Observables

BUILDING WEB APPLICATIONS USING ANGULAR

Observables

- A few patterns have evolved over time for dealing with asynchronous data in JavaScript including Promises, Events and Callbacks - Observables are another pattern
- Where as Promises work very well for the fetching of a single piece of asynchronous data, reacting to that piece when it arrives, it does not work so well in certain circumstances such as repetitive requests
- Angular works with Observables built in, but often we need to extend this behaviour and to do so we can use Reactive Extensions for JavaScript (RxJS)

Observables / Observers / Disposables

- These are the core concepts of RxJS
- Observables are representations of a data source that can be observed. Observables are subscribed to by Observers which returns a Subscription, whose primary function is to unsubscribe when required.
- The Observable will call one of three publication events on a subscribed observer as appropriate
  - onNext – when the Observable has new data available
  - onError – when the Observable encounters an error
  - onCompleted – when the Observable has finished sending data

Creating an Observable

- To create an Observable in Angular we need to import some symbols

```
import { Observable } from 'rxjs/Observable';
import { Subscription } from 'rxjs/Subscription';
```

- From there it is fairly simple to create an Observable

```
observable: Observable<string> = Observable.create((observer) => {
   setInterval(() => {
      observer.next('hello world!');
   },1000)
})
```

Creating an Observable

- For an Observable to be useful, you have to subscribe to it!

```
let sub:Subscription = this.observable.subscribe(
   message => console.log(message),
   error => console.error(error),
   ()=>console.log('All done')
);
```

- Note we have passed in three callback functions for each call to onNext(), onError() and onCompleted()
- We have to be careful to unsubscribe from the Observable to not create memory leaks

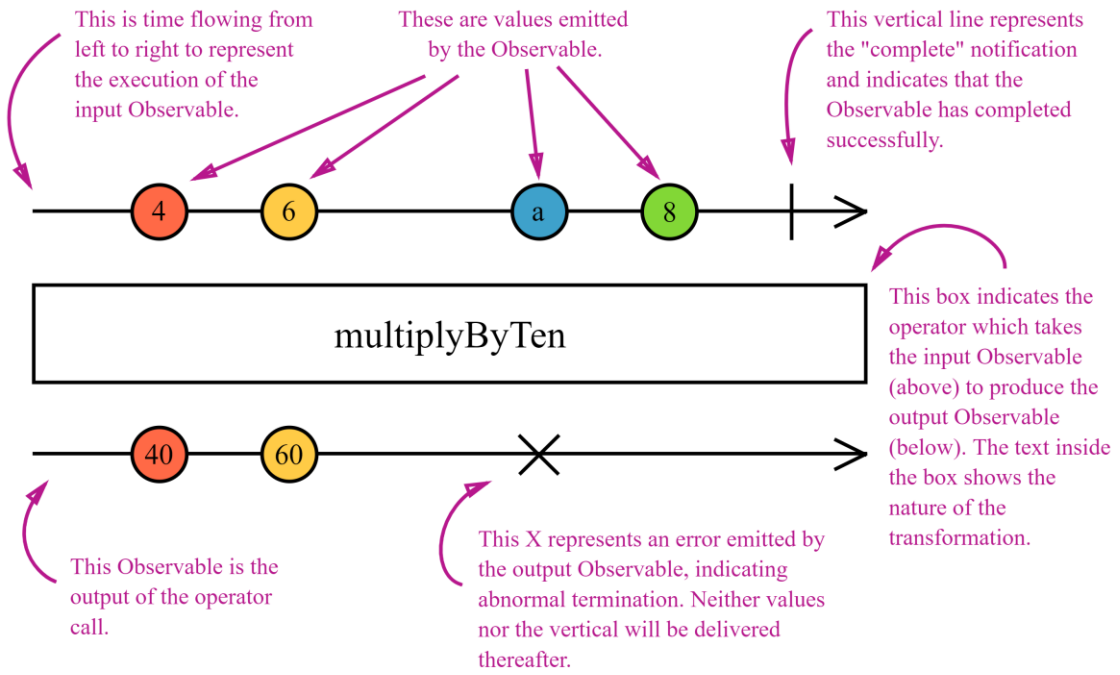
```
setTimeout(()=>{
   console.log("unsubscribing");
   sub.unsubscribe();
},10000)
```

Operators

- Setting up an Observable can be fairly straightforward, and most of the time we simply use Observables provided to us
- Much of the power of using Observables comes from their operators, most of which are not included by default so we need to import
- Operators are methods, pure functions actually, that enable us to deal with Observables in the functional programming style.
  - Being a pure function, they do not change the existing Observable – they return a wholly new Observable whose subscription logic is based on the input Observable
  - Subscribing to the output observable also subscribes to the input observable

Marble Diagrams

- Marble diagrams provide a visual queue as to how operators work
- Given many operators are in some way related to time, text descriptions are often insufficient and confusing



Lettable Operators

- RxJS 5.5 introduced ‘pipeable’ operators that differ in their use from pre-5.5 “patch” operators

```
//pipeable operators
import { switchMap } from 'rxjs/operators';
```

```
//patch operators
Import ‘rxjs/add/operator/switchMap’;
```

- This improved the use of operators by not ‘patching’ the prototype, meaning:
  - No blind dependencies (if you need it, you must import it!)
  - Operators become ‘tree shakable’
  - Linters can easily detect unused operators
  - You can easily to build your own operators

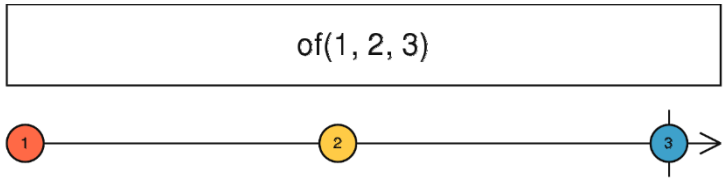
Pipeable Operators

- Patch operators were chained together using dot notation, let operators need to use the pipe function

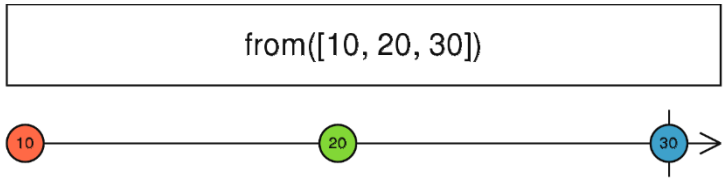
```
inputObservable.pipe(
        operator1(),
        operator2(),
        operator3(),
        …
).subscribe(outputValues=>console.log(outputValues))
```

Operators – Of and From

- RxJS comes with numerous 'Creation Operators' two popular ones are Of and From which allow us to create Observables from values.
- **Of** will create an Observable that emits the arguments you provide and then completes



- **From** will create an Observable from "almost anything"

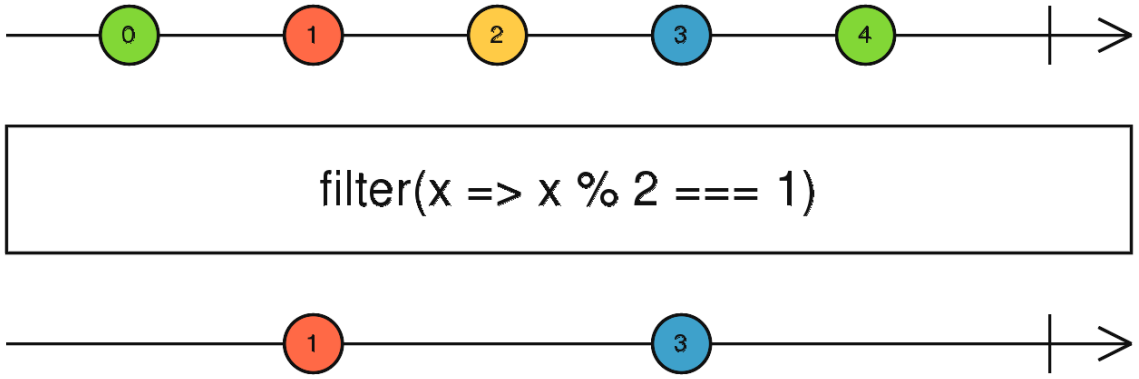


Operators – Of and From

- Other popular creation operators:
  - **fromEvent** – creates an Observable that is linked to DOM Events, Node EventEmitters of similar
  - **Interval** – creates an Observable that emits an ever-increasing integer at a specified interval
  - **Create** – creates an Observable that does what you tell it to
  - **Range** – creates an Observable that emits a range of sequential integers

Operators - Filter

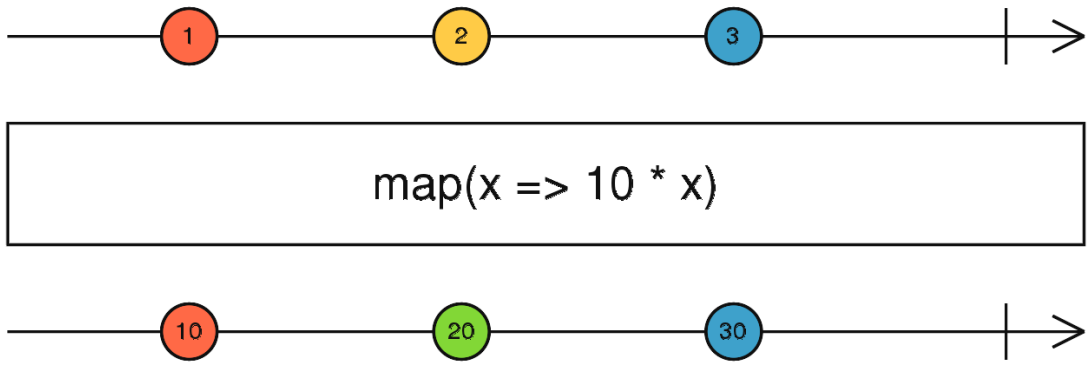
- The Operator equivalent to Array.prototype.filter this Operator takes values emitted by the input Observable and only emits them if they fulfil the criteria provided by the predicate function




```
inputObservable.pipe(
    filter(inputValue=>inputValue%2 === 0)
).subscribe(outputValue=>console.log(outputValue)); //Only even numbers
```

Operators - Map

- The map operator applies a transformation function to each emitted value from the input observable and emits the resulting value from the output observable
- An Observable version of Array.prototype.map()



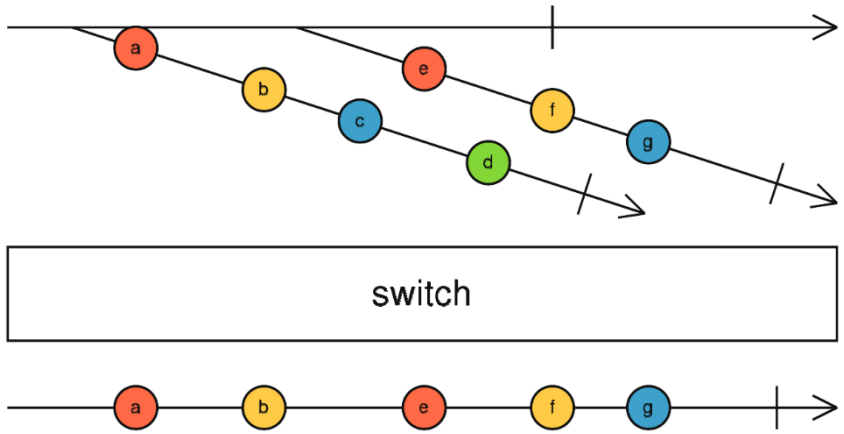

```
inputObservable.pipe(
    map(()=>return `always something special`)
).subscribe(outputValue=>console.log(outputValue)); //always something special
```

Operators - Switch

- Switch is an example of a combination operator
- It takes a higher-order observable (an observable of observables) and emits only the values from the latest emitted observable
- That is to say, it emits the values emitted by the first observable emitted, and then **switches** to emitting the values emitted by the next emitted observable ad infinitum

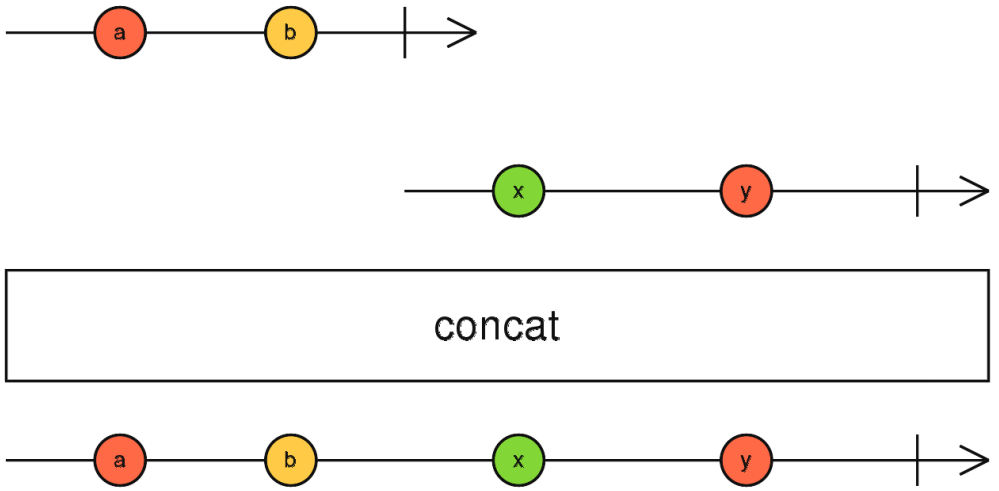
```
inputObservable.pipe(
    map((evt: MouseEvent)=>timer(1000,1000)),
    switchAll()
).subscribe(outputValue=>console.log(outputValue));
```

- Each time the above inputObservable emits, the timer restarts



Operators – Concat

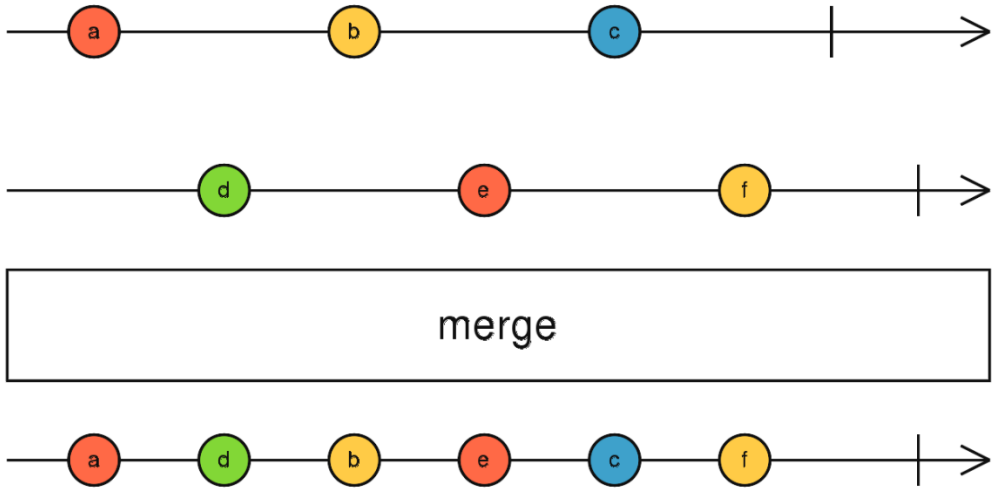
- Concat outputs an Observable that emits from the first stream until it completes, then emits values from the next stream




```
firstObservable.pipe(
    concat(secondObservable)
).subscribe((value: number)=>console.log(value));
```

Operators – Merge

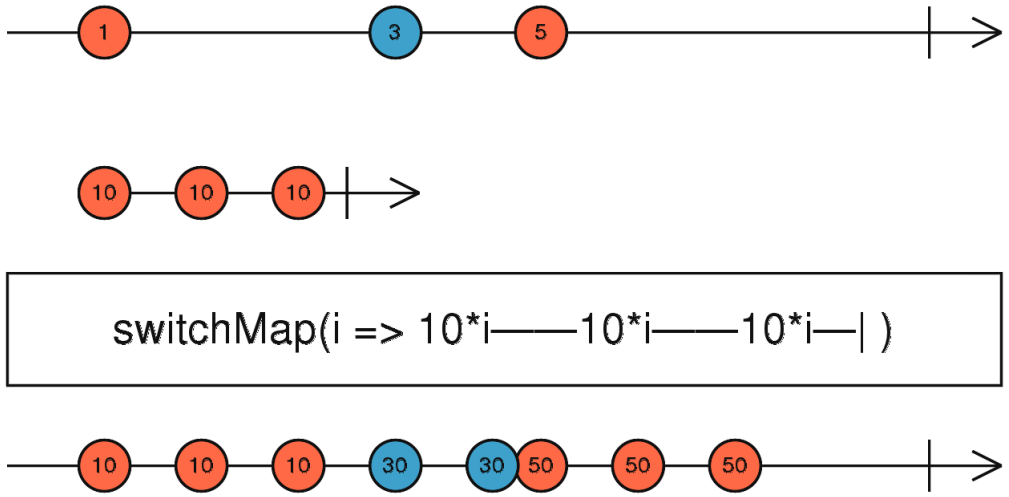
- Merge outputs one Observable that emits from whatever stream is currently active




```
firstObservable.pipe(
    merge(secondObservable)
).subscribe((value: number)=>console.log(value));
```

Operators - SwitchMap

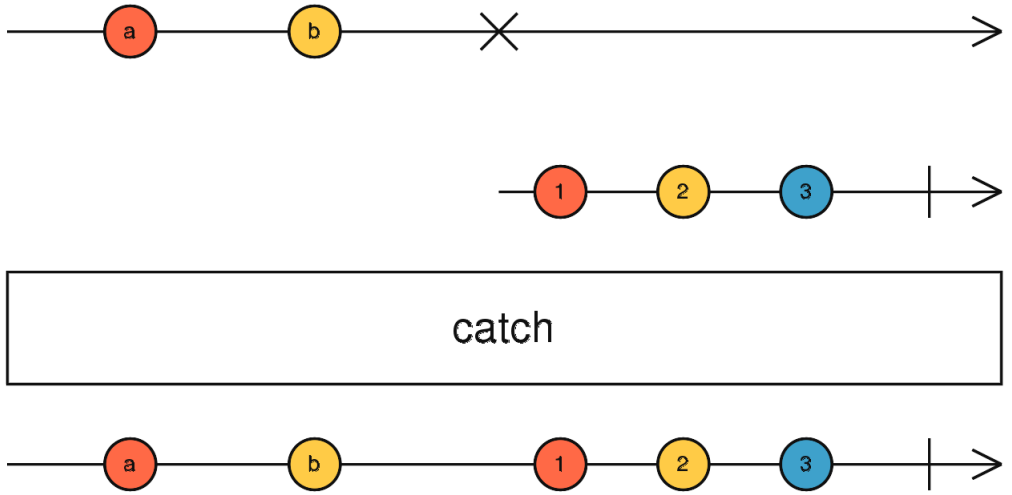
- Takes an **input** Observable and **maps** each emitted value to an **inner** Observable. The inner observable is then merged into the **output** Observable. Should the input Observable emit a new value the output Observable will **switch** to emit the new inner Observable.




```
input.pipe(
    switchMap(inputValue=>getJSON(inputValue))
)
.subscribe(ouputValue=>console.log(outputValue));
```

Operators - catchError

- Catches errors on the input Observable which needs to be handled by returning a new Observable or throwing an Error




```
input.pipe(
    catchError(err=>of(-1))
).subscribe(ouputValue=>{
          if (outputValue === -1) {
                    //handle error
          }
});
```

Exercise

- Using observables and any appropriate operators you feel are necessary create a digital clock that ticks every second and stops after 10 seconds.