
ECMAScript Observable

This proposal introduces an **Observable** type to the ECMAScript standard library. The **Observable** type can be used to model push-based data sources such as DOM events, timer intervals, and sockets. In addition, observables are:

- *Compositional*: Observables can be composed with higher-order combinators.
- *Lazy*: Observables do not start emitting data until an **observer** has subscribed.

Example: Observing Keyboard Events

Using the **Observable** constructor, we can create a function which returns an observable stream of events for an arbitrary DOM element and event type.

```
1 function listen(element, eventName) {
2   return new Observable(observer => {
3     // Create an event handler which sends data to the sink
4     let handler = event => observer.next(event);
5
6     // Attach the event handler
7     element.addEventListener(eventName, handler, true);
8
9     // Return a cleanup function which will cancel the event stream
10    return () => {
11      // Detach the event handler from the element
12      element.removeEventListener(eventName, handler, true);
13    };
14  });
15 }
```

We can then use standard combinators to filter and map the events in the stream, just like we would with an array.

```
1 // Return an observable of special key down commands
2 function commandKeys(element) {
3   let keyCommands = { "38": "up", "40": "down" };
4
5   return listen(element, "keydown")
6     .filter(event => event.keyCode in keyCommands)
7     .map(event => keyCommands[event.keyCode])
8 }
```

Note: The “filter” and “map” methods are not included in this proposal. They may be added in a future version of this specification.

When we want to consume the event stream, we subscribe with an **observer**.

```
1 let subscription = commandKeys(inputElement).subscribe({
2   next(val) { console.log("Received key command: " + val) },
3   error(err) { console.log("Received an error: " + err) },
4   complete() { console.log("Stream complete") },
5 });
```

The object returned by **subscribe** will allow us to cancel the subscription at any time. Upon cancellation, the Observable's cleanup function will be executed.

```
1 // After calling this function, no more events will be sent
2 subscription.unsubscribe();
```

Motivation

The Observable type represents one of the fundamental protocols for processing asynchronous streams of data. It is particularly effective at modeling streams of data which originate from the environment and are pushed into the application, such as user interface events. By offering Observable as a component of the ECMAScript standard library, we allow platforms and applications to share a common push-based stream protocol.

Implementations

- RxJS 5
- core-js
- zen-observable
- fate-observable

Running Tests

To run the unit tests, install the **es-observable-tests** package into your project.

```
1 npm install es-observable-tests
```

Then call the exported `runTests` function with the constructor you want to test.

```
1 require("es-observable-tests").runTests(MyObservable);
```

API

Observable An Observable represents a sequence of values which may be observed.

```
1 interface Observable {
2
3     constructor(subscriber : SubscriberFunction);
4
5     // Subscribes to the sequence with an observer
6     subscribe(observer : Observer) : Subscription;
7
8     // Subscribes to the sequence with callbacks
9     subscribe(onNext : Function,
10              onError? : Function,
11              onComplete? : Function) : Subscription;
12
13     // Returns itself
14     [Symbol.observable]() : Observable;
15
16     // Converts items to an Observable
17     static of(...items) : Observable;
18
19     // Converts an observable or iterable to an Observable
20     static from(observable) : Observable;
21 }
22
23 interface Subscription {
24
25     // Cancels the subscription
26     unsubscribe() : void;
27
28     // A boolean value indicating whether the subscription is closed
29     get closed() : Boolean;
30 }
31
32 function SubscriberFunction(observer: SubscriptionObserver) : (void =>
33     void) | Subscription;
```

Observable.of `Observable.of` creates an Observable of the values provided as arguments. The values are delivered synchronously when `subscribe` is called.

```
1 Observable.of("red", "green", "blue").subscribe({
2     next(color) {
3         console.log(color);
4     }
5 });
6
7 /*
8 > "red"
9 > "green"
10 > "blue"
11 */
```

Observable.from `Observable.from` converts its argument to an Observable.

- If the argument has a `Symbol.observable` method, then it returns the result of invoking that method. If the resulting object is not an instance of Observable, then it is wrapped in an Observable which will delegate subscription.
- Otherwise, the argument is assumed to be an iterable and the iteration values are delivered synchronously when `subscribe` is called.

Converting from an object which supports `Symbol.observable` to an Observable:

```
1 Observable.from({
2   [Symbol.observable]() {
3     return new Observable(observer => {
4       setTimeout(() => {
5         observer.next("hello");
6         observer.next("world");
7         observer.complete();
8       }, 2000);
9     });
10  }
11 }).subscribe({
12   next(value) {
13     console.log(value);
14   }
15 });
16
17 /*
18 > "hello"
19 > "world"
20 */
21
22 let observable = new Observable(observer => {});
23 Observable.from(observable) === observable; // true
```

Converting from an iterable to an Observable:

```
1 Observable.from(["mercury", "venus", "earth"]).subscribe({
2   next(value) {
3     console.log(value);
4   }
5 });
6
7 /*
8 > "mercury"
9 > "venus"
10 > "earth"
```

```
11 */
```

Observer An Observer is used to receive data from an Observable, and is supplied as an argument to **subscribe**.

All methods are optional.

```
1 interface Observer {
2
3     // Receives the subscription object when `subscribe` is called
4     start(subscription : Subscription);
5
6     // Receives the next value in the sequence
7     next(value);
8
9     // Receives the sequence error
10    error(errorValue);
11
12    // Receives a completion notification
13    complete();
14 }
```

SubscriptionObserver A SubscriptionObserver is a normalized Observer which wraps the observer object supplied to **subscribe**.

```
1 interface SubscriptionObserver {
2
3     // Sends the next value in the sequence
4     next(value);
5
6     // Sends the sequence error
7     error(errorValue);
8
9     // Sends the completion notification
10    complete();
11
12    // A boolean value indicating whether the subscription is closed
13    get closed() : Boolean;
14 }
```