
(`Stride`) of five.
    ///
    ///     let sub =
Timer.publish(every: 1, on: .main,
in: .default)
    ///         .autoconnect()
    ///         .collect(.byTime(RunLoop.
main, .seconds(5)))
```

```
    ///          .sink { print("\($0)",
terminator: "\n\n") }
    ///
    ///     // Prints: "[2020-01-24
00:54:46 +0000, 2020-01-24 00:54:47
+0000,
    ///     //          2020-01-24
00:54:48 +0000, 2020-01-24 00:54:49
+0000,
    ///     //          2020-01-24
00:54:50 +0000]"
    ///
    /// > Note: When this publisher
receives a request for `.max(n)`
elements, it requests `.max(count * n)`
from the upstream publisher.
    ///
    /// - Parameters:
    ///   - strategy: The timing group
strategy used by the operator to collect
and publish elements.
    ///   - options: Scheduler options to
use for the strategy.
    /// - Returns: A publisher that
collects elements by a given strategy,
and emits a single array of the
collection.
    public func collect<S>(_ strategy:
Publishers.TimeGroupingStrategy<S>,
options: S.SchedulerOptions? = nil) ->
Publishers.CollectByTime<Self, S> where S
: Scheduler
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Specifies the scheduler on which
to receive elements from the publisher.
    ///
    /// You use the
``Publisher/receive(on:options:)``
operator to receive results and
completion on a specific scheduler, such
as performing UI work on the main run
loop. In contrast with
``Publisher/subscribe(on:options:)``,
which affects upstream messages,
``Publisher/receive(on:options:)``
changes the execution context of
downstream messages.
    ///
    /// In the following example, the
``Publisher/subscribe(on:options:)``
operator causes `jsonPublisher` to
receive requests on `backgroundQueue`,
while the
    ///
``Publisher/receive(on:options:)`` causes
`labelUpdater` to receive elements and
completion on `RunLoop.main`.
    ///
    ///     let jsonPublisher =
MyJSONLoaderPublisher() // Some
publisher.
```

```
    ///     let labelUpdater =
MyLabelUpdateSubscriber() // Some
subscriber that updates the UI.
    ///
    ///     jsonPublisher
    ///         .subscribe(on:
backgroundQueue)
    ///         .receive(on:
RunLoop.main)
    ///         .subscribe(labelUpdater)
    ///
    ///
    /// Prefer
``Publisher/receive(on:options:)`` over
explicit use of dispatch queues when
performing work in subscribers. For
example, instead of the following
pattern:
    ///
    ///     pub.sink {
    ///         DispatchQueue.main.async
{
    ///             // Do something.
    ///         }
    ///     }
    ///
    /// Use this pattern instead:
    ///
    ///     pub.receive(on:
DispatchQueue.main).sink {
    ///         // Do something.
    ///     }
    ///
```

```
    ///  > Note: ``Publisher/receive(on:options:)`` doesn’t affect the scheduler used to call the subscriber’s ``Subscriber/receive(subscription:)`` method.
    ///
    /// - Parameters:
    ///   - scheduler: The scheduler the publisher uses for element delivery.
    ///   - options: Scheduler options used to customize element delivery.
    /// - Returns: A publisher that delivers elements using the specified scheduler.
    public func receive<S>(on scheduler: S, options: S.SchedulerOptions? = nil) -> Publishers.ReceiveOn<Self, S> where S : Scheduler
}

@available(macOS 10.15, iOS 13.0, tvOS 13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes the value of a key path.
    ///
    /// In the following example, the ``Publisher/map(_:)-6sm0a`` operator uses the Swift key path syntax to access the `die` member of the `DiceRoll` structure published by the ``Just`` publisher.
```

```
    ///
    /// The downstream sink subscriber
receives only the value of this `Int`,
not the entire `DiceRoll`.
    ///
    ///     struct DiceRoll {
    ///         let die: Int
    ///     }
    ///
    ///     cancellable =
Just(DiceRoll(die:Int.random(in:1...6)))
    ///         .map(\.die)
    ///         .sink {
    ///             print ("Rolled: \
($0)")
    ///         }
    ///     // Prints "Rolled: 3" (or
some other random value).
    ///
    /// - Parameter keyPath: The key path
of a property on `Output`.
    /// - Returns: A publisher that
publishes the value of the key path.
    public func map<T>(_ keyPath:
KeyPath<Self.Output, T>) ->
Publishers.MapKeyPath<Self, T>

    /// Publishes the values of two key
paths as a tuple.
    ///
    /// In the following example, the
``Publisher/map(_:_:)`` operator uses the
Swift key path syntax to access the
```

```
`die1` and `die2` members of the
`DiceRoll` structure published by the
``Just`` publisher.
    ///
    /// The downstream sink subscriber
receives only these two values (as an
`(Int, Int)` tuple), not the entire
`DiceRoll`.
    ///
    ///     struct DiceRoll {
    ///         let die1: Int
    ///         let die2: Int
    ///     }
    ///
    ///     cancellable =
Just(DiceRoll(die1:Int.random(in:1...6),
    ///
die2: Int.random(in:1...6)))
    ///         .map(\.die1, \.die2)
    ///         .sink { values in
    ///             print ("Rolled: \
(values.0), \(values.1) (total: \
(values.0 + values.1))")
    ///         }
    ///     // Prints "Rolled: 6, 4
(total: 10)" (or other random values).
    ///
    /// - Parameters:
    ///   - keyPath0: The key path of a
property on `Output`.
    ///   - keyPath1: The key path of
another property on `Output`.
    /// - Returns: A publisher that
```

```
publishes the values of two key paths as
a tuple.
    public func map<T0, T1>(_ keyPath0:
KeyPath<Self.Output, T0>, _ keyPath1:
KeyPath<Self.Output, T1>) ->
Publishers.MapKeyPath2<Self, T0, T1>

    /// Publishes the values of three key
paths as a tuple.
    ///
    /// In the following example, the
``Publisher/map(_:_:_:)`` operator uses
the Swift key path syntax to access the
`die1`, `die2`, and `die3` members of the
`DiceRoll` structure published by the
``Just`` publisher.
    ///
    /// The downstream sink subscriber
receives only these three values (as an
`(Int, Int, Int)` tuple), not the entire
`DiceRoll`.
    ///
    ///     struct DiceRoll {
    ///         let die1: Int
    ///         let die2: Int
    ///         let die3: Int
    ///     }
    ///
    ///     cancellable =
Just(DiceRoll(die1:Int.random(in:1...6),
    ///
die2: Int.random(in:1...6),
    ///
```

```
die3: Int.random(in:1...6)))
    ///          .map(\.die1, \.die2,
\.die3)
    ///          .sink { values in
    ///              print ("Rolled: \
(values.0), \(values.1), \(values.2)
(total \(values.0 + values.1 +
values.2))")
    ///          }
    ///     // Prints "Rolled: 5, 4, 2
(total 11)" (or other random values).
    ///
    /// - Parameters:
    ///   - keyPath0: The key path of a
property on `Output`.
    ///   - keyPath1: The key path of a
second property on `Output`.
    ///   - keyPath2: The key path of a
third property on `Output`.
    /// - Returns: A publisher that
publishes the values of three key paths
as a tuple.
    public func map<T0, T1, T2>(_
keyPath0: KeyPath<Self.Output, T0>, _
keyPath1: KeyPath<Self.Output, T1>, _
keyPath2: KeyPath<Self.Output, T2>) ->
Publishers.MapKeyPath3<Self, T0, T1, T2>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Failure ==
Never {
```

```
    /// The elements produced by the
publisher, as an asynchronous sequence.
    ///
    /// This property provides an
``AsyncPublisher``, which allows you to
use the Swift `async`-`await` syntax to
receive the publisher's elements. Because
``AsyncPublisher`` conforms to
<doc://com.apple.documentation/documentat
ion/Swift/AsyncSequence>, you iterate
over its elements with a `for`-`await`-
`in` loop, rather than attaching a
subscriber.
    ///
    /// The following example shows how
to use the `values` property to receive
elements asynchronously. The example
adapts a code snippet from the
``Publisher/filter(_:)`` operator's
documentation, which filters a sequence
to only emit even integers. This example
replaces the ``Subscribers/Sink``
subscriber with a `for`-`await`-`in` loop
that iterates over the ``AsyncPublisher``
provided by the `values` property.
    ///
    ///     let numbers: [Int] = [1, 2,
3, 4, 5]
    ///     let filtered =
numbers.publisher
    ///         .filter { $0 % 2 == 0 }
    ///
```

```
    ///     for await number in
filtered.values
    ///     {
    ///         print("\(number)",
terminator: " ")
    ///     }
    ///
    @available(macOS 12.0, iOS 15.0, tvOS
15.0, watchOS 8.0, *)
    public var values:
AsyncPublisher<Self> { get }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// The elements produced by the
publisher, as a throwing asynchronous
sequence.
    ///
    /// This property provides an
``AsyncThrowingPublisher``, which allows
you to use the Swift `async`-`await`
syntax to receive the publisher's
elements. Because ``AsyncPublisher``
conforms to
<doc://com.apple.documentation/documentat
ion/Swift/AsyncSequence>, you iterate
over its elements with a `for`-`await`-
`in` loop, rather than attaching a
subscriber. If the publisher terminates
with an error, the awaiting caller
```

```
receives the error as a `throw`.
    ///
    /// The following example shows how
to use the `values` property to receive
elements asynchronously. The example
adapts a code snippet from the
``Publisher/tryFilter(_:)`` operator's
documentation, which filters a sequence
to only emit even integers, and terminate
with an error on a `0`. This example
replaces the ``Subscribers/Sink``
subscriber with a `for`-`await`-`in` loop
that iterates over the ``AsyncPublisher``
provided by the `values` property. With
this approach, the error handling
previously provided in the sink
subscriber's
``Subscribers/Sink/receiveCompletion``
closure goes instead in a `catch` block.
    ///
    ///     let numbers: [Int] = [1, 2,
3, 4, 0, 5]
    ///     let filterPublisher =
numbers.publisher
    ///         .tryFilter{
    ///             if $0 == 0 {
    ///                 throw ZeroError()
    ///             } else {
    ///                 return $0 % 2 ==
0
    ///             }
    ///         }
    ///
```

```
    ///     do {
    ///         for try await number in
filterPublisher.values {
    ///             print ("\(number)",
terminator: " ")
    ///         }
    ///     } catch {
    ///         print ("\(error)")
    ///     }
    ///
    ///
    @available(macOS 12.0, iOS 15.0, tvOS
15.0, watchOS 8.0, *)
    public var values:
AsyncThrowingPublisher<Self> { get }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Republishes elements until
another publisher emits an element.
    ///
    /// After the second publisher
publishes an element, the publisher
returned by this method finishes.
    ///
    /// - Parameter publisher: A second
publisher.
    /// - Returns: A publisher that
republishes elements until the second
publisher publishes an element.
```

```
    public func prefix<P>(untilOutputFrom
publisher: P) ->
Publishers.PrefixUntilOutput<Self, P>
where P : Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Attaches the specified subject to
this publisher.
    ///
    /// - Parameter subject: The subject
to attach to this publisher.
    public func subscribe<S>(_ subject:
S) -> AnyCancellable where S : Subject,
Self.Failure == S.Failure, Self.Output ==
S.Output
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Applies a closure that collects
each element of a stream and publishes a
final result upon completion.
    ///
    /// Use ``Publisher/reduce(_:_:)`` to
collect a stream of elements and produce
an accumulated value based on a closure
you provide.
```

```
    ///
    /// In the following example, the
``Publisher/reduce(_:_:)`` operator
collects all the integer values it
receives from its upstream publisher:
    ///
    ///     let numbers = (0...10)
    ///     cancellable =
numbers.publisher
    ///         .reduce(0, { accum, next
in accum + next })
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "55"
    ///
    /// - Parameters:
    ///   - initialResult: The value that
the closure receives the first time it’s
called.
    ///   - nextPartialResult: A closure
that produces a new value by taking the
previously-accumulated value and the next
element it receives from the upstream
publisher.
    /// - Returns: A publisher that
applies the closure to all received
elements and produces an accumulated
value when the upstream publisher
finishes. If ``Publisher/reduce(_:_:)``
receives an error from the upstream
publisher, the operator delivers it to
the downstream subscriber, the publisher
terminates and publishes no value.
```

```
    public func reduce<T>(_
initialResult: T, _ nextPartialResult:
@escaping (T, Self.Output) -> T) ->
Publishers.Reduce<Self, T>

    /// Applies an error-throwing closure
that collects each element of a stream
and publishes a final result upon
completion.
    ///
    /// Use ``Publisher/tryReduce(_:_:)``
to collect a stream of elements and
produce an accumulated value based on an
error-throwing closure you provide.
    /// If the closure throws an error,
the publisher fails and passes the error
to its subscriber.
    ///
    /// In the example below, the
publisher's `0` element causes the
`myDivide(_:_:)` function to throw an
error and publish the
<doc://com.apple.documentation/documentat
ion/Swift/Double/nan> result:
    ///
    ///     struct DivisionByZeroError:
Error {}
    ///     func myDivide(_ dividend:
Double, _ divisor: Double) throws ->
Double {
    ///         guard divisor != 0 else {
throw DivisionByZeroError() }
    ///         return dividend / divisor
```

```
    ///     }
    ///
    ///     var numbers: [Double] = [5,
4, 3, 2, 1, 0]
    ///     numbers.publisher
    ///         .tryReduce(numbers.first!
, { accum, next in try myDivide(accum,
next) })
    ///         .catch({ _ in
Just(Double.nan) })
    ///         .sink { print("\($0)") }
    ///
    /// - Parameters:
    ///   - initialResult: The value that
the closure receives the first time it's
called.
    ///   - nextPartialResult: An error-
throwing closure that takes the
previously-accumulated value and the next
element from the upstream publisher to
produce a new value.
    ///
    /// - Returns: A publisher that
applies the closure to all received
elements and produces an accumulated
value when the upstream publisher
finishes.
    public func tryReduce<T>(_
initialResult: T, _ nextPartialResult:
@escaping (T, Self.Output) throws -> T)
-> Publishers.TryReduce<Self, T>
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Calls a closure with each
received element and publishes any
returned optional that has a value.
    ///
    /// Combine's
``Publisher/compactMap(_:)`` operator
performs a function similar to that of
<doc://com.apple.documentation/documentat
ion/Swift/Sequence/compactMap(_:)> in the
Swift standard library: the
``Publisher/compactMap(_:)`` operator in
Combine removes `nil` elements in a
publisher's stream and republishes non-
`nil` elements to the downstream
subscriber.
    ///
    /// The example below uses a range of
numbers as the source for a collection
based publisher. The
``Publisher/compactMap(_:)`` operator
consumes each element from the `numbers`
publisher attempting to access the
dictionary using the element as the key.
If the example's dictionary returns a
`nil`, due to a non-existent key,
``Publisher/compactMap(_:)`` filters out
the `nil` (missing) elements.
    ///
    ///     let numbers = (0...5)
```

```
    ///     let romanNumeralDict: [Int :
String] =
    ///         [1: "I", 2: "II", 3:
"III", 5: "V"]
    ///
    ///     cancellable =
numbers.publisher
    ///         .compactMap
{ romanNumeralDict[$0] }
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "I II III V"
    ///
    /// - Parameter transform: A closure
that receives a value and returns an
optional value.
    /// - Returns: Any non-`nil` optional
results of the calling the supplied
closure.
    public func compactMap<T>(_
transform: @escaping (Self.Output) -> T?)
-> Publishers.CompactMap<Self, T>

    /// Calls an error-throwing closure
with each received element and publishes
any returned optional that has a value.
    ///
    /// Use
``Publisher/tryCompactMap(_:)`` to remove
`nil` elements from a publisher's stream
based on an error-throwing closure you
provide. If the closure throws an error,
```

```
the publisher cancels the upstream
publisher and sends the thrown error to
the downstream subscriber as a
``Publisher/Failure``.
    ///
    /// The following example uses an
array of numbers as the source for a
collection-based publisher. A
``Publisher/tryCompactMap(_:)`` operator
consumes each integer from the publisher
and uses a dictionary to transform the
numbers from its Arabic to Roman
numerals, as an optional
<doc://com.apple.documentation/documentat
ion/Swift/String>.
    ///
    /// If the closure called by
``Publisher/tryCompactMap(_:)`` fails to
look up a Roman numeral, it returns the
optional String `(unknown)`.
    ///
    /// If the closure called by
``Publisher/tryCompactMap(_:)``
determines the input is `0`, it throws an
error. The
``Publisher/tryCompactMap(_:)`` operator
catches this error and stops publishing,
sending a
``Subscribers/Completion/failure(_:)``
that wraps the error.
    ///
    ///     struct ParseError: Error {}
    ///     func romanNumeral(from: Int)
```

```
throws -> String? {
    ///         let romanNumeralDict:
[Int : String] =
    ///             [1: "I", 2: "II", 3:
"III", 4: "IV", 5: "V"]
    ///         guard from != 0 else
{ throw ParseError() }
    ///         return
romanNumeralDict[from]
    ///     }
    ///     let numbers = [6, 5, 4, 3, 2,
1, 0]
    ///     cancellable =
numbers.publisher
    ///         .tryCompactMap { try
romanNumeral(from: $0) }
    ///         .sink(
    ///               receiveCompletion:
{ print ("\($0)") },
    ///               receiveValue:
{ print ("\($0)", terminator: " ") }
    ///          )
    ///
    ///     // Prints: "(Unknown) V IV
III II I failure(ParseError())"
    ///
    /// - Parameter transform: An error-
throwing closure that receives a value
and returns an optional value.
    /// - Returns: Any non-`nil` optional
results of calling the supplied closure.
    public func tryCompactMap<T>(_
transform: @escaping (Self.Output) throws
```

```
-> T?) -> Publishers.TryCompactMap<Self,
T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Combines elements from this
publisher with those from another
publisher, delivering an interleaved
sequence of elements.
    ///
    /// Use
``Publisher/merge(with:)-7fk3a`` when you
want to receive a new element whenever
any of the upstream publishers emits an
element. To receive tuples of the most-
recent value from all the upstream
publishers whenever any of them emit a
value, use
``Publisher/combineLatest(_:)``. To
combine elements from multiple upstream
publishers, use ``Publisher/zip(_:)``.
    ///
    /// In this example, as
``Publisher/merge(with:)-7fk3a`` receives
input from either upstream publisher, it
republishes it to the downstream:
    ///
    ///     let publisher =
PassthroughSubject<Int, Never>()
    ///     let pub2 =
```

```
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = publisher
    ///         .merge(with: pub2)
    ///         .sink { print("\($0)",
terminator: " " )}
    ///
    ///     publisher.send(2)
    ///     pub2.send(2)
    ///     publisher.send(3)
    ///     pub2.send(22)
    ///     publisher.send(45)
    ///     pub2.send(22)
    ///     publisher.send(17)
    ///
    ///     // Prints: "2 2 3 22 45 22
17"
    ///
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameter other: Another
publisher.
    /// - Returns: A publisher that emits
an event when either upstream publisher
emits an event.
    public func merge<P>(with other: P)
-> Publishers.Merge<Self, P> where P :
```

```
Publisher, Self.Failure == P.Failure,
Self.Output == P.Output

    /// Combines elements from this
publisher with those from two other
publishers, delivering an interleaved
sequence of elements.
    ///
    /// Use ``Publisher/merge(with:_:)``
when you want to receive a new element
whenever any of the upstream publishers
emits an element. To receive tuples of
the most-recent value from all the
upstream publishers whenever any of them
emit a value, use
``Publisher/combineLatest(_:_:)-5crqg``.
    /// To combine elements from multiple
upstream publishers, use
``Publisher/zip(_:_:)-8d7k7``.
    ///
    /// In this example, as
``Publisher/merge(with:_:)`` receives
input from the upstream publishers, it
republishes the interleaved elements to
the downstream:
    ///
    ///     let pubA =
PassthroughSubject<Int, Never>()
    ///     let pubB =
PassthroughSubject<Int, Never>()
    ///     let pubC =
PassthroughSubject<Int, Never>()
    ///
```

```
    ///     cancellable = pubA
    ///         .merge(with: pubB, pubC)
    ///         .sink { print("\($0)",
terminator: " " )}
    ///
    ///     pubA.send(1)
    ///     pubB.send(40)
    ///     pubC.send(90)
    ///     pubA.send(2)
    ///     pubB.send(50)
    ///     pubC.send(100)
    ///
    ///     // Prints: "1 40 90 2 50 100"
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameters:
    ///   - b: A second publisher.
    ///   - c: A third publisher.
    /// - Returns: A publisher that emits
an event when any upstream publisher
emits an event.
    public func merge<B, C>(with b: B, _
c: C) -> Publishers.Merge3<Self, B, C>
where B : Publisher, C : Publisher,
Self.Failure == B.Failure, Self.Output ==
B.Output, B.Failure == C.Failure,
B.Output == C.Output
```

```
    /// Combines elements from this
publisher with those from three other
publishers, delivering an interleaved
sequence of elements.
    ///
    /// Use
``Publisher/merge(with:_:_:)`` when you
want to receive a new element whenever
any of the upstream publishers emits an
element. To receive tuples of the most-
recent value from all the upstream
publishers whenever any of them emit a
value, use
``Publisher/combineLatest(_:_:_:)-48buc``
.
    /// To combine elements from multiple
upstream publishers, use
``Publisher/zip(_:_:_:)-16rcy``.
    ///
    /// In this example, as
``Publisher/merge(with:_:_:)`` receives
input from the upstream publishers, it
republishes the interleaved elements to
the downstream:
    ///
    ///     let pubA =
PassthroughSubject<Int, Never>()
    ///     let pubB =
PassthroughSubject<Int, Never>()
    ///     let pubC =
PassthroughSubject<Int, Never>()
    ///     let pubD =
```

```
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pubA
    ///         .merge(with: pubB, pubC,
pubD)
    ///         .sink { print("\($0)",
terminator: " " )}
    ///
    ///     pubA.send(1)
    ///     pubB.send(40)
    ///     pubC.send(90)
    ///     pubD.send(-1)
    ///     pubA.send(2)
    ///     pubB.send(50)
    ///     pubC.send(100)
    ///     pubD.send(-2)
    ///
    ///     // Prints: "1 40 90 -1 2 50
100 -2 "
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameters:
    ///   - b: A second publisher.
    ///   - c: A third publisher.
    ///   - d: A fourth publisher.
    /// - Returns: A publisher that emits
an event when any upstream publisher
```

```
emits an event.
    public func merge<B, C, D>(with b: B,
_ c: C, _ d: D) ->
Publishers.Merge4<Self, B, C, D> where
B : Publisher, C : Publisher, D :
Publisher, Self.Failure == B.Failure,
Self.Output == B.Output, B.Failure ==
C.Failure, B.Output == C.Output,
C.Failure == D.Failure, C.Output ==
D.Output

    /// Combines elements from this
publisher with those from four other
publishers, delivering an interleaved
sequence of elements.
    ///
    /// Use
``Publisher/merge(with:_:_:_:)`` when you
want to receive a new element whenever
any of the upstream publishers emits an
element. To receive tuples of the most-
recent value from all the upstream
publishers whenever any of them emit a
value, use
``Publisher/combineLatest(_:_:_:)-48buc``
.
    /// To combine elements from multiple
upstream publishers, use
``Publisher/zip(_:_:_:)-16rcy``.
    ///
    /// In this example, as
``Publisher/merge(with:_:_:_:)`` receives
input from the upstream publishers, it
```

```
republishes the interleaved elements to
the downstream:
    ///
    ///      let pubA =
PassthroughSubject<Int, Never>()
    ///      let pubB =
PassthroughSubject<Int, Never>()
    ///      let pubC =
PassthroughSubject<Int, Never>()
    ///      let pubD =
PassthroughSubject<Int, Never>()
    ///      let pubE =
PassthroughSubject<Int, Never>()
    ///
    ///      cancellable = pubA
    ///          .merge(with: pubB, pubC,
pubD, pubE)
    ///          .sink { print("\($0)",
terminator: " " ) }
    ///
    ///      pubA.send(1)
    ///      pubB.send(40)
    ///      pubC.send(90)
    ///      pubD.send(-1)
    ///      pubE.send(33)
    ///      pubA.send(2)
    ///      pubB.send(50)
    ///      pubC.send(100)
    ///      pubD.send(-2)
    ///      pubE.send(33)
    ///
    ///      // Prints: "1 40 90 -1 33 2
50 100 -2 33"
```

```
    ///
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameters:
    ///   - b: A second publisher.
    ///   - c: A third publisher.
    ///   - d: A fourth publisher.
    ///   - e: A fifth publisher.
    /// - Returns: A publisher that emits
an event when any upstream publisher
emits an event.
    public func merge<B, C, D, E>(with b:
B, _ c: C, _ d: D, _ e: E) ->
Publishers.Merge5<Self, B, C, D, E> where
B : Publisher, C : Publisher, D :
Publisher, E : Publisher, Self.Failure ==
B.Failure, Self.Output == B.Output,
B.Failure == C.Failure, B.Output ==
C.Output, C.Failure == D.Failure,
C.Output == D.Output, D.Failure ==
E.Failure, D.Output == E.Output

    /// Combines elements from this
publisher with those from five other
publishers, delivering an interleaved
sequence of elements.
    ///
```

```
    /// Use
``Publisher/merge(with:_:_:_:_:)`` when
you want to receive a new element
whenever any of the upstream publishers
emits an element. To receive tuples of
the most-recent value from all the
upstream publishers whenever any of them
emit a value, use
``Publisher/combineLatest(_:_:_:)-48buc``
.
    /// To combine elements from multiple
upstream publishers, use
``Publisher/zip(_:_:_:)-16rcy``.
    ///
    /// In this example, as
``Publisher/merge(with:_:_:_:_:)``
receives input from the upstream
publishers, it republishes the
interleaved elements to the downstream:
    ///
    ///     let pubA =
PassthroughSubject<Int, Never>()
    ///     let pubB =
PassthroughSubject<Int, Never>()
    ///     let pubC =
PassthroughSubject<Int, Never>()
    ///     let pubD =
PassthroughSubject<Int, Never>()
    ///     let pubE =
PassthroughSubject<Int, Never>()
    ///     let pubF =
PassthroughSubject<Int, Never>()
    ///
```

```
    ///     cancellable = pubA
    ///         .merge(with: pubB, pubC,
pubD, pubE, pubF)
    ///         .sink { print("\($0)",
terminator: " " ) }
    ///
    ///     pubA.send(1)
    ///     pubB.send(40)
    ///     pubC.send(90)
    ///     pubD.send(-1)
    ///     pubE.send(33)
    ///     pubF.send(44)
    ///
    ///     pubA.send(2)
    ///     pubB.send(50)
    ///     pubC.send(100)
    ///     pubD.send(-2)
    ///     pubE.send(33)
    ///     pubF.send(33)
    ///
    ///     //Prints: "1 40 90 -1 33 44 2
50 100 -2 33 33"
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameters:
    ///   - b: A second publisher.
    ///   - c: A third publisher.
```

```
    ///   - d: A fourth publisher.
    ///   - e: A fifth publisher.
    ///   - f: A sixth publisher.
    /// - Returns: A publisher that emits
an event when any upstream publisher
emits an event.
    public func merge<B, C, D, E, F>(with
b: B, _ c: C, _ d: D, _ e: E, _ f: F) ->
Publishers.Merge6<Self, B, C, D, E, F>
where B : Publisher, C : Publisher, D :
Publisher, E : Publisher, F : Publisher,
Self.Failure == B.Failure, Self.Output ==
B.Output, B.Failure == C.Failure,
B.Output == C.Output, C.Failure ==
D.Failure, C.Output == D.Output,
D.Failure == E.Failure, D.Output ==
E.Output, E.Failure == F.Failure,
E.Output == F.Output

    /// Combines elements from this
publisher with those from six other
publishers, delivering an interleaved
sequence of elements.
    ///
    /// Use
``Publisher/merge(with:_:_:_:_:)`` when
you want to receive a new element
whenever any of the upstream publishers
emits an element. To receive tuples of
the most-recent value from all the
upstream publishers whenever any of them
emit a value, use
``Publisher/combineLatest(_:_:_:)-48buc``
```

```
.
    /// To combine elements from multiple
upstream publishers, use
``Publisher/zip(_:_:_:)-16rcy``.
    ///
    /// In this example, as
``Publisher/merge(with:_:_:_:_:)``
receives input from the upstream
publishers; it republishes the
interleaved elements to the downstream:
    ///
    ///     let pubA =
PassthroughSubject<Int, Never>()
    ///     let pubB =
PassthroughSubject<Int, Never>()
    ///     let pubC =
PassthroughSubject<Int, Never>()
    ///     let pubD =
PassthroughSubject<Int, Never>()
    ///     let pubE =
PassthroughSubject<Int, Never>()
    ///     let pubF =
PassthroughSubject<Int, Never>()
    ///     let pubG =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pubA
    ///         .merge(with: pubB, pubC,
pubD, pubE, pubE, pubG)
    ///         .sink { print("\($0)",
terminator: " " ) }
    ///
    ///     pubA.send(1)
```

```
    ///     pubB.send(40)
    ///     pubC.send(90)
    ///     pubD.send(-1)
    ///     pubE.send(33)
    ///     pubF.send(44)
    ///     pubG.send(54)
    ///
    ///     pubA.send(2)
    ///     pubB.send(50)
    ///     pubC.send(100)
    ///     pubD.send(-2)
    ///     pubE.send(33)
    ///     pubF.send(33)
    ///     pubG.send(54)
    ///
    ///     //Prints: "1 40 90 -1 33 44
54 2 50 100 -2 33 33 54"
    ///
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameters:
    ///   - b: A second publisher.
    ///   - c: A third publisher.
    ///   - d: A fourth publisher.
    ///   - e: A fifth publisher.
    ///   - f: A sixth publisher.
    ///   - g: A seventh publisher.
```

```
    /// - Returns: A publisher that emits
an event when any upstream publisher
emits an event.
    public func merge<B, C, D, E, F,
G>(with b: B, _ c: C, _ d: D, _ e: E, _
f: F, _ g: G) -> Publishers.Merge7<Self,
B, C, D, E, F, G> where B : Publisher,
C : Publisher, D : Publisher, E :
Publisher, F : Publisher, G : Publisher,
Self.Failure == B.Failure, Self.Output ==
B.Output, B.Failure == C.Failure,
B.Output == C.Output, C.Failure ==
D.Failure, C.Output == D.Output,
D.Failure == E.Failure, D.Output ==
E.Output, E.Failure == F.Failure,
E.Output == F.Output, F.Failure ==
G.Failure, F.Output == G.Output

    /// Combines elements from this
publisher with those from seven other
publishers, delivering an interleaved
sequence of elements.
    ///
    /// Use
``Publisher/merge(with:_:_:_:_:_:)``
when you want to receive a new element
whenever any of the upstream publishers
emits an element. To receive tuples of
the most-recent value from all the
upstream publishers whenever any of them
emit a value, use
``Publisher/combineLatest(_:_:_:)-48buc``
.
```

```
    /// To combine elements from multiple
upstream publishers, use
``Publisher/zip(_:_:_:)-16rcy``.
    ///
    /// In this example, as
``Publisher/merge(with:_:_:_:_:_:)``
receives input from the upstream
publishers, it republishes the
interleaved elements to the downstream:
    ///
    ///     let pubA =
PassthroughSubject<Int, Never>()
    ///     let pubB =
PassthroughSubject<Int, Never>()
    ///     let pubC =
PassthroughSubject<Int, Never>()
    ///     let pubD =
PassthroughSubject<Int, Never>()
    ///     let pubE =
PassthroughSubject<Int, Never>()
    ///     let pubF =
PassthroughSubject<Int, Never>()
    ///     let pubG =
PassthroughSubject<Int, Never>()
    ///     let pubH =
PassthroughSubject<Int, Never>()
    ///
    ///     cancellable = pubA
    ///         .merge(with: pubB, pubC,
pubD, pubE, pubF, pubG, pubH)
    ///         .sink { print("\($0)",
terminator: " " ) }
    ///
```

```
    ///     pubA.send(1)
    ///     pubB.send(40)
    ///     pubC.send(90)
    ///     pubD.send(-1)
    ///     pubE.send(33)
    ///     pubF.send(44)
    ///     pubG.send(54)
    ///     pubH.send(1000)
    ///
    ///     pubA.send(2)
    ///     pubB.send(50)
    ///     pubC.send(100)
    ///     pubD.send(-2)
    ///     pubE.send(33)
    ///     pubF.send(33)
    ///     pubG.send(54)
    ///     pubH.send(1001)
    ///
    ///     //Prints: "1 40 90 -1 33 44
54 1000 2 50 100 -2 33 33 54 1001"
    ///
    /// The merged publisher continues to
emit elements until all upstream
publishers finish.
    /// If an upstream publisher produces
an error, the merged publisher fails with
that error.
    ///
    /// - Parameters:
    ///   - b: A second publisher.
    ///   - c: A third publisher.
    ///   - d: A fourth publisher.
    ///   - e: A fifth publisher.
```

```
    ///   - f: A sixth publisher.
    ///   - g: A seventh publisher.
    ///   - h: An eighth publisher.
    /// - Returns: A publisher that emits
an event when any upstream publisher
emits an event.
    public func merge<B, C, D, E, F, G,
H>(with b: B, _ c: C, _ d: D, _ e: E, _
f: F, _ g: G, _ h: H) ->
Publishers.Merge8<Self, B, C, D, E, F, G,
H> where B : Publisher, C : Publisher,
D : Publisher, E : Publisher, F :
Publisher, G : Publisher, H : Publisher,
Self.Failure == B.Failure, Self.Output ==
B.Output, B.Failure == C.Failure,
B.Output == C.Output, C.Failure ==
D.Failure, C.Output == D.Output,
D.Failure == E.Failure, D.Output ==
E.Output, E.Failure == F.Failure,
E.Output == F.Output, F.Failure ==
G.Failure, F.Output == G.Output,
G.Failure == H.Failure, G.Output ==
H.Output

    /// Combines elements from this
publisher with those from another
publisher of the same type, delivering an
interleaved sequence of elements.
    ///
    /// - Parameter other: Another
publisher of this publisher's type.
    /// - Returns: A publisher that emits
an event when either upstream publisher
```

```
emits an event.
    public func merge(with other: Self)
-> Publishers.MergeMany<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Transforms elements from the
upstream publisher by providing the
current
    /// element to a closure along with
the last value returned by the closure.
    ///
    /// Use ``Publisher/scan(_:_:)`` to
accumulate all previously-published
values into a single
    /// value, which you then combine
with each newly-published value.
    ///
    /// The following example logs a
running total of all values received
    /// from the sequence publisher.
    ///
    ///     let range = (0...5)
    ///     cancellable = range.publisher
    ///         .scan(0) { return $0 + $1
}
    ///         .sink { print ("\($0)",
terminator: " ") }
    ///      // Prints: "0 1 3 6 10 15 ".
    ///
```

```
    /// - Parameters:
    ///   - initialResult: The previous
result returned by the
`nextPartialResult` closure.
    ///   - nextPartialResult: A closure
that takes as its arguments the previous
value returned by the closure and the
next element emitted from the upstream
publisher.
    /// - Returns: A publisher that
transforms elements by applying a closure
that receives its previous return value
and the next element from the upstream
publisher.
    public func scan<T>(_ initialResult:
T, _ nextPartialResult: @escaping (T,
Self.Output) -> T) ->
Publishers.Scan<Self, T>

    /// Transforms elements from the
upstream publisher by providing the
current element to an error-throwing
closure along with the last value
returned by the closure.
    ///
    /// Use ``Publisher/tryScan(_:_:)``
to accumulate all previously-published
values into a single value, which you
then combine with each newly-published
value.
    /// If your accumulator closure
throws an error, the publisher terminates
with the error.
```

```
    ///
    /// In the example below, ``Publisher/tryScan(_:_:)`` calls a division function on elements of a collection publisher. The ``Publishers/TryScan`` publisher publishes each result until the function encounters a `DivisionByZeroError`, which terminates the publisher.
    ///
    ///     struct DivisionByZeroError: Error {}
    ///
    ///     /// A function that throws a DivisionByZeroError if `current` provided by the TryScan publisher is zero.
    ///     func myThrowingFunction(_ lastValue: Int, _ currentValue: Int) throws -> Int {
    ///         guard currentValue != 0 else { throw DivisionByZeroError() }
    ///         return (lastValue + currentValue) / currentValue
    ///      }
    ///
    ///     let numbers = [1,2,3,4,5,0,6,7,8,9]
    ///     cancellable = numbers.publisher
    ///         .tryScan(10) { try myThrowingFunction($0, $1) }
    ///         .sink(
    ///             receiveCompletion:
```

```
{ print ("\($0)") },
    ///             receiveValue: { print
("\($0)", terminator: " ") }
    ///           )
    ///
    ///     // Prints: "11 6 3 1 1 -1
failure(DivisionByZeroError())".
    ///
    /// If the closure throws an error,
the publisher fails with the error.
    ///
    /// - Parameters:
    ///   - initialResult: The previous
result returned by the
`nextPartialResult` closure.
    ///   - nextPartialResult: An error-
throwing closure that takes as its
arguments the previous value returned by
the closure and the next element emitted
from the upstream publisher.
    /// - Returns: A publisher that
transforms elements by applying a closure
that receives its previous return value
and the next element from the upstream
publisher.
    public func tryScan<T>(_
initialResult: T, _ nextPartialResult:
@escaping (T, Self.Output) throws -> T)
-> Publishers.TryScan<Self, T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
```

```
extension Publisher {

    /// Publishes the number of elements
received from the upstream publisher.
    ///
    /// Use ``Publisher/count()`` to
determine the number of elements received
from the upstream publisher before it
completes:
    ///
    ///     let numbers = (0...10)
    ///     cancellable =
numbers.publisher
    ///         .count()
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "11"
    ///
    /// - Returns: A publisher that
consumes all elements until the upstream
publisher finishes, then emits a single
value with the total number of elements
received.
    public func count() ->
Publishers.Count<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes the last element of a
stream that satisfies a predicate
```

```
closure, after upstream finishes.
    ///
    /// Use ``Publisher/last(where:)`` when you need to republish only the last element of a stream that satisfies a closure you specify.
    ///
    /// In the example below, a range publisher emits the last element that satisfies the closure’s criteria, then finishes normally:
    ///
    ///     let numbers = (-10...10)
    ///     cancellable = numbers.publisher
    ///         .last { $0 < 6 }
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "5"
    ///
    /// - Parameter predicate: A closure that takes an element as its parameter and returns a Boolean value that indicates whether to publish the element.
    /// - Returns: A publisher that only publishes the last element satisfying the given predicate.
    public func last(where predicate: @escaping (Self.Output) -> Bool) -> Publishers.LastWhere<Self>

    /// Publishes the last element of a stream that satisfies an error-throwing
```

```
predicate closure, after the stream
finishes.
    ///
    /// Use ``Publisher/tryLast(where:)``
when you need to republish the last
element that satisfies an error-throwing
closure you specify. If the predicate
closure throws an error, the publisher
fails.
    ///
    /// In the example below, a publisher
emits the last element that satisfies the
error-throwing closure, then finishes
normally:
    ///
    ///     struct RangeError: Error {}
    ///
    ///     let numbers = [-62, 1, 6, 10,
9, 22, 41, -1, 5]
    ///     cancellable =
numbers.publisher
    ///         .tryLast {
    ///             guard 0 != 0  else
{throw RangeError()}
    ///             return true
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)", terminator:
" ") },
    ///             receiveValue: { print
("\($0)", terminator: " ") }
    ///         )
```

```
    ///     // Prints: "5 completion:
finished"
    ///     // If instead the numbers
array had contained a `0`, the `tryLast`
operator would terminate publishing with
a RangeError."
    ///
    /// - Parameter predicate: A closure
that takes an element as its parameter
and returns a Boolean value that
indicates whether to publish the element.
    /// - Returns: A publisher that only
publishes the last element satisfying the
given predicate.
    public func tryLast(where predicate:
@escaping (Self.Output) throws -> Bool)
-> Publishers.TryLastWhere<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Ignores all upstream elements,
but passes along the upstream publisher's
completion state (finished or failed).
    ///
    /// Use the
``Publisher/ignoreOutput()`` operator to
determine if a publisher is able to
complete successfully or would fail.
    ///
    /// In the example below, the array
```

```
publisher (`numbers`) delivers the first
five of its elements successfully, as
indicated by the
``Publisher/ignoreOutput()`` operator.
The operator consumes, but doesn’t
republish the elements downstream.
However, the sixth element, `0`, causes
the error throwing closure to catch a
`NoZeroValuesAllowedError` that
terminates the stream.
    ///
    ///     struct
NoZeroValuesAllowedError: Error {}
    ///     let numbers = [1, 2, 3, 4, 5,
0, 6, 7, 8, 9]
    ///     cancellable =
numbers.publisher
    ///         .tryFilter({ anInt in
    ///             guard anInt != 0 else
{ throw NoZeroValuesAllowedError() }
    ///             return anInt < 20
    ///         })
    ///         .ignoreOutput()
    ///         .sink(receiveCompletion:
{print("completion: \($0)")},
    ///               receiveValue:
{print("value \($0)")})
    ///
    ///     // Prints: "completion:
failure(NoZeroValuesAllowedError())"
    ///
    /// The output type of this publisher
is
```

```
<doc://com.apple.documentation/documentation/Swift/Never>.
    ///
    /// - Returns: A publisher that ignores all upstream elements.
    public func ignoreOutput() -> Publishers.IgnoreOutput<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS 13.0, watchOS 6.0, *)
extension Publisher where Self.Failure == Never {

    /// Assigns each element from a publisher to a property on an object.
    ///
    /// Use the ``Publisher/assign(to:on:)`` subscriber when you want to set a given property each time a publisher produces a value.
    ///
    /// In this example, the ``Publisher/assign(to:on:)`` sets the value of the `anInt` property on an instance of `MyClass`:
    ///
    ///     class MyClass {
    ///         var anInt: Int = 0 {
    ///             didSet {
    ///                 print("anInt was set to: \(anInt)", terminator: "; ")
    ///             }
```

```
    ///         }
    ///     }
    ///
    ///     var myObject = MyClass()
    ///     let myRange = (0...2)
    ///     cancellable =
myRange.publisher
    ///         .assign(to: \.anInt, on:
myObject)
    ///
    ///     // Prints: "anInt was set to:
0; anInt was set to: 1; anInt was set to:
2"
    ///
    ///  > Important: The
``Subscribers/Assign`` instance created
by this operator maintains a strong
reference to `object`, and sets it to
`nil` when the upstream publisher
completes (either normally or with an
error).
    ///
    /// - Parameters:
    ///   - keyPath: A key path that
indicates the property to assign. See
[Key-Path Expression]
(https://developer.apple.com/library/arch
ive/documentation/Swift/Conceptual/
Swift_Programming_Language/
Expressions.html#//apple_ref/doc/uid/
TP40014097-CH32-ID563) in _The Swift
Programming Language_ to learn how to use
key paths to specify a property of an
```

```
object.
    ///   - object: The object that
contains the property. The subscriber
assigns the object’s property every time
it receives a new value.
    /// - Returns: An ``AnyCancellable``
instance. Call ``Cancellable/cancel()``
on this instance when you no longer want
the publisher to automatically assign the
property. Deinitializing this instance
will also cancel automatic assignment.
    public func assign<Root>(to keyPath:
ReferenceWritableKeyPath<Root,
Self.Output>, on object: Root) ->
AnyCancellable
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Failure ==
Self.Output.Failure, Self.Output :
Publisher {

    /// Republishes elements sent by the
most recently received publisher.
    ///
    /// This operator works with an
upstream publisher of publishers,
flattening the stream of elements to
appear as if they were coming from a
single stream of elements. It switches
the inner publisher as new ones arrive
but keeps the outer publisher constant
```

```
for downstream subscribers.
    ///
    /// For example, given the type
`AnyPublisher<URLSession.DataTaskPublishe
r, NSError>`, calling `switchToLatest()`
results in the type
`SwitchToLatest<(Data, URLResponse),
URLError>`. The downstream subscriber
sees a continuous stream of `(Data,
URLResponse)` elements from what looks
like a single
<doc://com.apple.documentation/documentat
ion/Foundation/URLSession/
DataTaskPublisher> even though the
elements are coming from different
upstream publishers.
    ///
    /// When this operator receives a new
publisher from the upstream publisher, it
cancels its previous subscription. Use
this feature to prevent earlier
publishers from performing unnecessary
work, such as creating network request
publishers from frequently updating user
interface publishers.
    ///
    /// The following example updates a
``PassthroughSubject`` with a new value
every `0.1` seconds. A
``Publisher/map(_:)-99evh`` operator
receives the new value and uses it to
create a new
<doc://com.apple.documentation/documentat
```

```
ion/Foundation/URLSession/
DataTaskPublisher>. By using the
`switchToLatest()` operator, the
downstream sink subscriber receives the
`(Data, URLResponse)` output type from
the data task publishers, rather than the
<doc://com.apple.documentation/documentat
ion/Foundation/URLSession/
DataTaskPublisher> type produced by the
``Publisher/map(_:)-99evh`` operator.
Furthermore, creating each new data task
publisher cancels the previous data task
publisher.
    ///
    ///     let subject =
PassthroughSubject<Int, Never>()
    ///     cancellable = subject
    ///         .setFailureType(to:
URLError.self)
    ///         .map() { index ->
URLSession.DataTaskPublisher in
    ///             let url = URL(string:
"https://example.org/get?index=\
(index)")!
    ///             return
URLSession.shared.dataTaskPublisher(for:
url)
    ///         }
    ///         .switchToLatest()
    ///         .sink(receiveCompletion:
{ print("Complete: \($0)") },
    ///               receiveValue:
{ (data, response) in
```

```
    ///                 guard let url =
response.url else { print("Bad
response."); return }
    ///                 print("URL: \
(url)")
    ///         })
    ///
    ///     for index in 1...5 {
    ///
DispatchQueue.main.asyncAfter(deadline: .
now() + TimeInterval(index/10)) {
    ///             subject.send(index)
    ///         }
    ///     }
    ///
    ///     // Prints "URL:
https://example.org/get?index=5"
    ///
    /// The exact behavior of this
example depends on the value of
`asyncAfter` and the speed of the network
connection. If the delay value is longer,
or the network connection is fast, the
earlier data tasks may complete before
`switchToLatest()` can cancel them. If
this happens, the output includes
multiple URLs whose tasks complete before
cancellation.
    public func switchToLatest() ->
Publishers.SwitchToLatest<Self.Output,
Self>
}
```

```
@available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
extension Publisher where Self.Failure ==
Never, Self.Output : Publisher {

    /// Republishes elements sent by the
most recently received publisher.
    ///
    /// This operator works with an
upstream publisher of publishers,
flattening the stream of elements to
appear as if they were coming from a
single stream of elements. It switches
the inner publisher as new ones arrive
but keeps the outer publisher constant
for downstream subscribers.
    ///
    /// When this operator receives a new
publisher from the upstream publisher, it
cancels its previous subscription. Use
this feature to prevent earlier
publishers from performing unnecessary
work, such as creating network request
publishers from frequently updating user
interface publishers.
    public func switchToLatest() ->
Publishers.SwitchToLatest<Self.Output,
Publishers.SetFailureType<Self,
Self.Output.Failure>>
}

@available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
```

```
extension Publisher where Self.Failure ==
Never, Self.Output : Publisher,
Self.Output.Failure == Never {

    /// Republishes elements sent by the
most recently received publisher.
    ///
    /// This operator works with an
upstream publisher of publishers,
flattening the stream of elements to
appear as if they were coming from a
single stream of elements. It switches
the inner publisher as new ones arrive
but keeps the outer publisher constant
for downstream subscribers.
    ///
    /// When this operator receives a new
publisher from the upstream publisher, it
cancels its previous subscription. Use
this feature to prevent earlier
publishers from performing unnecessary
work, such as creating network request
publishers from frequently updating user
interface publishers.
    public func switchToLatest() ->
Publishers.SwitchToLatest<Self.Output,
Self>
}

@available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
extension Publisher where Self.Output :
Publisher, Self.Output.Failure == Never {
```

```
    /// Republishes elements sent by the
most recently received publisher.
    ///
    /// This operator works with an
upstream publisher of publishers,
flattening the stream of elements to
appear as if they were coming from a
single stream of elements. It switches
the inner publisher as new ones arrive
but keeps the outer publisher constant
for downstream subscribers.
    ///
    /// When this operator receives a new
publisher from the upstream publisher, it
cancels its previous subscription. Use
this feature to prevent earlier
publishers from performing unnecessary
work, such as creating network request
publishers from frequently updating user
interface publishers.
    public func switchToLatest() ->
Publishers.SwitchToLatest<Publishers.SetF
ailureType<Self.Output, Self.Failure>,
Publishers.Map<Self,
Publishers.SetFailureType<Self.Output,
Self.Failure>>>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {
```

```
    /// Attempts to recreate a failed
subscription with the upstream publisher
up to the number of times you specify.
    ///
    /// Use ``Publisher/retry(_:)`` to
try a connecting to an upstream publisher
after a failed connection attempt.
    ///
    /// In the example below, a
<doc://com.apple.documentation/documentat
ion/Foundation/URLSession/
DataTaskPublisher> attempts to connect to
a remote URL. If the connection attempt
succeeds, it publishes the remote
service's HTML to the downstream
publisher and completes normally.
Otherwise, the retry operator attempts to
reestablish the connection. If after
three attempts the publisher still can't
connect to the remote URL, the
``Publisher/catch(_:)`` operator replaces
the error with a new publisher that
publishes a "connection timed out" HTML
page. After the downstream subscriber
receives the timed out message, the
stream completes normally.
    ///
    ///     struct WebSiteData: Codable {
    ///         var rawHTML: String
    ///     }
    ///
    ///     let myURL = URL(string:
"https://www.example.com")
```

```
    ///
    ///     cancellable =
URLSession.shared.dataTaskPublisher(for:
myURL!)
    ///         .retry(3)
    ///         .map({ (page) ->
WebSiteData in
    ///             return
WebSiteData(rawHTML: String(decoding:
page.data, as: UTF8.self))
    ///         })
    ///         .catch { error in
    ///             return
Just(WebSiteData(rawHTML: "<HTML>Unable
to load page - timed out.</HTML>"))
    ///     }
    ///     .sink(receiveCompletion:
{ print ("completion: \($0)") },
    ///           receiveValue: { print
("value: \($0)") }
    ///       )
    ///
    ///     // Prints: The HTML content
from the remote URL upon a successful
connection,
    ///     //         or returns
"<HTML>Unable to load page - timed
out.</HTML>" if the number of retries
exceeds the specified value.
    ///
    /// After exceeding the specified
number of retries, the publisher passes
the failure to the downstream receiver.
```

```
    /// - Parameter retries: The number
of times to attempt to recreate the
subscription.
    /// - Returns: A publisher that
attempts to recreate its subscription to
a failed upstream publisher.
    public func retry(_ retries: Int) ->
Publishers.Retry<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Converts any failure from the
upstream publisher into a new error.
    ///
    /// Use the
``Publisher/mapError(_:)`` operator when
you need to replace one error type with
another, or where a downstream operator
needs the error types of its inputs to
match.
    ///
    /// The following example uses a
``Publisher/tryMap(_:)`` operator to
divide `1` by each element produced by a
sequence publisher. When the publisher
produces a `0`, the
``Publisher/tryMap(_:)`` fails with a
    /// `DivisionByZeroError`. The
``Publisher/mapError(_:)`` operator
converts this into a `MyGenericError`.
```

```
    ///
    ///     struct DivisionByZeroError:
Error {}
    ///     struct MyGenericError: Error
{ var wrappedError: Error }
    ///
    ///     func myDivide(_ dividend:
Double, _ divisor: Double) throws ->
Double {
    ///             guard divisor != 0
else { throw DivisionByZeroError() }
    ///             return dividend /
divisor
    ///         }
    ///
    ///     let divisors: [Double] = [5,
4, 3, 2, 1, 0]
    ///     divisors.publisher
    ///         .tryMap { try myDivide(1,
$0) }
    ///         .mapError
{ MyGenericError(wrappedError: $0) }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)") ,
    ///             receiveValue: { print
("value: \($0)", terminator: " ") }
    ///          )
    ///
    ///     // Prints: "0.2 0.25
0.3333333333333333 0.5 1.0 completion:
failure(MyGenericError(wrappedError:
DivisionByZeroError()))"
```

```
    ///
    /// - Parameter transform: A closure
that takes the upstream failure as a
parameter and returns a new error for the
publisher to terminate with.
    /// - Returns: A publisher that
replaces any upstream failure with a new
error produced by the `transform`
closure.
    public func mapError<E>(_ transform:
@escaping (Self.Failure) -> E) ->
Publishers.MapError<Self, E> where E :
Error
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes either the most-recent
or first element published by the
upstream publisher in the specified time
interval.
    ///
    /// Use
``Publisher/throttle(for:scheduler:latest
:)`` to selectively republish elements
from an upstream publisher during an
interval you specify. Other elements
received from the upstream in the
throttling interval aren't republished.
    ///
    /// In the example below, a
```

```
<doc://com.apple.documentation/documentation/Foundation/Timer/TimerPublisher> produces elements on one-second intervals; the ``Publisher/throttle(for:scheduler:latest:)`` operator delivers the first event, then republishes only the latest event in the following ten second intervals:
    ///
    ///     cancellable = Timer.publish(every: 3.0, on: .main, in: .default)
    ///         .autoconnect()
    ///         .print("\(Date().description)")
    ///         .throttle(for: 10.0, scheduler: RunLoop.main, latest: true)
    ///         .sink(
    ///             receiveCompletion: { print ("Completion: \($0).") },
    ///             receiveValue: { print("Received Timestamp \($0).") }
    ///           )
    ///
    ///     // Prints:
    ///      //    Publish at: 2020-03-19 18:26:54 +0000: receive value: (2020-03-19 18:26:57 +0000)
    ///      //    Received Timestamp 2020-03-19 18:26:57 +0000.
    ///      //    Publish at: 2020-03-19 18:26:54 +0000: receive value: (2020-03-19 18:27:00 +0000)
```

```
    ///      //    Publish at: 2020-03-19
18:26:54 +0000: receive value: (2020-03-
19 18:27:03 +0000)
    ///      //    Publish at: 2020-03-19
18:26:54 +0000: receive value: (2020-03-
19 18:27:06 +0000)
    ///      //    Publish at: 2020-03-19
18:26:54 +0000: receive value: (2020-03-
19 18:27:09 +0000)
    ///      //    Received Timestamp
2020-03-19 18:27:09 +0000.
    ///
    /// - Parameters:
    ///   - interval: The interval at
which to find and emit either the most
recent or the first element, expressed in
the time system of the scheduler.
    ///   - scheduler: The scheduler on
which to publish elements.
    ///   - latest: A Boolean value that
indicates whether to publish the most
recent element. If `false`, the publisher
emits the first element received during
the interval.
    /// - Returns: A publisher that emits
either the most-recent or first element
received during the specified interval.
    public func throttle<S>(for interval:
S.SchedulerTimeType.Stride, scheduler: S,
latest: Bool) ->
Publishers.Throttle<Self, S> where S :
Scheduler
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Shares the output of an upstream
publisher with multiple subscribers.
    ///
    /// The publisher returned by this
operator supports multiple subscribers,
all of whom receive unchanged elements
and completion states from the upstream
publisher.
    ///
    ///  - Tip: ``Publishers/Share`` is
effectively a combination of the
``Publishers/Multicast`` and
``PassthroughSubject`` publishers, with
an implicit
``ConnectablePublisher/autoconnect()``.
    ///
    /// The following example uses a
sequence publisher as a counter to
publish three random numbers, generated
by a ``Publisher/map(_:)-99evh``
operator. It uses a ``Publisher/share()``
operator to share the same random number
to each of two subscribers. This example
uses a
``Publisher/delay(for:tolerance:scheduler
:options:)`` operator only to prevent the
first subscriber from exhausting the
sequence publisher immediately; an
```

```
asynchronous publisher wouldn’t need
this.
    ///
    ///     let pub = (1...3).publisher
    ///         .delay(for: 1, scheduler:
DispatchQueue.main)
    ///         .map( { _ in return
Int.random(in: 0...100) } )
    ///         .print("Random")
    ///         .share()
    ///
    ///     cancellable1 = pub
    ///         .sink { print ("Stream 1
received: \($0)")}
    ///     cancellable2 = pub
    ///         .sink { print ("Stream 2
received: \($0)")}
    ///
    ///     // Prints:
    ///     // Random: receive value:
(20)
    ///     // Stream 1 received: 20
    ///     // Stream 2 received: 20
    ///     // Random: receive value:
(85)
    ///     // Stream 1 received: 85
    ///     // Stream 2 received: 85
    ///     // Random: receive value:
(98)
    ///     // Stream 1 received: 98
    ///     // Stream 2 received: 98
    ///
    ///
```

```
    /// Without the ``Publisher/share()``
operator, stream 1 receives three random
values, followed by stream 2 receiving
three different random values.
    ///
    /// Also note that
``Publishers/Share`` is a class rather
than a structure like most other
publishers. This means you can use this
operator to create a publisher instance
that uses reference semantics.
    /// - Returns: A class instance that
shares elements received from its
upstream to multiple subscribers.
    public func share() ->
Publishers.Share<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher where Self.Output :
Comparable {

    /// Publishes the minimum value
received from the upstream publisher,
after it finishes.
    ///
    /// Use ``Publisher/min(by:)`` to
find the minimum value in a stream of
elements from an upstream publisher.
    ///
    /// In the example below, the
``Publisher/min(by:)`` operator emits a
```

```
value when the publisher finishes, that
value is the minimum of the values
received from upstream, which is `-1`.
    ///
    ///     let numbers = [-1, 0, 10, 5]
    ///     numbers.publisher
    ///         .min()
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "-1"
    ///
    /// After this publisher receives a
request for more than 0 items, it
requests unlimited items from its
upstream publisher.
    /// - Returns: A publisher that
publishes the minimum value received from
the upstream publisher, after the
upstream publisher finishes.
    public func min() ->
Publishers.Comparison<Self>

    /// Publishes the maximum value
received from the upstream publisher,
after it finishes.
    ///
    /// Use ``Publisher/max()`` to
determine the maximum value in the stream
of elements from an upstream publisher.
    ///
    /// In the example below, the
``Publisher/max()`` operator emits a
value when the publisher finishes, that
```

```
value is the maximum of the values
received from upstream, which is `10`.
    ///
    ///     let numbers = [0, 10, 5]
    ///     cancellable =
numbers.publisher
    ///         .max()
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "10"
    ///
    /// After this publisher receives a
request for more than 0 items, it
requests unlimited items from its
upstream publisher.
    /// - Returns: A publisher that
publishes the maximum value received from
the upstream publisher, after the
upstream publisher finishes.
    public func max() ->
Publishers.Comparison<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes the minimum value
received from the upstream publisher,
after it finishes.
    ///
    /// Use ``Publisher/min(by:)`` to
determine the minimum value in the stream
```

```
of elements from an upstream publisher
using a comparison operation you specify.
    ///
    /// This operator is useful when the
value received from the upstream
publisher isn't
<doc://com.apple.documentation/documentat
ion/Swift/Comparable>.
    ///
    /// In the example below an array
publishes enumeration elements
representing playing card ranks. The
``Publisher/min(by:)`` operator compares
the current and next elements using the
`rawValue` property of each enumeration
value in the user supplied closure and
prints the minimum value found after
publishing all of the elements.
    ///
    ///     enum Rank: Int {
    ///         case ace = 1, two, three,
four, five, six, seven, eight, nine, ten,
jack, queen, king
    ///     }
    ///
    ///     let cards: [Rank] =
[.five, .queen, .ace, .eight, .king]
    ///     cancellable = cards.publisher
    ///         .min {
    ///             return  $0.rawValue <
$1.rawValue
    ///         }
    ///         .sink { print("\($0)") }
```

```
    ///
    ///     // Prints: "ace"
    ///
    /// After this publisher receives a
request for more than 0 items, it
requests unlimited items from its
upstream publisher.
    ///
    /// - Parameter areInIncreasingOrder:
A closure that receives two elements and
returns true if they’re in increasing
order.
    /// - Returns: A publisher that
publishes the minimum value received from
the upstream publisher, after the
upstream publisher finishes.
    public func min(by
areInIncreasingOrder: @escaping
(Self.Output, Self.Output) -> Bool) ->
Publishers.Comparison<Self>

    /// Publishes the minimum value
received from the upstream publisher,
using the provided error-throwing closure
to order the items.
    ///
    /// Use ``Publisher/tryMin(by:)`` to
determine the minimum value of elements
received from the upstream publisher
using an error-throwing closure you
specify.
    ///
    /// In the example below, an array
```

```
publishes elements. The
``Publisher/tryMin(by:)`` operator
executes the error-throwing closure that
throws when the `first` element is an odd
number, terminating the publisher.
    ///
    ///     struct IllegalValueError:
Error {}
    ///
    ///     let numbers: [Int]  = [0, 10,
6, 13, 22, 22]
    ///     numbers.publisher
    ///         .tryMin { first, second
-> Bool in
    ///             if (first % 2 != 0) {
    ///                 throw
IllegalValueError()
    ///             }
    ///             return first < second
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)") },
    ///             receiveValue: { print
("value: \($0)") }
    ///         )
    ///
    ///     // Prints: "completion:
failure(IllegalValueError())"
    ///
    /// After this publisher receives a
request for more than 0 items, it
requests unlimited items from its
```

```
upstream publisher.
    ///
    /// - Parameter areInIncreasingOrder:
A throwing closure that receives two
elements and returns `true` if they’re in
increasing order. If this closure throws,
the publisher terminates with a
``Subscribers/Completion/failure(_:)``.
    /// - Returns: A publisher that
publishes the minimum value received from
the upstream publisher, after the
upstream publisher finishes.
    public func tryMin(by
areInIncreasingOrder: @escaping
(Self.Output, Self.Output) throws ->
Bool) -> Publishers.TryComparison<Self>

    /// Publishes the maximum value
received from the upstream publisher,
using the provided ordering closure.
    ///
    /// Use ``Publisher/max(by:)`` to
determine the maximum value of elements
received from the upstream publisher
based on an ordering closure you specify.
    ///
    /// In the example below, an array
publishes enumeration elements
representing playing card ranks. The
``Publisher/max(by:)`` operator compares
the current and next elements using the
`rawValue` property of each enumeration
value in the user supplied closure and
```

```
prints the maximum value found after
publishing all of the elements.
    ///
    ///     enum Rank: Int {
    ///         case ace = 1, two, three,
four, five, six, seven, eight, nine, ten,
jack, queen, king
    ///     }
    ///
    ///     let cards: [Rank] =
[.five, .queen, .ace, .eight, .jack]
    ///     cancellable = cards.publisher
    ///         .max {
    ///             return  $0.rawValue >
$1.rawValue
    ///         }
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "queen"
    ///
    /// After this publisher receives a
request for more than 0 items, it
requests unlimited items from its
upstream publisher.
    ///
    /// - Parameter areInIncreasingOrder:
A closure that receives two elements and
returns true if they’re in increasing
order.
    /// - Returns: A publisher that
publishes the maximum value received from
the upstream publisher, after the
upstream publisher finishes.
```

```
    public func max(by
areInIncreasingOrder: @escaping
(Self.Output, Self.Output) -> Bool) ->
Publishers.Comparison<Self>

    /// Publishes the maximum value
received from the upstream publisher,
using the provided error-throwing closure
to order the items.
    ///
    /// Use ``Publisher/tryMax(by:)`` to
determine the maximum value of elements
received from the upstream publisher
using an error-throwing closure you
specify.
    ///
    /// In the example below, an array
publishes elements. The
``Publisher/tryMax(by:)`` operator
executes the error-throwing closure that
throws when the `first` element is an odd
number, terminating the publisher.
    ///
    ///     struct IllegalValueError:
Error {}
    ///
    ///     let numbers: [Int]  = [0, 10,
6, 13, 22, 22]
    ///     cancellable =
numbers.publisher
    ///         .tryMax { first, second
-> Bool in
    ///             if (first % 2 != 0) {
```

```
    ///                     throw
IllegalValueError()
    ///             }
    ///             return first > second
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)") },
    ///             receiveValue: { print
("value: \($0)") }
    ///         )
    ///
    ///     // Prints: completion:
failure(IllegalValueError())
    ///
    /// After this publisher receives a
request for more than 0 items, it
requests unlimited items from its
upstream publisher.
    ///
    /// - Parameter areInIncreasingOrder:
A throwing closure that receives two
elements and returns `true` if they’re in
increasing order. If this closure throws,
the publisher terminates with a
``Subscribers/Completion/failure(_:)``.
    ///
    /// - Returns: A publisher that
publishes the maximum value received from
the upstream publisher, after the
upstream publisher finishes.
    public func tryMax(by
areInIncreasingOrder: @escaping
```

```
(Self.Output, Self.Output) throws ->
Bool) -> Publishers.TryComparison<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Replaces nil elements in the
stream with the provided element.
    ///
    /// The
``Publisher/replaceNil(with:)`` operator
enables replacement of `nil` values in a
stream with a substitute value. In the
example below, a collection publisher
contains a nil value. The
``Publisher/replaceNil(with:)`` operator
replaces this with `0.0`.
    ///
    ///     let numbers: [Double?] =
[1.0, 2.0, nil, 3.0]
    ///     numbers.publisher
    ///         .replaceNil(with: 0.0)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "Optional(1.0)
Optional(2.0) Optional(0.0)
Optional(3.0)"
    ///
    /// - Parameter output: The element
to use when replacing `nil`.
```

```
    /// - Returns: A publisher that
replaces `nil` elements from the upstream
publisher with the provided element.
    public func replaceNil<T>(with
output: T) -> Publishers.Map<Self, T>
where Self.Output == T?
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Replaces any errors in the stream
with the provided element.
    ///
    /// If the upstream publisher fails
with an error, this publisher emits the
provided element, then finishes normally.
    ///
    /// In the example below, a publisher
of strings fails with a `MyError`
instance, which sends a failure
completion downstream. The
``Publisher/replaceError(with:)``
operator handles the failure by
publishing the string `(replacement
element)` and completing normally.
    ///
    ///     struct MyError: Error {}
    ///     let fail = Fail<String,
MyError>(error: MyError())
    ///     cancellable = fail
    ///         .replaceError(with:
```

```
"(replacement element)")
    ///          .sink(
    ///              receiveCompletion:
{ print ("\($0)") },
    ///              receiveValue: { print
("\($0)", terminator: " ") }
    ///          )
    ///
    ///      // Prints: "(replacement
element) finished".
    ///
    /// This
``Publisher/replaceError(with:)``
functionality is useful when you want to
handle an error by sending a single
replacement element and end the stream.
Use ``Publisher/catch(_:)`` to recover
from an error and provide a replacement
publisher to continue providing elements
to the downstream subscriber.
    ///
    /// - Parameter output: An element to
emit when the upstream publisher fails.
    /// - Returns: A publisher that
replaces an error from the upstream
publisher with the provided output
element.
    public func replaceError(with output:
Self.Output) ->
Publishers.ReplaceError<Self>

    /// Replaces an empty stream with the
provided element.
```

```
    ///
    /// Use
``Publisher/replaceEmpty(with:)`` to
provide a replacement element if the
upstream publisher finishes without
producing any elements.
    ///
    /// In the example below, the empty
`Double` array publisher doesn’t produce
any elements, so
``Publisher/replaceEmpty(with:)``
publishes `Double.nan` and finishes
normally.
    ///
    ///     let numbers: [Double] = []
    ///     cancellable =
numbers.publisher
    ///         .replaceEmpty(with:
Double.nan)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints "(nan)".
    ///
    /// Conversely, providing a non-empty
publisher publishes all elements and the
publisher then terminates normally:
    ///
    ///     let otherNumbers: [Double] =
[1.0, 2.0, 3.0]
    ///     cancellable2 =
otherNumbers.publisher
    ///         .replaceEmpty(with:
```

```
Double.nan)
    ///          .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: 1.0 2.0 3.0
    ///
    /// - Parameter output: An element to
emit when the upstream publisher finishes
without emitting any elements.
    /// - Returns: A publisher that
replaces an empty stream with the
provided output element.
    public func replaceEmpty(with output:
Self.Output) ->
Publishers.ReplaceEmpty<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Raises a fatal error when its
upstream publisher fails, and otherwise
republishes all received input.
    ///
    /// Use `assertNoFailure()` for
internal integrity checks that are active
during testing. However, it is important
to note that, like its Swift counterpart
`fatalError(_:)`, the `assertNoFailure()`
operator asserts a fatal exception when
triggered during development and testing,
_and_ in shipping versions of code.
```

```
    ///
    /// In the example below, a
`CurrentValueSubject` publishes the
initial and second values successfully.
The third value, containing a
`genericSubjectError`, causes the
`assertNoFailure()` operator to assert a
fatal exception stopping the process:
    ///
    ///     public enum SubjectError:
Error {
    ///         case genericSubjectError
    ///     }
    ///
    ///     let subject =
CurrentValueSubject<String,
Error>("initial value")
    ///     subject
    ///         .assertNoFailure()
    ///         .sink(receiveCompletion:
{ print ("completion: \($0)") },
    ///               receiveValue:
{ print ("value: \($0).") }
    ///         )
    ///
    ///     subject.send("second value")
    ///     subject.send(completion:
Subscribers.Completion<Error>.failure(Sub
jectError.genericSubjectError))
    ///
    ///     // Prints:
    ///     //  value: initial value.
    ///     //  value: second value.
```

```
    ///     //  The process then
terminates in the debugger as the
assertNoFailure operator catches the
genericSubjectError.
    ///
    /// - Parameters:
    ///   - prefix: A string used at the
beginning of the fatal error message.
    ///   - file: A filename used in the
error message. This defaults to `#file`.
    ///   - line: A line number used in
the error message. This defaults to
`#line`.
    /// - Returns: A publisher that
raises a fatal error when its upstream
publisher fails.
    public func assertNoFailure(_ prefix:
String = "", file: StaticString = #file,
line: UInt = #line) ->
Publishers.AssertNoFailure<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Ignores elements from the
upstream publisher until it receives an
element from a second publisher.
    ///
    /// Use
``Publisher/drop(untilOutputFrom:)`` to
ignore elements from the upstream
```

```
publisher until another, second,
publisher delivers its first element.
    /// This publisher requests a single
value from the second publisher, and it
ignores (drops) all elements from the
upstream publisher until the second
publisher produces a value. After the
second publisher produces an element,
``Publisher/drop(untilOutputFrom:)``
cancels its subscription to the second
publisher, and allows events from the
upstream publisher to pass through.
    ///
    /// After this publisher receives a
subscription from the upstream publisher,
it passes through backpressure requests
from downstream to the upstream
publisher. If the upstream publisher acts
on those requests before the other
publisher produces an item, this
publisher drops the elements it receives
from the upstream publisher.
    ///
    /// In the example below, the `pub1`
publisher defers publishing its elements
until the `pub2` publisher delivers its
first element:
    ///
    ///     let upstream =
PassthroughSubject<Int,Never>()
    ///     let second =
PassthroughSubject<String,Never>()
    ///     cancellable = upstream
```

```
    ///         .drop(untilOutputFrom:
second)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     upstream.send(1)
    ///     upstream.send(2)
    ///     second.send("A")
    ///     upstream.send(3)
    ///     upstream.send(4)
    ///     // Prints "3 4"
    ///
    /// - Parameter publisher: A
publisher to monitor for its first
emitted element.
    /// - Returns: A publisher that drops
elements from the upstream publisher
until the `other` publisher produces a
value.
    public func drop<P>(untilOutputFrom
publisher: P) ->
Publishers.DropUntilOutput<Self, P> where
P : Publisher, Self.Failure == P.Failure
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Performs the specified closures
when publisher events occur.
    ///
    /// Use
```

```
``Publisher/handleEvents(receiveSubscription:receiveOutput:receiveCompletion:receiveCancel:receiveRequest:)`` when you want to examine elements as they progress through the stages of the publisher's lifecycle.
    ///
    /// In the example below, a publisher of integers shows the effect of printing debugging information at each stage of the element-processing lifecycle:
    ///
    ///     let integers = (0...2)
    ///     cancellable = integers.publisher
    ///         .handleEvents(receiveSubscription: { subs in
    ///             print("Subscription: \(subs.combineIdentifier)")
    ///         }, receiveOutput: { anInt in
    ///             print("in output handler, received \(anInt)")
    ///         }, receiveCompletion: { _ in
    ///             print("in completion handler")
    ///         }, receiveCancel: {
    ///             print("received cancel")
    ///         }, receiveRequest: { (demand) in
    ///             print("received
```

```
demand: \(demand.description)")
    ///          })
    ///          .sink { _ in return }
    ///
    ///     // Prints:
    ///     //   received demand:
unlimited
    ///     //   Subscription:
0x7f81284734c0
    ///     //   in output handler,
received 0
    ///     //   in output handler,
received 1
    ///     //   in output handler,
received 2
    ///     //   in completion handler
    ///
    ///
    /// - Parameters:
    ///   - receiveSubscription: An
optional closure that executes when the
publisher receives the subscription from
the upstream publisher. This value
defaults to `nil`.
    ///   - receiveOutput: An optional
closure that executes when the publisher
receives a value from the upstream
publisher. This value defaults to `nil`.
    ///   - receiveCompletion: An
optional closure that executes when the
upstream publisher finishes normally or
terminates with an error. This value
defaults to `nil`.
```

```
    ///   - receiveCancel: An optional
closure that executes when the downstream
receiver cancels publishing. This value
defaults to `nil`.
    ///   - receiveRequest: An optional
closure that executes when the publisher
receives a request for more elements.
This value defaults to `nil`.
    /// - Returns: A publisher that
performs the specified closures when
publisher events occur.
    public func
handleEvents(receiveSubscription: ((any
Subscription) -> Void)? = nil,
receiveOutput: ((Self.Output) -> Void)? =
nil, receiveCompletion:
((Subscribers.Completion<Self.Failure>)
-> Void)? = nil, receiveCancel: (() ->
Void)? = nil, receiveRequest:
((Subscribers.Demand) -> Void)? = nil) ->
Publishers.HandleEvents<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Prefixes a publisher's output
with the specified values.
    ///
    /// Use
``Publisher/prepend(_:)-7wk5l`` when you
need to prepend specific elements before
```

```
the output of a publisher.
    ///
    /// In the example below, the
``Publisher/prepend(_:)-7wk5l`` operator
publishes the provided elements before
republishing all elements from
`dataElements`:
    ///
    ///     let dataElements = (0...10)
    ///     cancellable =
dataElements.publisher
    ///         .prepend(0, 1, 255)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 255 0 1 2 3 4
5 6 7 8 9 10"
    ///
    /// - Parameter elements: The
elements to publish before this
publisher's elements.
    /// - Returns: A publisher that
prefixes the specified elements prior to
this publisher's elements.
    public func prepend(_ elements:
Self.Output...) ->
Publishers.Concatenate<Publishers.Sequenc
e<[Self.Output], Self.Failure>, Self>

    /// Prefixes a publisher's output
with the specified sequence.
    ///
    /// Use ``Publisher/prepend(_:)-
```

```
v9sb`` to publish values from two
publishers when you need to prepend one
publisher’s elements to another.
    ///
    /// In this example the
``Publisher/prepend(_:)-v9sb`` operator
publishes the provided sequence before
republishing all elements from
`dataElements`:
    ///
    ///     let prefixValues = [0, 1,
255]
    ///     let dataElements = (0...10)
    ///     cancellable =
dataElements.publisher
    ///         .prepend(prefixValues)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 255 0 1 2 3 4
5 6 7 8 9 10"
    ///
    /// - Parameter elements: A sequence
of elements to publish before this
publisher’s elements.
    /// - Returns: A publisher that
prefixes the sequence of elements prior
to this publisher’s elements.
    public func prepend<S>(_ elements: S)
->
Publishers.Concatenate<Publishers.Sequenc
e<S, Self.Failure>, Self> where S :
Sequence, Self.Output == S.Element
```

```
    /// Prefixes the output of this
publisher with the elements emitted by
the given publisher.
    ///
    /// Use
``Publisher/prepend(_:)-5dj9c`` to
publish values from two publishers when
you need to prepend one publisher’s
elements to another.
    ///
    /// In the example below, a publisher
of `prefixValues` publishes its elements
before the `dataElements` publishes its
elements:
    ///
    ///     let prefixValues = [0, 1,
255]
    ///     let dataElements = (0...10)
    ///     cancellable =
dataElements.publisher
    ///         .prepend(prefixValues.pub
lisher)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 255 0 1 2 3 4
5 6 7 8 9 10"
    ///
    /// - Parameter publisher: The
prefixing publisher.
    /// - Returns: A publisher that
prefixes the prefixing publisher’s
```

```
elements prior to this publisher’s
elements.
    public func prepend<P>(_ publisher:
P) -> Publishers.Concatenate<P, Self>
where P : Publisher, Self.Failure ==
P.Failure, Self.Output == P.Output

    /// Appends a publisher’s output with
the specified elements.
    ///
    /// Use
``Publisher/append(_:)-1qb8d`` when you
need to prepend specific elements after
the output of a publisher.
    ///
    /// In the example below, the
``Publisher/append(_:)-1qb8d`` operator
publishes the provided elements after
republishing all elements from
`dataElements`:
    ///
    ///     let dataElements = (0...10)
    ///     cancellable =
dataElements.publisher
    ///         .append(0, 1, 255)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 2 3 4 5 6 7 8
9 10 0 1 255"
    ///
    ///
    /// - Parameter elements: Elements to
```

```
publish after this publisher’s elements.
    /// - Returns: A publisher that
appends the specifiecd elements after
this publisher’s elements.
    public func append(_ elements:
Self.Output...) ->
Publishers.Concatenate<Self,
Publishers.Sequence<[Self.Output],
Self.Failure>>

    /// Appends a publisher’s output with
the specified sequence.
    ///
    /// Use
``Publisher/append(_:)-69sdn`` to append
a sequence to the end of a publisher’s
output.
    ///
    /// In the example below, the
``Publisher/append(_:)-69sdn`` publisher
republishes all elements from
`groundTransport` until it finishes, then
publishes the members of `airTransport`:
    ///
    ///     let groundTransport = ["car",
"bus", "truck", "subway", "bicycle"]
    ///     let airTransport =
["parasail", "jet", "helicopter",
"rocket"]
    ///     cancellable =
groundTransport.publisher
    ///         .append(airTransport)
    ///         .sink { print("\($0)",
```

```
terminator: " ") }
    ///
    ///     // Prints: "car bus truck
subway bicycle parasail jet helicopter
rocket"
    ///
    /// - Parameter elements: A sequence
of elements to publish after this
publisher's elements.
    /// - Returns: A publisher that
appends the sequence of elements after
this publisher's elements.
    public func append<S>(_ elements: S)
-> Publishers.Concatenate<Self,
Publishers.Sequence<S, Self.Failure>>
where S : Sequence, Self.Output ==
S.Element

    /// Appends the output of this
publisher with the elements emitted by
the given publisher.
    ///
    /// Use
``Publisher/append(_:)-5yh02`` to append
the output of one publisher to another.
The ``Publisher/append(_:)-5yh02``
operator produces no elements until this
publisher finishes. It then produces this
publisher's elements, followed by the
given publisher's elements. If this
publisher fails with an error, the given
publishers elements aren't published.
    ///
```

```
    /// In the example below, the
`append` publisher republishes all
elements from the `numbers` publisher
until it finishes, then publishes all
elements from the `otherNumbers`
publisher:
    ///
    ///     let numbers = (0...10)
    ///     let otherNumbers = (25...35)
    ///     cancellable =
numbers.publisher
    ///         .append(otherNumbers.publ
isher)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "0 1 2 3 4 5 6 7 8
9 10 25 26 27 28 29 30 31 32 33 34 35 "
    ///
    /// - Parameter publisher: The
appending publisher.
    /// - Returns: A publisher that
appends the appending publisher’s
elements after this publisher’s elements.
    public func append<P>(_ publisher: P)
-> Publishers.Concatenate<Self, P> where
P : Publisher, Self.Failure == P.Failure,
Self.Output == P.Output
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {
```

```
    /// Publishes elements only after a
specified time interval elapses between
events.
    ///
    /// Use the
``Publisher/debounce(for:scheduler:option
s:)`` operator to control the number of
values and time between delivery of
values from the upstream publisher. This
operator is useful to process bursty or
high-volume event streams where you need
to reduce the number of values delivered
to the downstream to a rate you specify.
    ///
    /// In this example, a
``PassthroughSubject`` publishes elements
on a schedule defined by the `bounces`
array. The array is composed of tuples
representing a value sent by the
``PassthroughSubject``, and a
<doc://com.apple.documentation/documentat
ion/Foundation/TimeInterval> ranging from
one-quarter second up to 2 seconds that
drives a delivery timer. As the queue
builds, elements arriving faster than
one-half second `debounceInterval` are
discarded, while elements arriving at a
rate slower than `debounceInterval` are
passed through to the
``Publisher/sink(receiveValue:)``
operator.
    ///
```

```
    ///     let bounces:
[(Int,TimeInterval)] = [
    ///         (0, 0),
    ///         (1, 0.25),  // 0.25s
interval since last index
    ///         (2, 1),     // 0.75s
interval since last index
    ///         (3, 1.25),  // 0.25s
interval since last index
    ///         (4, 1.5),   // 0.25s
interval since last index
    ///         (5, 2)      // 0.5s
interval since last index
    ///     ]
    ///
    ///     let subject =
PassthroughSubject<Int, Never>()
    ///     cancellable = subject
    ///         .debounce(for: .seconds(0
.5), scheduler: RunLoop.main)
    ///         .sink { index in
    ///             print ("Received
index \(index)")
    ///         }
    ///
    ///     for bounce in bounces {
    ///
DispatchQueue.main.asyncAfter(deadline: .
now() + bounce.1) {
    ///
subject.send(bounce.0)
    ///         }
    ///     }
```

```
    ///
    ///     // Prints:
    ///     //  Received index 1
    ///     //  Received index 4
    ///     //  Received index 5
    ///
    ///     //  Here is the event flow shown from the perspective of time, showing value delivery through the `debounce()` operator:
    ///
    ///     //  Time 0: Send index 0.
    ///     //  Time 0.25: Send index 1. Index 0 was waiting and is discarded.
    ///     //  Time 0.75: Debounce period ends, publish index 1.
    ///     //  Time 1: Send index 2.
    ///     //  Time 1.25: Send index 3. Index 2 was waiting and is discarded.
    ///     //  Time 1.5: Send index 4. Index 3 was waiting and is discarded.
    ///     //  Time 2: Debounce period ends, publish index 4. Also, send index 5.
    ///     //  Time 2.5: Debounce period ends, publish index 5.
    ///
    /// - Parameters:
    ///   - dueTime: The time the publisher should wait before publishing an element.
    ///   - scheduler: The scheduler on which this publisher delivers elements
```

```
    ///   - options: Scheduler options
that customize this publisher’s delivery
of elements.
    /// - Returns: A publisher that
publishes events only after a specified
time elapses.
    public func debounce<S>(for dueTime:
S.SchedulerTimeType.Stride, scheduler: S,
options: S.SchedulerOptions? = nil) ->
Publishers.Debounce<Self, S> where S :
Scheduler
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes the last element of a
stream, after the stream finishes.
    ///
    /// Use ``Publisher/last()`` when you
need to emit only the last element from
an upstream publisher.
    ///
    /// In the example below, the range
publisher only emits the last element
from the sequence publisher, `10`, then
finishes normally.
    ///
    ///     let numbers = (-10...10)
    ///     cancellable =
numbers.publisher
    ///         .last()
```

```
    ///          .sink { print("\($0)") }
    ///
    ///     // Prints: "10"
    ///
    /// - Returns: A publisher that only
publishes the last element of a stream.
    public func last() ->
Publishers.Last<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Transforms all elements from the
upstream publisher with a provided
closure.
    ///
    /// Combine's
``Publisher/map(_:)-99evh`` operator
performs a function similar to that of
<doc://com.apple.documentation/documentat
ion/Swift/Sequence/map(_:)> in the Swift
standard library: it uses a closure to
transform each element it receives from
the upstream publisher. You use
``Publisher/map(_:)-99evh`` to transform
from one kind of element to another.
    ///
    /// The following example uses an
array of numbers as the source for a
collection based publisher. A
``Publisher/map(_:)-99evh`` operator
```

```
consumes each integer from the publisher
and uses a dictionary to transform it
from its Arabic numeral to a Roman
equivalent, as a
<doc://com.apple.documentation/documentat
ion/Swift/String>.
    /// If the
``Publisher/map(_:)-99evh``’s closure
fails to look up a Roman numeral, it
returns the string `(unknown)`.
    ///
    ///     let numbers = [5, 4, 3, 2, 1,
0]
    ///     let romanNumeralDict: [Int :
String] =
    ///         [1:"I", 2:"II", 3:"III",
4:"IV", 5:"V"]
    ///     cancellable =
numbers.publisher
    ///             .map
{ romanNumeralDict[$0] ?? "(unknown)" }
    ///             .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "V IV III II I
(unknown)"
    ///
    /// If your closure can throw an
error, use Combine’s
``Publisher/tryMap(_:)`` operator
instead.
    ///
    /// - Parameter transform: A closure
```

```
that takes one element as its parameter
and returns a new element.
    /// - Returns: A publisher that uses
the provided closure to map elements from
the upstream publisher to new elements
that it then publishes.
    public func map<T>(_ transform:
@escaping (Self.Output) -> T) ->
Publishers.Map<Self, T>

    /// Transforms all elements from the
upstream publisher with a provided error-
throwing closure.
    ///
    /// Combine's
``Publisher/tryMap(_:)`` operator
performs a function similar to that of
<doc://com.apple.documentation/documentat
ion/Swift/Sequence/map(_:)> in the Swift
standard library: it uses a closure to
transform each element it receives from
the upstream publisher. You use
``Publisher/tryMap(_:)`` to transform
from one kind of element to another, and
to terminate publishing when the map's
closure throws an error.
    ///
    /// The following example uses an
array of numbers as the source for a
collection based publisher. A
``Publisher/tryMap(_:)`` operator
consumes each integer from the publisher
and uses a dictionary to transform it
```

```
from its Arabic numeral to a Roman
equivalent, as a
<doc://com.apple.documentation/documentat
ion/Swift/String>.
    /// If the ``Publisher/tryMap(_:)``'s
closure fails to look up a Roman numeral,
it throws an error. The
``Publisher/tryMap(_:)`` operator catches
this error and terminates publishing,
sending a
``Subscribers/Completion/failure(_:)``
that wraps the error.
    ///
    ///     struct ParseError: Error {}
    ///     func romanNumeral(from:Int)
throws -> String {
    ///         let romanNumeralDict:
[Int : String] =
    ///             [1:"I", 2:"II",
3:"III", 4:"IV", 5:"V"]
    ///         guard let numeral =
romanNumeralDict[from] else {
    ///             throw ParseError()
    ///         }
    ///         return numeral
    ///     }
    ///     let numbers = [5, 4, 3, 2, 1,
0]
    ///     cancellable =
numbers.publisher
    ///         .tryMap { try
romanNumeral(from: $0) }
    ///         .sink(
```

```
    ///             receiveCompletion:
{ print ("completion: \($0)") },
    ///             receiveValue: { print
("\($0)", terminator: " ") }
    ///         )
    ///
    ///     // Prints: "V IV III II I
completion: failure(ParseError())"
    ///
    /// If your closure doesn’t throw,
use ``Publisher/map(_:)-99evh`` instead.
    ///
    /// - Parameter transform: A closure
that takes one element as its parameter
and returns a new element. If the closure
throws an error, the publisher fails with
the thrown error.
    /// - Returns: A publisher that uses
the provided closure to map elements from
the upstream publisher to new elements
that it then publishes.
    public func tryMap<T>(_ transform:
@escaping (Self.Output) throws -> T) ->
Publishers.TryMap<Self, T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Terminates publishing if the
upstream publisher exceeds the specified
time interval without producing an
```

```
element.
    ///
    /// Use
``Publisher/timeout(_:scheduler:options:c
ustomError:)`` to terminate a publisher
if an element isn’t delivered within a
timeout interval you specify.
    ///
    /// In the example below, a
``PassthroughSubject`` publishes
<doc://com.apple.documentation/documentat
ion/Swift/String> elements and is
configured to time out if no new elements
are received within its `TIME_OUT` window
of 5 seconds. A single value is published
after the specified 2-second `WAIT_TIME`,
after which no more elements are
available; the publisher then times out
and completes normally.
    ///
    ///     var WAIT_TIME : Int = 2
    ///     var TIMEOUT_TIME : Int = 5
    ///
    ///     let subject =
PassthroughSubject<String, Never>()
    ///     let cancellable = subject
    ///         .timeout(.seconds(TIMEOUT
_TIME), scheduler: DispatchQueue.main,
options: nil, customError:nil)
    ///         .sink(
    ///               receiveCompletion:
{ print ("completion: \($0) at \
(Date())") },
```

```
    ///                 receiveValue:
{ print ("value: \($0) at \(Date())") }
    ///             )
    ///
    ///
DispatchQueue.main.asyncAfter(deadline: .
now() + .seconds(WAIT_TIME),
    ///
execute: { subject.send("Some data - sent
after a delay of \(WAIT_TIME) seconds") }
)
    ///
    ///     // Prints: value: Some data -
sent after a delay of 2 seconds at 2020-
03-10 23:47:59 +0000
    ///     //          completion:
finished at 2020-03-10 23:48:04 +0000
    ///
    ///
    /// If `customError` is `nil`, the
publisher completes normally; if you
provide a closure for the `customError`
argument, the upstream publisher is
instead terminated upon timeout, and the
error is delivered to the downstream.
    ///
    /// - Parameters:
    ///   - interval: The maximum time
interval the publisher can go without
emitting an element, expressed in the
time system of the scheduler.
    ///   - scheduler: The scheduler on
which to deliver events.
```

```
    ///   - options: Scheduler options
that customize the delivery of elements.
    ///   - customError: A closure that
executes if the publisher times out. The
publisher sends the failure returned by
this closure to the subscriber as the
reason for termination.
    /// - Returns: A publisher that
terminates if the specified interval
elapses with no events received from the
upstream publisher.
    public func timeout<S>(_ interval:
S.SchedulerTimeType.Stride, scheduler: S,
options: S.SchedulerOptions? = nil,
customError: (() -> Self.Failure)? = nil)
-> Publishers.Timeout<Self, S> where S :
Scheduler
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Buffers elements received from an
upstream publisher.
    ///
    /// Use
``Publisher/buffer(size:prefetch:whenFull
:)`` to collect a specific number of
elements from an upstream publisher
before republishing them to the
downstream subscriber according to the
``Publishers/BufferingStrategy`` and
```

```
``Publishers/PrefetchStrategy`` strategy
you specify.
    ///
    /// If the publisher completes before
reaching the `size` threshold, it buffers
the elements and publishes them
downstream prior to completion.
    ///
    /// - Parameters:
    ///   - size: The maximum number of
elements to store.
    ///   - prefetch: The strategy to
initially populate the buffer.
    ///   - whenFull: The action to take
when the buffer becomes full.
    /// - Returns: A publisher that
buffers elements received from an
upstream publisher.
    public func buffer(size: Int,
prefetch: Publishers.PrefetchStrategy,
whenFull:
Publishers.BufferingStrategy<Self.Failure
>) -> Publishers.Buffer<Self>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Combines elements from another
publisher and deliver pairs of elements
as tuples.
    ///
```

```
    /// Use ``Publisher/zip(_:)`` to
combine the latest elements from two
publishers and emit a tuple to the
downstream. The returned publisher waits
until both publishers have emitted an
event, then delivers the oldest
unconsumed event from each publisher
together as a tuple to the subscriber.
    ///
    /// Much like a zipper or zip
fastener on a piece of clothing pulls
together rows of teeth to link the two
sides, ``Publisher/zip(_:)`` combines
streams from two different publishers by
linking pairs of elements from each side.
    ///
    /// In this example, `numbers` and
`letters` are ``PassthroughSubject``s
that emit values; once
``Publisher/zip(_:)`` receives one value
from each, it publishes the pair as a
tuple to the downstream subscriber. It
then waits for the next pair of values.
    ///
    ///      let numbersPub =
PassthroughSubject<Int, Never>()
    ///      let lettersPub =
PassthroughSubject<String, Never>()
    ///
    ///      cancellable = numbersPub
    ///          .zip(lettersPub)
    ///          .sink { print("\($0)") }
    ///      numbersPub.send(1)    //
```

```
numbersPub: 1      lettersPub:         zip
output: <none>
    ///      numbersPub.send(2)    //
numbersPub: 1,2    lettersPub:         zip
output: <none>
    ///      letters.send("A")     //
numbers: 1,2       letters:"A"         zip
output: <none>
    ///      numbers.send(3)       //
numbers: 1,2,3     letters:            zip
output: (1,"A")
    ///      letters.send("B")     //
numbers: 1,2,3     letters: "B"        zip
output: (2,"B")
    ///
    ///      // Prints:
    ///      //  (1, "A")
    ///      //  (2, "B")
    ///
    /// If either upstream publisher
finishes successfully or fails with an
error, the zipped publisher does the
same.
    ///
    /// - Parameter other: Another
publisher.
    /// - Returns: A publisher that emits
pairs of elements from the upstream
publishers as tuples.
    public func zip<P>(_ other: P) ->
Publishers.Zip<Self, P> where P :
Publisher, Self.Failure == P.Failure
```

```
    /// Combines elements from another
publisher and delivers a transformed
output.
    ///
    /// Use ``Publisher/zip(_:_:)-4xn21``
to return a new publisher that combines
the elements from two publishers using a
transformation you specify to publish a
new value to the downstream.  The
returned publisher waits until both
publishers have emitted an event, then
delivers the oldest unconsumed event from
each publisher together that the operator
uses in the transformation.
    ///
    /// In this example,
``PassthroughSubject`` instances
`numbersPub` and `lettersPub` emit
values; ``Publisher/zip(_:_:)-4xn21``
receives the oldest value from each
publisher, uses the `Int` from
`numbersPub` and publishes a string that
repeats the
<doc://com.apple.documentation/documentat
ion/Swift/String> from `lettersPub` that
many times.
    ///
    ///     let numbersPub =
PassthroughSubject<Int, Never>()
    ///     let lettersPub =
PassthroughSubject<String, Never>()
    ///     cancellable = numbersPub
    ///         .zip(lettersPub) { anInt,
```

```
aLetter in
    ///             String(repeating:
aLetter, count: anInt)
    ///         }
    ///         .sink { print("\($0)") }
    ///     numbersPub.send(1)     //
numbersPub: 1      lettersPub:       zip
output: <none>
    ///     numbersPub.send(2)     //
numbersPub: 1,2    lettersPub:       zip
output: <none>
    ///     numbersPub.send(3)     //
numbersPub: 1,2,3  lettersPub:       zip
output: <none>
    ///     lettersPub.send("A")   //
numbersPub: 1,2,3  lettersPub: "A"   zip
output: "A"
    ///     lettersPub.send("B")   //
numbersPub: 2,3    lettersPub: "B"   zip
output: "BB"
    ///     // Prints:
    ///     //  A
    ///     //  BB
    ///
    /// If either upstream publisher
finishes successfully or fails with an
error, the zipped publisher does the
same.
    ///
    /// - Parameters:
    ///   - other: Another publisher.
    ///   - transform: A closure that
receives the most-recent value from each
```

```
publisher and returns a new value to
publish.
    /// - Returns: A publisher that uses
the `transform` closure to emit new
elements, produced by combining the most
recent value from two upstream
publishers.
    public func zip<P, T>(_ other: P, _
transform: @escaping (Self.Output,
P.Output) -> T) ->
Publishers.Map<Publishers.Zip<Self, P>,
T> where P : Publisher, Self.Failure ==
P.Failure

    /// Combines elements from two other
publishers and delivers groups of
elements as tuples.
    ///
    /// Use ``Publisher/zip(_:_:)-8d7k7``
to return a new publisher that combines
the elements from two additional
publishers to publish a tuple to the
downstream. The returned publisher waits
until all three publishers have emitted
an event, then delivers the oldest
unconsumed event from each publisher as a
tuple to the subscriber.
    ///
    /// In this example, `numbersPub`,
`lettersPub` and `emojiPub` are each a
``PassthroughSubject``;
    /// ``Publisher/zip(_:_:)-8d7k7``
receives the oldest unconsumed value from
```

```
each publisher and combines them into a
tuple that it republishes to the
downstream:
    ///
    ///     let numbersPub =
PassthroughSubject<Int, Never>()
    ///     let lettersPub =
PassthroughSubject<String, Never>()
    ///     let emojiPub =
PassthroughSubject<String, Never>()
    ///
    ///     cancellable = numbersPub
    ///         .zip(lettersPub,
emojiPub)
    ///         .sink { print("\($0)") }
    ///     numbersPub.send(1)     //
numbersPub: 1      lettersPub:
emojiPub:        zip output: <none>
    ///     numbersPub.send(2)     //
numbersPub: 1,2    lettersPub:
emojiPub:        zip output: <none>
    ///     numbersPub.send(3)     //
numbersPub: 1,2,3  lettersPub:
emojiPub:        zip output: <none>
    ///     lettersPub.send("A")   //
numbersPub: 1,2,3  lettersPub: "A"
emojiPub:        zip output: <none>
    ///     emojiPub.send("😀")    //
numbersPub: 2,3    lettersPub: "A"
emojiPub: "😀"   zip output: (1, "A",
"😀")
    ///     lettersPub.send("B")   //
numbersPub: 2,3    lettersPub: "B"
```

```
emojiPub:         zip output: <none>
    ///     emojiPub.send("🥰")    //
numbersPub: 3       lettersPub:
emojiPub:         zip output: (2, "B",
"🥰")
    ///
    ///     // Prints:
    ///     //  (1, "A", "😀")
    ///     //  (2, "B", "🥰")
    ///
    /// If any upstream publisher
finishes successfully or fails with an
error, so too does the zipped publisher.
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher.
    ///   - publisher2: A third
publisher.
    /// - Returns: A publisher that emits
groups of elements from the upstream
publishers as tuples.
    public func zip<P, Q>(_ publisher1:
P, _ publisher2: Q) ->
Publishers.Zip3<Self, P, Q> where P :
Publisher, Q : Publisher, Self.Failure ==
P.Failure, P.Failure == Q.Failure

    /// Combines elements from two other
publishers and delivers a transformed
output.
    ///
    /// Use
```

```
``Publisher/zip(_:_:_:)-9yqi1`` to return
a new publisher that combines the
elements from two other publishers using
a transformation you specify to publish a
new value to the downstream subscriber.
The returned publisher waits until all
three publishers have emitted an event,
then delivers the oldest unconsumed event
from each publisher together that the
operator uses in the transformation.
    ///
    /// In this example, `numbersPub`,
`lettersPub` and `emojiPub` are each a
``PassthroughSubject`` that emit values;
``Publisher/zip(_:_:_:)-9yqi1`` receives
the oldest value from each publisher and
uses the `Int` from `numbersPub` and
publishes a string that repeats the
<doc://com.apple.documentation/documentat
ion/Swift/String> from `lettersPub` and
`emojiPub` that many times.
    ///
    ///     let numbersPub =
PassthroughSubject<Int, Never>()
    ///     let lettersPub =
PassthroughSubject<String, Never>()
    ///     let emojiPub =
PassthroughSubject<String, Never>()
    ///
    ///     cancellable = numbersPub
    ///         .zip(letters, emoji)
{ anInt, aLetter, anEmoji in
    ///             ("\(String(repeating:
```

```
anEmoji, count: anInt)) \
(String(repeating: aLetter, count:
anInt))")
    ///         }
    ///         .sink { print("\($0)") }
    ///
    ///     numbersPub.send(1)     //
numbersPub: 1      lettersPub:
emojiPub:            zip output: <none>
    ///     numbersPub.send(2)     //
numbersPub: 1,2    lettersPub:
emojiPub:            zip output: <none>
    ///     numbersPub.send(3)     //
numbersPub: 1,2,3  lettersPub:
emojiPub:            zip output: <none>
    ///     lettersPub.send("A")   //
numbersPub: 1,2,3  lettersPub: "A"
emojiPub:            zip output: <none>
    ///     emojiPub.send("😀")    //
numbersPub: 2,3    lettersPub: "A"
emojiPub:"😀"         zip output: "😀 A"
    ///     lettersPub.send("B")   //
numbersPub: 2,3    lettersPub: "B"
emojiPub:            zip output: <none>
    ///     emojiPub.send("🥰")    //
numbersPub: 3      lettersPub:
emojiPub:"😀", "🥰"  zip output: "🥰🥰 BB"
    ///
    ///     // Prints:
    ///     // 😀 A
    ///     // 🥰🥰 BB
    ///
    /// If any upstream publisher
```

```
finishes successfully or fails with an
error, so too does the zipped publisher.
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher.
    ///   - publisher2: A third
publisher.
    ///   - transform: A closure that
receives the most-recent value from each
publisher and returns a new value to
publish.
    /// - Returns: A publisher that uses
the `transform` closure to emit new
elements, produced by combining the most
recent value from three upstream
publishers.
    public func zip<P, Q, T>(_
publisher1: P, _ publisher2: Q, _
transform: @escaping (Self.Output,
P.Output, Q.Output) -> T) ->
Publishers.Map<Publishers.Zip3<Self, P,
Q>, T> where P : Publisher, Q :
Publisher, Self.Failure == P.Failure,
P.Failure == Q.Failure

    /// Combines elements from three
other publishers and delivers groups of
elements as tuples.
    ///
    /// Use
``Publisher/zip(_:_:_:)-16rcy`` to return
a new publisher that combines the
```

```
elements from three other publishers to
publish a tuple to the downstream
subscriber. The returned publisher waits
until all four publishers have emitted an
event, then delivers the oldest
unconsumed event from each publisher as a
tuple to the subscriber.
    ///
    /// In this example, several
``PassthroughSubject`` instances emit
values; ``Publisher/zip(_:_:_:)-16rcy``
receives the oldest unconsumed value from
each publisher and combines them into a
tuple that it republishes to the
downstream:
    ///
    ///     let numbersPub =
PassthroughSubject<Int, Never>()
    ///     let lettersPub =
PassthroughSubject<String, Never>()
    ///     let emojiPub =
PassthroughSubject<String, Never>()
    ///     let fractionsPub  =
PassthroughSubject<Double, Never>()
    ///
    ///     cancellable = numbersPub
    ///         .zip(lettersPub,
emojiPub, fractionsPub)
    ///         .sink { print("\($0)") }
    ///     numbersPub.send(1)         //
numbersPub: 1       lettersPub:
emojiPub:       fractionsPub:         zip
output: <none>
```

```
    ///     numbersPub.send(2)          //
numbersPub: 1,2     lettersPub:
emojiPub:       fractionsPub:           zip
output: <none>
    ///     numbersPub.send(3)          //
numbersPub: 1,2,3   lettersPub:
emojiPub:       fractionsPub:           zip
output: <none>
    ///     fractionsPub.send(0.1)      //
numbersPub: 1,2,3   lettersPub: "A"
emojiPub:       fractionsPub: 0.1       zip
output: <none>
    ///     lettersPub.send("A")        //
numbersPub: 1,2,3   lettersPub: "A"
emojiPub:       fractionsPub: 0.1       zip
output: <none>
    ///     emojiPub.send("😀")         //
numbersPub: 2,3     lettersPub: "A"
emojiPub: "😀"  fractionsPub: 0.1       zip
output: (1, "A", "😀", 0.1)
    ///     lettersPub.send("B")        //
numbersPub: 2,3     lettersPub: "B"
emojiPub:       fractionsPub:           zip
output: <none>
    ///     fractionsPub.send(0.8)      //
numbersPub: 2,3     lettersPub: "B"
emojiPub:       fractionsPub: 0.8       zip
output: <none>
    ///     emojiPub.send("🥰")         //
numbersPub: 3       lettersPub: "B"
emojiPub:       fractionsPub: 0.8       zip
output: (2, "B", "🥰", 0.8)
    ///     // Prints:
```

```
    ///     //  (1, "A", "😀", 0.1)
    ///     //  (2, "B", "🥰", 0.8)
    ///
    ///
    /// If any upstream publisher
finishes successfully or fails with an
error, so too does the zipped publisher.
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher.
    ///   - publisher2: A third
publisher.
    ///   - publisher3: A fourth
publisher.
    /// - Returns: A publisher that emits
groups of elements from the upstream
publishers as tuples.
    public func zip<P, Q, R>(_
publisher1: P, _ publisher2: Q, _
publisher3: R) -> Publishers.Zip4<Self,
P, Q, R> where P : Publisher, Q :
Publisher, R : Publisher, Self.Failure ==
P.Failure, P.Failure == Q.Failure,
Q.Failure == R.Failure

    /// Combines elements from three
other publishers and delivers a
transformed output.
    ///
    /// Use ``Publisher/zip(_:_:_:)``
to return a new publisher that combines
the elements from three other publishers
```

```
using a transformation you specify to
publish a new value to the downstream
subscriber. The returned publisher waits
until all four publishers have emitted an
event, then delivers the oldest
unconsumed event from each publisher
together that the operator uses in the
transformation.
    ///
    /// In this example, the
``PassthroughSubject`` publishers,
`numbersPub`,
    /// `fractionsPub`, `lettersPub`, and
`emojiPub` emit values. The
``Publisher/zip(_:_:_:_:)`` operator
receives the oldest value from each
publisher and uses the `Int` from
`numbersPub` and publishes a string that
repeats the
<doc://com.apple.documentation/documentat
ion/Swift/String> from `lettersPub` and
`emojiPub` that many times and prints out
the value in `fractionsPub`.
    ///
    ///     let numbersPub =
PassthroughSubject<Int, Never>()      //
first publisher
    ///     let lettersPub =
PassthroughSubject<String, Never>()   //
second
    ///     let emojiPub =
PassthroughSubject<String,
Never>()     // third
```

```
    ///     let fractionsPub  =
PassthroughSubject<Double, Never>()//
fourth
    ///
    ///     cancellable = numbersPub
    ///         .zip(lettersPub,
emojiPub, fractionsPub) { anInt, aLetter,
anEmoji, aFraction  in
    ///             ("\(String(repeating:
anEmoji, count: anInt)) \
(String(repeating: aLetter, count:
anInt)) \(aFraction)")
    ///         }
    ///         .sink { print("\($0)") }
    ///
    ///     numbersPub.send(1)          //
numbersPub: 1       lettersPub:
emojiPub:           zip output: <none>
    ///     numbersPub.send(2)          //
numbersPub: 1,2     lettersPub:
emojiPub:           zip output: <none>
    ///     numbersPub.send(3)          //
numbersPub: 1,2,3   lettersPub:
emojiPub:           zip output: <none>
    ///     fractionsPub.send(0.1)      //
numbersPub: 1,2,3   lettersPub: "A"
emojiPub:           zip output: <none>
    ///     lettersPub.send("A")        //
numbersPub: 1,2,3   lettersPub: "A"
emojiPub:           zip output: <none>
    ///     emojiPub.send("😀")         //
numbersPub: 1,2,3   lettersPub: "A"
emojiPub:"😀"       zip output: "😀 A"
```

```
    ///     lettersPub.send("B")       //
numbersPub: 2,3     lettersPub: "B"
emojiPub:           zip output: <none>
    ///     fractionsPub.send(0.8)     //
numbersPub: 2,3     lettersPub: "A"
emojiPub:           zip output: <none>
    ///     emojiPub.send("🥰")        //
numbersPub: 3       lettersPub: "B"
emojiPub:           zip output: "🥰🥰 BB"
    ///     // Prints:
    ///     //1 😀 A 0.1
    ///     //2 🥰🥰 BB 0.8
    ///
    /// If any upstream publisher
finishes successfully or fails with an
error, so too does the zipped publisher.
    ///
    /// - Parameters:
    ///   - publisher1: A second
publisher.
    ///   - publisher2: A third
publisher.
    ///   - publisher3: A fourth
publisher.
    ///   - transform: A closure that
receives the most-recent value from each
publisher and returns a new value to
publish.
    /// - Returns: A publisher that uses
the `transform` closure to emit new
elements, produced by combining the most
recent value from four upstream
publishers.
```

```
    public func zip<P, Q, R, T>(_
publisher1: P, _ publisher2: Q, _
publisher3: R, _ transform: @escaping
(Self.Output, P.Output, Q.Output,
R.Output) -> T) ->
Publishers.Map<Publishers.Zip4<Self, P,
Q, R>, T> where P : Publisher, Q :
Publisher, R : Publisher, Self.Failure ==
P.Failure, P.Failure == Q.Failure,
Q.Failure == R.Failure
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes a specific element,
indicated by its index in the sequence of
published elements.
    ///
    /// Use ``Publisher/output(at:)``
when you need to republish a specific
element specified by its position in the
stream. If the publisher completes
normally or with an error before
publishing the specified element, then
the publisher doesn’t produce any
elements.
    ///
    /// In the example below, the array
publisher emits the fifth element in the
sequence of published elements:
    ///
```

```
    ///     let numbers = [1, 2, 3, 4, 5,
6, 7, 8, 9, 10]
    ///     numbers.publisher
    ///         .output(at: 5)
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "6"
    ///
    /// - Parameter index: The index that
indicates the element to publish.
    /// - Returns: A publisher that
publishes a specific indexed element.
    public func output(at index: Int) ->
Publishers.Output<Self>

    /// Publishes elements specified by
their range in the sequence of published
elements.
    ///
    /// Use ``Publisher/output(in:)`` to
republish a range indices you specify in
the published stream. After publishing
all elements, the publisher finishes
normally. If the publisher completes
normally or with an error before
producing all the elements in the range,
it doesn't publish the remaining
elements.
    ///
    /// In the example below, an array
publisher emits the subset of elements at
the indices in the specified range:
    ///
```

```
    ///     let numbers = [1, 1, 2, 2, 2,
3, 4, 5, 6]
    ///     numbers.publisher
    ///         .output(in: (3...5))
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "2 2 3"
    ///
    /// - Parameter range: A range that
indicates which elements to publish.
    /// - Returns: A publisher that
publishes elements specified by a range.
    public func output<R>(in range: R) ->
Publishers.Output<Self> where R :
RangeExpression, R.Bound == Int
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Handles errors from an upstream
publisher by replacing it with another
publisher.
    ///
    /// Use `catch()` to replace an error
from an upstream publisher with a new
publisher.
    ///
    /// In the example below, the
`catch()` operator handles the
`SimpleError` thrown by the upstream
```

```
publisher by replacing the error with a
`Just` publisher. This continues the
stream by publishing a single value and
completing normally.
    ///
    ///     struct SimpleError: Error {}
    ///     let numbers = [5, 4, 3, 2, 1,
0, 9, 8, 7, 6]
    ///     cancellable =
numbers.publisher
    ///         .tryLast(where: {
    ///             guard $0 != 0 else
{throw SimpleError()}
    ///             return true
    ///         })
    ///         .catch({ (error) in
    ///             Just(-1)
    ///         })
    ///         .sink { print("\($0)") }
    ///         // Prints: -1
    ///
    /// Backpressure note: This publisher
passes through `request` and `cancel` to
the upstream. After receiving an error,
the publisher sends sends any unfulfilled
demand to the new `Publisher`.
    /// SeeAlso: `replaceError`
    /// - Parameter handler: A closure
that accepts the upstream failure as
input and returns a publisher to replace
the upstream publisher.
    /// - Returns: A publisher that
handles errors from an upstream publisher
```

```
by replacing the failed publisher with
another publisher.
    public func `catch`<P>(_ handler:
@escaping (Self.Failure) -> P) ->
Publishers.Catch<Self, P> where P :
Publisher, Self.Output == P.Output

    /// Handles errors from an upstream
publisher by either replacing it with
another publisher or throwing a new
error.
    ///
    /// Use ``Publisher/tryCatch(_:)`` to
decide how to handle from an upstream
publisher by either replacing the
publisher with a new publisher, or
throwing a new error.
    ///
    /// In the example below, an array
publisher emits values that a
``Publisher/tryMap(_:)`` operator
evaluates to ensure the values are
greater than zero. If the values aren't
greater than zero, the operator throws an
error to the downstream subscriber to let
it know there was a problem. The
subscriber, ``Publisher/tryCatch(_:)``,
replaces the error with a new publisher
using ``Just`` to publish a final value
before the stream ends normally.
    ///
    ///     enum SimpleError: Error
{ case error }
```

```
    ///     var numbers = [5, 4, 3, 2, 1,
-1, 7, 8, 9, 10]
    ///
    ///     cancellable =
numbers.publisher
    ///         .tryMap { v in
    ///             if v > 0 {
    ///                 return v
    ///             } else {
    ///                 throw
SimpleError.error
    ///             }
    ///     }
    ///       .tryCatch { error in
    ///           Just(0) // Send a final
value before completing normally.
    ///                   //
Alternatively, throw a new error to
terminate the stream.
    ///     }
    ///       .sink(receiveCompletion:
{ print ("Completion: \($0).") },
    ///             receiveValue: { print
("Received \($0).") }
    ///       )
    ///     //    Received 5.
    ///     //    Received 4.
    ///     //    Received 3.
    ///     //    Received 2.
    ///     //    Received 1.
    ///     //    Received 0.
    ///     //    Completion: finished.
    ///
```

```
    /// - Parameter handler: A throwing
closure that accepts the upstream failure
as input. This closure can either replace
the upstream publisher with a new one, or
throw a new error to the downstream
subscriber.
    /// - Returns: A publisher that
handles errors from an upstream publisher
by replacing the failed publisher with
another publisher, or an error.
    public func tryCatch<P>(_ handler:
@escaping (Self.Failure) throws -> P) ->
Publishers.TryCatch<Self, P> where P :
Publisher, Self.Output == P.Output
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Transforms all elements from an
upstream publisher into a new publisher
up to a maximum number of publishers you
specify.
    ///
    /// Combine‘s
`flatMap(maxPublishers:_:)` operator
performs a similar function to the
<doc://com.apple.documentation/documentat
ion/Swift/Sequence/flatMap(_:)-jo2y>
operator in the Swift standard library,
but turns the elements from one kind of
publisher into a new publisher that is
```

```
sent to subscribers. Use
`flatMap(maxPublishers:_:)` when you want
to create a new series of events for
downstream subscribers based on the
received value. The closure creates the
new ``Publisher`` based on the received
value. The new ``Publisher`` can emit
more than one event, and successful
completion of the new ``Publisher`` does
not complete the overall stream. Failure
of the new ``Publisher`` causes the
overall stream to fail.
    ///
    /// In the example below, a
``PassthroughSubject`` publishes
`WeatherStation` elements. The
`flatMap(maxPublishers:_:)` receives each
element, creates a
<doc://com.apple.documentation/documentat
ion/Foundation/URL> from it, and produces
a new
<doc://com.apple.documentation/documentat
ion/Foundation/URLSession/
DataTaskPublisher>, which will publish
the data loaded from that
<doc://com.apple.documentation/documentat
ion/Foundation/URL>.
    ///
    ///     public struct WeatherStation
{
    ///         public let stationID:
String
    ///     }
```

```
    ///
    ///     var weatherPublisher =
PassthroughSubject<WeatherStation,
URLError>()
    ///
    ///     cancellable =
weatherPublisher.flatMap { station ->
URLSession.DataTaskPublisher in
    ///         let url =
URL(string:"https://weatherapi.example.co
m/stations/\(station.stationID)/
observations/latest")!
    ///         return
URLSession.shared.dataTaskPublisher(for:
url)
    ///     }
    ///     .sink(
    ///         receiveCompletion:
{ completion in
    ///             // Handle publisher
completion (normal or error).
    ///         },
    ///         receiveValue: {
    ///             // Process the
received data.
    ///         }
    ///      )
    ///
    ///
weatherPublisher.send(WeatherStation(stat
ionID: "KSFO")) // San Francisco, CA
    ///
weatherPublisher.send(WeatherStation(stat
```

```
ionID: "EGLC")) // London, UK
    ///
weatherPublisher.send(WeatherStation(stat
ionID: "ZBBB")) // Beijing, CN
    ///
    /// - Parameters:
    ///   - maxPublishers: Specifies the
maximum number of concurrent publisher
subscriptions, or
``Combine/Subscribers/Demand/unlimited``
if unspecified.
    ///   - transform: A closure that
takes an element as a parameter and
returns a publisher that produces
elements of that type.
    /// - Returns: A publisher that
transforms elements from an upstream
publisher into a publisher of that
element's type.
    public func flatMap<T,
P>(maxPublishers: Subscribers.Demand
= .unlimited, _ transform: @escaping
(Self.Output) -> P) ->
Publishers.FlatMap<P, Self> where T ==
P.Output, P : Publisher, Self.Failure ==
P.Failure
}

@available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
extension Publisher where Self.Failure ==
Never {
```

```
    /// Transforms all elements from an
upstream publisher into a new publisher
up to a maximum number of publishers you
specify.
    ///
    /// - Parameters:
    ///   - maxPublishers: Specifies the
maximum number of concurrent publisher
subscriptions, or
``Combine/Subscribers/Demand/unlimited``
if unspecified.
    ///   - transform: A closure that
takes an element as a parameter and
returns a publisher that produces
elements of that type.
    /// - Returns: A publisher that
transforms elements from an upstream
publisher into a publisher of that
element's type.
    public func flatMap<P>(maxPublishers:
Subscribers.Demand = .unlimited, _
transform: @escaping (Self.Output) -> P)
-> Publishers.FlatMap<P,
Publishers.SetFailureType<Self,
P.Failure>> where P : Publisher
}

@available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
extension Publisher where Self.Failure ==
Never {

    /// Transforms all elements from an
```

```
upstream publisher into a new publisher
up to a maximum number of publishers you
specify.
    ///
    /// - Parameters:
    ///   - maxPublishers: Specifies the
maximum number of concurrent publisher
subscriptions, or
``Combine/Subscribers/Demand/unlimited``
if unspecified.
    ///   - transform: A closure that
takes an element as a parameter and
returns a publisher that produces
elements of that type.
    /// - Returns: A publisher that
transforms elements from an upstream
publisher into a publisher of that
element's type.
    public func flatMap<P>(maxPublishers:
Subscribers.Demand = .unlimited, _
transform: @escaping (Self.Output) -> P)
-> Publishers.FlatMap<P, Self> where P :
Publisher, P.Failure == Never
}

@available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
extension Publisher {

    /// Transforms all elements from an
upstream publisher into a new publisher
up to a maximum number of publishers you
specify.
```

```
    ///
    /// - Parameters:
    ///   - maxPublishers: Specifies the
maximum number of concurrent publisher
subscriptions, or
``Combine/Subscribers/Demand/unlimited``
if unspecified.
    ///   - transform: A closure that
takes an element as a parameter and
returns a publisher that produces
elements of that type.
    /// - Returns: A publisher that
transforms elements from an upstream
publisher into a publisher of that
element's type.
    public func flatMap<P>(maxPublishers:
Subscribers.Demand = .unlimited, _
transform: @escaping (Self.Output) -> P)
->
Publishers.FlatMap<Publishers.SetFailureT
ype<P, Self.Failure>, Self> where P :
Publisher, P.Failure == Never
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Delays delivery of all output to
the downstream receiver by a specified
amount of time on a particular scheduler.
    ///
    /// Use
```

```
``Publisher/delay(for:tolerance:scheduler
:options:)`` when you need to delay the
delivery of elements to a downstream by a
specified amount of time.
    ///
    /// In this example, a
<doc://com.apple.documentation/documentat
ion/Foundation/Timer> publishes an event
every second. The
``Publisher/delay(for:tolerance:scheduler
:options:)`` operator holds the delivery
of the initial element for 3 seconds
(±0.5 seconds), after which each element
is delivered to the downstream on the
main run loop after the specified delay:
    ///
    ///     let df = DateFormatter()
    ///     df.dateStyle = .none
    ///     df.timeStyle = .long
    ///     cancellable =
Timer.publish(every: 1.0, on: .main,
in: .default)
    ///         .autoconnect()
    ///         .handleEvents(receiveOutp
ut: { date in
    ///             print ("Sending
Timestamp \'\(df.string(from: date))\' to
delay()")
    ///         })
    ///         .delay(for: .seconds(3),
scheduler: RunLoop.main, options: .none)
    ///         .sink(
    ///             receiveCompletion:
```

```
{ print ("completion: \($0)", terminator:
"\n") },
    ///             receiveValue: { value
in
    ///                 let now = Date()
    ///                 print ("At \
(df.string(from: now)) received
Timestamp \'\(df.string(from: value))\'
sent: \(String(format: "%.2f",
now.timeIntervalSince(value))) secs ago",
terminator: "\n")
    ///             }
    ///         )
    ///
    ///     // Prints:
    ///     //    Sending Timestamp
'5:02:33 PM PDT' to delay()
    ///     //    Sending Timestamp
'5:02:34 PM PDT' to delay()
    ///     //    Sending Timestamp
'5:02:35 PM PDT' to delay()
    ///     //    Sending Timestamp
'5:02:36 PM PDT' to delay()
    ///     //    At 5:02:36 PM PDT
received  Timestamp '5:02:33 PM PDT'
sent: 3.00 secs ago
    ///     //    Sending Timestamp
'5:02:37 PM PDT' to delay()
    ///     //    At 5:02:37 PM PDT
received  Timestamp '5:02:34 PM PDT'
sent: 3.00 secs ago
    ///     //    Sending Timestamp
'5:02:38 PM PDT' to delay()
```

```
    ///     //    At 5:02:38 PM PDT
received  Timestamp '5:02:35 PM PDT'
sent: 3.00 secs ago
    ///
    /// The delay affects the delivery of
elements and completion, but not of the
original subscription.
    ///
    /// - Parameters:
    ///   - interval: The amount of time
to delay.
    ///   - tolerance: The allowed
tolerance in delivering delayed events.
The `Delay` publisher may deliver
elements this much sooner or later than
the interval specifies.
    ///   - scheduler: The scheduler to
deliver the delayed events.
    ///   - options: Options relevant to
the scheduler's behavior.
    /// - Returns: A publisher that
delays delivery of elements and
completion to the downstream receiver.
    public func delay<S>(for interval:
S.SchedulerTimeType.Stride, tolerance:
S.SchedulerTimeType.Stride? = nil,
scheduler: S, options:
S.SchedulerOptions? = nil) ->
Publishers.Delay<Self, S> where S :
Scheduler
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Publisher {

    /// Omits the specified number of
elements before republishing subsequent
elements.
    ///
    /// Use ``Publisher/dropFirst(_:)``
when you want to drop the first `n`
elements from the upstream publisher, and
republish the remaining elements.
    ///
    /// The example below drops the first
five elements from the stream:
    ///
    ///     let numbers = [1, 2, 3, 4, 5,
6, 7, 8, 9, 10]
    ///     cancellable =
numbers.publisher
    ///         .dropFirst(5)
    ///         .sink { print("\($0)",
terminator: " ") }
    ///
    ///     // Prints: "6 7 8 9 10 "
    ///
    /// - Parameter count: The number of
elements to omit. The default is `1`.
    /// - Returns: A publisher that
doesn’t republish the first `count`
elements.
    public func dropFirst(_ count: Int =
1) -> Publishers.Drop<Self>
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Wraps this publisher with a type
eraser.
    ///
    /// Use
``Publisher/eraseToAnyPublisher()`` to
expose an instance of ``AnyPublisher`` to
the downstream subscriber, rather than
this publisher's actual type.
    /// This form of _type erasure_
preserves abstraction across API
boundaries, such as different modules.
    /// When you expose your publishers
as the ``AnyPublisher`` type, you can
change the underlying implementation over
time without affecting existing clients.
    ///
    /// The following example shows two
types that each have a `publisher`
property. `TypeWithSubject` exposes this
property as its actual type,
``PassthroughSubject``, while
`TypeWithErasedSubject` uses
``Publisher/eraseToAnyPublisher()`` to
expose it as an ``AnyPublisher``. As seen
in the output, a caller from another
module can access
`TypeWithSubject.publisher` as its native
type. This means you can't change your
```

```
publisher to a different type without
breaking the caller. By comparison,
`TypeWithErasedSubject.publisher` appears
to callers as an ``AnyPublisher``, so you
can change the underlying publisher type
at will.
    ///
    ///     public class TypeWithSubject
{
    ///         public let publisher:
some Publisher =
PassthroughSubject<Int,Never>()
    ///     }
    ///     public class
TypeWithErasedSubject {
    ///         public let publisher:
some Publisher =
PassthroughSubject<Int,Never>()
    ///             .eraseToAnyPublisher(
)
    ///     }
    ///
    ///     // In another module:
    ///     let nonErased =
TypeWithSubject()
    ///     if let subject =
nonErased.publisher as?
PassthroughSubject<Int,Never> {
    ///         print("Successfully cast
nonErased.publisher.")
    ///     }
    ///     let erased =
TypeWithErasedSubject()
```

```
    ///     if let subject =
erased.publisher as?
PassthroughSubject<Int,Never> {
    ///         print("Successfully cast
erased.publisher.")
    ///     }
    ///
    ///     // Prints "Successfully cast
nonErased.publisher."
    ///
    /// - Returns: An ``AnyPublisher``
wrapping this publisher.
    public func eraseToAnyPublisher() ->
AnyPublisher<Self.Output, Self.Failure>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publisher {

    /// Publishes the first element of a
stream, then finishes.
    ///
    /// Use ``Publisher/first()`` to
publish just the first element from an
upstream publisher, then finish normally.
The ``Publisher/first()`` operator
requests ``Subscribers/Demand/unlimited``
from its upstream as soon as downstream
requests at least one element. If the
upstream completes before
``Publisher/first()`` receives any
elements, it completes without emitting
```

```
any values.
    ///
    /// In this example, the
``Publisher/first()`` publisher
republishes the first element received
from the sequence publisher, `-10`, then
finishes normally.
    ///
    ///     let numbers = (-10...10)
    ///     cancellable =
numbers.publisher
    ///         .first()
    ///         .sink { print("\($0)") }
    ///
    ///     // Print: "-10"
    ///
    /// - Returns: A publisher that only
publishes the first element of a stream.
    public func first() ->
Publishers.First<Self>

    /// Publishes the first element of a
stream to satisfy a predicate closure,
then finishes normally.
    ///
    /// Use ``Publisher/first(where:)``
to republish only the first element of a
stream that satisfies a closure you
specify. The publisher ignores all
elements after the first element that
satisfies the closure and finishes
normally.
    /// If this publisher doesn’t receive
```

```
any elements, it finishes without
publishing.
    ///
    /// In the example below, the
provided closure causes the
``Publishers/FirstWhere`` publisher to
republish the first received element
that’s greater than `0`, then finishes
normally.
    ///
    ///     let numbers = (-10...10)
    ///     cancellable =
numbers.publisher
    ///         .first { $0 > 0 }
    ///         .sink { print("\($0)") }
    ///
    ///     // Prints: "1"
    ///
    /// - Parameter predicate: A closure
that takes an element as a parameter and
returns a Boolean value that indicates
whether to publish the element.
    /// - Returns: A publisher that only
publishes the first element of a stream
that satisfies the predicate.
    public func first(where predicate:
@escaping (Self.Output) -> Bool) ->
Publishers.FirstWhere<Self>

    /// Publishes the first element of a
stream to satisfy a throwing predicate
closure, then finishes normally.
    ///
```

```
    /// Use
``Publisher/tryFirst(where:)`` when you
need to republish only the first element
of a stream that satisfies an error-
throwing closure you specify.
    /// The publisher ignores all
elements after the first. If this
publisher doesn’t receive any elements,
it finishes without publishing. If the
predicate closure throws an error, the
publisher fails.
    ///
    /// In the example below, a range
publisher emits the first element in the
range then finishes normally:
    ///
    ///     let numberRange:
ClosedRange<Int> = (-1...50)
    ///     numberRange.publisher
    ///         .tryFirst {
    ///             guard $0 < 99 else
{throw RangeError()}
    ///             return true
    ///         }
    ///         .sink(
    ///             receiveCompletion:
{ print ("completion: \($0)", terminator:
" ") },
    ///             receiveValue: { print
("\($0)", terminator: " ") }
    ///           )
    ///
    ///     // Prints: "-1 completion:
```

```
finished"
    ///     // If instead the number
range were ClosedRange<Int> =
(100...200), the tryFirst operator would
terminate publishing with a RangeError.
    ///
    /// - Parameter predicate: A closure
that takes an element as a parameter and
returns a Boolean value that indicates
whether to publish the element.
    /// - Returns: A publisher that only
publishes the first element of a stream
that satisfies the predicate.
    public func tryFirst(where predicate:
@escaping (Self.Output) throws -> Bool)
-> Publishers.TryFirstWhere<Self>
}

/// A namespace for types that serve as
publishers.
///
/// The various operators defined as
extensions on ``Publisher`` implement
their functionality as classes or
structures that extend this enumeration.
For example, the `contains(_:)` operator
returns a `Publishers.Contains` instance.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public enum Publishers {
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that uses a subject
to deliver elements to multiple
subscribers.
    ///
    /// Use a multicast publisher when
you have multiple downstream subscribers,
but you want upstream publishers to only
process one ``Subscriber/receive(_:)``
call per event.
    final public class
Multicast<Upstream, SubjectType> :
ConnectablePublisher where Upstream :
Publisher, SubjectType : Subject,
Upstream.Failure == SubjectType.Failure,
Upstream.Output == SubjectType.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
```

```
Upstream.Failure

        /// The publisher from which this
publisher receives its elements.
        final public let upstream:
Upstream

        /// A closure that returns a
subject each time a subscriber attaches
to the multicast publisher.
        final public let createSubject:
() -> SubjectType

        /// Creates a multicast publisher
that applies a closure to create a
subject that delivers elements to
subscribers.
        ///
        /// - Parameter createSubject: A
closure that returns a ``Subject`` each
time a subscriber attaches to the
multicast publisher.
        public init(upstream: Upstream,
createSubject: @escaping () ->
SubjectType)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
```

```
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        final public func
receive<S>(subscriber: S) where S :
Subscriber, SubjectType.Failure ==
S.Failure, SubjectType.Output == S.Input

        /// Connects to the publisher,
allowing it to produce elements, and
returns an instance with which to cancel
publishing.
        ///
        /// - Returns: A ``Cancellable``
instance that you use to cancel
publishing.
        final public func connect() ->
any Cancellable
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that receives
elements from an upstream publisher on a
specific scheduler.
    public struct SubscribeOn<Upstream,
```

```
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The scheduler the publisher
should use to receive elements.
        public let scheduler: Context

        /// Scheduler options that
customize the delivery of elements.
        public let options:
Context.SchedulerOptions?

        /// Creates a publisher that
```

```
receives elements from an upstream
publisher on a specific scheduler.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - scheduler: The scheduler
the publisher should use to receive
elements.
        ///   - options: Scheduler
options that customize the delivery of
elements.
        public init(upstream: Upstream,
scheduler: Context, options:
Context.SchedulerOptions?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
```

```
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that measures and
emits the time interval between events
received from an upstream publisher.
    public struct
MeasureInterval<Upstream, Context> :
Publisher where Upstream : Publisher,
Context : Scheduler {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
elements of the provided scheduler's time
type's stride.
        public typealias Output =
Context.SchedulerTimeType.Stride

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure
```

```
        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The scheduler used for
tracking the timing of events.
        public let scheduler: Context

        /// Creates a publisher that
measures and emits the time interval
between events received from an upstream
publisher.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - scheduler: A scheduler to
use for tracking the timing of events.
        public init(upstream: Upstream,
scheduler: Context)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
```

```
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input ==
Context.SchedulerTimeType.Stride
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that omits elements
from an upstream publisher until a given
closure returns false.
    public struct DropWhile<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
```

```
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The closure that indicates
whether to drop the element.
        public let predicate:
(Publishers.DropWhile<Upstream>.Output)
-> Bool

        /// Creates a publisher that
omits elements from an upstream publisher
until a given closure returns false.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The closure
that indicates whether to drop the
element.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.DropWhile<Upstream>.Output)
-> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
```

```
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that omits elements
from an upstream publisher until a given
error-throwing closure returns false.
    public struct
TryDropWhile<Upstream> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
```

```
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The error-throwing closure
that indicates whether to drop the
element.
        public let predicate:
(Publishers.TryDropWhile<Upstream>.Output
) throws -> Bool

        /// Creates a publisher that
omits elements from an upstream publisher
until a given error-throwing closure
returns false.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The error-
throwing closure that indicates whether
to drop the element.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.TryDropWhile<Upstream>.Output
) throws -> Bool)
```

```
        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that republishes all
elements that match a provided closure.
    public struct Filter<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
```

```
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// A closure that indicates
whether to republish an element.
        public let isIncluded:
(Upstream.Output) -> Bool

        /// Creates a publisher that
republishes all elements that match a
provided closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - isIncluded: A closure
that indicates whether to republish an
element.
```

```
        public init(upstream: Upstream,
isIncluded: @escaping (Upstream.Output)
-> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that republishes all
elements that match a provided error-
throwing closure.
    public struct TryFilter<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
```

```
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// An error-throwing closure
that indicates whether this filter should
republish an element.
        public let isIncluded:
(Upstream.Output) throws -> Bool

        /// Creates a publisher that
republishes all elements that match a
provided error-throwing closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - isIncluded: An error-
throwing closure that indicates whether
```

```
this filter should republish an element.
        public init(upstream: Upstream,
isIncluded: @escaping (Upstream.Output)
throws -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that raises a
debugger signal when a provided closure
needs to stop the process in the
```

```
debugger.
    ///
    /// When any of the provided closures
returns `true`, this publisher raises the
`SIGTRAP` signal to stop the process in
the debugger.
    /// Otherwise, this publisher passes
through values and completions as-is.
    public struct Breakpoint<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// A closure that executes when
the publisher receives a subscription,
```

```
and can raise a debugger signal by
returning a true Boolean value.
        public let receiveSubscription:
((any Subscription) -> Bool)?

        /// A closure that executes when
the publisher receives output from the
upstream publisher, and can raise a
debugger signal by returning a true
Boolean value.
        public let receiveOutput:
((Upstream.Output) -> Bool)?

        /// A closure that executes when
the publisher receives completion, and
can raise a debugger signal by returning
a true Boolean value.
        public let receiveCompletion:
((Subscribers.Completion<Publishers.Break
point<Upstream>.Failure>) -> Bool)?

        /// Creates a breakpoint
publisher with the provided upstream
publisher and breakpoint-raising
closures.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - receiveSubscription: A
closure that executes when the publisher
receives a subscription, and can raise a
```

```
debugger signal by returning a true
Boolean value.
        ///   - receiveOutput: A closure
that executes when the publisher receives
output from the upstream publisher, and
can raise a debugger signal by returning
a true Boolean value.
        ///   - receiveCompletion: A
closure that executes when the publisher
receives completion, and can raise a
debugger signal by returning a true
Boolean value.
        public init(upstream: Upstream,
receiveSubscription: ((any Subscription)
-> Bool)? = nil, receiveOutput:
((Upstream.Output) -> Bool)? = nil,
receiveCompletion:
((Subscribers.Completion<Publishers.Break
point<Upstream>.Failure>) -> Bool)? =
nil)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
```

```
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes a
single Boolean value that indicates
whether all received elements pass a
given predicate.
    public struct AllSatisfy<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
Boolean elements.
        public typealias Output = Bool

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
```

```
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// A closure that evaluates each
received element.
        ///
        /// Return `true` to continue, or
`false` to cancel the upstream and
finish.
        public let predicate:
(Upstream.Output) -> Bool

        /// Creates a publisher that
publishes a single Boolean value that
indicates whether all received elements
pass a given predicate.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: A closure that
evaluates each received element.
        public init(upstream: Upstream,
predicate: @escaping (Upstream.Output) ->
Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
```

```
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == Bool
    }

    /// A publisher that publishes a
single Boolean value that indicates
whether all received elements pass a
given error-throwing predicate.
    public struct TryAllSatisfy<Upstream>
: Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
Boolean elements.
        public typealias Output = Bool

        /// The kind of errors this
publisher might publish.
        ///
```

```
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// A closure that evaluates each
received element.
        ///
        /// Return `true` to continue, or
`false` to cancel the upstream and
complete. The closure may throw, in which
case the publisher cancels the upstream
publisher and fails with the thrown
error.
        public let predicate:
(Upstream.Output) throws -> Bool

        /// Returns a publisher that
publishes a single Boolean value that
indicates whether all received elements
pass a given error-throwing predicate.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: A closure that
evaluates each received element.
        public init(upstream: Upstream,
```

```
predicate: @escaping (Upstream.Output)
throws -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, S.Failure == any Error,
S.Input == Bool
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes only
elements that don’t match the previous
element.
    public struct
RemoveDuplicates<Upstream> : Publisher
```

```
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The predicate closure used to
evaluate whether two elements are
duplicates.
        public let predicate:
(Publishers.RemoveDuplicates<Upstream>.Ou
tput,
Publishers.RemoveDuplicates<Upstream>.Out
put) -> Bool

        /// Creates a publisher that
publishes only elements that don’t match
```

```
the previous element, as evaluated by a
provided closure.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        /// - Parameter predicate: A
closure to evaluate whether two elements
are equivalent, for purposes of
filtering. Return `true` from this
closure to indicate that the second
element is a duplicate of the first.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.RemoveDuplicates<Upstream>.Ou
tput,
Publishers.RemoveDuplicates<Upstream>.Out
put) -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
```

```
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that publishes only
elements that don’t match the previous
element, as evaluated by a provided
error-throwing closure.
    public struct
TryRemoveDuplicates<Upstream> : Publisher
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// Use `Never` if this
`Publisher` does not publish errors.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// An error-throwing closure to
evaluate whether two elements are
equivalent, for purposes of filtering.
        public let predicate:
```

```
(Publishers.TryRemoveDuplicates<Upstream>
.Output,
Publishers.TryRemoveDuplicates<Upstream>.
Output) throws -> Bool

        /// Creates a publisher that
publishes only elements that don’t match
the previous element, as evaluated by a
provided error-throwing closure.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        /// - Parameter predicate: An
error-throwing closure to evaluate
whether two elements are equivalent, for
purposes of filtering. Return `true` from
this closure to indicate that the second
element is a duplicate of the first. If
this closure throws an error, the
publisher terminates with the thrown
error.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.TryRemoveDuplicates<Upstream>
.Output,
Publishers.TryRemoveDuplicates<Upstream>.
Output) throws -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
```

```
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that decodes elements
received from an upstream publisher,
using a given decoder.
    public struct Decode<Upstream,
Output, Coder> : Publisher where Upstream
: Publisher, Output : Decodable, Coder :
TopLevelDecoder, Upstream.Output ==
Coder.Input {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
```

```
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        public let upstream: Upstream

        /// Creates a publisher that
decodes elements received from an
upstream publisher, using a given
decoder.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - decoder: The decoder that
decodes elements received from the
upstream publisher.
        public init(upstream: Upstream,
decoder: Coder)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
```

```
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, S.Failure == any
Error
    }

    /// A publisher that encodes elements
received from an upstream publisher,
using a given encoder.
    public struct Encode<Upstream, Coder>
: Publisher where Upstream : Publisher,
Coder : TopLevelEncoder,
Upstream.Output : Encodable {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses the
encoder's output type.
        public typealias Output =
Coder.Output
```

```
        public let upstream: Upstream

        /// Creates a publisher that
decodes elements received from an
upstream publisher, using a given
decoder.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - encoder: The encoder that
decodes elements received from the
upstream publisher.
        public init(upstream: Upstream,
encoder: Coder)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Coder.Output == S.Input,
```

```
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that emits a Boolean
value when it receives a specific element
from its upstream publisher.
    public struct Contains<Upstream> :
Publisher where Upstream : Publisher,
Upstream.Output : Equatable {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
Boolean elements.
        public typealias Output = Bool

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream
```

```
        /// The element to match in the
upstream publisher.
        public let output:
Upstream.Output

        /// Creates a publisher that
emits a Boolean value when it receives a
specific element from its upstream
publisher.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - output: The element to
match in the upstream publisher.
        public init(upstream: Upstream,
output: Upstream.Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
```

```
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == Bool
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that receives and
combines the latest elements from two
publishers.
    public struct CombineLatest<A, B> :
Publisher where A : Publisher, B :
Publisher, A.Failure == B.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces two-
element tuples of the upstream
publishers' output types.
        public typealias Output =
(A.Output, B.Output)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
failure type shared by its upstream
publishers.
```

```
        public typealias Failure =
A.Failure

        public let a: A

        public let b: B

        /// Creates a publisher that
receives and combines the latest elements
from two publishers.
        /// - Parameters:
        ///   - a: The first upstream
publisher.
        ///   - b: The second upstream
publisher.
        public init(_ a: A, _ b: B)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
```

```
Subscriber, B.Failure == S.Failure,
S.Input == (A.Output, B.Output)
    }

    /// A publisher that receives and
combines the latest elements from three
publishers.
    public struct CombineLatest3<A, B, C>
: Publisher where A : Publisher, B :
Publisher, C : Publisher, A.Failure ==
B.Failure, B.Failure == C.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
three-element tuples of the upstream
publishers' output types.
        public typealias Output =
(A.Output, B.Output, C.Output)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
failure type shared by its upstream
publishers.
        public typealias Failure =
A.Failure

        public let a: A

        public let b: B
```

```
        public let c: C

        public init(_ a: A, _ b: B, _ c:
C)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, C.Failure == S.Failure,
S.Input == (A.Output, B.Output, C.Output)
    }

    /// A publisher that receives and
combines the latest elements from four
publishers.
    public struct CombineLatest4<A, B, C,
D> : Publisher where A : Publisher, B :
Publisher, C : Publisher, D : Publisher,
A.Failure == B.Failure, B.Failure ==
```

```
C.Failure, C.Failure == D.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces four-
element tuples of the upstream
publishers' output types.
        public typealias Output =
(A.Output, B.Output, C.Output, D.Output)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
failure type shared by its upstream
publishers.
        public typealias Failure =
A.Failure

        public let a: A

        public let b: B

        public let c: C

        public let d: D

        public init(_ a: A, _ b: B, _ c:
C, _ d: D)

        /// Attaches the specified
subscriber to this publisher.
```

```
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, D.Failure == S.Failure,
S.Input == (A.Output, B.Output, C.Output,
D.Output)
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that automatically
connects to an upstream connectable
publisher.
    ///
    /// This publisher calls
``ConnectablePublisher/connect()`` on the
upstream ``ConnectablePublisher`` when
first attached to by a subscriber.
    public class Autoconnect<Upstream> :
```

```
Publisher where Upstream :
ConnectablePublisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstram publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        final public let upstream:
Upstream

        /// Creates a publisher that
automatically connects to an upstream
connectable publisher.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
```

```
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that prints log
messages for all publishing events,
optionally prefixed with a given string.
    ///
    /// This publisher prints log
messages when receiving the following
events:
    ///
    /// - subscription
```

```
    /// - value
    /// - normal completion
    /// - failure
    /// - cancellation
    public struct Print<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// A string with which to prefix
all log messages.
        public let prefix: String

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        public let stream: (any
TextOutputStream)?
```

```
        /// Creates a publisher that
prints log messages for all publishing
events.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - prefix: A string with
which to prefix all log messages.
        public init(upstream: Upstream,
prefix: String, to stream: (any
TextOutputStream)? = nil)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
```

```
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that republishes
elements while a predicate closure
indicates publishing should continue.
    public struct PrefixWhile<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream
```

```
        /// The closure that determines
whether publishing should continue.
        public let predicate:
(Publishers.PrefixWhile<Upstream>.Output)
-> Bool

        /// Creates a publisher that
republishes elements while a predicate
closure indicates publishing should
continue.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The closure
that determines whether publishing should
continue.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.PrefixWhile<Upstream>.Output)
-> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
```

```
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that republishes
elements while an error-throwing
predicate closure indicates publishing
should continue.
    public struct
TryPrefixWhile<Upstream> : Publisher
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error
```

```
        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The error-throwing closure
that determines whether publishing should
continue.
        public let predicate:
(Publishers.TryPrefixWhile<Upstream>.Outp
ut) throws -> Bool

        /// Creates a publisher that
republishes elements while an error-
throwing predicate closure indicates
publishing should continue.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The error-
throwing closure that determines whether
publishing should continue.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.TryPrefixWhile<Upstream>.Outp
ut) throws -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
```

```
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that appears to send
a specified failure type.
    ///
    /// The publisher can't actually fail
with the specified type and finishes
normally. Use this publisher type when
you need to match the error types for two
mismatched publishers.
    public struct
SetFailureType<Upstream, Failure> :
Publisher where Upstream : Publisher,
Failure : Error, Upstream.Failure ==
Never {
```

```
        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
appears to send a specified failure type.
        ///
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
```

```
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Failure
== S.Failure, S : Subscriber,
Upstream.Output == S.Input

        public func setFailureType<E>(to
failure: E.Type) ->
Publishers.SetFailureType<Upstream, E>
where E : Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that emits a Boolean
value upon receiving an element that
satisfies the predicate closure.
    public struct ContainsWhere<Upstream>
: Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
Boolean elements.
        public typealias Output = Bool

        /// The kind of errors this
publisher might publish.
```

```
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The closure that determines
whether the publisher should consider an
element as a match.
        public let predicate:
(Upstream.Output) -> Bool

        /// Creates a publisher that
emits a Boolean value upon receiving an
element that satisfies the predicate
closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The closure
that determines whether the publisher
should consider an element as a match.
        public init(upstream: Upstream,
predicate: @escaping (Upstream.Output) ->
Bool)

        /// Attaches the specified
subscriber to this publisher.
```

```
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == Bool
    }

    /// A publisher that emits a Boolean
value upon receiving an element that
satisfies the throwing predicate closure.
    public struct
TryContainsWhere<Upstream> : Publisher
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
Boolean elements.
        public typealias Output = Bool

        /// The kind of errors this
```

```
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The error-throwing closure
that determines whether this publisher
should emit a Boolean true element.
        public let predicate:
(Upstream.Output) throws -> Bool

        /// Creates a publisher that
emits a Boolean value upon receiving an
element that satisfies the throwing
predicate closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The error-
throwing closure that determines whether
this publisher should emit a Boolean true
element.
        public init(upstream: Upstream,
predicate: @escaping (Upstream.Output)
throws -> Bool)
```

```
        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, S.Failure == any Error,
S.Input == Bool
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that provides
explicit connectability to another
publisher.
    ///
    /// ``Publishers/MakeConnectable`` is
a ``ConnectablePublisher``, which allows
you to perform configuration before
```

```
publishing any elements. Call
``ConnectablePublisher/connect()`` on
this publisher when you want to attach to
its upstream publisher and start
producing elements.
    ///
    /// Use the
``Publisher/makeConnectable()`` operator
to wrap an upstream publisher with an
instance of this publisher.
    public struct
MakeConnectable<Upstream> :
ConnectablePublisher where Upstream :
Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// Creates a connectable
publisher, attached to the provide
```

```
upstream publisher.
        ///
        /// - Parameter upstream: The
publisher from which to receive elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input

        /// Connects to the publisher,
allowing it to produce elements, and
returns an instance with which to cancel
publishing.
        ///
        /// - Returns: A ``Cancellable``
instance that you use to cancel
publishing.
```

```
        public func connect() -> any
Cancellable
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A strategy for collecting
received elements.
    public enum
TimeGroupingStrategy<Context> where
Context : Scheduler {

        /// A grouping that collects and
periodically publishes items.
        case byTime(Context,
Context.SchedulerTimeType.Stride)

        /// A grouping that collects and
publishes items periodically or when a
buffer reaches a maximum size.
        case byTimeOrCount(Context,
Context.SchedulerTimeType.Stride, Int)
    }

    /// A publisher that buffers and
periodically publishes its items.
    public struct CollectByTime<Upstream,
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {
```

```
        /// The kind of values published
by this publisher.
        ///
        /// This publisher publishes
arrays of its upstream publisher's output
type.
        public typealias Output =
[Upstream.Output]

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher that this
publisher receives elements from.
        public let upstream: Upstream

        /// The strategy with which to
collect and publish elements.
        public let strategy:
Publishers.TimeGroupingStrategy<Context>

        /// Scheduler options to use for
the strategy.
        public let options:
Context.SchedulerOptions?

        /// Creates a publisher that
buffers and periodically publishes its
```

```
items.
        /// - Parameters:
        ///   - upstream: The publisher
that this publisher receives elements
from.
        ///   - strategy: The strategy
with which to collect and publish
elements.
        ///   - options: `Scheduler`
options to use for the strategy.
        public init(upstream: Upstream,
strategy:
Publishers.TimeGroupingStrategy<Context>,
options: Context.SchedulerOptions?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == [Upstream.Output]
```

```
    }

    /// A publisher that buffers items.
    public struct Collect<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher publishes
arrays of its upstream publisher's output
type.
        public typealias Output =
[Upstream.Output]

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher that this
publisher receives elements from.
        public let upstream: Upstream

        /// Creates a publisher that
buffers items.
        /// - Parameter upstream: The
publisher that this publisher receives
elements from.
        public init(upstream: Upstream)
```

```
        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == [Upstream.Output]
    }

    /// A publisher that buffers a
maximum number of items.
    public struct
CollectByCount<Upstream> : Publisher
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher publishes
arrays of its upstream publisher's output
type.
```

```
        public typealias Output =
[Upstream.Output]

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher that this
publisher receives elements from.
        public let upstream: Upstream

        /// The maximum number of
received elements to buffer before
publishing.
        public let count: Int

        /// Creates a publisher that
buffers a maximum number of items.
        /// - Parameters:
        ///   - upstream: The publisher
that this publisher receives elements
from.
        ///   - count: The maximum number
of received elements to buffer before
publishing.
        public init(upstream: Upstream,
count: Int)

        /// Attaches the specified
```

```
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == [Upstream.Output]
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that delivers
elements to its downstream subscriber on
a specific scheduler.
    public struct ReceiveOn<Upstream,
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {

        /// The kind of values published
by this publisher.
```

```
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The scheduler the publisher
uses to deliver elements.
        public let scheduler: Context

        /// Scheduler options used to
customize element delivery.
        public let options:
Context.SchedulerOptions?

        /// Creates a publisher that
delivers elements to its downstream
subscriber on a specific scheduler.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
```

```
elements.
        ///   - scheduler: The scheduler
the publisher uses to deliver elements.
        ///   - options: Scheduler
options used to customize element
delivery.
        public init(upstream: Upstream,
scheduler: Context, options:
Context.SchedulerOptions?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
```

```
extension Publishers {

    /// A publisher that publishes the
value of a key path.
    public struct MapKeyPath<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The key path of a property to
publish.
        public let keyPath:
KeyPath<Upstream.Output, Output>

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
```

```
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, Upstream.Failure
== S.Failure
    }

    /// A publisher that publishes the
values of two key paths as a tuple.
    public struct MapKeyPath2<Upstream,
Output0, Output1> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces two-
element tuples, where each menber's type
matches the type of the corresponding key
path's property.
        public typealias Output =
(Output0, Output1)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
```

```
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The key path of a property to
publish.
        public let keyPath0:
KeyPath<Upstream.Output, Output0>

        /// The key path of a second
property to publish.
        public let keyPath1:
KeyPath<Upstream.Output, Output1>

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
```

```
Subscriber, Upstream.Failure ==
S.Failure, S.Input == (Output0, Output1)
    }

    /// A publisher that publishes the
values of three key paths as a tuple.
    public struct MapKeyPath3<Upstream,
Output0, Output1, Output2> : Publisher
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
three-element tuples, where each menber's
type matches the type of the
corresponding key path's property.
        public typealias Output =
(Output0, Output1, Output2)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The key path of a property to
```

```
publish.
        public let keyPath0:
KeyPath<Upstream.Output, Output0>

        /// The key path of a second
property to publish.
        public let keyPath1:
KeyPath<Upstream.Output, Output1>

        /// The key path of a third
property to publish.
        public let keyPath2:
KeyPath<Upstream.Output, Output2>

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == (Output0, Output1,
Output2)
```

```
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that republishes
elements until another publisher emits an
element.
    public struct
PrefixUntilOutput<Upstream, Other> :
Publisher where Upstream : Publisher,
Other : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
```

```
        public let upstream: Upstream

        /// Another publisher, whose
first output causes this publisher to
finish.
        public let other: Other

        /// Creates a publisher that
republishes elements until another
publisher emits an element.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - other: Another publisher,
the first output from which causes this
publisher to finish.
        public init(upstream: Upstream,
other: Other)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
```

```
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that applies a
closure to all received elements and
produces an accumulated value when the
upstream publisher finishes.
    public struct Reduce<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The initial value provided on
```

```
the first invocation of the closure.
        public let initial: Output

        /// A closure that takes the
previously-accumulated value and the next
element from the upstream publisher to
produce a new value.
        public let nextPartialResult:
(Output, Upstream.Output) -> Output

        /// Creates a publisher that
applies a closure to all received
elements and produces an accumulated
value when the upstream publisher
finishes.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - initial: The initial
value provided on the first invocation of
the closure.
        ///   - nextPartialResult: A
closure that takes the previously-
accumulated value and the next element
from the upstream publisher to produce a
new value.
        public init(upstream: Upstream,
initial: Output, nextPartialResult:
@escaping (Output, Upstream.Output) ->
Output)

        /// Attaches the specified
```

```
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, Upstream.Failure
== S.Failure
    }

    /// A publisher that applies an
error-throwing closure to all received
elements and produces an accumulated
value when the upstream publisher
finishes.
    public struct TryReduce<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
```

```
<doc://com.apple.documentation/documentation/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this publisher receives elements.
        public let upstream: Upstream

        /// The initial value provided on the first-use of the closure.
        public let initial: Output

        /// An error-throwing closure that takes the previously-accumulated value and the next element from the upstream to produce a new value.
        ///
        /// If this closure throws an error, the publisher fails and passes the error to its subscriber.
        public let nextPartialResult: (Output, Upstream.Output) throws -> Output

        /// Creates a publisher that applies an error-throwing closure to all received elements and produces an accumulated value when the upstream publisher finishes.
        /// - Parameters:
        ///   - upstream: The publisher from which this publisher receives elements.
```

```
        ///   - initial: The initial
value provided on the first-use of the
closure.
        ///   - nextPartialResult: An
error-throwing closure that takes the
previously-accumulated value and the next
element from the upstream to produce a
new value. If this closure throws an
error, the publisher fails and passes the
error to its subscriber.
        public init(upstream: Upstream,
initial: Output, nextPartialResult:
@escaping (Output, Upstream.Output)
throws -> Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, S.Failure == any
Error
```

```
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that republishes all
non-nil results of calling a closure with
each received element.
    public struct CompactMap<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// A closure that receives
values from the upstream publisher and
returns optional values.
        public let transform:
(Upstream.Output) -> Output?

        /// Creates a publisher that
```

```
republishes all non-`nil` results of
calling a closure with each received
element.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - transform: A closure that
receives values from the upstream
publisher and returns optional values.
        public init(upstream: Upstream,
transform: @escaping (Upstream.Output) ->
Output?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, Upstream.Failure
== S.Failure
    }
```

```
    /// A publisher that republishes all
non-nil results of calling an error-
throwing closure with each received
element.
    public struct TryCompactMap<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// An error-throwing closure
that receives values from the upstream
publisher and returns optional values.
        ///
        /// If this closure throws an
error, the publisher fails.
        public let transform:
(Upstream.Output) throws -> Output?

        public init(upstream: Upstream,
transform: @escaping (Upstream.Output)
```

```
throws -> Output?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, S.Failure == any
Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher created by applying
the merge function to two upstream
publishers.
    public struct Merge<A, B> : Publisher
where A : Publisher, B : Publisher,
A.Failure == B.Failure, A.Output ==
```

```
B.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// Creates a publisher created
by applying the merge function to two
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
        ///   - b: A second publisher to
merge.
        public init(_ a: A, _ b: B)
```

```
        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, B.Failure == S.Failure,
B.Output == S.Input

        public func merge<P>(with other:
P) -> Publishers.Merge3<A, B, P> where
P : Publisher, B.Failure == P.Failure,
B.Output == P.Output

        public func merge<Z, Y>(with z:
Z, _ y: Y) -> Publishers.Merge4<A, B, Z,
Y> where Z : Publisher, Y : Publisher,
B.Failure == Z.Failure, B.Output ==
Z.Output, Z.Failure == Y.Failure,
Z.Output == Y.Output

        public func merge<Z, Y, X>(with
z: Z, _ y: Y, _ x: X) ->
```

```
Publishers.Merge5<A, B, Z, Y, X> where
Z : Publisher, Y : Publisher, X :
Publisher, B.Failure == Z.Failure,
B.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output,
Y.Failure == X.Failure, Y.Output ==
X.Output

        public func merge<Z, Y, X,
W>(with z: Z, _ y: Y, _ x: X, _ w: W) ->
Publishers.Merge6<A, B, Z, Y, X, W> where
Z : Publisher, Y : Publisher, X :
Publisher, W : Publisher, B.Failure ==
Z.Failure, B.Output == Z.Output,
Z.Failure == Y.Failure, Z.Output ==
Y.Output, Y.Failure == X.Failure,
Y.Output == X.Output, X.Failure ==
W.Failure, X.Output == W.Output

        public func merge<Z, Y, X, W,
V>(with z: Z, _ y: Y, _ x: X, _ w: W, _
v: V) -> Publishers.Merge7<A, B, Z, Y, X,
W, V> where Z : Publisher, Y : Publisher,
X : Publisher, W : Publisher, V :
Publisher, B.Failure == Z.Failure,
B.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output,
Y.Failure == X.Failure, Y.Output ==
X.Output, X.Failure == W.Failure,
X.Output == W.Output, W.Failure ==
V.Failure, W.Output == V.Output

        public func merge<Z, Y, X, W, V,
```

```
U>(with z: Z, _ y: Y, _ x: X, _ w: W, _
v: V, _ u: U) -> Publishers.Merge8<A, B,
Z, Y, X, W, V, U> where Z : Publisher,
Y : Publisher, X : Publisher, W :
Publisher, V : Publisher, U : Publisher,
B.Failure == Z.Failure, B.Output ==
Z.Output, Z.Failure == Y.Failure,
Z.Output == Y.Output, Y.Failure ==
X.Failure, Y.Output == X.Output,
X.Failure == W.Failure, X.Output ==
W.Output, W.Failure == V.Failure,
W.Output == V.Output, V.Failure ==
U.Failure, V.Output == U.Output
    }

    /// A publisher created by applying
the merge function to three upstream
publishers.
    public struct Merge3<A, B, C> :
Publisher where A : Publisher, B :
Publisher, C : Publisher, A.Failure ==
B.Failure, A.Output == B.Output,
B.Failure == C.Failure, B.Output ==
C.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output
```

```
        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// A third publisher to merge.
        public let c: C

        /// Creates a publisher created
by applying the merge function to three
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
        ///   - b: A second publisher to
merge.
        ///   - c: A third publisher to
merge.
        public init(_ a: A, _ b: B, _ c:
C)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
```

```
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, C.Failure == S.Failure,
C.Output == S.Input

        public func merge<P>(with other:
P) -> Publishers.Merge4<A, B, C, P> where
P : Publisher, C.Failure == P.Failure,
C.Output == P.Output

        public func merge<Z, Y>(with z:
Z, _ y: Y) -> Publishers.Merge5<A, B, C,
Z, Y> where Z : Publisher, Y : Publisher,
C.Failure == Z.Failure, C.Output ==
Z.Output, Z.Failure == Y.Failure,
Z.Output == Y.Output

        public func merge<Z, Y, X>(with
z: Z, _ y: Y, _ x: X) ->
Publishers.Merge6<A, B, C, Z, Y, X> where
Z : Publisher, Y : Publisher, X :
Publisher, C.Failure == Z.Failure,
C.Output == Z.Output, Z.Failure ==
```

```
Y.Failure, Z.Output == Y.Output,
Y.Failure == X.Failure, Y.Output ==
X.Output

        public func merge<Z, Y, X,
W>(with z: Z, _ y: Y, _ x: X, _ w: W) ->
Publishers.Merge7<A, B, C, Z, Y, X, W>
where Z : Publisher, Y : Publisher, X :
Publisher, W : Publisher, C.Failure ==
Z.Failure, C.Output == Z.Output,
Z.Failure == Y.Failure, Z.Output ==
Y.Output, Y.Failure == X.Failure,
Y.Output == X.Output, X.Failure ==
W.Failure, X.Output == W.Output

        public func merge<Z, Y, X, W,
V>(with z: Z, _ y: Y, _ x: X, _ w: W, _
v: V) -> Publishers.Merge8<A, B, C, Z, Y,
X, W, V> where Z : Publisher, Y :
Publisher, X : Publisher, W : Publisher,
V : Publisher, C.Failure == Z.Failure,
C.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output,
Y.Failure == X.Failure, Y.Output ==
X.Output, X.Failure == W.Failure,
X.Output == W.Output, W.Failure ==
V.Failure, W.Output == V.Output
    }

    /// A publisher created by applying
the merge function to four upstream
publishers.
    public struct Merge4<A, B, C, D> :
```

```
Publisher where A : Publisher, B :
Publisher, C : Publisher, D : Publisher,
A.Failure == B.Failure, A.Output ==
B.Output, B.Failure == C.Failure,
B.Output == C.Output, C.Failure ==
D.Failure, C.Output == D.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// A third publisher to merge.
        public let c: C

        /// A fourth publisher to merge.
```

```
        public let d: D

        /// Creates a publisher created
by applying the merge function to four
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
        ///   - b: A second publisher to
merge.
        ///   - c: A third publisher to
merge.
        ///   - d: A fourth publisher to
merge.
        public init(_ a: A, _ b: B, _ c:
C, _ d: D)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, D.Failure == S.Failure,
```

```
D.Output == S.Input

        public func merge<P>(with other:
P) -> Publishers.Merge5<A, B, C, D, P>
where P : Publisher, D.Failure ==
P.Failure, D.Output == P.Output

        public func merge<Z, Y>(with z:
Z, _ y: Y) -> Publishers.Merge6<A, B, C,
D, Z, Y> where Z : Publisher, Y :
Publisher, D.Failure == Z.Failure,
D.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output

        public func merge<Z, Y, X>(with
z: Z, _ y: Y, _ x: X) ->
Publishers.Merge7<A, B, C, D, Z, Y, X>
where Z : Publisher, Y : Publisher, X :
Publisher, D.Failure == Z.Failure,
D.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output,
Y.Failure == X.Failure, Y.Output ==
X.Output

        public func merge<Z, Y, X,
W>(with z: Z, _ y: Y, _ x: X, _ w: W) ->
Publishers.Merge8<A, B, C, D, Z, Y, X, W>
where Z : Publisher, Y : Publisher, X :
Publisher, W : Publisher, D.Failure ==
Z.Failure, D.Output == Z.Output,
Z.Failure == Y.Failure, Z.Output ==
Y.Output, Y.Failure == X.Failure,
Y.Output == X.Output, X.Failure ==
```

```
W.Failure, X.Output == W.Output
    }

    /// A publisher created by applying
the merge function to five upstream
publishers.
    public struct Merge5<A, B, C, D, E> :
Publisher where A : Publisher, B :
Publisher, C : Publisher, D : Publisher,
E : Publisher, A.Failure == B.Failure,
A.Output == B.Output, B.Failure ==
C.Failure, B.Output == C.Output,
C.Failure == D.Failure, C.Output ==
D.Output, D.Failure == E.Failure,
D.Output == E.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
```

```
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// A third publisher to merge.
        public let c: C

        /// A fourth publisher to merge.
        public let d: D

        /// A fifth publisher to merge.
        public let e: E

        /// Creates a publisher created
by applying the merge function to five
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
        ///   - b: A second publisher to
merge.
        ///   - c: A third publisher to
merge.
        ///   - d: A fourth publisher to
merge.
        ///   - e: A fifth publisher to
merge.
        public init(_ a: A, _ b: B, _ c:
C, _ d: D, _ e: E)

        /// Attaches the specified
subscriber to this publisher.
        ///
```

```
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, E.Failure == S.Failure,
E.Output == S.Input

        public func merge<P>(with other:
P) -> Publishers.Merge6<A, B, C, D, E, P>
where P : Publisher, E.Failure ==
P.Failure, E.Output == P.Output

        public func merge<Z, Y>(with z:
Z, _ y: Y) -> Publishers.Merge7<A, B, C,
D, E, Z, Y> where Z : Publisher, Y :
Publisher, E.Failure == Z.Failure,
E.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output

        public func merge<Z, Y, X>(with
z: Z, _ y: Y, _ x: X) ->
Publishers.Merge8<A, B, C, D, E, Z, Y, X>
where Z : Publisher, Y : Publisher, X :
Publisher, E.Failure == Z.Failure,
```

```
E.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output,
Y.Failure == X.Failure, Y.Output ==
X.Output
    }

    /// A publisher created by applying
the merge function to six upstream
publishers.
    public struct Merge6<A, B, C, D, E,
F> : Publisher where A : Publisher, B :
Publisher, C : Publisher, D : Publisher,
E : Publisher, F : Publisher, A.Failure
== B.Failure, A.Output == B.Output,
B.Failure == C.Failure, B.Output ==
C.Output, C.Failure == D.Failure,
C.Output == D.Output, D.Failure ==
E.Failure, D.Output == E.Output,
E.Failure == F.Failure, E.Output ==
F.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
```

```
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// A third publisher to merge.
        public let c: C

        /// A fourth publisher to merge.
        public let d: D

        /// A fifth publisher to merge.
        public let e: E

        /// A sixth publisher to merge.
        public let f: F

        /// publisher created by applying
the merge function to six upstream
publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
        ///   - b: A second publisher to
merge.
        ///   - c: A third publisher to
merge.
        ///   - d: A fourth publisher to
merge.
```

```
        ///   - e: A fifth publisher to
merge.
        ///   - f: A sixth publisher to
merge.
        public init(_ a: A, _ b: B, _ c:
C, _ d: D, _ e: E, _ f: F)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, F.Failure == S.Failure,
F.Output == S.Input

        public func merge<P>(with other:
P) -> Publishers.Merge7<A, B, C, D, E, F,
P> where P : Publisher, F.Failure ==
P.Failure, F.Output == P.Output

        public func merge<Z, Y>(with z:
Z, _ y: Y) -> Publishers.Merge8<A, B, C,
```

```
D, E, F, Z, Y> where Z : Publisher, Y :
Publisher, F.Failure == Z.Failure,
F.Output == Z.Output, Z.Failure ==
Y.Failure, Z.Output == Y.Output
    }

    /// A publisher created by applying
the merge function to seven upstream
publishers.
    public struct Merge7<A, B, C, D, E,
F, G> : Publisher where A : Publisher,
B : Publisher, C : Publisher, D :
Publisher, E : Publisher, F : Publisher,
G : Publisher, A.Failure == B.Failure,
A.Output == B.Output, B.Failure ==
C.Failure, B.Output == C.Output,
C.Failure == D.Failure, C.Output ==
D.Output, D.Failure == E.Failure,
D.Output == E.Output, E.Failure ==
F.Failure, E.Output == F.Output,
F.Failure == G.Failure, F.Output ==
G.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output

        /// The kind of errors this
publisher might publish.
```

```
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// A third publisher to merge.
        public let c: C

        /// A fourth publisher to merge.
        public let d: D

        /// A fifth publisher to merge.
        public let e: E

        /// A sixth publisher to merge.
        public let f: F

        /// An seventh publisher to
merge.
        public let g: G

        /// Creates a publisher created
by applying the merge function to seven
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
```

```
        ///   - b: A second publisher to
merge.
        ///   - c: A third publisher to
merge.
        ///   - d: A fourth publisher to
merge.
        ///   - e: A fifth publisher to
merge.
        ///   - f: A sixth publisher to
merge.
        ///   - g: An seventh publisher
to merge.
        public init(_ a: A, _ b: B, _ c:
C, _ d: D, _ e: E, _ f: F, _ g: G)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, G.Failure == S.Failure,
G.Output == S.Input
```

```
        public func merge<P>(with other:
P) -> Publishers.Merge8<A, B, C, D, E, F,
G, P> where P : Publisher, G.Failure ==
P.Failure, G.Output == P.Output
    }

    /// A publisher created by applying
the merge function to eight upstream
publishers.
    public struct Merge8<A, B, C, D, E,
F, G, H> : Publisher where A : Publisher,
B : Publisher, C : Publisher, D :
Publisher, E : Publisher, F : Publisher,
G : Publisher, H : Publisher, A.Failure
== B.Failure, A.Output == B.Output,
B.Failure == C.Failure, B.Output ==
C.Output, C.Failure == D.Failure,
C.Output == D.Output, D.Failure ==
E.Failure, D.Output == E.Output,
E.Failure == F.Failure, E.Output ==
F.Output, F.Failure == G.Failure,
F.Output == G.Output, G.Failure ==
H.Failure, G.Output == H.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
A.Output
```

```
        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to merge.
        public let a: A

        /// A second publisher to merge.
        public let b: B

        /// A third publisher to merge.
        public let c: C

        /// A fourth publisher to merge.
        public let d: D

        /// A fifth publisher to merge.
        public let e: E

        /// A sixth publisher to merge.
        public let f: F

        /// An seventh publisher to
merge.
        public let g: G

        /// A eighth publisher to merge.
        public let h: H
```

```
        /// Creates a publisher created
by applying the merge function to eight
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to merge
        ///   - b: A second publisher to
merge.
        ///   - c: A third publisher to
merge.
        ///   - d: A fourth publisher to
merge.
        ///   - e: A fifth publisher to
merge.
        ///   - f: A sixth publisher to
merge.
        ///   - g: An seventh publisher
to merge.
        ///   - h: An eighth publisher to
merge.
        public init(_ a: A, _ b: B, _ c:
C, _ d: D, _ e: E, _ f: F, _ g: G, _ h:
H)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
```

```
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, H.Failure == S.Failure,
H.Output == S.Input
    }

    /// A publisher created by applying
the merge function to an arbitrary number
of upstream publishers.
    public struct MergeMany<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publishers' common output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
Upstream.Failure

        /// The array of upstream
```

```
publishers that this publisher merges
together.
        public let publishers: [Upstream]

        /// Creates a publisher created
by applying the merge function to an
arbitrary number of upstream publishers.
        /// - Parameter upstream: A
variadic parameter containing zero or
more publishers to merge with this
publisher.
        public init(_ upstream:
Upstream...)

        /// Creates a publisher created
by applying the merge function to a
sequence of upstream publishers.
        /// - Parameter upstream: A
sequence containing zero or more
publishers to merge with this publisher.
        public init<S>(_ upstream: S)
where Upstream == S.Element, S : Sequence

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
```

```
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input

        public func merge(with other:
Upstream) ->
Publishers.MergeMany<Upstream>
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that transforms
elements from the upstream publisher by
providing the current element to a
closure along with the last value
returned by the closure.
    public struct Scan<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
```

```
        public typealias Failure =
Upstream.Failure

        /// The publisher that this
publisher receives elements from.
        public let upstream: Upstream

        /// The previous result returned
by the `nextPartialResult` closure.
        public let initialResult: Output

        ///  An error-throwing closure
that takes as its arguments the previous
value returned by the closure and the
next element emitted from the upstream
publisher.
        public let nextPartialResult:
(Output, Upstream.Output) -> Output

        /// Creates a publisher that
transforms elements from the upstream
publisher by providing the current
element to a closure along with the last
value returned by the closure.
        /// - Parameters:
        ///   - upstream: The publisher
that this publisher receives elements
from.
        ///   - initialResult: The
previous result returned by the
`nextPartialResult` closure.
        ///   - nextPartialResult: A
closure that takes as its arguments the
```

```
previous value returned by the closure
and the next element emitted from the
upstream publisher.
        public init(upstream: Upstream,
initialResult: Output, nextPartialResult:
@escaping (Output, Upstream.Output) ->
Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, Upstream.Failure
== S.Failure
    }

    /// A publisher that transforms
elements from the upstream publisher by
providing the current element to a
failable closure along with the last
value returned by the closure.
```

```
    public struct TryScan<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher that this
publisher receives elements from.
        public let upstream: Upstream

        /// The previous result returned
by the `nextPartialResult` closure.
        public let initialResult: Output

        /// An error-throwing closure
that takes as its arguments the previous
value returned by the closure and the
next element emitted from the upstream
publisher.
        public let nextPartialResult:
(Output, Upstream.Output) throws ->
Output

        /// Creates a publisher that
transforms elements from the upstream
publisher by providing the current
```

```
element to a failable closure along with
the last value returned by the closure.
        /// - Parameters:
        ///   - upstream: The publisher
that this publisher receives elements
from.
        ///   - initialResult: The
previous result returned by the
`nextPartialResult` closure.
        ///   - nextPartialResult: An
error-throwing closure that takes as its
arguments the previous value returned by
the closure and the next element emitted
from the upstream publisher.
        public init(upstream: Upstream,
initialResult: Output, nextPartialResult:
@escaping (Output, Upstream.Output)
throws -> Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
```

```
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, S.Failure == any
Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes the
number of elements received from the
upstream publisher.
    public struct Count<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
integer elements.
        public typealias Output = Int

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
```

```
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
publishes the number of elements received
from the upstream publisher.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == Int
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that waits until
after the stream finishes and then
publishes the last element of the stream
that satisfies a predicate closure.
    public struct LastWhere<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The closure that determines
whether to publish an element.
        public let predicate:
```

```
(Publishers.LastWhere<Upstream>.Output)
-> Bool

        /// Creates a publisher that
waits until after the stream finishes and
then publishes the last element of the
stream that satisfies a predicate
closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The closure
that determines whether to publish an
element.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.LastWhere<Upstream>.Output)
-> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
```

```
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that waits until
after the stream finishes and then
publishes the last element of the stream
that satisfies an error-throwing
predicate closure.
    public struct
TryLastWhere<Upstream> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error
```

```
        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The error-throwing closure
that determines whether to publish an
element.
        public let predicate:
(Publishers.TryLastWhere<Upstream>.Output
) throws -> Bool

        /// Creates a publisher that
waits until after the stream finishes and
then publishes the last element of the
stream that satisfies an error-throwing
predicate closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - predicate: The error-
throwing closure that determines whether
to publish an element.
        public init(upstream: Upstream,
predicate: @escaping
(Publishers.TryLastWhere<Upstream>.Output
) throws -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
```

```
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that ignores all
upstream elements, but passes along the
upstream publisher's completion state
(finished or failed).
    public struct
IgnoreOutput<Upstream> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher never produces
elements.
```

```
        public typealias Output = Never

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
ignores all upstream elements, but passes
along the upstream publisher's completion
state (finish or failed).
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
```

```
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, S.Input == Never
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that flattens nested
publishers.
    ///
    /// Given a publisher that publishes
``Publisher`` instances, the
``Publishers/SwitchToLatest`` publisher
produces a sequence of events from only
the most recent one. For example, given
the type
`AnyPublisher<URLSession.DataTaskPublishe
r,
    /// NSError>`, calling
``Publisher/switchToLatest()`` results in
the type `SwitchToLatest<(Data,
URLResponse), URLError>`. The downstream
subscriber sees a continuous stream of
`(Data, URLResponse)` elements from what
looks like a single
```

```
<doc://com.apple.documentation/documentation/Foundation/URLSession/DataTaskPublisher> even though the elements are coming from different upstream publishers.
    ///
    /// When ``Publishers/SwitchToLatest`` receives a new publisher from the upstream publisher, it cancels its previous subscription. Use this feature to prevent earlier publishers from performing unnecessary work, such as creating network request publishers from frequently-updating user interface publishers.
    public struct SwitchToLatest<P, Upstream> : Publisher where P : Publisher, P == Upstream.Output, Upstream : Publisher, P.Failure == Upstream.Failure {

        /// The kind of values published by this publisher.
        ///
        /// This publisher produces elements of the type produced by the upstream publisher-of-publishers.
        public typealias Output = P.Output

        /// The kind of errors this publisher might publish.
```

```
        ///
        /// This publisher produces
errors of the type produced by the
upstream publisher-of-publishers.
        public typealias Failure =
P.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
“flattens” nested publishers.
        ///
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
```

```
        public func
receive<S>(subscriber: S) where S :
Subscriber, P.Output == S.Input,
Upstream.Failure == S.Failure
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that attempts to
recreate its subscription to a failed
upstream publisher.
    public struct Retry<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure
```

```
        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The maximum number of retry
attempts to perform.
        ///
        /// If `nil`, this publisher
attempts to reconnect with the upstream
publisher an unlimited number of times.
        public let retries: Int?

        /// Creates a publisher that
attempts to recreate its subscription to
a failed upstream publisher.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - retries: The maximum
number of retry attempts to perform. If
`nil`, this publisher attempts to
reconnect with the upstream publisher an
unlimited number of times.
        public init(upstream: Upstream,
retries: Int?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
```

```
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that converts any
failure from the upstream publisher into
a new error.
    public struct MapError<Upstream,
Failure> : Publisher where Upstream :
Publisher, Failure : Error {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
```

```
Upstream.Output

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The closure that converts the
upstream failure into a new error.
        public let transform:
(Upstream.Failure) -> Failure

        /// Creates a publisher that
converts any failure from the upstream
publisher into a new error.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - transform: The closure
that converts the upstream failure into a
new error.
        public init(upstream: Upstream,
transform: @escaping (Upstream.Failure)
-> Failure)

        public init(upstream: Upstream, _
map: @escaping (Upstream.Failure) ->
Failure)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
```

```
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Failure
== S.Failure, S : Subscriber,
Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes either
the most-recent or first element
published by the upstream publisher in a
specified time interval.
    public struct Throttle<Upstream,
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
```

```
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The interval in which to find
and emit the most recent element.
        public let interval:
Context.SchedulerTimeType.Stride

        /// The scheduler on which to
publish elements.
        public let scheduler: Context

        /// A Boolean value indicating
whether to publish the most recent
element.
        ///
        /// If `false`, the publisher
emits the first element received during
the interval.
        public let latest: Bool
```

```
        /// Creates a publisher that
publishes either the most-recent or first
element published by the upstream
publisher in a specified time interval.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - interval: The interval in
which to find and emit the most recent
element.
        ///   - scheduler: The scheduler
on which to publish elements.
        ///   - latest: A Boolean value
indicating whether to publish the most
recent element. If `false`, the publisher
emits the first element received during
the interval.
        public init(upstream: Upstream,
interval:
Context.SchedulerTimeType.Stride,
scheduler: Context, latest: Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
```

```
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that shares the
output of an upstream publisher with
multiple subscribers.
    ///
    /// This publisher type supports
multiple subscribers, all of whom receive
unchanged elements and completion states
from the upstream publisher.
    ///
    ///  > Tip: ``Publishers/Share`` is
effectively a combination of the
``Publishers/Multicast`` and
``PassthroughSubject`` publishers, with
an implicit
``ConnectablePublisher/autoconnect()``.
    ///
    /// Be aware that
```

```
``Publishers/Share`` is a class rather
than a structure like most other
publishers. Use this type when you need a
publisher instance that uses reference
semantics.
    final public class Share<Upstream> :
Publisher, Equatable where Upstream :
Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        final public let upstream:
Upstream

        /// Creates a publisher that
shares the output of an upstream
publisher with multiple subscribers.
```

```
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        final public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input

        /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
        /// - Parameters:
        ///   - lhs: A `Share` publisher
to compare for equality.
        ///   - rhs: Another `Share`
publisher to compare for equality.
        /// - Returns: `true` if the
```

```
publishers have reference equality
(`===`); otherwise `false`.
        public static func == (lhs:
Publishers.Share<Upstream>, rhs:
Publishers.Share<Upstream>) -> Bool
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that republishes
items from another publisher only if each
new item is in increasing order from the
previously-published item.
    public struct Comparison<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upsteam publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
```

```
Upstream.Failure

        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// A closure that receives two
elements and returns true if they are in
increasing order.
        public let areInIncreasingOrder:
(Upstream.Output, Upstream.Output) ->
Bool

        /// Creates a publisher that
republishes items from another publisher
only if each new item is in increasing
order from the previously-published item.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - areInIncreasingOrder: A
closure that receives two elements and
returns true if they are in increasing
order.
        public init(upstream: Upstream,
areInIncreasingOrder: @escaping
(Upstream.Output, Upstream.Output) ->
Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
```

```
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that republishes
items from another publisher only if each
new item is in increasing order from the
previously-published item, and fails if
the ordering logic throws an error.
    public struct TryComparison<Upstream>
: Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upsteam publisher's output type.
        public typealias Output =
Upstream.Output
```

```
        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// A closure that receives two
elements and returns true if they are in
increasing order.
        public let areInIncreasingOrder:
(Upstream.Output, Upstream.Output) throws
-> Bool

        /// Creates a publisher that
republishes items from another publisher
only if each new item is in increasing
order from the previously-published item,
and fails if the ordering logic throws an
error.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - areInIncreasingOrder: A
closure that receives two elements and
returns true if they are in increasing
```

```
order.
        public init(upstream: Upstream,
areInIncreasingOrder: @escaping
(Upstream.Output, Upstream.Output) throws
-> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that replaces an
empty stream with a provided element.
```

```
    public struct
ReplaceEmpty<Upstream> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The element to deliver when
the upstream publisher finishes without
delivering any elements.
        public let output:
Publishers.ReplaceEmpty<Upstream>.Output

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
replaces an empty stream with a provided
element.
```

```
        /// - Parameters:
        ///   - upstream: The element to
deliver when the upstream publisher
finishes without delivering any elements.
        ///   - output: The publisher
from which this publisher receives
elements.
        public init(upstream: Upstream,
output:
Publishers.ReplaceEmpty<Upstream>.Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that replaces any
errors in the stream with a provided
```

```
element.
    public struct
ReplaceError<Upstream> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher never fails.
        public typealias Failure = Never

        /// The element with which to
replace errors from the upstream
publisher.
        public let output:
Publishers.ReplaceError<Upstream>.Output

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
replaces any errors in the stream with a
provided element.
        /// - Parameters:
```

```
        ///   - upstream: The element
with which to replace errors from the
upstream publisher.
        ///   - output: The publisher
from which this publisher receives
elements.
        public init(upstream: Upstream,
output:
Publishers.ReplaceError<Upstream>.Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == Never
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
```

```
extension Publishers {

    /// A publisher that raises a fatal
error upon receiving any failure, and
otherwise republishes all received input.
    ///
    /// Use this function for internal
integrity checks that are active during
testing but don't affect performance of
shipping code.
    public struct
AssertNoFailure<Upstream> : Publisher
where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher never produces
errors.
        public typealias Failure = Never

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream
```

```
        /// The string used at the
beginning of the fatal error message.
        public let prefix: String

        /// The filename used in the
error message.
        public let file: StaticString

        /// The line number used in the
error message.
        public let line: UInt

        /// Creates a publisher that
raises a fatal error upon receiving any
failure, and otherwise republishes all
received input.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - prefix: The string used
at the beginning of the fatal error
message.
        ///   - file: The filename used
in the error message.
        ///   - line: The line number
used in the error message.
        public init(upstream: Upstream,
prefix: String, file: StaticString, line:
UInt)

        /// Attaches the specified
subscriber to this publisher.
```

```
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
S.Failure == Never
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that ignores elements
from the upstream publisher until it
receives an element from second
publisher.
    public struct
DropUntilOutput<Upstream, Other> :
Publisher where Upstream : Publisher,
Other : Publisher, Upstream.Failure ==
Other.Failure {
```

```
        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// A publisher to monitor for
its first emitted element.
        public let other: Other

        /// Creates a publisher that
ignores elements from the upstream
publisher until it receives an element
from another publisher.
        ///
        /// - Parameters:
        ///   - upstream: A publisher to
drop elements from while waiting for
another publisher to emit elements.
```

```
        ///   - other: A publisher to
monitor for its first emitted element.
        public init(upstream: Upstream,
other: Other)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
Other.Failure == S.Failure
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that performs the
specified closures when publisher events
occur.
```

```
    public struct
HandleEvents<Upstream> : Publisher where
Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// A closure that executes when
the publisher receives the subscription
from the upstream publisher.
        public var receiveSubscription:
((any Subscription) -> Void)?

        /// A closure that executes when
the publisher receives a value from the
upstream publisher.
```

```
        public var receiveOutput:
((Publishers.HandleEvents<Upstream>.Outpu
t) -> Void)?

        /// A closure that executes when
the upstream publisher finishes normally
or terminates with an error.
        public var receiveCompletion:
((Subscribers.Completion<Publishers.Handl
eEvents<Upstream>.Failure>) -> Void)?

        /// A closure that executes when
the downstream receiver cancels
publishing.
        public var receiveCancel: (() ->
Void)?

        /// A closure that executes when
the publisher receives a request for more
elements.
        public var receiveRequest:
((Subscribers.Demand) -> Void)?

        /// Creates a publisher that
performs the specified closures when
publisher events occur.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - receiveSubscription: A
closure that executes when the publisher
receives the subscription from the
```

```
upstream publisher.
        ///   - receiveOutput: A closure
that executes when the publisher receives
a value from the upstream publisher.
        ///   - receiveCompletion: A
closure that executes when the publisher
receives the completion from the upstream
publisher.
        ///   - receiveCancel: A closure
that executes when the downstream
receiver cancels publishing.
        ///   - receiveRequest: A closure
that executes when the publisher receives
a request for more elements.
        public init(upstream: Upstream,
receiveSubscription: ((any Subscription)
-> Void)? = nil, receiveOutput:
((Publishers.HandleEvents<Upstream>.Outpu
t) -> Void)? = nil, receiveCompletion:
((Subscribers.Completion<Publishers.Handl
eEvents<Upstream>.Failure>) -> Void)? =
nil, receiveCancel: (() -> Void)? = nil,
receiveRequest: ((Subscribers.Demand) ->
Void)?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
```

```
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that emits all of one
publisher's elements before those from
another publisher.
    public struct Concatenate<Prefix,
Suffix> : Publisher where Prefix :
Publisher, Suffix : Publisher,
Prefix.Failure == Suffix.Failure,
Prefix.Output == Suffix.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
source publishers' output type.
        public typealias Output =
Suffix.Output
```

```
        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
source publishers' failure type.
        public typealias Failure =
Suffix.Failure

        /// The publisher to republish,
in its entirety, before republishing
elements from `suffix`.
        public let prefix: Prefix

        /// The publisher to republish
only after `prefix` finishes.
        public let suffix: Suffix

        /// Creates a publisher that
emits all of one publisher’s elements
before those from another publisher.
        /// - Parameters:
        ///   - prefix: The publisher to
republish, in its entirety, before
republishing elements from `suffix`.
        ///   - suffix: The publisher to
republish only after `prefix` finishes.
        public init(prefix: Prefix,
suffix: Suffix)

        /// Attaches the specified
subscriber to this publisher.
        ///
```

```
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Suffix.Failure == S.Failure,
Suffix.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes
elements only after a specified time
interval elapses between events.
    public struct Debounce<Upstream,
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
```

```
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The amount of time the
publisher should wait before publishing
an element.
        public let dueTime:
Context.SchedulerTimeType.Stride

        /// The scheduler on which this
publisher delivers elements.
        public let scheduler: Context

        /// Scheduler options that
customize this publisher’s delivery of
elements.
        public let options:
Context.SchedulerOptions?

        /// Creates a publisher that
```

```
publishes elements only after a specified
time interval elapses between events.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - dueTime: The amount of
time the publisher should wait before
publishing an element.
        ///   - scheduler: The scheduler
on which this publisher delivers
elements.
        ///   - options: Scheduler
options that customize this publisher’s
delivery of elements.
        public init(upstream: Upstream,
dueTime:
Context.SchedulerTimeType.Stride,
scheduler: Context, options:
Context.SchedulerOptions?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
```

```
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that waits until
after the stream finishes, and then
publishes the last element of the stream.
    public struct Last<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
```

```
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
waits until after the stream finishes and
then publishes the last element of the
stream.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
```

```
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that transforms all
elements from the upstream publisher with
a provided closure.
    public struct Map<Upstream, Output> :
Publisher where Upstream : Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The closure that transforms
elements from the upstream publisher.
        public let transform:
(Upstream.Output) -> Output

        /// Creates a publisher that
transforms all elements from the upstream
publisher with a provided closure.
```

```
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - transform: The closure
that transforms elements from the
upstream publisher.
        public init(upstream: Upstream,
transform: @escaping (Upstream.Output) ->
Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, Upstream.Failure
== S.Failure
    }

    /// A publisher that transforms all
elements from the upstream publisher with
```

```
a provided error-throwing closure.
    public struct TryMap<Upstream,
Output> : Publisher where Upstream :
Publisher {

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The error-throwing closure
that transforms elements from the
upstream publisher.
        public let transform:
(Upstream.Output) throws -> Output

        /// Creates a publisher that
transforms all elements from the upstream
publisher with a provided error-throwing
closure.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - transform: The error-
```

```
throwing closure that transforms elements
from the upstream publisher.
        public init(upstream: Upstream,
transform: @escaping (Upstream.Output)
throws -> Output)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Output ==
S.Input, S : Subscriber, S.Failure == any
Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that terminates
publishing if the upstream publisher
```

```
exceeds a specified time interval without
producing an element.
    public struct Timeout<Upstream,
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The maximum time interval the
publisher can go without emitting an
element, expressed in the time system of
the scheduler.
        public let interval:
Context.SchedulerTimeType.Stride
```

```
        /// The scheduler on which to
deliver events.
        public let scheduler: Context

        /// Scheduler options that
customize the delivery of elements.
        public let options:
Context.SchedulerOptions?

        /// A closure that executes if
the publisher times out. The publisher
sends the failure returned by this
closure to the subscriber as the reason
for termination.
        public let customError: (() ->
Publishers.Timeout<Upstream,
Context>.Failure)?

        /// Creates a publisher that
terminates publishing if the upstream
publisher exceeds the specified time
interval without producing an element.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - interval: The maximum
time interval the publisher can go
without emitting an element, expressed in
the time system of the scheduler.
        ///   - scheduler: The scheduler
on which to deliver events.
        ///   - options: Scheduler
```

```
options that customize the delivery of
elements.
        ///   - customError: A closure
that executes if the publisher times out.
The publisher sends the failure returned
by this closure to the subscriber as the
reason for termination.
        public init(upstream: Upstream,
interval:
Context.SchedulerTimeType.Stride,
scheduler: Context, options:
Context.SchedulerOptions?, customError:
(() -> Publishers.Timeout<Upstream,
Context>.Failure)?)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
```

```
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A strategy for filling a buffer.
    public enum PrefetchStrategy {

        /// A strategy to fill the buffer
at subscription time, and keep it full
thereafter.
        ///
        /// This strategy starts by
making a demand equal to the buffer's
size from the upstream when the
subscriber first connects. Afterwards, it
continues to demand elements from the
upstream to try to keep the buffer full.
        case keepFull

        /// A strategy that avoids
prefetching and instead performs requests
on demand.
        ///
        /// This strategy just forwards
the downstream's requests to the upstream
publisher.
        case byRequest

        /// Returns a Boolean value
indicating whether two values are equal.
```

```
        ///
        /// Equality is the inverse of
inequality. For any values `a` and `b`,
        /// `a == b` implies that `a !=
b` is `false`.
        ///
        /// - Parameters:
        ///   - lhs: A value to compare.
        ///   - rhs: Another value to
compare.
        public static func == (a:
Publishers.PrefetchStrategy, b:
Publishers.PrefetchStrategy) -> Bool

        /// Hashes the essential
components of this value by feeding them
into the
        /// given hasher.
        ///
        /// Implement this method to
conform to the `Hashable` protocol. The
        /// components used for hashing
must be the same as the components
compared
        /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
        /// with each of these
components.
        ///
        /// - Important: In your
implementation of `hash(into:)`,
        ///   don't call `finalize()` on
the `hasher` instance provided,
```

```
        ///   or replace it with a
different instance.
        ///   Doing so may become a
compile-time error in the future.
        ///
        /// - Parameter hasher: The
hasher to use when combining the
components
        ///   of this instance.
        public func hash(into hasher:
inout Hasher)

        /// The hash value.
        ///
        /// Hash values are not
guaranteed to be equal across different
executions of
        /// your program. Do not save
hash values to use during a future
execution.
        ///
        /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
        ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
        ///   The compiler provides an
implementation for `hashValue` for you.
        public var hashValue: Int { get }
    }

    /// A strategy that handles
```

```
exhaustion of a buffer’s capacity.
    public enum
BufferingStrategy<Failure> where
Failure : Error {

        /// When the buffer is full,
discard the newly received element.
        case dropNewest

        /// When the buffer is full,
discard the oldest element in the buffer.
        case dropOldest

        /// When the buffer is full,
execute the closure to provide a custom
error.
        case customError(() -> Failure)
    }

    /// A publisher that buffers elements
from an upstream publisher.
    public struct Buffer<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
```

```
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The maximum number of
elements to store.
        public let size: Int

        /// The strategy for initially
populating the buffer.
        public let prefetch:
Publishers.PrefetchStrategy

        /// The action to take when the
buffer becomes full.
        public let whenFull:
Publishers.BufferingStrategy<Publishers.B
uffer<Upstream>.Failure>

        /// Creates a publisher that
buffers elements received from an
upstream publisher.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        /// - Parameter size: The maximum
```

```
number of elements to store.
        /// - Parameter prefetch: The
strategy for initially populating the
buffer.
        /// - Parameter whenFull: The
action to take when the buffer becomes
full.
        public init(upstream: Upstream,
size: Int, prefetch:
Publishers.PrefetchStrategy, whenFull:
Publishers.BufferingStrategy<Publishers.B
uffer<Upstream>.Failure>)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes a
given sequence of elements.
    ///
    /// When the publisher exhausts the
elements in the sequence, the next
request causes the publisher to finish.
    public struct Sequence<Elements,
Failure> : Publisher where Elements :
Sequence, Failure : Error {

        /// The kind of values published
by this publisher.
        public typealias Output =
Elements.Element

        /// The sequence of elements to
publish.
        public let sequence: Elements

        /// Creates a publisher for a
sequence of elements.
        ///
        /// - Parameter sequence: The
sequence of elements to publish.
        public init(sequence: Elements)

        /// Attaches the specified
subscriber to this publisher.
```

```
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Failure
== S.Failure, S : Subscriber,
Elements.Element == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher created by applying
the zip function to two upstream
publishers.
    ///
    /// Use `Publishers.Zip` to combine
the latest elements from two publishers
and emit a tuple to the downstream. The
returned publisher waits until both
publishers have emitted an event, then
delivers the oldest unconsumed event from
```

```
each publisher together as a tuple to the
subscriber.
    ///
    /// Much like a zipper or zip
fastener on a piece of clothing pulls
together rows of teeth to link the two
sides, `Publishers.Zip` combines streams
from two different publishers by linking
pairs of elements from each side.
    ///
    /// If either upstream publisher
finishes successfully or fails with an
error, so too does the zipped publisher.
    public struct Zip<A, B> : Publisher
where A : Publisher, B : Publisher,
A.Failure == B.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces two-
element tuples, whose members' types
correspond to the types produced by the
upstream publishers.
        public typealias Output =
(A.Output, B.Output)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
```

```
A.Failure

        /// A publisher to zip.
        public let a: A

        /// Another publisher to zip.
        public let b: B

        /// Creates a publisher that
applies the zip function to two upstream
publishers.
        /// - Parameters:
        ///   - a: A publisher to zip.
        ///   - b: Another publisher to
zip.
        public init(_ a: A, _ b: B)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
```

```
Subscriber, B.Failure == S.Failure,
S.Input == (A.Output, B.Output)
    }

    /// A publisher created by applying
the zip function to three upstream
publishers.
    ///
    /// Use a `Publishers.Zip3` to
combine the latest elements from three
publishers and emit a tuple to the
downstream. The returned publisher waits
until all three publishers have emitted
an event, then delivers the oldest
unconsumed event from each publisher as a
tuple to the subscriber.
    ///
    /// If any upstream publisher
finishes successfully or fails with an
error, so too does the zipped publisher.
    public struct Zip3<A, B, C> :
Publisher where A : Publisher, B :
Publisher, C : Publisher, A.Failure ==
B.Failure, B.Failure == C.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces
three-element tuples, whose members'
types correspond to the types produced by
the upstream publishers.
        public typealias Output =
```

```
(A.Output, B.Output, C.Output)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to zip.
        public let a: A

        /// A second publisher to zip.
        public let b: B

        /// A third publisher to zip.
        public let c: C

        /// Creates a publisher that
applies the zip function to three
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to zip.
        ///   - b: A second publisher to
zip.
        ///   - c: A third publisher to
zip.
        public init(_ a: A, _ b: B, _ c:
C)

        /// Attaches the specified
subscriber to this publisher.
```

```
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, C.Failure == S.Failure,
S.Input == (A.Output, B.Output, C.Output)
    }

    /// A publisher created by applying
the zip function to four upstream
publishers.
    ///
    /// Use a `Publishers.Zip4` to
combine the latest elements from four
publishers and emit a tuple to the
downstream. The returned publisher waits
until all four publishers have emitted an
event, then delivers the oldest
unconsumed event from each publisher as a
tuple to the subscriber.
    ///
    /// If any upstream publisher
finishes successfully or fails with an
```

```
error, so too does the zipped publisher.
    public struct Zip4<A, B, C, D> :
Publisher where A : Publisher, B :
Publisher, C : Publisher, D : Publisher,
A.Failure == B.Failure, B.Failure ==
C.Failure, C.Failure == D.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher produces four-
element tuples, whose members' types
correspond to the types produced by the
upstream publishers.
        public typealias Output =
(A.Output, B.Output, C.Output, D.Output)

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publishers' common failure type.
        public typealias Failure =
A.Failure

        /// A publisher to zip.
        public let a: A

        /// A second publisher to zip.
        public let b: B

        /// A third publisher to zip.
        public let c: C
```

```
        /// A fourth publisher to zip.
        public let d: D

        /// Creates a publisher created
by applying the zip function to four
upstream publishers.
        /// - Parameters:
        ///   - a: A publisher to zip.
        ///   - b: A second publisher to
zip.
        ///   - c: A third publisher to
zip.
        ///   - d: A fourth publisher to
zip.
        public init(_ a: A, _ b: B, _ c:
C, _ d: D)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
```

```
receive<S>(subscriber: S) where S :
Subscriber, D.Failure == S.Failure,
S.Input == (A.Output, B.Output, C.Output,
D.Output)
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes
elements specified by a range in the
sequence of published elements.
    public struct Output<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure
```

```
        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// The range of elements to
publish.
        public let range:
CountableRange<Int>

        /// Creates a publisher that
publishes elements specified by a range.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - range: The range of
elements to publish.
        public init(upstream: Upstream,
range: CountableRange<Int>)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
```

```
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that handles errors
from an upstream publisher by replacing
the failed publisher with another
publisher.
    public struct Catch<Upstream,
NewPublisher> : Publisher where
Upstream : Publisher, NewPublisher :
Publisher, Upstream.Output ==
NewPublisher.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
```

```
        ///
        /// This publisher uses the
replacement publisher's failure type.
        public typealias Failure =
NewPublisher.Failure

        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// A closure that accepts the
upstream failure as input and returns a
publisher to replace the upstream
publisher.
        public let handler:
(Upstream.Failure) -> NewPublisher

        /// Creates a publisher that
handles errors from an upstream publisher
by replacing the failed publisher with
another publisher.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - handler: A closure that
accepts the upstream failure as input and
returns a publisher to replace the
upstream publisher.
        public init(upstream: Upstream,
handler: @escaping (Upstream.Failure) ->
NewPublisher)
```

```
        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, NewPublisher.Failure ==
S.Failure, NewPublisher.Output == S.Input
    }

    /// A publisher that handles errors
from an upstream publisher by replacing
the failed publisher with another
publisher or producing a new error.
    ///
    /// Because this publisher's handler
can throw an error,
``Publishers/TryCatch`` defines its
``Publisher/Failure`` type as `Error`.
This is different from
``Publishers/Catch``, which gets its
failure type from the replacement
```

```
publisher.
    public struct TryCatch<Upstream,
NewPublisher> : Publisher where
Upstream : Publisher, NewPublisher :
Publisher, Upstream.Output ==
NewPublisher.Output {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error

        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// A closure that accepts the
upstream failure as input and either
returns a publisher to replace the
upstream publisher or throws an error.
        public let handler:
```

```
(Upstream.Failure) throws -> NewPublisher

        /// Creates a publisher that
handles errors from an upstream publisher
by replacing the failed publisher with
another publisher or by throwing an
error.
        ///
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - handler: A closure that
accepts the upstream failure as input and
either returns a publisher to replace the
upstream publisher. If this closure
throws an error, the publisher terminates
with the thrown error.
        public init(upstream: Upstream,
handler: @escaping (Upstream.Failure)
throws -> NewPublisher)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
```

```
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, NewPublisher.Output ==
S.Input, S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that transforms
elements from an upstream publisher into
a new publisher.
    public struct FlatMap<NewPublisher,
Upstream> : Publisher where
NewPublisher : Publisher, Upstream :
Publisher, NewPublisher.Failure ==
Upstream.Failure {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses the
output type declared by the new
publisher.
        public typealias Output =
NewPublisher.Output

        /// The kind of errors this
```

```
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The maximum number of
concurrent publisher subscriptions
        public let maxPublishers:
Subscribers.Demand

        /// A closure that takes an
element as a parameter and returns a
publisher that produces elements of that
type.
        public let transform:
(Upstream.Output) -> NewPublisher

        /// Creates a publisher that
transforms elements from an upstream
publisher into a new publisher.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives
elements.
        ///   - maxPublishers: The
maximum number of concurrent publisher
subscriptions.
```

```
        ///   - transform: A closure that
takes an element as a parameter and
returns a publisher that produces
elements of that type.
        public init(upstream: Upstream,
maxPublishers: Subscribers.Demand,
transform: @escaping (Upstream.Output) ->
NewPublisher)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, NewPublisher.Output ==
S.Input, Upstream.Failure == S.Failure
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {
```

```
    /// A publisher that delays delivery
of elements and completion to the
downstream receiver.
    public struct Delay<Upstream,
Context> : Publisher where Upstream :
Publisher, Context : Scheduler {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives its elements.
        public let upstream: Upstream

        /// The amount of time to delay.
        public let interval:
Context.SchedulerTimeType.Stride

        /// The allowed tolerance in
```

```
firing delayed events.
        public let tolerance:
Context.SchedulerTimeType.Stride

        /// The scheduler to deliver the
delayed events.
        public let scheduler: Context

        /// Options relevant to the
scheduler's behavior.
        public let options:
Context.SchedulerOptions?

        /// Creates a publisher that
delays delivery of elements and
completion to the downstream receiver.
        /// - Parameters:
        ///   - upstream: The publisher
from which this publisher receives its
elements.
        ///   - interval: The amount of
time to delay.
        ///   - tolerance: The allowed
tolerance in delivering delayed events.
The `Delay` publisher may deliver
elements this much sooner or later than
the interval specifies.
        ///   - scheduler: The scheduler
to deliver the delayed events.
        ///   - options: Options relevant
to the scheduler's behavior.
        public init(upstream: Upstream,
interval:
```

```
Context.SchedulerTimeType.Stride,
tolerance:
Context.SchedulerTimeType.Stride,
scheduler: Context, options:
Context.SchedulerOptions? = nil)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that omits a
specified number of elements before
```

```
republishing later elements.
    public struct Drop<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The number of elements to
drop.
        public let count: Int

        /// Creates a publisher that
omits a specified number of elements
before republishing later elements.
        /// - Parameters:
        ///   - upstream: The publisher
```

```
from which this publisher receives
elements.
        ///   - count: The number of
elements to drop.
        public init(upstream: Upstream,
count: Int)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers {

    /// A publisher that publishes the
```

```
first element of a stream, then finishes.
    public struct First<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// Creates a publisher that
publishes the first element of a stream,
then finishes.
        /// - Parameter upstream: The
publisher from which this publisher
receives elements.
        public init(upstream: Upstream)

        /// Attaches the specified
```

```
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that only publishes
the first element of a stream to satisfy
a predicate closure.
    public struct FirstWhere<Upstream> :
Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output
```

```
        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher uses its
upstream publisher's failure type.
        public typealias Failure =
Upstream.Failure

        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The closure that determines
whether to publish an element.
        public let predicate:
(Publishers.FirstWhere<Upstream>.Output)
-> Bool

        public init(upstream: Upstream,
predicate: @escaping
(Publishers.FirstWhere<Upstream>.Output)
-> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
```

```
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Failure ==
S.Failure, Upstream.Output == S.Input
    }

    /// A publisher that only publishes
the first element of a stream to satisfy
a throwing predicate closure.
    public struct TryFirstWhere<Upstream>
: Publisher where Upstream : Publisher {

        /// The kind of values published
by this publisher.
        ///
        /// This publisher uses its
upstream publisher's output type.
        public typealias Output =
Upstream.Output

        /// The kind of errors this
publisher might publish.
        ///
        /// This publisher produces the
Swift
<doc://com.apple.documentation/documentat
ion/Swift/Error> type.
        public typealias Failure = Error
```

```
        /// The publisher from which this
publisher receives elements.
        public let upstream: Upstream

        /// The error-throwing closure
that determines whether to publish an
element.
        public let predicate:
(Publishers.TryFirstWhere<Upstream>.Outpu
t) throws -> Bool

        public init(upstream: Upstream,
predicate: @escaping
(Publishers.TryFirstWhere<Upstream>.Outpu
t) throws -> Bool)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where S :
Subscriber, Upstream.Output == S.Input,
```

```
S.Failure == any Error
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Filter {

    public func filter(_ isIncluded:
@escaping
(Publishers.Filter<Upstream>.Output) ->
Bool) -> Publishers.Filter<Upstream>

    public func tryFilter(_ isIncluded:
@escaping
(Publishers.Filter<Upstream>.Output)
throws -> Bool) ->
Publishers.TryFilter<Upstream>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.TryFilter {

    public func filter(_ isIncluded:
@escaping
(Publishers.TryFilter<Upstream>.Output)
-> Bool) ->
Publishers.TryFilter<Upstream>

    public func tryFilter(_ isIncluded:
@escaping
(Publishers.TryFilter<Upstream>.Output)
```

```
throws -> Bool) ->
Publishers.TryFilter<Upstream>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Contains : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A contains publisher to
compare for equality.
    ///   - rhs: Another contains
publisher to compare for equality.
    /// - Returns: `true` if the two
publishers' `upstream` and `output`
properties are equal; otherwise `false`.
    public static func == (lhs:
Publishers.Contains<Upstream>, rhs:
Publishers.Contains<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.CombineLatest :
Equatable where A : Equatable, B :
Equatable {

    /// Returns a Boolean value that
```

```
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A combineLatest
publisher to compare for equality.
    ///   - rhs: Another combineLatest
publisher to compare for equality.
    /// - Returns: `true` if the
corresponding upstream publishers of each
combineLatest publisher are equal;
otherwise `false`.
    public static func == (lhs:
Publishers.CombineLatest<A, B>, rhs:
Publishers.CombineLatest<A, B>) -> Bool
}

/// Returns a Boolean value that
indicates whether two publishers are
equivalent.
///
/// - Parameters:
///   - lhs: A combineLatest publisher to
compare for equality.
///   - rhs: Another combineLatest
publisher to compare for equality.
/// - Returns: `true` if the
corresponding upstream publishers of each
combineLatest publisher are equal;
otherwise `false`.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.CombineLatest3 :
```

```
Equatable where A : Equatable, B :
Equatable, C : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
Publishers.CombineLatest3<A, B, C>, rhs:
Publishers.CombineLatest3<A, B, C>) ->
Bool
}

/// Returns a Boolean value that
indicates whether two publishers are
equivalent.
///
/// - Parameters:
///   - lhs: A combineLatest publisher to
compare for equality.
///   - rhs: Another combineLatest
publisher to compare for equality.
/// - Returns: `true` if the
corresponding upstream publishers of each
combineLatest publisher are equal;
```

```
otherwise `false`.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.CombineLatest4 :
Equatable where A : Equatable, B :
Equatable, C : Equatable, D : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (lhs:
Publishers.CombineLatest4<A, B, C, D>,
rhs: Publishers.CombineLatest4<A, B, C,
D>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.SetFailureType :
Equatable where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
```

```
    /// - Parameters:
    ///   - lhs: A `SetFailureType`
publisher to compare for equality.
    ///   - rhs: Another `SetFailureType`
publisher to compare for equality.
    /// - Returns: `true` if the
publishers have equal `upstream`
properties; otherwise `false`.
    public static func == (lhs:
Publishers.SetFailureType<Upstream,
Failure>, rhs:
Publishers.SetFailureType<Upstream,
Failure>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Collect : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `Collect` instance to
compare.
    ///   - rhs: Another `Collect`
instance to compare.
    /// - Returns: `true` if the
corresponding `upstream` properties of
each publisher are equal; otherwise
`false`.
    public static func == (lhs:
```

```
Publishers.Collect<Upstream>, rhs:
Publishers.Collect<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.CollectByCount :
Equatable where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `CollectByCount`
instance to compare.
    ///   - rhs: Another `CollectByCount`
instance to compare.
    /// - Returns: `true` if the
corresponding `upstream` and `count`
properties of each publisher are equal;
otherwise `false`.
    public static func == (lhs:
Publishers.CollectByCount<Upstream>, rhs:
Publishers.CollectByCount<Upstream>) ->
Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.CompactMap {

    public func compactMap<T>(_
transform: @escaping (Output) -> T?) ->
```

```
Publishers.CompactMap<Upstream, T>

    public func map<T>(_ transform:
@escaping (Output) -> T) ->
Publishers.CompactMap<Upstream, T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.TryCompactMap {

    public func compactMap<T>(_
transform: @escaping (Output) throws ->
T?) -> Publishers.TryCompactMap<Upstream,
T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge : Equatable
where A : Equatable, B : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
publisher to compare for equality..
    /// - Returns: `true` if the two
merging - rhs: Another merging publisher
```

```
to compare for equality.
    public static func == (lhs:
Publishers.Merge<A, B>, rhs:
Publishers.Merge<A, B>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge3 : Equatable
where A : Equatable, B : Equatable, C :
Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
publisher to compare for equality.
    /// - Returns: `true` if the two
merging publishers have equal source
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Merge3<A, B, C>, rhs:
Publishers.Merge3<A, B, C>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge4 : Equatable
where A : Equatable, B : Equatable, C :
```

```
Equatable, D : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
publisher to compare for equality.
    /// - Returns: `true` if the two
merging publishers have equal source
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Merge4<A, B, C, D>, rhs:
Publishers.Merge4<A, B, C, D>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge5 : Equatable
where A : Equatable, B : Equatable, C :
Equatable, D : Equatable, E : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
```

```
publisher to compare for equality.
    /// - Returns: `true` if the two
merging publishers have equal source
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Merge5<A, B, C, D, E>, rhs:
Publishers.Merge5<A, B, C, D, E>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge6 : Equatable
where A : Equatable, B : Equatable, C :
Equatable, D : Equatable, E : Equatable,
F : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
publisher to compare for equality.
    /// - Returns: `true` if the two
merging publishers have equal source
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Merge6<A, B, C, D, E, F>, rhs:
Publishers.Merge6<A, B, C, D, E, F>) ->
Bool
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge7 : Equatable
where A : Equatable, B : Equatable, C :
Equatable, D : Equatable, E : Equatable,
F : Equatable, G : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
publisher to compare for equality.
    /// - Returns: `true` if the two
merging publishers have equal source
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Merge7<A, B, C, D, E, F, G>,
rhs: Publishers.Merge7<A, B, C, D, E, F,
G>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Merge8 : Equatable
where A : Equatable, B : Equatable, C :
Equatable, D : Equatable, E : Equatable,
F : Equatable, G : Equatable, H :
Equatable {
```

```
    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A merging publisher to
compare for equality.
    ///   - rhs: Another merging
publisher to compare for equality.
    /// - Returns: `true` if the two
merging publishers have equal source
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Merge8<A, B, C, D, E, F, G,
H>, rhs: Publishers.Merge8<A, B, C, D, E,
F, G, H>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.MergeMany :
Equatable where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `MergeMany` publisher
to compare for equality.
    ///   - rhs: Another `MergeMany`
publisher to compare for equality.
    /// - Returns: `true` if the
```

```
publishers have equal `publishers`
properties; otherwise `false`.
    public static func == (lhs:
Publishers.MergeMany<Upstream>, rhs:
Publishers.MergeMany<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Count : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///     /// - Parameters:
    ///   - lhs: A `Count` instance to
compare.
    ///   - rhs: Another `Count` instance
to compare.
    /// - Returns: `true` if the two
publishers' `upstream` properties are
equal; otherwise `false`.
    public static func == (lhs:
Publishers.Count<Upstream>, rhs:
Publishers.Count<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.IgnoreOutput :
Equatable where Upstream : Equatable {
```

```
    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: An ignore output
publisher to compare for equality.
    ///   - rhs: Another ignore output
publisher to compare for equality.
    /// - Returns: `true` if the two
publishers have equal upstream
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.IgnoreOutput<Upstream>, rhs:
Publishers.IgnoreOutput<Upstream>) ->
Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Retry : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `Retry` publisher to
compare for equality.
    ///   - rhs: Another `Retry`
publisher to compare for equality.
    /// - Returns: `true` if the
publishers have equal `upstream` and
```

```
`retries` properties; otherwise `false`.
    public static func == (lhs:
Publishers.Retry<Upstream>, rhs:
Publishers.Retry<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.ReplaceEmpty :
Equatable where Upstream : Equatable,
Upstream.Output : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A replace empty
publisher to compare for equality.
    ///   - rhs: Another replace empty
publisher to compare for equality.
    /// - Returns: `true` if the two
publishers have equal upstream publishers
and output elements; otherwise `false`.
    public static func == (lhs:
Publishers.ReplaceEmpty<Upstream>, rhs:
Publishers.ReplaceEmpty<Upstream>) ->
Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.ReplaceError :
```

```
Equatable where Upstream : Equatable,
Upstream.Output : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A replace error
publisher to compare for equality.
    ///   - rhs: Another replace error
publisher to compare for equality.
    /// - Returns: `true` if the two
publishers have equal upstream publishers
and output elements; otherwise `false`.
    public static func == (lhs:
Publishers.ReplaceError<Upstream>, rhs:
Publishers.ReplaceError<Upstream>) ->
Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.DropUntilOutput :
Equatable where Upstream : Equatable,
Other : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A
`Publishers.DropUntilOutput` instance to
```

```
compare for equality.
    ///   - rhs: Another
`Publishers.DropUntilOutput` instance to
compare for equality.
    /// - Returns: `true` if the
publishers have equal `upstream` and
`other` properties; otherwise `false`.
    public static func == (lhs:
Publishers.DropUntilOutput<Upstream,
Other>, rhs:
Publishers.DropUntilOutput<Upstream,
Other>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Concatenate :
Equatable where Prefix : Equatable,
Suffix : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A concatenate publisher
to compare for equality.
    ///   - rhs: Another concatenate
publisher to compare for equality.
    /// - Returns: `true` if the two
publishers' `prefix` and `suffix`
properties are equal; otherwise `false`.
    public static func == (lhs:
```

```
Publishers.Concatenate<Prefix, Suffix>,
rhs: Publishers.Concatenate<Prefix,
Suffix>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Last : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A last publisher to
compare for equality.
    ///   - rhs: Another last publisher
to compare for equality.
    /// - Returns: `true` if the two
publishers have equal upstream
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.Last<Upstream>, rhs:
Publishers.Last<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Map {

    public func map<T>(_ transform:
@escaping (Output) -> T) ->
```

```
Publishers.Map<Upstream, T>

    public func tryMap<T>(_ transform:
@escaping (Output) throws -> T) ->
Publishers.TryMap<Upstream, T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.TryMap {

    public func map<T>(_ transform:
@escaping (Output) -> T) ->
Publishers.TryMap<Upstream, T>

    public func tryMap<T>(_ transform:
@escaping (Output) throws -> T) ->
Publishers.TryMap<Upstream, T>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.PrefetchStrategy :
Equatable {
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.PrefetchStrategy :
Hashable {
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Failure == Never {

    public func min(by
areInIncreasingOrder:
(Publishers.Sequence<Elements,
Failure>.Output,
Publishers.Sequence<Elements,
Failure>.Output) -> Bool) ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher

    public func max(by
areInIncreasingOrder:
(Publishers.Sequence<Elements,
Failure>.Output,
Publishers.Sequence<Elements,
Failure>.Output) -> Bool) ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher

    public func first(where predicate:
(Publishers.Sequence<Elements,
Failure>.Output) -> Bool) ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence {
```

```
    public func allSatisfy(_ predicate:
(Publishers.Sequence<Elements,
Failure>.Output) -> Bool) -> Result<Bool,
Failure>.Publisher

    public func tryAllSatisfy(_
predicate: (Publishers.Sequence<Elements,
Failure>.Output) throws -> Bool) ->
Result<Bool, any Error>.Publisher

    public func collect() ->
Result<[Publishers.Sequence<Elements,
Failure>.Output], Failure>.Publisher

    public func compactMap<T>(_
transform: (Publishers.Sequence<Elements,
Failure>.Output) -> T?) ->
Publishers.Sequence<[T], Failure>

    public func contains(where predicate:
(Publishers.Sequence<Elements,
Failure>.Output) -> Bool) -> Result<Bool,
Failure>.Publisher

    public func tryContains(where
predicate: (Publishers.Sequence<Elements,
Failure>.Output) throws -> Bool) ->
Result<Bool, any Error>.Publisher

    public func drop(while predicate:
(Elements.Element) -> Bool) ->
Publishers.Sequence<DropWhileSequence<Ele
ments>, Failure>
```

```
    public func dropFirst(_ count: Int =
1) ->
Publishers.Sequence<DropFirstSequence<Ele
ments>, Failure>

    public func filter(_ isIncluded:
(Publishers.Sequence<Elements,
Failure>.Output) -> Bool) ->
Publishers.Sequence<[Publishers.Sequence<
Elements, Failure>.Output], Failure>

    public func ignoreOutput() ->
Empty<Publishers.Sequence<Elements,
Failure>.Output, Failure>

    public func map<T>(_ transform:
(Elements.Element) -> T) ->
Publishers.Sequence<[T], Failure>

    public func prefix(_ maxLength: Int)
->
Publishers.Sequence<PrefixSequence<Elemen
ts>, Failure>

    public func prefix(while predicate:
(Elements.Element) -> Bool) ->
Publishers.Sequence<[Elements.Element],
Failure>

    public func reduce<T>(_
initialResult: T, _ nextPartialResult:
@escaping (T,
```

```
Publishers.Sequence<Elements,
Failure>.Output) -> T) -> Result<T,
Failure>.Publisher

    public func tryReduce<T>(_
initialResult: T, _ nextPartialResult:
@escaping (T,
Publishers.Sequence<Elements,
Failure>.Output) throws -> T) ->
Result<T, any Error>.Publisher

    public func replaceNil<T>(with
output: T) ->
Publishers.Sequence<[Publishers.Sequence<
Elements, Failure>.Output], Failure>
where Elements.Element == T?

    public func scan<T>(_ initialResult:
T, _ nextPartialResult: @escaping (T,
Publishers.Sequence<Elements,
Failure>.Output) -> T) ->
Publishers.Sequence<[T], Failure>

    public func setFailureType<E>(to
error: E.Type) ->
Publishers.Sequence<Elements, E> where
E : Error
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements.Element : Equatable {
```

```
    public func removeDuplicates() ->
Publishers.Sequence<[Publishers.Sequence<
Elements, Failure>.Output], Failure>

    public func contains(_ output:
Elements.Element) -> Result<Bool,
Failure>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Failure == Never, Elements.Element :
Comparable {

    public func min() ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher

    public func max() ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : Collection, Failure == Never {

    public func first() ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher
```

```
    public func output(at index:
Elements.Index) ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : Collection {

    public func count() -> Result<Int,
Failure>.Publisher

    public func output(in range:
Range<Elements.Index>) ->
Publishers.Sequence<[Publishers.Sequence<
Elements, Failure>.Output], Failure>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : BidirectionalCollection,
Failure == Never {

    public func last() ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher

    public func last(where predicate:
(Publishers.Sequence<Elements,
```

```
Failure>.Output) -> Bool) ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : RandomAccessCollection,
Failure == Never {

    public func output(at index:
Elements.Index) ->
Optional<Publishers.Sequence<Elements,
Failure>.Output>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : RandomAccessCollection {

    public func output(in range:
Range<Elements.Index>) ->
Publishers.Sequence<[Publishers.Sequence<
Elements, Failure>.Output], Failure>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : RandomAccessCollection,
Failure == Never {
```

```
    public func count() -> Just<Int>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : RandomAccessCollection {

    public func count() -> Result<Int,
Failure>.Publisher
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence where
Elements : RangeReplaceableCollection {

    public func prepend(_ elements:
Publishers.Sequence<Elements,
Failure>.Output...) ->
Publishers.Sequence<Elements, Failure>

    public func prepend<S>(_ elements: S)
-> Publishers.Sequence<Elements, Failure>
where S : Sequence, Elements.Element ==
S.Element

    public func prepend(_ publisher:
Publishers.Sequence<Elements, Failure>)
-> Publishers.Sequence<Elements, Failure>

    public func append(_ elements:
```

```
Publishers.Sequence<Elements,
Failure>.Output...) ->
Publishers.Sequence<Elements, Failure>

    public func append<S>(_ elements: S)
-> Publishers.Sequence<Elements, Failure>
where S : Sequence, Elements.Element ==
S.Element

    public func append(_ publisher:
Publishers.Sequence<Elements, Failure>)
-> Publishers.Sequence<Elements, Failure>
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Sequence : Equatable
where Elements : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A `Sequence` publisher
to compare for equality.
    ///   - rhs: Another `Sequewnce`
publisher to compare for equality.
    /// - Returns: `true` if the
publishers have equal `sequence`
properties; otherwise `false`.
    public static func == (lhs:
Publishers.Sequence<Elements, Failure>,
rhs: Publishers.Sequence<Elements,
```

```
Failure>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Zip : Equatable
where A : Equatable, B : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A zip publisher to
compare for equality.
    ///   - rhs: Another zip publisher to
compare for equality.
    /// - Returns: `true` if the
corresponding upstream publishers of each
zip publisher are equal; otherwise
`false`.
    public static func == (lhs:
Publishers.Zip<A, B>, rhs:
Publishers.Zip<A, B>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Zip3 : Equatable
where A : Equatable, B : Equatable, C :
Equatable {

    /// Returns a Boolean value that
```

```
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A zip publisher to
compare for equality.
    ///   - rhs: Another zip publisher to
compare for equality.
    /// - Returns: `true` if the
corresponding upstream publishers of each
zip publisher are equal; otherwise
`false`.
    public static func == (lhs:
Publishers.Zip3<A, B, C>, rhs:
Publishers.Zip3<A, B, C>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Zip4 : Equatable
where A : Equatable, B : Equatable, C :
Equatable, D : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    ///
    /// - Parameters:
    ///   - lhs: A zip publisher to
compare for equality.
    ///   - rhs: Another zip publisher to
compare for equality.
    /// - Returns: `true` if the
```

```
corresponding upstream publishers of each
zip publisher are equal; otherwise
`false`.
    public static func == (lhs:
Publishers.Zip4<A, B, C, D>, rhs:
Publishers.Zip4<A, B, C, D>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Output : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: An `Output` publisher to
compare for equality.
    ///   - rhs: Another `Output`
publisher to compare for equality.
    /// - Returns: `true` if the
publishers have equal `upstream` and
`range` properties; otherwise `false`.
    public static func == (lhs:
Publishers.Output<Upstream>, rhs:
Publishers.Output<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.Drop : Equatable
where Upstream : Equatable {
```

```
    /// Returns a Boolean value that
indicates whether two publishers are
equivalent.
    /// - Parameters:
    ///   - lhs: A drop publisher to
compare for equality.
    ///   - rhs: Another drop publisher
to compare for equality.
    /// - Returns: `true` if the
publishers have equal `upstream` and
`count` properties; otherwise `false`.
    public static func == (lhs:
Publishers.Drop<Upstream>, rhs:
Publishers.Drop<Upstream>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Publishers.First : Equatable
where Upstream : Equatable {

    /// Returns a Boolean value that
indicates whether two first publishers
have equal upstream publishers.
    ///
    /// - Parameters:
    ///   - lhs: A drop publisher to
compare for equality.
    ///   - rhs: Another drop publisher
to compare for equality.
    /// - Returns: `true` if the two
publishers have equal upstream
```

```
publishers; otherwise `false`.
    public static func == (lhs:
Publishers.First<Upstream>, rhs:
Publishers.First<Upstream>) -> Bool
}

/// A publisher that allows for recording
a series of inputs and a completion, for
later playback to each subscriber.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public struct Record<Output, Failure> :
Publisher where Failure : Error {

    /// The recorded output and
completion.
    public let recording: Record<Output,
Failure>.Recording

    /// Creates a publisher to
interactively record a series of outputs
and a completion.
    ///
    /// - Parameter record: A recording
instance that can be retrieved after
completion to create new record
publishers to replay the recording.
    public init(record: (inout
Record<Output, Failure>.Recording) ->
Void)

    /// Creates a record publisher from
an existing recording.
```

```
    ///
    /// - Parameter recording: A
previously-recorded recording of
published elements and a completion.
    public init(recording: Record<Output,
Failure>.Recording)

    /// Creates a record publisher to
publish the provided elements, followed
by the provided completion value.
    ///
    /// - Parameters:
    ///   - output: An array of output
elements to publish.
    ///   - completion: The completion
value with which to end publishing.
    public init(output: [Output],
completion:
Subscribers.Completion<Failure>)

    /// Attaches the specified subscriber
to this publisher.
    ///
    /// Implementations of ``Publisher``
must implement this method.
    ///
    /// The provided implementation of
``Publisher/subscribe(_:)-4u8kn``calls
this method.
    ///
    /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
```

```
values.
    public func receive<S>(subscriber: S)
where Output == S.Input, Failure ==
S.Failure, S : Subscriber

    /// A recorded sequence of outputs,
followed by a completion value.
    public struct Recording {

        public typealias Input = Output

        /// The output which will be sent
to a `Subscriber`.
        public var output: [Output] { get
}

        /// The completion which will be
sent to a `Subscriber`.
        public var completion:
Subscribers.Completion<Failure> { get }

        /// Set up a recording in a state
ready to receive output.
        public init()

        /// Set up a complete recording
with the specified output and completion.
        public init(output: [Output],
completion:
Subscribers.Completion<Failure>
= .finished)

        /// Add an output to the
```

```
recording.
        ///
        /// A `fatalError` will be raised
if output is added after adding
completion.
        public mutating func receive(_
input: Record<Output,
Failure>.Recording.Input)

        /// Add a completion to the
recording.
        ///
        /// A `fatalError` will be raised
if more than one completion is added.
        public mutating func
receive(completion:
Subscribers.Completion<Failure>)
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Record : Codable where Output :
Decodable, Output : Encodable, Failure :
Decodable, Failure : Encodable {

    /// Encodes this value into the given
encoder.
    ///
    /// If the value fails to encode
anything, `encoder` will encode an empty
    /// keyed container in its place.
    ///
```

```
    /// This function throws an error if
any values are invalid for the given
    /// encoder's format.
    ///
    /// - Parameter encoder: The encoder
to write data to.
    public func encode(to encoder: any
Encoder) throws

    /// Creates a new instance by
decoding from the given decoder.
    ///
    /// This initializer throws an error
if reading from the decoder fails, or
    /// if the data read is corrupted or
otherwise invalid.
    ///
    /// - Parameter decoder: The decoder
to read data from.
    public init(from decoder: any
Decoder) throws
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Record.Recording : Codable
where Output : Decodable, Output :
Encodable, Failure : Decodable, Failure :
Encodable {

    /// Creates a new instance by
decoding from the given decoder.
    ///
```

```
    /// This initializer throws an error
if reading from the decoder fails, or
    /// if the data read is corrupted or
otherwise invalid.
    ///
    /// - Parameter decoder: The decoder
to read data from.
    public init(from decoder: any
Decoder) throws

    public func encode(into encoder: any
Encoder) throws

    /// Encodes this value into the given
encoder.
    ///
    /// If the value fails to encode
anything, `encoder` will encode an empty
    /// keyed container in its place.
    ///
    /// This function throws an error if
any values are invalid for the given
    /// encoder's format.
    ///
    /// - Parameter encoder: The encoder
to write data to.
    public func encode(to encoder: any
Encoder) throws
}

/// A protocol that defines when and how
to execute a closure.
///
```

```
/// You can use a scheduler to execute
code as soon as possible, or after a
future date.
/// Individual scheduler implementations
use whatever time-keeping system makes
sense for them. Schedulers express this
as their `SchedulerTimeType`. Since this
type conforms to
``SchedulerTimeIntervalConvertible``, you
can always express these times with the
convenience functions like
`.milliseconds(500)`. Schedulers can
accept options to control how they
execute the actions passed to them. These
options may control factors like which
threads or dispatch queues execute the
actions.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol
Scheduler<SchedulerTimeType> {

    /// Describes an instant in time for
this scheduler.
    associatedtype SchedulerTimeType :
Strideable where
Self.SchedulerTimeType.Stride :
SchedulerTimeIntervalConvertible

    /// A type that defines options
accepted by the scheduler.
    ///
    /// This type is freely definable by
```

```
each `Scheduler`. Typically, operations
that take a `Scheduler` parameter will
also take `SchedulerOptions`.
    associatedtype SchedulerOptions

    /// This scheduler's definition of
the current moment in time.
    var now: Self.SchedulerTimeType { get
}

    /// The minimum tolerance allowed by
the scheduler.
    var minimumTolerance:
Self.SchedulerTimeType.Stride { get }

    /// Performs the action at the next
possible opportunity.
    func schedule(options:
Self.SchedulerOptions?, _ action:
@escaping () -> Void)

    /// Performs the action at some time
after the specified date.
    func schedule(after date:
Self.SchedulerTimeType, tolerance:
Self.SchedulerTimeType.Stride, options:
Self.SchedulerOptions?, _ action:
@escaping () -> Void)

    /// Performs the action at some time
after the specified date, at the
specified frequency, optionally taking
into account tolerance if possible.
```

```
    func schedule(after date:
Self.SchedulerTimeType, interval:
Self.SchedulerTimeType.Stride, tolerance:
Self.SchedulerTimeType.Stride, options:
Self.SchedulerOptions?, _ action:
@escaping () -> Void) -> any Cancellable
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Scheduler {

    /// Performs the action at some time
after the specified date, using the
scheduler's minimum tolerance.
    ///
    /// The immediate scheduler ignores
`date` and performs the action
immediately.
    public func schedule(after date:
Self.SchedulerTimeType, _ action:
@escaping () -> Void)

    /// Performs the action at the next
possible opportunity, without options.
    public func schedule(_ action:
@escaping () -> Void)

    /// Performs the action at some time
after the specified date.
    ///
    /// The immediate scheduler ignores
`date` and performs the action
```

```
immediately.
    public func schedule(after date:
Self.SchedulerTimeType, tolerance:
Self.SchedulerTimeType.Stride, _ action:
@escaping () -> Void)

    /// Performs the action at some time
after the specified date, at the
specified frequency, taking into account
tolerance if possible.
    ///
    /// The immediate scheduler ignores
`date` and performs the action
immediately.
    public func schedule(after date:
Self.SchedulerTimeType, interval:
Self.SchedulerTimeType.Stride, tolerance:
Self.SchedulerTimeType.Stride, _ action:
@escaping () -> Void) -> any Cancellable

    /// Performs the action at some time
after the specified date, at the
specified frequency, using minimum
tolerance possible for this Scheduler.
    ///
    /// The immediate scheduler ignores
`date` and performs the action
immediately.
    public func schedule(after date:
Self.SchedulerTimeType, interval:
Self.SchedulerTimeType.Stride, _ action:
@escaping () -> Void) -> any Cancellable
}
```

```
/// A protocol that provides a scheduler
with an expression for relative time.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol
SchedulerTimeIntervalConvertible {

    /// Converts the specified number of
seconds into an instance of this
scheduler time type.
    static func seconds(_ s: Int) -> Self

    /// Converts the specified number of
seconds, as a floating-point value, into
an instance of this scheduler time type.
    static func seconds(_ s: Double) ->
Self

    /// Converts the specified number of
milliseconds into an instance of this
scheduler time type.
    static func milliseconds(_ ms: Int)
-> Self

    /// Converts the specified number of
microseconds into an instance of this
scheduler time type.
    static func microseconds(_ us: Int)
-> Self

    /// Converts the specified number of
nanoseconds into an instance of this
```

```
scheduler time type.
    static func nanoseconds(_ ns: Int) ->
Self
}

/// A publisher that exposes a method for
outside callers to publish elements.
///
/// A subject is a publisher that you can
use to ”inject” values into a stream, by
calling its ``Subject/send(_:)`` method.
This can be useful for adapting existing
imperative code to the Combine model.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol Subject<Output,
Failure> : AnyObject, Publisher {

    /// Sends a value to the subscriber.
    ///
    /// - Parameter value: The value to
send.
    func send(_ value: Self.Output)

    /// Sends a completion signal to the
subscriber.
    ///
    /// - Parameter completion: A
`Completion` instance which indicates
whether publishing has finished normally
or failed with an error.
    func send(completion:
Subscribers.Completion<Self.Failure>)
```

```
    /// Sends a subscription to the
subscriber.
    ///
    /// This call provides the
``Subject`` an opportunity to establish
demand for any new upstream
subscriptions.
    ///
    /// - Parameter subscription: The
subscription instance through which the
subscriber can request elements.
    func send(subscription: any
Subscription)
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subject where Self.Output == ()
{

    /// Sends a void value to the
subscriber.
    ///
    /// Use `Void` inputs and outputs
when you want to signal that an event has
occurred, but don’t need to send the
event itself.
    public func send()
}

/// A protocol that declares a type that
can receive input from a publisher.
```

```
///
/// A ``Subscriber`` instance receives a
stream of elements from a ``Publisher``,
along with life cycle events describing
changes to their relationship. A given
subscriber's ``Subscriber/Input`` and
``Subscriber/Failure`` associated types
must match the ``Publisher/Output`` and
``Publisher/Failure`` of its
corresponding publisher.
///
/// You connect a subscriber to a
publisher by calling the publisher's
``Publisher/subscribe(_:)-4u8kn`` method.
After making this call, the publisher
invokes the subscriber's
``Subscriber/receive(subscription:)``
method. This gives the subscriber a
``Subscription`` instance, which it uses
to demand elements from the publisher,
and to optionally cancel the
subscription. After the subscriber makes
an initial demand, the publisher calls
``Subscriber/receive(_:)``, possibly
asynchronously, to deliver newly-
published elements. If the publisher
stops publishing, it calls
``Subscriber/receive(completion:)``,
using a parameter of type
``Subscribers/Completion`` to indicate
whether publishing completes normally or
with an error.
///
```

```
/// Combine provides the following
subscribers as operators on the
``Publisher`` type:
///
/// -
``Publisher/sink(receiveCompletion:receiv
eValue:)`` executes arbitrary closures
when it receives a completion signal and
each time it receives a new element.
/// - ``Publisher/assign(to:on:)`` writes
each newly-received value to a property
identified by a key path on a given
instance.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol Subscriber<Input,
Failure> :
CustomCombineIdentifierConvertible {

    /// The kind of values this
subscriber receives.
    associatedtype Input

    /// The kind of errors this
subscriber might receive.
    ///
    /// Use `Never` if this `Subscriber`
cannot receive errors.
    associatedtype Failure : Error

    /// Tells the subscriber that it has
successfully subscribed to the publisher
and may request items.
```

```
    ///
    /// Use the received ``Subscription``
to request items from the publisher.
    /// - Parameter subscription: A
subscription that represents the
connection between publisher and
subscriber.
    func receive(subscription: any
Subscription)

    /// Tells the subscriber that the
publisher has produced an element.
    ///
    /// - Parameter input: The published
element.
    /// - Returns: A `Subscribers.Demand`
instance indicating how many more
elements the subscriber expects to
receive.
    func receive(_ input: Self.Input) ->
Subscribers.Demand

    /// Tells the subscriber that the
publisher has completed publishing,
either normally or with an error.
    ///
    /// - Parameter completion: A
``Subscribers/Completion`` case
indicating whether publishing completed
normally or with an error.
    func receive(completion:
Subscribers.Completion<Self.Failure>)
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscriber where Self.Input ==
() {

    /// Tells the subscriber that a
publisher of void elements is ready to
receive further requests.
    ///
    /// Use `Void` inputs and outputs
when you want to signal that an event has
occurred, but don’t need to send the
event itself.
    /// - Returns: A
``Subscribers/Demand`` instance
indicating how many more elements the
subscriber expects to receive.
    public func receive() ->
Subscribers.Demand
}

/// A namespace for types that serve as
subscribers.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public enum Subscribers {
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers {
```

```
    /// A simple subscriber that requests
an unlimited number of values upon
subscription.
    final public class Sink<Input,
Failure> : Subscriber, Cancellable,
CustomStringConvertible,
CustomReflectable,
CustomPlaygroundDisplayConvertible where
Failure : Error {

        /// The closure to execute on
receipt of a value.
        final public var receiveValue:
(Input) -> Void { get }

        /// The closure to execute on
completion.
        final public var
receiveCompletion:
(Subscribers.Completion<Failure>) -> Void
{ get }

        /// A textual representation of
this instance.
        ///
        /// Calling this property
directly is discouraged. Instead, convert
an
        /// instance of any type to a
string by using the `String(describing:)`
        /// initializer. This initializer
works with any type, and uses the custom
        /// `description` property for
```

```
types that conform to
        /// `CustomStringConvertible`:
        ///
        ///     struct Point:
CustomStringConvertible {
        ///         let x: Int, y: Int
        ///
        ///         var description:
String {
        ///             return "(\(x), \
(y))"
        ///         }
        ///     }
        ///
        ///     let p = Point(x: 21, y:
30)
        ///     let s =
String(describing: p)
        ///     print(s)
        ///     // Prints "(21, 30)"
        ///
        /// The conversion of `p` to a
string in the assignment to `s` uses the
        /// `Point` type's `description`
property.
        final public var description:
String { get }

        /// The custom mirror for this
instance.
        ///
        /// If this type has value
semantics, the mirror should be
```

```
unaffected by
        /// subsequent mutations of the
instance.
        final public var customMirror:
Mirror { get }

        /// A custom playground
description for this instance.
        final public var
playgroundDescription: Any { get }

        /// Initializes a sink with the
provided closures.
        ///
        /// - Parameters:
        ///   - receiveCompletion: The
closure to execute on completion.
        ///   - receiveValue: The closure
to execute on receipt of a value.
        public init(receiveCompletion:
@escaping
((Subscribers.Completion<Failure>) ->
Void), receiveValue: @escaping ((Input)
-> Void))

        /// Tells the subscriber that it
has successfully subscribed to the
publisher and may request items.
        ///
        /// Use the received
``Subscription`` to request items from
the publisher.
        /// - Parameter subscription: A
```

```
subscription that represents the
connection between publisher and
subscriber.
        final public func
receive(subscription: any Subscription)

        /// Tells the subscriber that the
publisher has produced an element.
        ///
        /// - Parameter input: The
published element.
        /// - Returns: A
`Subscribers.Demand` instance indicating
how many more elements the subscriber
expects to receive.
        final public func receive(_
value: Input) -> Subscribers.Demand

        /// Tells the subscriber that the
publisher has completed publishing,
either normally or with an error.
        ///
        /// - Parameter completion: A
``Subscribers/Completion`` case
indicating whether publishing completed
normally or with an error.
        final public func
receive(completion:
Subscribers.Completion<Failure>)

        /// Cancel the activity.
        ///
        /// When implementing
```

```
``Cancellable`` in support of a custom
publisher, implement `cancel()` to
request that your publisher stop calling
its downstream subscribers. Combine
doesn't require that the publisher stop
immediately, but the `cancel()` call
should take effect quickly. Canceling
should also eliminate any strong
references it currently holds.
        ///
        /// After you receive one call to
`cancel()`, subsequent calls shouldn't do
anything. Additionally, your
implementation must be thread-safe, and
it shouldn't block the caller.
        ///
        /// > Tip: Keep in mind that your
`cancel()` may execute concurrently with
another call to `cancel()` --- including
the scenario where an ``AnyCancellable``
is deallocating --- or to
``Subscription/request(_:)``.
        final public func cancel()
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers {

    /// A signal that a publisher doesn’t
produce additional elements, either due
to normal completion or an error.
```

```
    @frozen public enum
Completion<Failure> where Failure : Error
{

        /// The publisher finished
normally.
        case finished

        /// The publisher stopped
publishing due to the indicated error.
        case failure(Failure)
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers {

    /// A requested number of items, sent
to a publisher from a subscriber through
the subscription.
    @frozen public struct Demand :
Equatable, Comparable, Hashable, Codable,
CustomStringConvertible {

        /// A request for as many values
as the publisher can produce.
        public static let unlimited:
Subscribers.Demand

        /// A request for no elements
from the publisher.
        ///
```

```
        /// This is equivalent to
`Demand.max(0)`.
        public static let none:
Subscribers.Demand

        /// Creates a demand for the
given maximum number of elements.
        ///
        /// The publisher is free to send
fewer than the requested maximum number
of elements.
        ///
        /// - Parameter value: The
maximum number of elements. Providing a
negative value for this parameter results
in a fatal error.
        @inlinable public static func
max(_ value: Int) -> Subscribers.Demand

        /// A textual representation of
this instance.
        ///
        /// Calling this property
directly is discouraged. Instead, convert
an
        /// instance of any type to a
string by using the `String(describing:)`
        /// initializer. This initializer
works with any type, and uses the custom
        /// `description` property for
types that conform to
        /// `CustomStringConvertible`:
        ///
```

```
        ///     struct Point:
CustomStringConvertible {
        ///         let x: Int, y: Int
        ///
        ///         var description:
String {
        ///             return "(\(x), \
(y))"
        ///         }
        ///     }
        ///
        ///     let p = Point(x: 21, y:
30)
        ///     let s =
String(describing: p)
        ///     print(s)
        ///     // Prints "(21, 30)"
        ///
        /// The conversion of `p` to a
string in the assignment to `s` uses the
        /// `Point` type's `description`
property.
        public var description: String {
get }

        /// Returns the result of adding
two demands.
        /// When adding any value to
`.unlimited`, the result is `.unlimited`.
        @inlinable public static func +
(lhs: Subscribers.Demand, rhs:
Subscribers.Demand) -> Subscribers.Demand
```

```
        /// Adds two demands, and assigns
the result to the first demand.
        ///
        /// When adding any value to
`.unlimited`, the result is `.unlimited`.
        @inlinable public static func +=
(lhs: inout Subscribers.Demand, rhs:
Subscribers.Demand)

        /// Returns the result of adding
an integer to a demand.
        ///
        /// When adding any value to
`.unlimited`, the result is `.unlimited`.
        @inlinable public static func +
(lhs: Subscribers.Demand, rhs: Int) ->
Subscribers.Demand

        /// Adds an integer to a demand,
and assigns the result to the demand.
        ///
        /// When adding any value to
`.unlimited`, the result is `.unlimited`.
        @inlinable public static func +=
(lhs: inout Subscribers.Demand, rhs: Int)

        /// Returns the result of
multiplying a demand by an integer.
        ///
        /// When multiplying any value by
`.unlimited`, the result is `.unlimited`.
If
        /// the multiplication operation
```

```
overflows, the result is `.unlimited`.
        public static func * (lhs:
Subscribers.Demand, rhs: Int) ->
Subscribers.Demand

        /// Multiplies a demand by an
integer, and assigns the result to the
demand.
        ///
        /// When multiplying any value by
`.unlimited`, the result is `.unlimited`.
If
        /// the multiplication operation
overflows, the result is `.unlimited`.
        @inlinable public static func *=
(lhs: inout Subscribers.Demand, rhs: Int)

        /// Returns the result of
subtracting one demand from another.
        ///
        /// When subtracting any value
(including `.unlimited`) from
`.unlimited`, the result is still
`.unlimited`. Subtracting `.unlimited`
from any value (except `.unlimited`)
results in `.max(0)`. A negative demand
is impossible; when an operation would
result in a negative value, Combine
adjusts the value to `.max(0)`.
        @inlinable public static func -
(lhs: Subscribers.Demand, rhs:
Subscribers.Demand) -> Subscribers.Demand
```

```
        /// Subtracts one demand from
another, and assigns the result to the
first demand.
        ///
        /// When subtracting any value
(including `.unlimited`) from
`.unlimited`, the result is still
`.unlimited`. Subtracting `.unlimited`
from any value (except `.unlimited`)
results in `.max(0)`. A negative demand
is impossible; when an operation would
result in a negative value, Combine
adjusts the value to `.max(0)`.
        @inlinable public static func -=
(lhs: inout Subscribers.Demand, rhs:
Subscribers.Demand)

        /// Returns the result of
subtracting an integer from a demand.
        ///
        /// When subtracting any value
from `.unlimited`, the result is still
`.unlimited`. A negative demand is
possible, but be aware that it isn't
usable when requesting values in a
subscription.
        @inlinable public static func -
(lhs: Subscribers.Demand, rhs: Int) ->
Subscribers.Demand

        /// Subtracts an integer from a
demand, and assigns the result to the
demand.
```

```
        ///
        /// When subtracting any value
from `.unlimited`, the result is still
`.unlimited`. A negative demand is
impossible; when an operation would
result in a negative value, Combine
adjusts the value to `.max(0)`.
        @inlinable public static func -=
(lhs: inout Subscribers.Demand, rhs: Int)

        /// Returns a Boolean that
indicates whether the demand requests
more than the given number of elements.
        ///
        /// If `lhs` is `.unlimited`,
then the result is always `true`.
Otherwise, the operator compares the
demand's `max` value to `rhs`.
        @inlinable public static func >
(lhs: Subscribers.Demand, rhs: Int) ->
Bool

        /// Returns a Boolean that
indicates whether the first demand
requests more or the same number of
elements as the second.
        ///
        /// If `lhs` is `.unlimited`,
then the result is always `true`.
Otherwise, the operator compares the
demand's `max` value to `rhs`.
        @inlinable public static func >=
(lhs: Subscribers.Demand, rhs: Int) ->
```

```
Bool

        /// Returns a Boolean that
indicates a given number of elements is
greater than the maximum specified by the
demand.
        ///
        /// If `rhs` is `.unlimited`,
then the result is always `false`.
Otherwise, the operator compares the
demand's `max` value to `lhs`.
        @inlinable public static func >
(lhs: Int, rhs: Subscribers.Demand) ->
Bool

        /// Returns a Boolean that
indicates a given number of elements is
greater than or equal to the maximum
specified by the demand.
        ///
        /// If `rhs` is `.unlimited`,
then the result is always `false`.
Otherwise, the operator compares the
demand's `max` value to `lhs`.
        @inlinable public static func >=
(lhs: Int, rhs: Subscribers.Demand) ->
Bool

        /// Returns a Boolean that
indicates whether the demand requests
fewer than the given number of elements.
        ///
        /// If `lhs` is `.unlimited`,
```

```
then the result is always `false`.
Otherwise, the operator compares the
demand's `max` value to `rhs`.
        @inlinable public static func <
(lhs: Subscribers.Demand, rhs: Int) ->
Bool

        /// Returns a Boolean that
indicates a given number of elements is
less than the maximum specified by the
demand.
        ///
        /// If `rhs` is `.unlimited`,
then the result is always `true`.
Otherwise, the operator compares the
demand's `max` value to `lhs`.
        @inlinable public static func <
(lhs: Int, rhs: Subscribers.Demand) ->
Bool

        /// Returns a Boolean that
indicates whether the demand requests
fewer or the same number of elements as
the given integer.
        ///
        /// If `lhs` is `.unlimited`,
then the result is always `false`.
Otherwise, the operator compares the
demand's `max` value to `rhs`.
        @inlinable public static func <=
(lhs: Subscribers.Demand, rhs: Int) ->
Bool
```

```
        /// Returns a Boolean value that
indicates a given number of elements is
less than or equal the maximum specified
by the demand.
        ///
        /// If `rhs` is `.unlimited`,
then the result is always `true`.
Otherwise, the operator compares the
demand's `max` value to `lhs`.
        @inlinable public static func <=
(lhs: Int, rhs: Subscribers.Demand) ->
Bool

        /// Returns a Boolean that
indicates whether the first demand
requests fewer elements than the second.
        ///
        /// If both sides are
`.unlimited`, the result is always
`false`. If `lhs` is `.unlimited`, then
the result is always `false`. If `rhs` is
`.unlimited` then the result is always
`true`. Otherwise, this operator compares
the demands' `max` values.
        @inlinable public static func <
(lhs: Subscribers.Demand, rhs:
Subscribers.Demand) -> Bool

        /// Returns a Boolean value that
indicates whether the first demand
requests fewer or the same number of
elements as the second.
        ///
```

```
        /// If both sides are
`.unlimited`, the result is always
`true`. If `lhs` is `.unlimited`, then
the result is always `false`. If `rhs` is
unlimited then the result is always
`true`. Otherwise, this operator compares
the demands' `max` values.
        @inlinable public static func <=
(lhs: Subscribers.Demand, rhs:
Subscribers.Demand) -> Bool

        /// Returns a Boolean that
indicates whether the first demand
requests more or the same number of
elements as the second.
        ///
        /// If both sides are
`.unlimited`, the result is always
`true`. If `lhs` is `.unlimited`, then
the result is always `true`. If `rhs` is
`.unlimited` then the result is always
`false`. Otherwise, this operator
compares the demands' `max` values.
        @inlinable public static func >=
(lhs: Subscribers.Demand, rhs:
Subscribers.Demand) -> Bool

        /// Returns a Boolean that
indicates whether the first demand
requests more elements than the second.
        ///
        /// If both sides are
`.unlimited`, the result is always
```

```
`false`. If `lhs` is `.unlimited`, then
the result is always `true`. If `rhs` is
`.unlimited` then the result is always
`false`. Otherwise, this operator
compares the demands' `max` values.
        @inlinable public static func >
(lhs: Subscribers.Demand, rhs:
Subscribers.Demand) -> Bool

        /// Returns a Boolean value that
indicates whether a demand requests the
given number of elements.
        ///
        /// An `.unlimited` demand
doesn't match any integer.
        @inlinable public static func ==
(lhs: Subscribers.Demand, rhs: Int) ->
Bool

        /// Returns a Boolean value that
indicates whether a demand isn't equal to
an integer.
        ///
        /// The `.unlimited` value isn't
equal to any integer.
        @inlinable public static func !=
(lhs: Subscribers.Demand, rhs: Int) ->
Bool

        /// Returns a Boolean value that
indicates whether a given number of
elements matches the request of a given
demand.
```

```
        ///
        /// An `.unlimited` demand
doesn’t match any integer.
        @inlinable public static func ==
(lhs: Int, rhs: Subscribers.Demand) ->
Bool

        /// Returns a Boolean value that
indicates whether an integer is unequal
to a demand.
        ///
        /// The `.unlimited` value isn’t
equal to any integer.
        @inlinable public static func !=
(lhs: Int, rhs: Subscribers.Demand) ->
Bool

        /// The number of requested
values.
        ///
        /// The value is `nil` if the
demand is
``Subscribers/Demand/unlimited``.
        @inlinable public var max: Int? {
get }

        /// Creates a demand instance
from a decoder.
        ///
        /// - Parameter decoder: The
decoder of a previously-encoded
``Subscribers/Demand`` instance.
        public init(from decoder: any
```

```
Decoder) throws

        /// Encodes the demand to the
provide encoder.
        ///
        /// - Parameter encoder: An
encoder instance.
        public func encode(to encoder:
any Encoder) throws

        /// Returns a Boolean value
indicating whether two values are equal.
        ///
        /// Equality is the inverse of
inequality. For any values `a` and `b`,
        /// `a == b` implies that `a !=
b` is `false`.
        ///
        /// - Parameters:
        ///   - lhs: A value to compare.
        ///   - rhs: Another value to
compare.
        public static func == (a:
Subscribers.Demand, b:
Subscribers.Demand) -> Bool

        /// Hashes the essential
components of this value by feeding them
into the
        /// given hasher.
        ///
        /// Implement this method to
conform to the `Hashable` protocol. The
```

```
        /// components used for hashing
must be the same as the components
compared
        /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
        /// with each of these
components.
        ///
        /// - Important: In your
implementation of `hash(into:)`,
        ///   don't call `finalize()` on
the `hasher` instance provided,
        ///   or replace it with a
different instance.
        ///   Doing so may become a
compile-time error in the future.
        ///
        /// - Parameter hasher: The
hasher to use when combining the
components
        ///   of this instance.
        public func hash(into hasher:
inout Hasher)

        /// The hash value.
        ///
        /// Hash values are not
guaranteed to be equal across different
executions of
        /// your program. Do not save
hash values to use during a future
execution.
        ///
```

```
        /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
        ///   conform to `Hashable`,
implement the `hash(into:)` requirement
instead.
        ///   The compiler provides an
implementation for `hashValue` for you.
        public var hashValue: Int { get }
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers {

    /// A simple subscriber that assigns
received elements to a property indicated
by a key path.
    final public class Assign<Root,
Input> : Subscriber, Cancellable,
CustomStringConvertible,
CustomReflectable,
CustomPlaygroundDisplayConvertible {

        /// The kind of errors this
subscriber might receive.
        ///
        /// Use `Never` if this
`Subscriber` cannot receive errors.
        public typealias Failure = Never

        /// The object that contains the
```

```
property to assign.
        ///
        /// The subscriber holds a strong
reference to this object until the
upstream publisher calls
``Subscriber/receive(completion:)``, at
which point the subscriber sets this
property to `nil`.
        final public var object: Root? {
get }

        /// The key path that indicates
the property to assign.
        final public let keyPath:
ReferenceWritableKeyPath<Root, Input>

        /// A textual representation of
this subscriber.
        final public var description:
String { get }

        /// A mirror that reflects the
subscriber.
        final public var customMirror:
Mirror { get }

        /// A custom playground
description for this subscriber.
        final public var
playgroundDescription: Any { get }

        /// Creates a subscriber to
assign the value of a property indicated
```

```
by a key path.
        ///
        /// - Parameters:
        ///   - object: The object that
contains the property. The subscriber
assigns the object's property every time
it receives a new value.
        ///   - keyPath: A key path that
indicates the property to assign. See
[Key-Path Expression]
(https://developer.apple.com/library/arch
ive/documentation/Swift/Conceptual/
Swift_Programming_Language/
Expressions.html#//apple_ref/doc/uid/
TP40014097-CH32-ID563) in _The Swift
Programming Language_ to learn how to use
key paths to specify a property of an
object.
        public init(object: Root,
keyPath: ReferenceWritableKeyPath<Root,
Input>)

        /// Tells the subscriber that it
has successfully subscribed to the
publisher and may request items.
        ///
        /// Use the received
``Subscription`` to request items from
the publisher.
        ///
        /// - Parameter subscription: A
subscription that represents the
connection between publisher and
```

```
subscriber.
        final public func
receive(subscription: any Subscription)

        /// Tells the subscriber that the
publisher has produced an element.
        ///
        /// A ``Subscribers/Demand``
instance indicating how many more
elements the subscriber expects to
receive.
        ///
        /// - Parameter input: The
published element.
        final public func receive(_
value: Input) -> Subscribers.Demand

        /// Tells the subscriber that the
publisher has completed publishing,
either normally or with an error.
        ///
        /// - Parameter completion: A
``Subscribers/Completion`` case
indicating whether publishing completed
normally or with an error.
        final public func
receive(completion:
Subscribers.Completion<Never>)

        /// Cancel the activity.
        final public func cancel()
    }
}
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Completion :
Equatable where Failure : Equatable {

    /// Returns a Boolean value
indicating whether two values are equal.
    ///
    /// Equality is the inverse of
inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is
`false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to
compare.
    public static func == (a:
Subscribers.Completion<Failure>, b:
Subscribers.Completion<Failure>) -> Bool
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Completion :
Hashable where Failure : Hashable {

    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// Implement this method to conform
```

```
to the `Hashable` protocol. The
    /// components used for hashing must
be the same as the components compared
    /// in your type's `==` operator
implementation. Call `hasher.combine(_:)`
    /// with each of these components.
    ///
    /// - Important: In your
implementation of `hash(into:)`,
    ///   don't call `finalize()` on the
`hasher` instance provided,
    ///   or replace it with a different
instance.
    ///   Doing so may become a compile-
time error in the future.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    public func hash(into hasher: inout
Hasher)

    /// The hash value.
    ///
    /// Hash values are not guaranteed to
be equal across different executions of
    /// your program. Do not save hash
values to use during a future execution.
    ///
    /// - Important: `hashValue` is
deprecated as a `Hashable` requirement.
To
    ///   conform to `Hashable`,
```

```
implement the `hash(into:)` requirement
instead.
    ///   The compiler provides an
implementation for `hashValue` for you.
    public var hashValue: Int { get }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Completion :
Encodable where Failure : Encodable {

    /// Encodes this value into the given
encoder.
    ///
    /// If the value fails to encode
anything, `encoder` will encode an empty
    /// keyed container in its place.
    ///
    /// This function throws an error if
any values are invalid for the given
    /// encoder's format.
    ///
    /// - Parameter encoder: The encoder
to write data to.
    public func encode(to encoder: any
Encoder) throws
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Completion :
Decodable where Failure : Decodable {
```

```
    /// Creates a new instance by
decoding from the given decoder.
    ///
    /// This initializer throws an error
if reading from the decoder fails, or
    /// if the data read is corrupted or
otherwise invalid.
    ///
    /// - Parameter decoder: The decoder
to read data from.
    public init(from decoder: any
Decoder) throws
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Completion :
Sendable {
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Demand : Sendable {
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscribers.Demand :
BitwiseCopyable {
}

/// A protocol representing the
```

```
connection of a subscriber to a
publisher.
///
/// Subscriptions are class constrained
because a ``Subscription`` has identity,
defined by the moment in time a
particular subscriber attached to a
publisher. Canceling a ``Subscription``
must be thread-safe.
///
/// You can only cancel a
``Subscription`` once.
///
/// Canceling a subscription frees up any
resources previously allocated by
attaching the ``Subscriber``.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol Subscription :
Cancellable,
CustomCombineIdentifierConvertible {

    /// Tells a publisher that it may
send more values to the subscriber.
    func request(_ demand:
Subscribers.Demand)
}

/// A namespace for symbols related to
subscriptions.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public enum Subscriptions {
```

```
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Subscriptions {

    /// Returns the “empty” subscription.
    ///
    /// Use the empty subscription when
you need a ``Subscription`` that ignores
requests and cancellation.
    public static var empty: any
Subscription { get }
}

/// A type that defines methods for
decoding.
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol TopLevelDecoder {

    /// The type this decoder accepts.
    associatedtype Input

    /// Decodes an instance of the
indicated type.
    func decode<T>(_ type: T.Type, from:
Self.Input) throws -> T where T :
Decodable
}

/// A type that defines methods for
encoding.
```

```
@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
public protocol TopLevelEncoder {

    /// The type this encoder produces.
    associatedtype Output

    /// Encodes an instance of the
indicated type.
    ///
    /// - Parameter value: The instance
to encode.
    func encode<T>(_ value: T) throws ->
Self.Output where T : Encodable
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Optional {

    /// A Combine publisher that
publishes this instance's value to each
subscriber exactly once, if it has any
value at all.
    ///
    /// In the following example, the
publisher for an `Int?` optional
publishes its value once, then finishes
normally:
    ///
    ///     let optional1: Int? = 1
    ///         optional1.publisher
    ///             .sink(receiveCompleti
```

```
on: { print("optional1 completed.") },
    ///                     receiveValue: {
print("optional1 = \($0)") }
    ///                 )
    ///
    ///     // Prints:
    ///     // optional1 = 1.
    ///     // optional1 completed.
    ///
    /// In contrast with the
<doc://com.apple.documentation/documentat
ion/Combine/Just> publisher, which always
publishes a single value, this publisher
might not send any values and instead
finish normally if the optional's
`output` is `nil`. In the next example,
an `Int?` optional that's `nil`
immediately sends the
<doc://com.apple.documentation/documentat
ion/Combine/Subscribers/Completion/
finished> completion, without producing
any values.
    ///
    ///     let optional2: Int? = nil
    ///     optional2.publisher
    ///                 .sink(receiveCompleti
on: { print("optional2 completed.") },
    ///                     receiveValue: {
print("optional2 = \($0)") }
    ///                 )
    ///
    ///     // Prints:
    ///     // optional2 completed.
```

```
    @available(macOS 11.0, iOS 14.0, tvOS
14.0, watchOS 7.0, *)
    public var publisher:
Optional<Wrapped>.Publisher { get }

    /// The type of a Combine publisher
that publishes the value of a Swift
optional instance to each subscriber
exactly once, if the instance has any
value at all.
    ///
    /// In contrast with the
<doc://com.apple.documentation/documentat
ion/Combine/Just> publisher, which always
produces a single value, this publisher
might not send any values and instead
finish normally, if ``output`` is `nil`.
    @available(macOS 10.15, iOS 13.0,
tvOS 13.0, watchOS 6.0, *)
    public struct Publisher : Publisher {

        /// The kind of value published
by this publisher.
        ///
        /// This publisher produces the
type wrapped by the optional.
        public typealias Output = Wrapped

        /// The kind of error this
publisher might publish.
        ///
        /// The optional publisher never
produces errors.
```

```
        public typealias Failure = Never

        /// The output to deliver to each
subscriber.
        public let output:
Optional<Wrapped>.Publisher.Output?

        /// Creates a publisher to emit
the value of the optional, or to finish
immediately if the optional doesn't have
a value.
        ///
        /// - Parameter output: The
result to deliver to each subscriber.
        public init(_ output:
Optional<Wrapped>.Publisher.Output?)

        /// Implements the Publisher
protocol by accepting the subscriber and
immediately publishing the optional’s
value if it has one, or finishing
normally if it doesn’t.
        ///
        /// - Parameter subscriber: The
subscriber to add.
        public func
receive<S>(subscriber: S) where Wrapped
== S.Input, S : Subscriber, S.Failure ==
Never
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
```

```
13.0, watchOS 6.0, *)
extension Result {

    /// A Combine publisher that
publishes this instance’s result to each
subscriber exactly once, or fails
immediately if the result indicates
failure.
    ///
    /// In the following example,
`goodResult` provides a successful result
with the integer value `1`. A sink
subscriber connected to the result's
publisher receives the output `1`,
followed by a normal completion
(<doc://com.apple.documentation/documenta
tion/Combine/Subscribers/Completion/
finished>).
    ///
    ///      let goodResult: Result<Int,
MyError> = .success(1)
    ///      goodResult.publisher
    ///          .sink(receiveCompletion:
{ print("goodResult done: \($0)")},
    ///                receiveValue:
{ print("goodResult value: \($0)")} )
    ///      // Prints:
    ///      // goodResult value: 1
    ///      // goodResult done: finished
    ///
    /// In contrast with the
<doc://com.apple.documentation/documentat
ion/Combine/Just> publisher, which always
```

```
publishes a single value, this publisher
might not send any values and instead
terminate with an error, if the result is
``/Swift/Result/failure``. In the next
example, `badResult` is a failure result
that wraps a custom error. A sink
subscriber connected to this result's
publisher immediately receives a
termination
(<doc://com.apple.documentation/documenta
tion/Combine/Subscribers/Completion/
failure(_:)>).
    ///
    ///      struct MyError: Error,
CustomDebugStringConvertible {
    ///          var debugDescription:
String = "MyError"
    ///      }
    ///      let badResult: Result<Int,
MyError> = .failure(MyError())
    ///      badResult.publisher
    ///          .sink(receiveCompletion:
{ print("badResult done: \($0)")},
    ///                receiveValue:
{ print("badResult value: \($0)")} )
    ///      // Prints:
    ///      // badResult done:
failure(MyError)
    ///
    public var publisher: Result<Success,
Failure>.Publisher { get }

    /// The type of a Combine publisher
```

```
that publishes this instance’s result to
each subscriber exactly once, or fails
immediately if the result indicates
failure.
    ///
    /// If the result is
``Swift/Result/success``, then the
publisher waits until it receives a
request for at least one value, then
sends the output to all subscribers and
finishes normally. If the result is
``/Swift/Result/failure``, then the
publisher sends the failure immediately
upon subscription. This latter behavior
is a contrast with
<doc://com.apple.documentation/documentat
ion/Combine/Just>, which always publishes
a single value.
    @available(macOS 10.15, iOS 13.0,
tvOS 13.0, watchOS 6.0, *)
    public struct Publisher : Publisher {

        /// The kind of values published
by this publisher.
        public typealias Output = Success

        /// The result to deliver to each
subscriber.
        public let result:
Result<Result<Success,
Failure>.Publisher.Output, Failure>

        /// Creates a publisher that
```

```
delivers the specified result.
        ///
        /// If `result` is
``Swift/Result/success``, then the
publisher waits until it receives a
request for at least one value, then
sends the output to all subscribers and
finishes normally. If `result` is
``Swift/Result/failure``, then the
publisher sends the failure immediately
upon subscription.
        /// - Parameter result: The
result to deliver to each subscriber.
        public init(_ result:
Result<Result<Success,
Failure>.Publisher.Output, Failure>)

        /// Creates a publisher that
sends the specified output to all
subscribers and finishes normally.
        ///
        /// - Parameter output: The
output to deliver to each subscriber.
        public init(_ output:
Result<Success,
Failure>.Publisher.Output)

        /// Creates a publisher that
immediately terminates upon subscription
with the given failure.
        ///
        /// - Parameter failure: The
failure to send when terminating.
```

```
        public init(_ failure: Failure)

        /// Attaches the specified
subscriber to this publisher.
        ///
        /// Implementations of
``Publisher`` must implement this method.
        ///
        /// The provided implementation
of ``Publisher/subscribe(_:)-4u8kn``calls
this method.
        ///
        /// - Parameter subscriber: The
subscriber to attach to this
``Publisher``, after which it can receive
values.
        public func
receive<S>(subscriber: S) where Success
== S.Input, Failure == S.Failure, S :
Subscriber
    }
}

@available(macOS 10.15, iOS 13.0, tvOS
13.0, watchOS 6.0, *)
extension Sequence {

    public var publisher:
Publishers.Sequence<Self, Never> { get }
}
```