Intro to RxJS

Angelo Rendina

Push/Pull systems

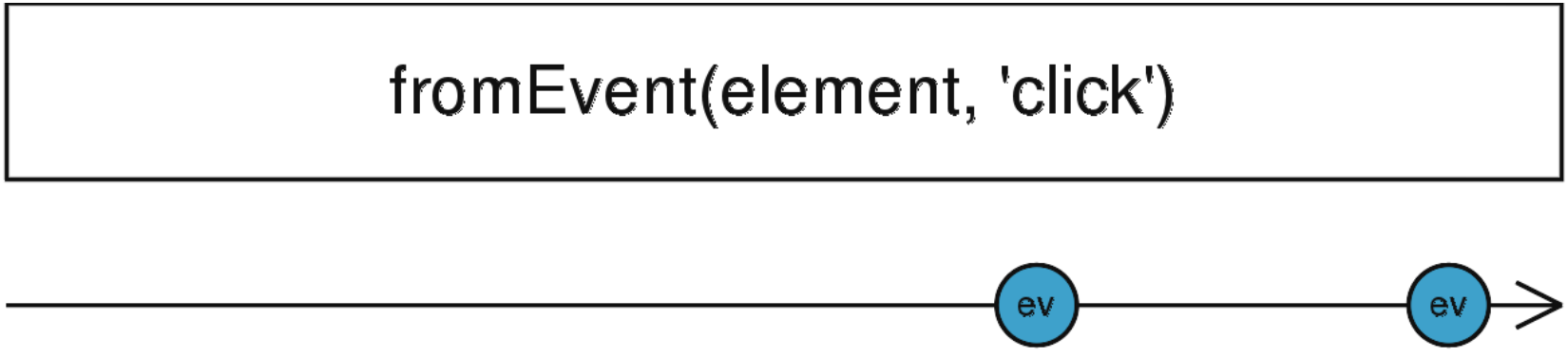
In a *pull system*, the Consumer gets data on request from the Producer; e.g. invoking a function.
In a *push system*, the Consumer is sent data from the Producer (once ready); e.g. waiting for a Promise to resolve.
In both examples above, only one result is returned.

Observables

A (RxJS) *Observable* is a push system that can return an arbitrary number of results, both synchronously and asynchronously.
Similarly to registering a *then* callback to a Promise, we need to *subscribe* a callback to an Observable in order to receive data.

Marble diagrams

We can represent the behaviour of an Observable with a marble diagram: the horizontal line is the flow of time, and each marble is the emission of a value.
In the picture, we represent the *fromEvent* Observable, which emits the event itself each time it is raised.



Operators

A (RxJS) *Operator* transforms an Observable into another Observable by modifying its emission.
As an example, the *map* Operator acts on Observables by applying the provided map to the emitted values. For instance, the Operator

$$\mathrm{map}(x \Rightarrow 2x)$$

would transform an Observable which emits the value 5 every second into one that emits the value 10 every second.

Piping

Formally, if Θ denotes an Observable and ϕ, ψ, χ are Operators, we would write

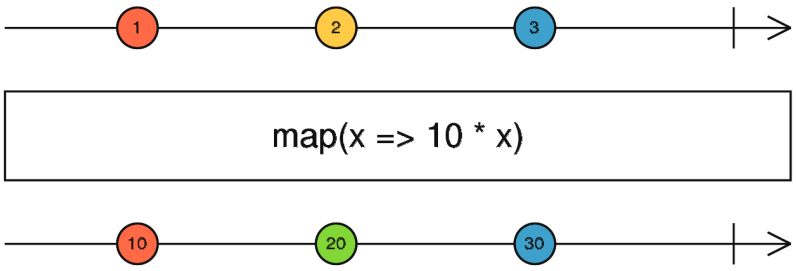
$$\phi\psi\chi\Theta$$

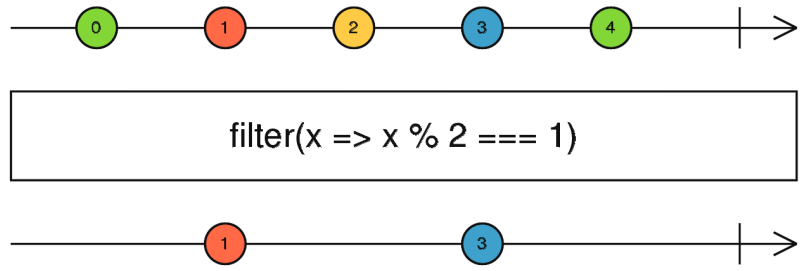
for the Observable obtained by sequentially applying χ, ψ and then ϕ (in this order!).
This is hard to read when the Operators are more than one character long. We always prefer the *pipe* notation, as follows:

$$\Theta.\mathsf{pipe}(\chi, \psi, \phi)$$

Marble diagrams and Operators

Marble diagrams are useful to visualise the action of an Operator. In the following pictures, the first line is the original Observable and the second line is the result of applying the Operator to it.



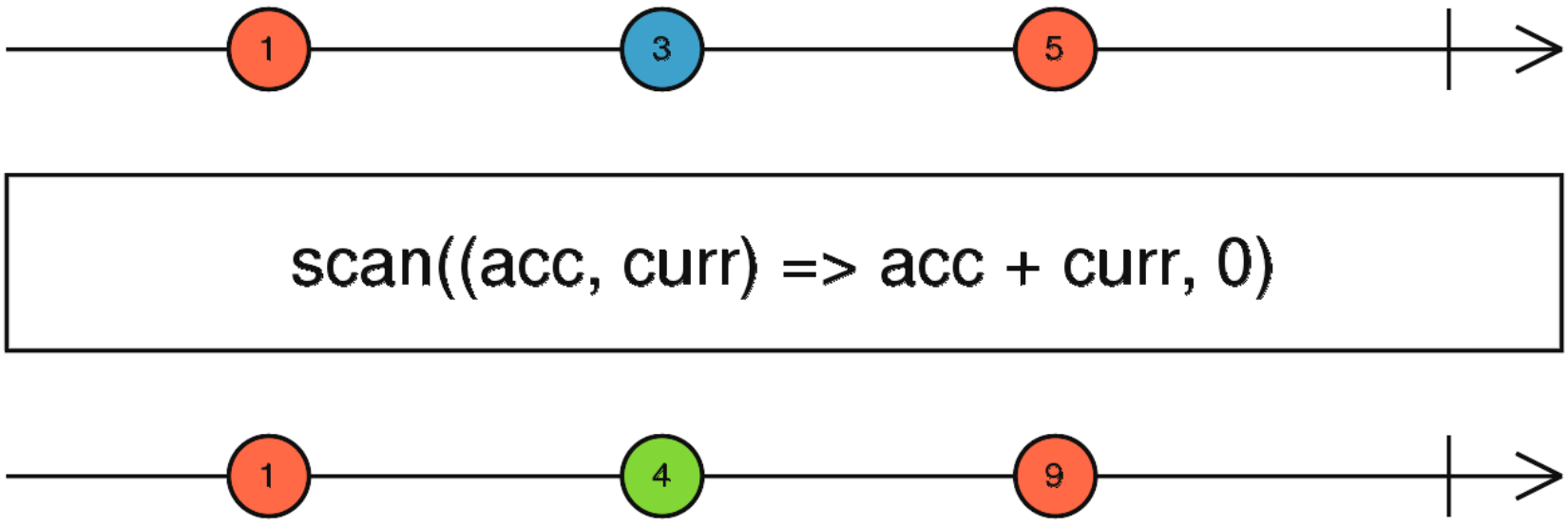


The Scan Operator

(The implementations of) Operators need not be pure functions: they can store any data in closures.
The main example of this is the Scan operator

$$\text{scan}((y, x) \Rightarrow f(y, x), y_0)$$

which transforms the emitted value x into $f(y, x)$, where y is an internally stored accumulator, primed as y_0 and updated to $f(y, x)$ at each emission.



Observers

An (RxjS) Observer is any object exposing the methods

1. next(any);
2. complete();
3. error(any).

An Observable is constructed by providing a function taking Observer as its only input.
Every time we subscribe to an Observable, this function is run and we receive every next, complete or error emission.

Unsubscribing

Every subscription returns a handle, that can be used to unsubscribe from the Observable.
When instantiating a custom Observable, we need to return a function named *unsubscribe* to comply with this interface (and dispose of any persistent resource, e.g. intervals or timeouts).

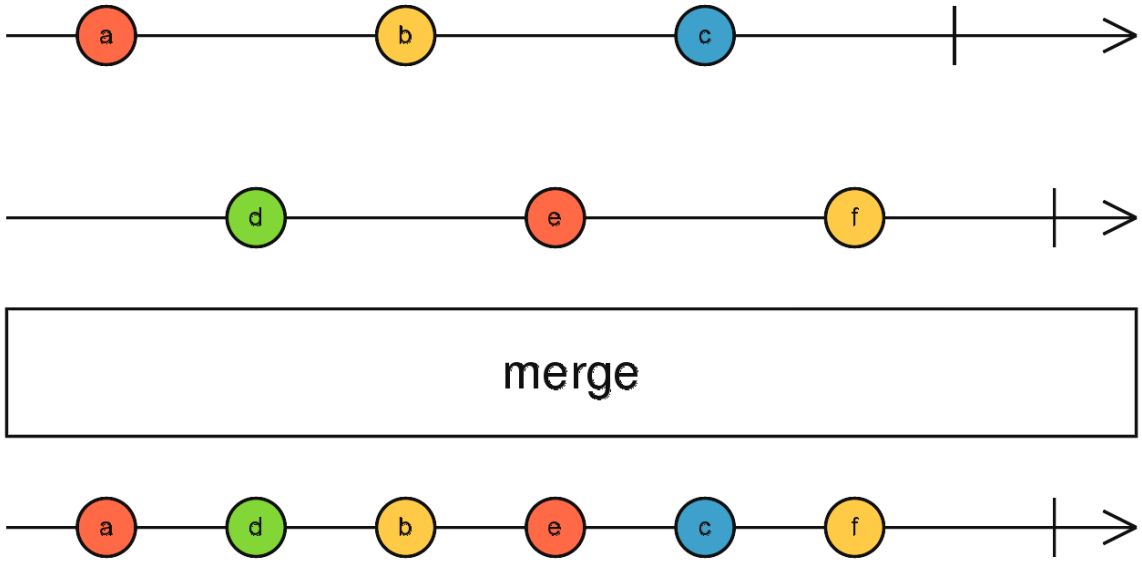
Subjects

A (RxJS) *Subject* is an object that behaves as both Observable and Observer.
It acts as a multicast event emitter: when it emits a value, this is broadcasted to each of its subscribers.
It exposes the *next* method, which emits the provided value.

Observers lifecycle

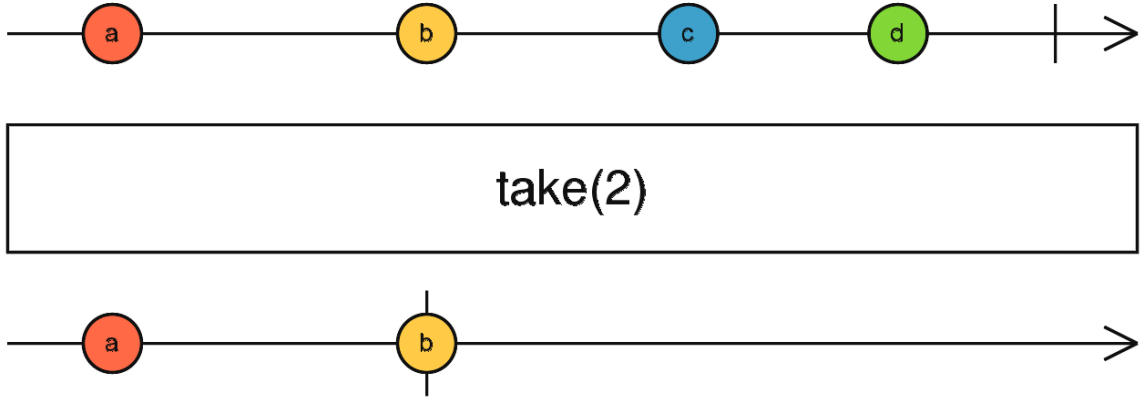
Calling *next* of an Observer emits the given value.
Calling *complete* does not emit a value, but puts the Observer in a *completed* state. Any further *next* invocation is ignored.
Calling *error* raises the argument as an Error, and puts the Observer in an *error* state. Any further *next* invocation is ignored.
In both cases, no unsubscription takes place. The callbacks are still subscribed to the Observable: it's the Observable itself that does not emit further events.

More marble diagrams

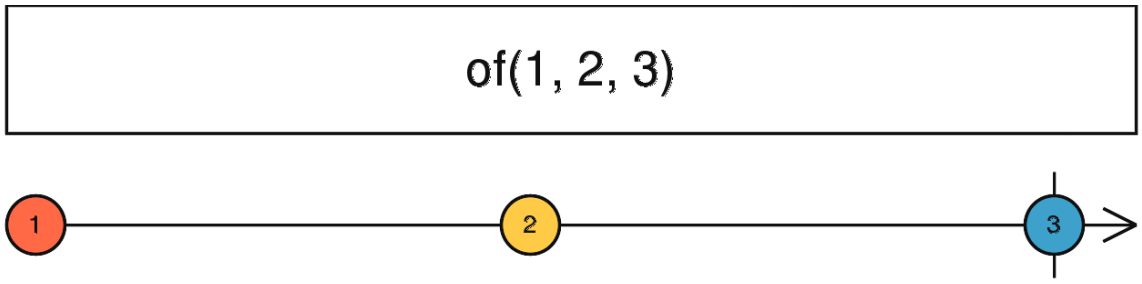


The *merge* operator creates an Observable that emits all the events of its operands, and completes after all do.

The *take*(*n*) operator creates an Observable that emits the first *n* values and immediately completes.



More marble diagrams



The *of* operator creates an Observable that immediately emits its parameters in sequence and completes.

The *switchMap*(*f*) operator creates an Observable with the same behaviour as the Observable returned by *f*, each time the source emits, ignoring completes and errors.