Observable

Draft Community Group Report, 28 February 2025

This version:
https://wicg.github.io/observable/

Editor:
Dominic Farolino (Google) domfarolino@gmail.com

Participate:
GitHub WICG/observable (new issue, open issues)

Commits:
GitHub spec.bs commits

Test Suite:
https://wpt.fyi/results/dom/observable/

Chrome 135	Firefox 137	Safari 18	Edge 135
0/0	0/0	0/0	0/0

Abstract

The Observable API provides a composable, ergonomic way of handling an asynchronous stream of events

Status of this document

This specification was published by the Web Platform Incubator Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.

Table of Contents

§ 1. Introduction

This section is non-normative.

§ 2. Core infrastructure

§ 2.1. The `Subscriber` interface

```
[Exposed=*]
interface Subscriber {
  undefined next(any value);
  undefined error(any error);
  undefined complete();
  undefined addTeardown(VoidFunction teardown);

  // True after the Subscriber is created, up until either
```

```
  // complete()/error() are invoked, or the subscriber unsubscribes. Inside
  // complete()/error(), this attribute is true.
  readonly attribute boolean active;

  readonly attribute AbortSignal signal;
};
```

Each `Subscriber` has an ordered set of ***internal observers***, initially empty.

Each `Subscriber` has a ***teardown callbacks***, which is a list of `VoidFunction`s, initially empty.

Each `Subscriber` has a ***subscription controller***, which is an `AbortController`.

Each `Subscriber` has a ***active*** boolean, initially true.

> NOTE: This is a bookkeeping variable to ensure that a `Subscriber` never calls any of the callbacks it owns after it has been closed.

The ***`active`*** getter steps are to return this's active boolean.

The ***`signal`*** getter steps are to return this's subscription controller's signal.

The ***`next(value)`*** method steps are:

1. If this's active is false, then return.
2. If this's relevant global object is a `Window` object, and its associated Document is not fully active, then return.
3. For each *observer* of this's internal observers:

    1. Run *observer*'s next steps given *value*.

        Assert: No exception was thrown.

        > Note: No exception can be thrown here because in the case where the internal observer's next steps is just a wrapper around a script-provided callback, the process observer steps take care to wrap these callbacks in logic that, when invoking them, catches any exceptions, and reports them to the global.
        >
        > When the next steps is a spec algorithm, those steps take care to not throw any exceptions outside of itself, to appease this assert.

The ***`error(error)`*** method steps are:

1. If this's active is false, report the exception *error*, then return.
2. If this's relevant global object is a `Window` object, and its associated Document is not fully active, then return.
3. Close this.
4. For each *observer* of this's internal observers:

    1. Run *observer*'s error steps given *error*.

        Assert: No exception was thrown.

        > NOTE: See the documentation in `next()` for details on why this is true.

The ***`complete()`*** method steps are:

1. If this's active is false, then return.
2. If this's relevant global object is a `Window` object, and its associated Document is not fully active, then return.
3. Close this.
4. For each *observer* of this's internal observers:

    1. Run *observer*'s complete steps.

        Assert: No exception was thrown.

        > NOTE: See the documentation in `next()` for details on why this is true.

The ***`addTeardown(teardown)`*** method steps are:

1. If this's relevant global object is a `Window` object, and its associated Document is not fully active, then return.
2. If this's active is true, then append *teardown* to this's teardown callbacks list.
3. Otherwise, invoke *teardown*.

    If an exception *E* was thrown, then report the exception *E*.

To ***close a subscription*** given a `Subscriber` *subscriber*, and an optional `any` *reason*, run these steps:

1. If *subscriber*'s active is false, then return.

This guards against re-entrant invocation, which can happen in the "producer-initiated" unsubscription case. Consider the following example:

EXAMPLE 1 ¶

```
const outerController = new AbortController();
const observable = new Observable(subscriber => {
  subscriber.addTeardown(() => {
    // 2.) This teardown executes inside the "Close" algorithm, while it's
    //     running. Aborting the downstream signal run its abort algorithms,
    //     one of which is the currently-running "Close" algorithm.
    outerController.abort();
  });

  // 1.) This immediately invokes the "Close" algorithm, which
  //     sets subscriber.active to false.
  subscriber.complete();
});

observable.subscribe({}, {signal: outerController.signal});
```

2. Set *subscriber*'s active boolean to false.
3. Signal abort *subscriber*'s subscription controller with *reason*, if it is given.
4. For each *teardown* of *subscriber*'s teardown callbacks sorted in reverse insertion order:
    1. If *subscriber*'s relevant global object is a `Window` object, and its associated Document is not fully active, then abort these steps.

        NOTE: This step runs repeatedly because each *teardown* could result in the above `Document` becoming inactive.

    2. Invoke *teardown*.

        If an exception *E* was thrown, then report the exception *E*.

§ 2.2. The `Observable` interface

```
// SubscribeCallback is where the Observable "creator's" code lives. It's
// called when subscribe() is called, to set up a new subscription.
callback SubscribeCallback = undefined (Subscriber subscriber);
callback ObservableSubscriptionCallback = undefined (any value);

dictionary SubscriptionObserver {
  ObservableSubscriptionCallback next;
  ObservableSubscriptionCallback error;
  VoidFunction complete;
};

callback ObservableInspectorAbortHandler = undefined (any value);

dictionary ObservableInspector {
  ObservableSubscriptionCallback next;
  ObservableSubscriptionCallback error;
  VoidFunction complete;

  VoidFunction subscribe;
  ObservableInspectorAbortHandler abort;
};

typedef (ObservableSubscriptionCallback or SubscriptionObserver) ObserverUnion;
typedef (ObservableSubscriptionCallback or ObservableInspector) ObservableInspectorUnion;

dictionary SubscribeOptions {
  AbortSignal signal;
};

callback Predicate = boolean (any value, unsigned long long index);
callback Reducer = any (any accumulator, any currentValue, unsigned long long index);
callback Mapper = any (any value, unsigned long long index);
// Differs from Mapper only in return type, since this callback is exclusively
// used to visit each element in a sequence, not transform it.
callback Visitor = undefined (any value, unsigned long long index);

// This callback returns an `any` that must convert into an `Observable`, via
// the `Observable` conversion semantics.
callback CatchCallback = any (any value);

[Exposed=*]
interface Observable {
  constructor(SubscribeCallback callback);
  undefined subscribe(optional ObserverUnion observer = {}, optional SubscribeOptions opti

  // Constructs a native Observable from value if it's any of the following:
  //   - Observable
  //   - AsyncIterable
  //   - Iterable
  //   - Promise
```

```
  static Observable from(any value);

  // Observable-returning operators. See "Operators" section in the spec.
  //
  // takeUntil() can consume promises, iterables, async iterables, and other
  // observables.
  Observable takeUntil(any value);
  Observable map(Mapper mapper);
  Observable filter(Predicate predicate);
  Observable take(unsigned long long amount);
  Observable drop(unsigned long long amount);
  Observable flatMap(Mapper mapper);
  Observable switchMap(Mapper mapper);
  Observable inspect(optional ObservableInspectorUnion inspectorUnion = {});
  Observable catch(CatchCallback callback);
  Observable finally(VoidFunction callback);

  // Promise-returning operators.
  Promise<sequence<any>> toArray(optional SubscribeOptions options = {});
  Promise<undefined> forEach(Visitor callback, optional SubscribeOptions options = {});
  Promise<boolean> every(Predicate predicate, optional SubscribeOptions options = {});
  Promise<any> first(optional SubscribeOptions options = {});
  Promise<any> last(optional SubscribeOptions options = {});
  Promise<any> find(Predicate predicate, optional SubscribeOptions options = {});
  Promise<boolean> some(Predicate predicate, optional SubscribeOptions options = {});
  Promise<any> reduce(Reducer reducer, optional any initialValue, optional SubscribeOption
};
```

Each `Observable` has a ***subscribe callback***, which is a `SubscribeCallback` or a set of steps that take in a `Subscriber`.

Each `Observable` has a ***weak subscriber***, which is a weak reference to a `Subscriber`-or-null, initially null.

> NOTE: The "union" of these types is to support both `Observable`s created by JavaScript (that are always constructed with a `SubscribeCallback`), and natively-constructed `Observable` objects (whose subscribe callback could be an arbitrary set of native steps, not a JavaScript callback). The return value of `when()` is an example of the latter.

The ***`new Observable(callback)`*** constructor steps are:

1. Set this's subscribe callback to *callback*.

   > NOTE: This callback will get invoked later when `subscribe()` is called.

The ***`subscribe(observer, options)`*** method steps are:

1. Subscribe to this given *observer* and *options*.

§ 2.2.1. Supporting concepts

The ***default error algorithm*** is an algorithm that takes an `any` *error*, and runs these steps:

1. Report the exception *error*.

> NOTE: We pull this default out separately so that every place in this specification that natively subscribes to an `Observable` (i.e., subscribes from spec prose, not going through the `subscribe()` method) doesn't have to redundantly define these steps.

An ***internal observer*** is a struct with the following items:

next steps
: An algorithm that takes a single parameter of type `any`. Initially, these steps do nothing.

error steps
: An algorithm that takes a single parameter of type `any`. Initially, the default error algorithm.

complete steps
: An algorithm with no parameters. Initially, these steps do nothing.

> The internal observer struct is used to mirror the `next`, `error`, and `complete` callback functions. For any `Observable` that is subscribed by JavaScript via the `subscribe()` method, these algorithm "steps" will just be a wrapper around invoking the corresponding `next`, `error`, and `complete` callback functions provided by script.
>
> But when internal spec prose (not user script) subscribes to an `Observable`, these "steps" are arbitrary spec algorithms that are not provided via an `ObserverUnion` packed with Web IDL callback functions. See the § 2.3.3 Promise-returning operators that make use of this, for example.

To ***convert to an Observable*** an `any` *value*, run these steps:

> NOTE: We split this algorithm out from the Web IDL `from()` method, so that spec prose can convert values to without going through the Web IDL bindings.

1. If Type(*value*) is not Object, throw a `TypeError`.

> NOTE: This prevents primitive types from being coerced into iterables (e.g., String). See discussion in WICG/observable#125.

2. ***From Observable***: If *value*'s specific type is an `Observable`, then return *value*.
3. ***From async iterable***: Let *asyncIteratorMethod* be ? GetMethod(*value*, `%Symbol.asyncIterator%`).

> NOTE: We use GetMethod instead of GetIterator because we're only probing for async iterator protocol support, and we don't want to throw if it's not implemented. GetIterator throws errors in BOTH of the following cases: (a) no iterator protocol is implemented, (b) an iterator protocol is implemented, but isn't callable or its getter throws. GetMethod lets us ONLY throw in the latter case.

4. If *asyncIteratorMethod*'s is undefined or null, then jump to the step labeled From iterable.
5. Let *nextAlgorithm* be the following steps, given a `Subscriber` *subscriber* and an Iterator Record *iteratorRecord*:

    1. If *subscriber*'s subscription controller's signal is aborted, then return.
    2. Let *nextPromise* be a `Promise`-or-undefined, initially undefined.
    3. Let *nextCompletion* be IteratorNext(*iteratorRecord*).

        > NOTE: We use IteratorNext here instead of IteratorStepValue, because IteratorStepValue expects the iterator's `next()` method to return an object that can immediately be inspected for a value, whereas in the async iterator case, `next()` is expected to return a Promise/thenable (which we wrap in a Promise and react to to get that value).

    4. If *nextCompletion* is a throw completion, then:

        1. Assert: *iteratorRecord*'s [[Done]] is true.
        2. Set *nextPromise* to a promise rejected with *nextRecord*'s [[Value]].

    5. Otherwise, if *nextRecord* is normal completion, then set *nextPromise* to a promise resolved with *nextRecord*'s [[Value]].

        > NOTE: This is done in case *nextRecord*'s [[Value]] is not *itself* already a `Promise`.

    6. React to *nextPromise*:

        - If *nextPromise* was fulfilled with value *iteratorResult*, then:

            1. If Type(*iteratorResult*) is not Object, then run *subscriber*'s `error()` method with a `TypeError` and abort these steps.
            2. Let *done* be IteratorComplete(*iteratorResult*).
            3. If *done* is a throw completion, then run *subscriber*'s `error()` method with *done*'s [[Value]] and abort these steps.
            4. If *done*'s [[Value]] is true, then run *subscriber*'s `complete()` and abort these steps.
            5. Let *value* be IteratorValue(*iteratorResult*).
            6. If *value* is a throw completion, then run *subscriber*'s `error()` method with *value*'s [[Value]] and abort these steps.
            7. Run *subscriber*'s `next()` given *value*'s [[Value]].
            8. Run *nextAlgorithm* given *subscriber* and *iteratorRecord*.

        - If *nextPromise* was rejected with reason *r*, then run *subscriber*'s `error()` method given *r*.

6. Return a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:

    1. If *subscriber*'s subscription controller's signal is aborted, then return.
    2. Let *iteratorRecordCompletion* be GetIterator(*value*, async).

        > NOTE: This both re-invokes any `%Symbol.asyncIterator%` method getters on *value*—note that whether this is desirable is an extreme corner case, but it matches test expectations; see issue#127 for discussion—and invokes the protocol itself to obtain an Iterator Record.

    3. If *iteratorRecordCompletion* is a throw completion, then run *subscriber*'s `error()` method with *iteratorRecordCompletion*'s [[Value]] and abort these steps.

        > NOTE: This means we invoke the `error()` method synchronously with respect to subscription, which is the only time this can happen for async iterables that are converted to `Observable`s. In all other cases, errors are propagated to the observer asynchronously, with microtask timing, by virtue of being wrapped in a rejected `Promise` that *nextAlgorithm* reacts to. This synchronous-error-propagation behavior is consistent with language constructs, i.e., **for-await of** loops that invoke `%Symbol.asyncIterator%` and synchronously re-throw exceptions to catch blocks outside the loop, before any Awaiting takes place.

    4. Let *iteratorRecord* be ! *iteratorRecordCompletion*.
    5. Assert: *iteratorRecord* is an Iterator Record.
    6. If *subscriber*'s subscription controller's signal is aborted, then return.
    7. Add the following abort algorithm to *subscriber*'s subscription controller's signal:

        1. Run AsyncIteratorClose(*iteratorRecord*, NormalCompletion(*subscriber*'s subscription controller's abort reason)).

    8. Run *nextAlgorithm* given *subscriber* and *iteratorRecord*.

7. ***From iterable***: Let *iteratorMethod* be ? GetMethod(*value*, `%Symbol.iterator%`).
8. If *iteratorMethod* is undefined, then jump to the step labeled From Promise.

Otherwise, return a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:

1. If *subscriber*'s subscription controller's signal is aborted, then return.
2. Let *iteratorRecordCompletion* be GetIterator(*value*, sync).
3. If *iteratorRecordCompletion* is a throw completion, then run *subscriber*'s `error()` method, given *iteratorRecordCompletion*'s [[Value]], and abort these steps.
4. Let *iteratorRecord* be ! *iteratorRecordCompletion*.
5. If *subscriber*'s subscription controller's signal is aborted, then return.
6. Add the following abort algorithm to *subscriber*'s subscription controller's signal:
    1. Run IteratorClose(*iteratorRecord*, NormalCompletion(UNUSED)).
7. While true:
    1. Let *next* be IteratorStepValue(*iteratorRecord*).
    2. If *next* is a throw completion, then run *subscriber*'s `error()` method, given *next*'s [[Value]], and break.
    3. Set *next* to ! *next*.
    4. If *next* is done, then:
        1. Assert: *iteratorRecord*'s [[Done]] is true.
        2. Run *subscriber*'s `complete()`.
        3. Return.
    5. Run *subscriber*'s `next()` given *next*.
    6. If *subscriber*'s subscription controller's signal is aborted, then break.
9. ***From Promise***: If IsPromise(*value*) is true, then:
    1. Return a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
        1. React to *value*:
            1. If *value* was fulfilled with value *v*, then:
                1. Run *subscriber*'s `next()` method, given *v*.
                2. Run *subscriber*'s `complete()` method.
            2. If *value* was rejected with reason *r*, then run *subscriber*'s `error()` method, given *r*.
10. Throw a `TypeError`.

▼ TESTS

observable-from.any.js (live test) (source)

To ***subscribe to an `Observable`*** given an `ObserverUnion`-or-internal observer *observer*, and a `SubscribeOptions` *options*, run these steps:

NOTE: We split this algorithm out from the Web IDL `subscribe()` method, so that spec prose can subscribe to an `Observable` without going through the Web IDL bindings. See w3c/IntersectionObserver#464 for similar context, where "internal" prose must not go through Web IDL bindings on objects whose properties could be mutated by JavaScript. See § 2.3.3 Promise-returning operators for usage of this.

1. If this's relevant global object is a `Window` object, and its associated Document is not fully active, then return.
2. Let *internal observer* be a new internal observer.
3. Process *observer* as follows:

↦ If *observer* is an `ObservableSubscriptionCallback`

Set *internal observer*'s next steps to these steps that take an `any` *value*:

1. Invoke *observer* with *value*.

   If an exception *E* was thrown, then report the exception *E*.

↪ If *observer* is a `SubscriptionObserver`

1. If *observer*'s `next` exists, then set *internal observer*'s next steps to these steps that take an `any` *value*:
    1. Invoke *observer*'s `next` with *value*.

       If an exception *E* was thrown, then report the exception *E*.
2. If *observer*'s `error` exists, then set *internal observer*'s error steps to these steps that take an `any` *error*:
    1. Invoke *observer*'s `error` with *error*.

       If an exception *E* was thrown, then report the exception *E*.
3. If *observer*'s `complete` exists, then set *internal observer*'s complete steps to these steps:
    1. Invoke *observer*'s `complete`.

       If an exception *E* was thrown, then report the exception *E*.

↪ If *observer* is an internal observer

Set *internal observer* to *observer*.

4. Assert: *internal observer*'s error steps is either the default error algorithm, or an algorithm that invokes the provided `error` callback function.
5. If this's weak subscriber is not null and this's weak subscriber's active is true:
    1. Let *subscriber* be this's weak subscriber.
    2. Append *internal observer* to *subscriber*'s internal observers.
    3. If *options*'s `signal` exists, then:
        1. If *options*'s `signal` is aborted, then remove *internal observer* from *subscriber*'s internal observers.
        2. Otherwise, add the following abort algorithm to *options*'s `signal`:
            1. If *subscriber*'s active is false, then abort these steps.
            2. Remove *internal observer* from *subscriber*'s internal observers.
            3. If *subscriber*'s internal observers is empty, then close *subscriber* with *options*'s `signal`'s abort reason.
    4. Return.
6. Let *subscriber* be a new `Subscriber`.
7. Append *internal observer* to *subscriber*'s internal observers.
8. Set this's weak subscriber to *subscriber*.
9. If *options*'s `signal` exists, then:
    1. If *options*'s `signal` is aborted, then close *subscriber* given *options*'s `signal` abort reason.
    2. Otherwise, add the following abort algorithm to *options*'s `signal`:
        1. If *subscriber*'s active is false, then abort these steps.
        2. Remove *internal observer* from *subscriber*'s internal observers.
        3. If *subscriber*'s internal observers is empty, then close *subscriber* with *options*'s `signal`'s abort reason.
10. If this's subscribe callback is a `SubscribeCallback`, invoke it with *subscriber*.

    If an exception *E* was thrown, call *subscriber*'s `error()` method with *E*.
11. Otherwise, run the steps given by this's subscribe callback, given *subscriber*.

▼ TESTS

observable-constructor.any.js (live test) (source)
observable-constructor.window.js (live test) (source)

2.3. Operators

For now, see https://github.com/wicg/observable#operators.

2.3.1. `from()`

The ***`from(value)`*** method steps are:

1. Return the result of converting *value* to an `Observable`. Rethrow any exceptions.

2.3.2. `Observable`-returning operators

The ***`takeUntil(value)`*** method steps are:

1. Let *sourceObservable* be this.
2. Let *notifier* be the result of converting *value* to an `Observable`.
3. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:

    Note that this method involves Subscribing to two `Observable`s: (1) *notifier*, and (2) *sourceObservable*. We "unsubscribe" from **both** of them in the following situations:
    1. *notifier* starts emitting values (either "next" or "error"). In this case, we unsubscribe from *notifier* since we got all we need from it, and no longer need it to keep producing values. We also unsubscribe from *sourceObservable*, because it no longer needs to produce values that get plumbed through this method's returned *observable*, because we're manually ending the subscription to *observable*, since *notifier* finally produced a value.
    2. *sourceObservable* either `error()`s or `complete()`s itself. In this case, we unsubscribe from *notifier* since we no longer need to listen for values it emits in order to determine when *observable* can stop mirroring values from *sourceObservable* (since *sourceObservable* ran to completion by itself). Unsubscribing from *sourceObservable* isn't necessary, since its subscription has been exhausted by itself.

    1. Let *notifierObserver* be a new internal observer, initialized as follows:

        next steps
        Run *subscriber*'s `complete()` method.

NOTE: This will "unsubscribe" from *sourceObservable*, if it has been subscribed to by this point. This is because *sourceObservable* is subscribed to with the "outer" *subscriber*'s subscription controller's signal as an input signal, and that signal will get aborted when the "outer" *subscriber*'s `complete()` is called above (and below).

error steps
Run *subscriber*'s `complete()` method.

NOTE: We do not specify complete steps, because if the *notifier* `Observable` completes itself, we do not need to complete the *subscriber* associated with the *observable* returned from this method. Rather, the *observable* will continue to mirror *sourceObservable* uninterrupted.

2. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
3. Subscribe to *notifier* given *notifierObserver* and *options*.
4. If *subscriber*'s active is false, then return.

NOTE: This means that *sourceObservable*'s subscribe callback will not even get invoked once, if *notifier* synchronously emits a value. If *notifier* only "completes" synchronously though (without emitting a "next" or "error" value), then *subscriber*'s active will still be true, and we proceed to subscribe to *sourceObservable*, which *observable* will mirror uninterrupted.

5. Let *sourceObserver* be a new internal observer, initialized as follows:

next steps
Run *subscriber*'s `next()` method, given the passed in *value*.

error steps
Run *subscriber*'s `error()` method, given the passed in *error*.

complete steps
Run *subscriber*'s `complete()` method.

NOTE: *sourceObserver* is mostly a pass-through, mirroring everything that *sourceObservable* emits, with the exception of having the ability to unsubscribe from the *notifier* `Observable` in the case where *sourceObservable* is exhausted before *notifier* emits anything.

6. Subscribe to *sourceObservable* given *sourceObserver* and *options*.

4. Return *observable*.

▼ TESTS
observable-takeUntil.any.js (live test) (source)
observable-takeUntil.window.js (live test) (source)

The ***`map(mapper)`*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:

1. Let *idx* be an `unsigned long long`, initially 0.
2. Let *sourceObserver* be a new internal observer, initialized as follows:

next steps
1. Invoke *mapper* with the passed in *value*, and *idx*, and let *mappedValue* be the returned value.

If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

2. Increment *idx*.
3. Run *subscriber*'s `next()` method, given *mappedValue*.

error steps
Run *subscriber*'s `error()` method, given the passed in *error*.

complete steps
Run *subscriber*'s `complete()` method.

3. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
4. Subscribe to *sourceObservable* given *sourceObserver* and *options*.

3. Return *observable*.

▼ TESTS
observable-map.any.js (live test) (source)
observable-map.window.js (live test) (source)

The ***`filter(predicate)`*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:

1. Let *idx* be an `unsigned long long`, initially 0.
2. Let *sourceObserver* be a new internal observer, initialized as follows:

next steps
1. Invoke *predicate* with the passed in *value* and *idx*, and let *matches* be the returned value.

If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

2. Set *idx* to *idx* + 1.
3. If *matches* is true, then run *subscriber*'s `next()` method, given *value*.

error steps
Run *subscriber*'s `error()` method, given the passed in *error*.

complete steps
Run *subscriber*'s `complete()` method.

3. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
4. Subscribe to *sourceObservable* given *sourceObserver* and *options*.

3. Return *observable*.

▼ TESTS
observable-filter.any.js (live test) (source)

The ***take(amount)*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
    1. Let *remaining* be *amount*.
    2. If *remaining* is 0, then run *subscriber*'s `complete()` method and abort these steps.
    3. Let *sourceObserver* be a new internal observer, initialized as follows:

        next steps
        1. Run *subscriber*'s `next()` method with the passed in *value*.
        2. Decrement *remaining*.
        3. If *remaining* is 0, then run *subscriber*'s `complete()` method.

        error steps
        Run *subscriber*'s `error()` method, given the passed in *error*.

        complete steps
        Run *subscriber*'s `complete()` method.

    4. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
    5. Subscribe to *sourceObservable* given *sourceObserver* and *options*.
3. Return *observable*.

▼ TESTS
observable-take.any.js (live test) (source)

The ***drop(amount)*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
    1. Let *remaining* be *amount*.
    2. Let *sourceObserver* be a new internal observer, initialized as follows:

        next steps
        1. If *remaining* is > 0, then decrement *remaining* and abort these steps.
        2. Assert: *remaining* is 0.
        3. Run *subscriber*'s `next()` method with the passed in *value*.

        error steps
        Run *subscriber*'s `error()` method, given the passed in *error*.

        complete steps
        Run *subscriber*'s `complete()` method.

    3. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
    4. Subscribe to *sourceObservable* given *sourceObserver* and *options*.
3. Return *observable*.

▼ TESTS
observable-drop.any.js (live test) (source)

The ***flatMap(mapper)*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
    1. Let *idx* be an `unsigned long long`, initially 0.
    2. Let *outerSubscriptionHasCompleted* to a boolean, initially false.
    3. Let *queue* be a new list of `any` values, initially empty.

NOTE: This *queue* is used to store any `Observable`s emitted by *sourceObservable*, while *observable* is currently subscribed to an `Observable` emitted earlier by *sourceObservable* that has not yet been exhausted.

4. Let *activeInnerSubscription* be a boolean, initially false.
5. Let *sourceObserver* be a new internal observer, initialized as follows:

   next steps

   1. If *activeInnerSubscription* is true, then:
      1. Append *value* to *queue*.

         NOTE: This *value* will eventually be processed once the `Observable` that is currently subscribed-to (as indicated by *activeInnerSubscription*) is exhausted.
      2. Otherwise:
         1. Set *activeInnerSubscription* to true.
         2. Run the flatmap process next value steps with *value*, *subscriber*, *mapper*, and **references** to all of the following: *queue*, *activeInnerSubscription*, *outerSubscriptionHasCompleted*, and *idx*.

            Note: This flatmap process next value steps will subscribe to the `Observable` derived from *value* (if one such can be derived) and keep processing values from it until its subscription becomes inactive (either by error or completion). If this "inner" `Observable` completes, then the processing steps will recursively invoke themselves with the next `any` in *queue*.

            If no such value exists, then the processing steps will terminate, **unsetting** *activeInnerSubscription*, so that future values emitted from *sourceObservable* are processed correctly.

   error steps

   Run *subscriber*'s `error()` method, given the passed in *error*.

   complete steps

   1. Set *outerSubscriptionHasCompleted* to true.

      NOTE: If *activeInnerSubscription* is true, then the below step will *not* complete *subscriber*. In that case, the flatmap process next value steps will be responsible for completing *subscriber* when *queue* is empty, after the "inner" subscription becomes inactive.
   2. If *activeInnerSubscription* is false and *queue* is empty, run *subscriber*'s `complete()` method.
6. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
7. Subscribe to *sourceObservable* given *sourceObserver* and *options*.

3. Return *observable*.

The ***flatmap process next value steps***, given an `any` *value*, a `Subscriber` *subscriber*, a `Mapper` *mapper*, and **references** to all of the following: a list of `any` values *queue*, a boolean *activeInnerSubscription*, a boolean *outerSubscriptionHasCompleted*, and an `unsigned long long` *idx*:

1. Let *mappedResult* be the result of invoking *mapper* with *value* and *idx*.

   If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.
2. Set *idx* to *idx* + 1.
3. Let *innerObservable* be the result of calling `from()` with *mappedResult*.

   If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

   ISSUE 1 We shouldn't invoke `from()` directly. Rather, we should call some internal algorithm that passes-back the exceptions for us to handle properly here, since we want to pipe them to *subscriber*.
4. Let *innerObserver* be a new internal observer, initialized as follows:

   next steps

   Run *subscriber*'s `next()` method, given the passed in *value*.

   error steps

   Run *subscriber*'s `error()` method, given the passed in *error*.

   complete steps

   1. If *queue* is not empty, then:
      1. Let *nextValue* be the first item in *queue*; remove remove this item from *queue*.
      2. Run flatmap process next value steps given *nextValue*, *subscriber*, *mapper*, and **references** to *queue* and *activeInnerSubscription*.
   2. Otherwise:
      1. Set *activeInnerSubscription* to false.

         NOTE: Because *activeInnerSubscription* is a reference, this has the effect of ensuring that all subsequent values emitted from the "outer" `Observable` (called *sourceObservable*.
      2. If *outerSubscriptionHasCompleted* is true, run *subscriber*'s `complete()` method.

> NOTE: This means the "outer" `Observable` has already completed, but did not proceed to complete *subscriber* yet because there was at least one more pending "inner" `Observable` (i.e., *innerObservable*) that had already been queued and had not yet completed. Until right now!

5. Let *innerOptions* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
6. Subscribe to *innerObservable* given *innerObserver* and *innerOptions*.

The ***`switchMap(mapper)`*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
    1. Let *idx* be an `unsigned long long`, initially 0.
    2. Let *outerSubscriptionHasCompleted* be a boolean, initially false.
    3. Let *activeInnerAbortController* be an `AbortController`-or-null, initially null.

        > NOTE: This `AbortController` is assigned to a new `AbortController` only by this algorithm's next steps (below), and only assigned to null by the switchmap process next value steps, when the "inner" `Observable` either completes or errors. This variable is used as a marker for whether there is currently an active "inner" subscription. The complete steps below care about this, because if *sourceObservable* completes while there is an active "inner" subscription, we do not immediately complete *subscriber*. In that case, *subscriber*'s completion becomes blocked on the "inner" subscription's completion.

    4. Let *sourceObserver* be a new internal observer, initialized as follows:

    next steps
        1. If *activeInnerAbortController* is not null, then signal abort *activeInnerAbortController*.

            > NOTE: This "unsubscribes" from the "inner" `Observable` that was derived from the value that was *last* pushed from *sourceObservable*. Then we immediately subscribe to the *new* `Observable` that we're about to derive from *value*, i.e., the *most-recently* pushed value from *sourceObservable*.

        2. Set *activeInnerAbortController* to a new `AbortController`.
        3. Run the switchmap process next value steps with *value*, *subscriber*, *mapper*, and **references** to all of the following: *activeInnerAbortController*, *outerSubscriptionHasCompleted*, and *idx*.

            > NOTE: The switchmap process next value steps will subscribe to the `Observable` derived from *value* (if one such can be derived) and keep processing values from it until either (1) its subscription becomes inactive (either by error or completion), or (2) *activeInnerAbortController* gets aborted, due to *sourceObservable* pushing another *newer* value that will replace the current "inner" subscription.

    error steps
        Run *subscriber*'s `error()` method, given the passed in *error*.

    complete steps
        1. Set *outerSubscriptionHasCompleted* to true.

            > NOTE: If *activeInnerAbortController* is not null, then we don't immediately complete *subscriber*. Instead, the switchmap process next value steps will complete *subscriber* when the inner subscription finally completes itself.

        2. If *activeInnerAbortController* is null, run *subscriber*'s `complete()` method.

    5. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
    6. Subscribe to *sourceObservable* given *sourceObserver* and *options*.
3. Return *observable*.

The ***switchmap process next value steps***, given an `any` *value*, a `Subscriber` *subscriber*, a `Mapper` *mapper*, and **references** to all of the following: an `AbortController` *activeInnerAbortController*, a boolean *outerSubscriptionHasCompleted*, and an `unsigned long long` *idx* are to run these steps:

1. Let *mappedResult* be the result of invoking *mapper* with *value* and *idx*.

    If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

2. Set *idx* to *idx* + 1.
3. Let *innerObservable* be the result of calling `from()` with *mappedResult*.

    If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

4. Let *innerObserver* be a new internal observer, initialized as follows:

    next steps
        Run *subscriber*'s `next()` method, given the passed in *value*.

    error steps
        Run *subscriber*'s `error()` method, given the passed in *error*.

        > NOTE: We don't have to set *activeInnerAbortController* to null here, to signal to the `switchMap()` method steps above that the inner "subscription" has been canceled. That's because calling *subscriber*'s `error()` method already unsubscribes from the "outer" source Observable, so it will not be able to push any more values to the `switchMap()` internal observer.

complete steps

1. If *outerSubscriptionHasCompleted* is true, run *subscriber*'s `complete()` method.
2. Otherwise, set *activeInnerAbortController* to null.

   NOTE: Because this variable is a reference, it signals to the switchMap complete steps that there is no active inner subscription.

5. Let *innerOptions* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*activeInnerAbortController*'s signal, *subscriber*'s subscription controller's signal», using `AbortSignal`, and the current realm.
6. Subscribe to *innerObservable* given *innerObserver* and *innerOptions*.

The ***`inspect(inspectorUnion)`*** method steps are:

1. Let *subscribe callback* be a `VoidFunction`-or-null, initially null.
2. Let *next callback* be a `ObservableSubscriptionCallback`-or-null, initially null.
3. Let *error callback* be a `ObservableSubscriptionCallback`-or-null, initially null.
4. Let *complete callback* be a `VoidFunction`-or-null, initially null.
5. Let *abort callback* be a `ObservableInspectorAbortHandler`-or-null, initially null.
6. Process *inspectorUnion* as follows:

↪ **If *inspectorUnion* is an `ObservableSubscriptionCallback`**

1. Set *next callback* to *inspectorUnion*.

↪ **If *inspectorUnion* is an `ObservableInspector`**

1. If `subscribe` exists in *inspectorUnion*, then set *subscribe callback* to it.
2. If `next` exists in *inspectorUnion*, then set *next callback* to it.
3. If `error` exists in *inspectorUnion*, then set *error callback* to it.
4. If `complete` exists in *inspectorUnion*, then set *complete callback* to it.
5. If `abort` exists in *inspectorUnion*, then set *abort callback* to it.

7. Let *sourceObservable* be this.
8. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:

   1. If *subscribe callback* is not null, then invoke it.

      If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

      NOTE: The result of this is that *sourceObservable* is never subscribed to.

   2. If *abort callback* is not null, then add the following abort algorithm to *subscriber*'s subscription controller's signal:

      1. Invoke *abort callback* with *subscriber*'s subscription controller's signal's abort reason.

         If an exception *E* was thrown, then report the exception *E*.

   3. Let *sourceObserver* be a new internal observer, initialized as follows:

      next steps

      1. If *next callback* is not null, then invoke *next callback* with the passed in *value*.

         If an exception *E* was thrown, then:

         1. Remove *abort callback* from *subscriber*'s subscription controller's signal.

            NOTE: This step is important, because the *abort callback* is only meant to be called for *consumer-initiated* unsubscriptions. When the producer terminates the subscription (via *subscriber*'s `error()` or `complete()` methods) like below, we have to ensure that *abort callback* is not run.

            ISSUE 2 This matches Chromium's implementation, but consider holding a reference to the originally-passed-in `SubscribeOptions`'s `signal` and just invoking *abort callback* when *it* aborts. The result is likely the same, but needs investigation.

         2. Run *subscriber*'s `error()` method, given *E*, and abort these steps.

      2. Run *subscriber*'s `next()` method with the passed in *value*.

      error steps

      1. Remove *abort callback* from *subscriber*'s subscription controller's signal.
      2. If *error callback* is not null, then invoke *error callback* given the passed in *error*.

         If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

      3. Run *subscriber*'s `error()` method, given the passed in *error*.

      complete steps

      1. Remove *abort callback* from *subscriber*'s subscription controller's signal.
      2. If *complete callback* is not null, then invoke *complete callback*.

         If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

      3. Run *subscriber*'s `complete()` method.

4. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
5. Subscribe to *sourceObservable* given *sourceObserver* and *options*.

9. Return *observable*.

▼ TESTS
observable-inspect.any.js (live test) (source)

The ***`catch(callback)`*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
    1. Let *sourceObserver* be a new internal observer, initialized as follows:

        next steps
        Run *subscriber*'s `next()` method, given the passed in *value*.

        error steps
        1. Invoke *callback* with the passed in *error*. Let *result* be the returned value.

            If an exception *E* was thrown, then run *subscriber*'s `error()` with *E*, and abort these steps.
        2. Let *innerObservable* be the result of calling `from()` with *result*.

            If an exception *E* was thrown, then run *subscriber*'s `error()` method, given *E*, and abort these steps.

            ISSUE 3 We shouldn't invoke `from()` directly. Rather, we should call some internal algorithm that passes-back the exceptions for us to handle properly here, since we want to pipe them to *subscriber*.
        3. Let *innerObserver* be a new internal observer, initialized as follows:

            next steps
            Run *subscriber*'s `next()` method, given the passed in *value*.

            error steps
            Run *subscriber*'s `error()` method, given the passed in *error*.

            complete steps
            Run *subscriber*'s `complete()` method.
        4. Let *innerOptions* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
        5. Subscribe to *innerObservable* given *innerObserver* and *innerOptions*.

            NOTE: We're free to subscribe to *innerObservable* here without first "unsubscribing" from *sourceObservable*, and without fear that *sourceObservable* will keep emitting values, because all of this is happening inside of the error steps associated with *sourceObservable*. This means *sourceObservable* has already completed its subscription and will no longer produce any values, and we are free to safely switch our source of values to *innerObservable*.

        complete steps
        Run *subscriber*'s `complete()` method.
    2. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
    3. Subscribe to *sourceObservable* given *sourceObserver* and *options*.
3. Return *observable*.

The ***`finally(callback)`*** method steps are:

1. Let *sourceObservable* be this.
2. Let *observable* be a new `Observable` whose subscribe callback is an algorithm that takes a `Subscriber` *subscriber* and does the following:
    1. Run *subscriber*'s `addTeardown()` method with *callback*.
    2. Let *sourceObserver* be a new internal observer, initialized as follows:

        next steps
        Run *subscriber*'s `next()` method, given the passed in *value*.

        error steps
        1. Run *subscriber*'s `error()` method, given the passed in *error*.

        complete steps
        1. Run *subscriber*'s `complete()` method.
    3. Let *options* be a new `SubscribeOptions` whose `signal` is *subscriber*'s subscription controller's signal.
    4. Subscribe to *sourceObservable* given *sourceObserver* and *options*.
3. Return *observable*.

2.3.3. `Promise`-returning operators

The ***`toArray(options)`*** method steps are:

1. Let *p* a new promise.

2. If *options*'s `signal` is not null:

    1. If *options*'s `signal` is aborted, then:

        1. Reject *p* with *options*'s `signal`'s abort reason.

        2. Return *p*.

    2. Add the following abort algorithm to *options*'s `signal`:

        1. Reject *p* with *options*'s `signal`'s abort reason.

    NOTE: All we have to do here is reject *p*. Note that the subscription to this `Observable` will also be closed automatically, since the "inner" Subscriber gets closed in response to *options*'s `signal` getting signal abort.

3. Let *values* be a new list.

4. Let *observer* be a new internal observer, initialized as follows:

    next steps
    Append the passed in *value* to *values*.

    error steps
    Reject *p* with the passed in *error*.

    complete steps
    Resolve *p* with *values*.

5. Subscribe to this given *observer* and *options*.

6. Return *p*.

▼ TESTS
observable-toArray.any.js (live test) (source)

The ***forEach(callback, options)*** method steps are:

1. Let *p* a new promise.

2. Let *visitor callback controller* be a new `AbortController`.

3. Let *internal options* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*visitor callback controller*'s signal, *options*'s `signal` if non-null», using `AbortSignal`, and the current realm.

    Many trivial internal observers act as pass-throughs, and do not control the subscription to the `Observable` that they represent; that is, their error steps and complete steps are called when the subscription is terminated, and their next steps simply pass some version of the given value along the chain.

    For this operator, however, the below *observer*'s next steps are responsible for actually aborting the underlying subscription to this, in the event that *callback* throws an exception. In that case, the `SubscribeOptions`'s `signal` we pass through to "Subscribe to an `Observable`", needs to be a dependent signal derived from *options*'s `signal`, **and** the `AbortSignal` of an `AbortController` that the next steps below has access to, and can signal abort when needed.

4. If *internal options*'s `signal` is aborted, then:

    1. Reject *p* with *internal options*'s `signal`'s abort reason.

    2. Return *p*.

5. Add the following abort algorithm to *internal options*'s `signal`:

    1. Reject *p* with *internal options*'s `signal`'s abort reason.

    NOTE: The fact that rejection of *p* is tied to *internal options*'s `signal`, and not *options*'s `signal` means, that any microtasks queued during the firing of *options*'s `signal`'s `abort` event will run before *p*'s rejection handler runs.

6. Let *idx* be an `unsigned long long`, initially 0.

7. Let *observer* be a new internal observer, initialized as follows:

    next steps

    1. Invoke *callback* with the passed in *value*, and *idx*.

        If an exception *E* was thrown, then reject *p* with *E*, and signal abort *visitor callback controller* with *E*.

    2. Increment *idx*.

    error steps
    Reject *p* with the passed in *error*.

    complete steps
    Resolve *p* with `undefined`.

8. Subscribe to this given *observer* and *internal options*.

9. Return *p*.

▼ TESTS
observable-forEach.any.js (live test) (source)

The ***every(predicate, options)*** method steps are:

1. Let *p* a new promise.

2. Let *controller* be a new `AbortController`.

3. Let *internal options* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*controller*'s signal, *options*'s `signal` if non-null», using `AbortSignal`, and the current realm.
4. If *internal options*'s `signal` is aborted, then:
    1. Reject *p* with *internal options*'s `signal`'s abort reason.
    2. Return *p*.
5. Add the following abort algorithm to *internal options*'s `signal`:
    1. Reject *p* with *internal options*'s `signal`'s abort reason.
6. Let *idx* be an `unsigned long long`, initially 0.
7. Let *observer* be a new internal observer, initialized as follows:

    next steps
    1. Invoke *predicate* with the passed in *value* and *idx*, and let *passed* be the returned value.

       If an exception *E* was thrown, then reject *p* with *E*, and signal abort *controller* with *E*.
    2. Set *idx* to *idx* + 1.
    3. If *passed* is false, then resolve *p* with false, and signal abort *controller*.

    error steps
    Reject *p* with the passed in *error*.

    complete steps
    Resolve *p* with true.
8. Subscribe to this given *observer* and *internal options*.
9. Return *p*.

The ***`first(options)`*** method steps are:

1. Let *p* a new promise.
2. Let *controller* be a new `AbortController`.
3. Let *internal options* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*controller*'s signal, *options*'s `signal` if non-null», using `AbortSignal`, and the current realm.
4. If *internal options*'s `signal` is aborted, then:
    1. Reject *p* with *internal options*'s `signal`'s abort reason.
    2. Return *p*.
5. Add the following abort algorithm to *internal options*'s `signal`:
    1. Reject *p* with *internal options*'s `signal`'s abort reason.
6. Let *internal observer* be a new internal observer, initialized as follows:

    next steps
    1. Resolve *p* with the passed in *value*.
    2. Signal abort *controller*.

    error steps
    Reject *p* with the passed in *error*.

    complete steps
    Reject *p* with a new `RangeError`.

    > NOTE: This is only reached when the source `Observable` completes *before* it emits a single value.
7. Subscribe to this given *internal observer* and *internal options*.
8. Return *p*.

The ***`last(options)`*** method steps are:

1. Let *p* a new promise.
2. If *options*'s `signal` is not null:
    1. If *options*'s `signal` is aborted, then:
        1. Reject *p* with *options*'s `signal`'s abort reason.
        2. Return *p*.
    2. Add the following abort algorithm to *options*'s `signal`:
        1. Reject *p* with *options*'s `signal`'s abort reason.
3. Let *lastValue* be an `any`-or-null, initially null.
4. Let *hasLastValue* be a boolean, initially false.
5. Let *observer* be a new internal observer, initialized as follows:

    next steps
    1. Set *hasLastValue* to true.
    2. Set *lastValue* to the passed in *value*.

    error steps
    Reject *p* with the passed in *error*.

    complete steps
    1. If *hasLastValue* is true, resolve *p* with *lastValue*.

1. Otherwise, reject *p* with a new `RangeError`.

   NOTE: See the note in `first()`.

6. Subscribe to this given *observer* and *options*.
7. Return *p*.

The ***find(predicate, options)*** method steps are:

1. Let *p* a new promise.
2. Let *controller* be a new `AbortController`.
3. Let *internal options* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*controller*'s signal, *options*'s `signal` if non-null», using `AbortSignal`, and the current realm.
4. If *internal options*'s `signal` is aborted, then:
   1. Reject *p* with *internal options*'s `signal`'s abort reason.
   2. Return *p*.
5. Add the following abort algorithm to *internal options*'s `signal`:
   1. Reject *p* with *internal options*'s `signal`'s abort reason.
6. Let *idx* be an `unsigned long long`, initially 0.
7. Let *observer* be a new internal observer, initialized as follows:

   next steps
   1. Invoke *predicate* with the passed in *value* an *idx*, and let *passed* be the returned value.

      If an exception *E* was thrown, then reject *p* with *E*, and signal abort *controller* with *E*.
   2. Set *idx* to *idx* + 1.
   3. If *passed* is true, then resolve *p* with *value*, and signal abort *controller*.

   error steps
   Reject *p* with the passed in *error*.

   complete steps
   Resolve *p* with `undefined`.

8. Subscribe to this given *observer* and *internal options*.
9. Return *p*.

The ***some(predicate, options)*** method steps are:

1. Let *p* a new promise.
2. Let *controller* be a new `AbortController`.
3. Let *internal options* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*controller*'s signal, *options*'s `signal` if non-null», using `AbortSignal`, and the current realm.
4. If *internal options*'s `signal` is aborted, then:
   1. Reject *p* with *internal options*'s `signal`'s abort reason.
   2. Return *p*.
5. Add the following abort algorithm to *internal options*'s `signal`:
   1. Reject *p* with *internal options*'s `signal`'s abort reason.
6. Let *idx* be an `unsigned long long`, initially 0.
7. Let *observer* be a new internal observer, initialized as follows:

   next steps
   1. Invoke *predicate* with the passed in *value* and *idx*, and let *passed* be the returned value.

      If an exception *E* was thrown, then reject *p* with *E*, and signal abort *controller* with *E*.
   2. Set *idx* to *idx* + 1.
   3. If *passed* is true, then resolve *p* with true, and signal abort *controller*.

   error steps
   Reject *p* with the passed in *error*.

   complete steps
   Resolve *p* with false.

8. Subscribe to this given *observer* and *internal options*.
9. Return *p*.

The ***reduce(reducer, initialValue, options)*** method steps are:

1. Let *p* a new promise.
2. Let *controller* be a new `AbortController`.
3. Let *internal options* be a new `SubscribeOptions` whose `signal` is the result of creating a dependent abort signal from the list «*controller*'s signal, *options*'s `signal` if non-null», using `AbortSignal`, and the current realm.
4. If *internal options*'s `signal` is aborted, then:
   1. Reject *p* with *internal options*'s `signal`'s abort reason.
   2. Return *p*.

5. Add the following abort algorithm to *internal options*'s `signal`:

    1. Reject *p* with *internal options*'s `signal`'s abort reason.

6. Let *idx* be an `unsigned long long`, initially 0.

7. Let *accumulator* be *initialValue* if it is given, and uninitialized otherwise.

8. Let *observer* be a new internal observer, initialized as follows:

    next steps

    1. If *accumulator* is uninitialized (meaning no *initialValue* was passed in), then set *accumulator* to the passed in *value*, set *idx* to *idx* + 1, and abort these steps.

        NOTE: This means that *reducer* will not be called with the first *value* that this produces set as the `currentValue`. Rather, when the *second* value is eventually emitted, we will call *reducer* with *it* as the `currentValue`, and the first value (that we're saving here) as the `accumulator`.

    2. Invoke *reducer* with *accumulator* as `accumulator`, the passed in *value* as `currentValue`, and *idx* as `index`. Let *result* be the returned value.

        If an exception *E* was thrown, then reject *p* with *E*, and signal abort *controller* with *E*.

    3. Set *idx* to *idx* + 1.

    4. Set *accumulator* to *result*.

    error steps

    Reject *p* with the passed in *error*.

    complete steps

    1. If *accumulator* is not "`unset`", then resolve *p* with *accumulator*.

        Otherwise, reject *p* with a `TypeError`.

9. Subscribe to this given *observer* and *internal options*.

10. Return *p*.

§ 3. `EventTarget` integration

```
dictionary ObservableEventListenerOptions {
  boolean capture = false;
  boolean passive;
};

partial interface EventTarget {
  Observable when(DOMString type, optional ObservableEventListenerOptions options = {});
};
```

The ***when(type, options)*** method steps are:

1. If this's relevant global object is a `Window` object, and its associated Document is not fully active, then return.

2. Let *event target* be this.

3. Let *observable* be a new `Observable`, initialized as follows:

    subscribe callback

    An algorithm that takes a `Subscriber` *subscriber* and runs these steps:

    1. If *event target* is null, abort these steps.

        NOTE: This is meant to capture the fact that *event target* can be garbage collected by the time this algorithm runs upon subscription.

    2. If *subscriber*'s subscription controller's signal is aborted, abort these steps.

    3. Add an event listener with *event target* and an event listener defined as follows:

        type

        type

        callback

        The result of creating a new Web IDL `EventListener` instance representing a reference to a function of one argument of type `Event` *event*. This function executes the observable event listener invoke algorithm given *subscriber* and *event*.

        capture

        options's `capture`

        passive

        options's `passive`

        once

        false

        signal

        subscriber's subscription controller's signal

        NOTE: This ensures that the event listener is cleaned up when subscription controller's signal is aborted, regardless of an engine's ownership model.

4. Return *observable*.

The ***observable event listener invoke algorithm*** takes a `Subscriber` *subscriber* and an `Event` *event*, and runs these steps:

1. Run *subscriber*'s `next()` method with *event*.

▼ TESTS

observable-event-target.any.js (live test) (source)

observable-event-target.window.js (live test) (source)

§ 4. Security & Privacy Considerations

This material is being upstreamed from our explainer into this specification, and in the meantime you can consult the following resources:

- TAG Security/Privacy Questionnaire

§ 5. Acknowledgements

A special thanks to Ben Lesh for much of the design input for the `Observable` API, and his many years of work maintaining userland Observable code that made this contribution to the web platform possible.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[DOM] defines the following terms:

- AbortController
- AbortSignal
- Document
- Event
- EventListener
- EventTarget
- abort
- abort reason
- aborted
- add
- add an event listener
- callback
- capture
- create a dependent abort signal
- event listener
- once
- passive
- remove
- signal (for AbortController)
- signal (for event listener)
- signal abort (for AbortController)
- signal abort (for AbortSignal)
- type

[ECMASCRIPT] defines the following terms:

- !
- %Symbol.asyncIterator%
- %Symbol.iterator%
- ?
- AsyncIteratorClose
- Await
- GetIterator
- GetMethod
- IsPromise
- IteratorClose
- IteratorComplete
- IteratorNext
- IteratorStepValue
- IteratorValue
- Type
- current realm
- Iterator Record
- normal completion
- NormalCompletion
- Object
- throw completion

[HTML] defines the following terms:

- Window
- associated Document
- fully active
- microtask
- queue a microtask
- relevant global object
- report the exception

[INFRA] defines the following terms:

- append (for list)
- append (for set)
- assert
- boolean
- break
- empty
- exist (for list)
- exist (for map)
- for each
- item
- list
- ordered set
- remove
- struct
- while

[WEBIDL] defines the following terms:

- DOMString
- Promise
- RangeError
- TypeError
- VoidFunction
- a new promise
- a promise rejected with
- a promise resolved with
- an exception was thrown
- any
- boolean
- callback function
- invoke
- new
- react
- reject
- resolve
- sequence
- specific type
- this
- throw
- undefined
- unsigned long long

§ References

§ Normative References

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMASCRIPT]
ECMAScript Language Specification. URL: https://tc39.es/ecma262/multipage/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

§ IDL Index

```
[Exposed=*]
interface Subscriber {
  undefined next(any value);
  undefined error(any error);
  undefined complete();
  undefined addTeardown(VoidFunction teardown);

  // True after the Subscriber is created, up until either
  // complete()/error() are invoked, or the subscriber unsubscribes. Inside
  // complete()/error(), this attribute is true.
  readonly attribute boolean active;

  readonly attribute AbortSignal signal;
};

// SubscribeCallback is where the Observable "creator's" code lives. It's
// called when subscribe() is called, to set up a new subscription.
```

```
callback SubscribeCallback = undefined (Subscriber subscriber);
callback ObservableSubscriptionCallback = undefined (any value);

dictionary SubscriptionObserver {
  ObservableSubscriptionCallback next;
  ObservableSubscriptionCallback error;
  VoidFunction complete;
};

callback ObservableInspectorAbortHandler = undefined (any value);

dictionary ObservableInspector {
  ObservableSubscriptionCallback next;
  ObservableSubscriptionCallback error;
  VoidFunction complete;

  VoidFunction subscribe;
  ObservableInspectorAbortHandler abort;
};

typedef (ObservableSubscriptionCallback or SubscriptionObserver) ObserverUnion;
typedef (ObservableSubscriptionCallback or ObservableInspector) ObservableInspectorUnion;

dictionary SubscribeOptions {
  AbortSignal signal;
};

callback Predicate = boolean (any value, unsigned long long index);
callback Reducer = any (any accumulator, any currentValue, unsigned long long index);
callback Mapper = any (any value, unsigned long long index);
// Differs from Mapper only in return type, since this callback is exclusively
// used to visit each element in a sequence, not transform it.
callback Visitor = undefined (any value, unsigned long long index);

// This callback returns an `any` that must convert into an `Observable`, via
// the `Observable` conversion semantics.
callback CatchCallback = any (any value);

[Exposed=*]
interface Observable {
  constructor(SubscribeCallback callback);
  undefined subscribe(optional ObserverUnion observer = {}, optional SubscribeOptions opti

  // Constructs a native Observable from value if it's any of the following:
  //   - Observable
  //   - AsyncIterable
  //   - Iterable
  //   - Promise
  static Observable from(any value);

  // Observable-returning operators. See "Operators" section in the spec.
  //
  // takeUntil() can consume promises, iterables, async iterables, and other
  // observables.
  Observable takeUntil(any value);
  Observable map(Mapper mapper);
  Observable filter(Predicate predicate);
  Observable take(unsigned long long amount);
  Observable drop(unsigned long long amount);
  Observable flatMap(Mapper mapper);
  Observable switchMap(Mapper mapper);
  Observable inspect(optional ObservableInspectorUnion inspectorUnion = {});
  Observable catch(CatchCallback callback);
  Observable finally(VoidFunction callback);

  // Promise-returning operators.
  Promise<sequence<any>> toArray(optional SubscribeOptions options = {});
  Promise<undefined> forEach(Visitor callback, optional SubscribeOptions options = {});
  Promise<boolean> every(Predicate predicate, optional SubscribeOptions options = {});
  Promise<any> first(optional SubscribeOptions options = {});
  Promise<any> last(optional SubscribeOptions options = {});
  Promise<any> find(Predicate predicate, optional SubscribeOptions options = {});
  Promise<boolean> some(Predicate predicate, optional SubscribeOptions options = {});
  Promise<any> reduce(Reducer reducer, optional any initialValue, optional SubscribeOption
};

dictionary ObservableEventListenerOptions {
  boolean capture = false;
  boolean passive;
};

partial interface EventTarget {
  Observable when(DOMString type, optional ObservableEventListenerOptions options = {});
};
```

§ Issues Index

ISSUE 1 We shouldn't invoke `from()` directly. Rather, we should call some internal algorithm that passes-back the exceptions for us to handle properly here, since we want to pipe them to *subscriber*.

ISSUE 2 This matches Chromium's implementation, but consider holding a reference to the originally-passed-in `SubscribeOptions`'s `signal` and just invoking *abort callback* when *it* aborts. The result is likely the same, but needs investigation.

ISSUE 3 We shouldn't invoke `from()` directly. Rather, we should call some internal algorithm that passes-back the exceptions for us to handle properly here, since we want to pipe them to *subscriber*.