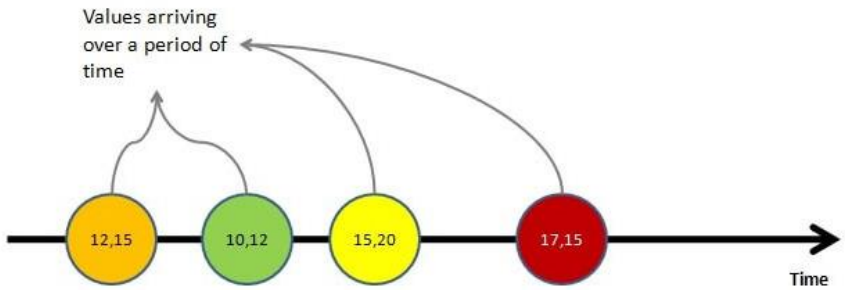
9

Reactive programming with RxJS

Reactive programming, RxJS, Observables and Subjects, Operators

Reactive programming

- Design paradigm that relies on asynchronous programming logic
- Reactive programming is all about streams
  - Time-ordered sequence of related messages
- Responds to events rather asking for data
- Combination of ”observer” and ”handler” functions



Reactive programming

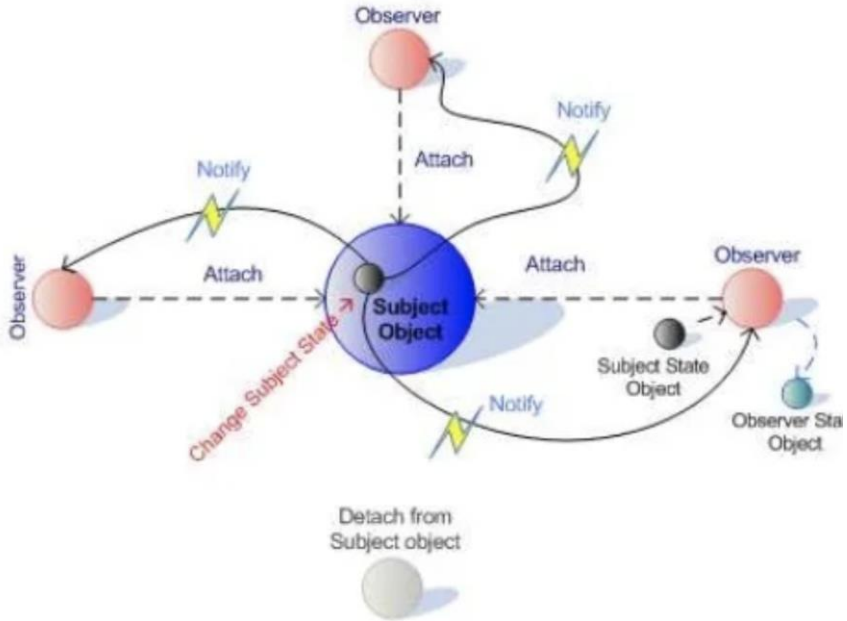
Angular is a reactive system

- Responds to events and propagate changes
- Highly relies on **observer** pattern
- Use the **RxJS** library (Reactive eXtensitions for JS)
  - A library for reactive programming
  - Compose asynchronous or callback based code

Observer design pattern

- Observer design pattern is behavioral patten
- Used to assure consitency between objects
- Define one-to-many dependecy between objects so that when object changes state, all its dependents are notified (or updated)
- The key objects of in this pattern
  - **Subject:** object holding a value and the list of it's dependent
  - **Observer:** the dependent object, is notified by the subject

Observer design pattern



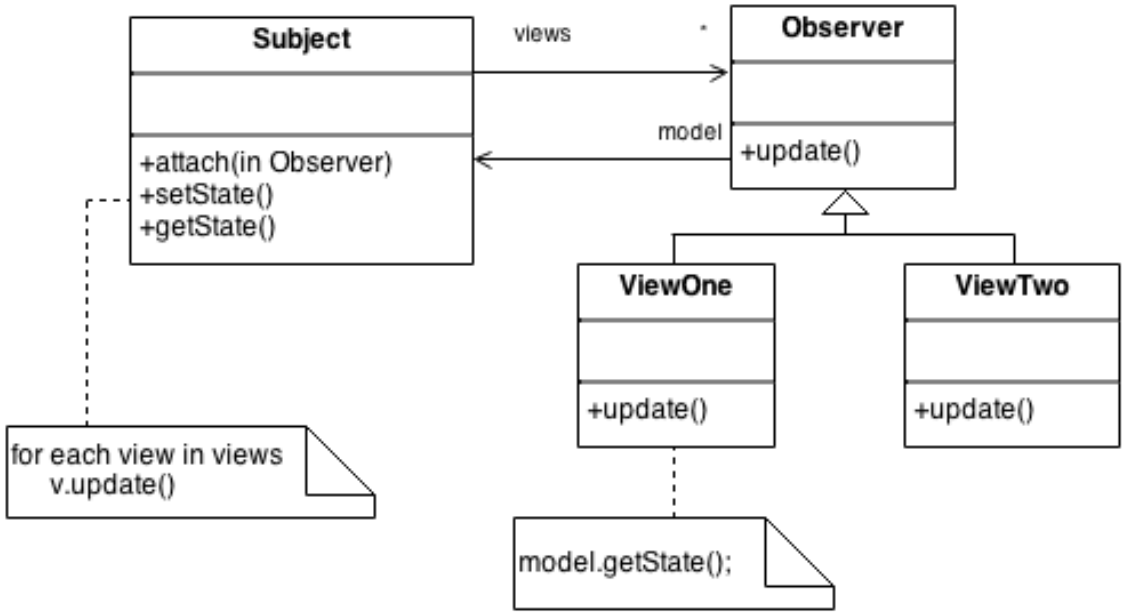
1. Observer is **attached** to subject

2. When subject state is changed, it **notifies** the attached observers

3. When observer no longer intersted, it **detach** from subject

Observer design pattern

Diagram



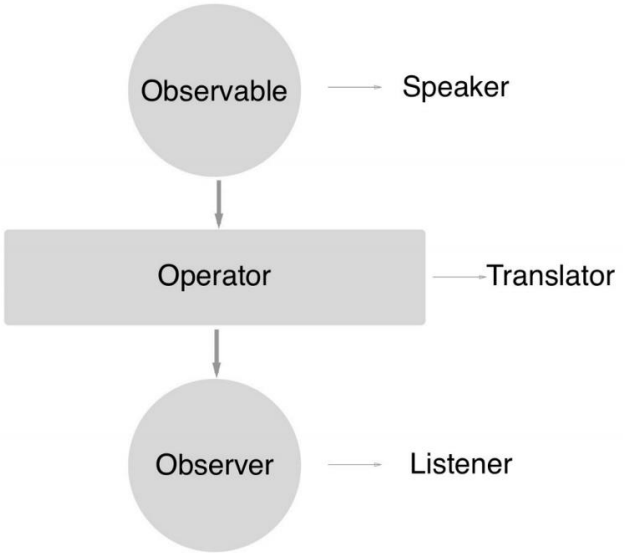
RxJS library

Reactive eXtensions for JavaScript

- Create and manipulate streams of events and data
- Provides an implementation of observer pattern
- "**Observable**" type is the core of RxJS
- Provides utility functions called (**Operators**) for:
  - Mapping
  - Transforming
  - Filtering
  - Composing
- Part of **ReactiveX** libraries: RxJava, Rx.NET, RxPy, ...

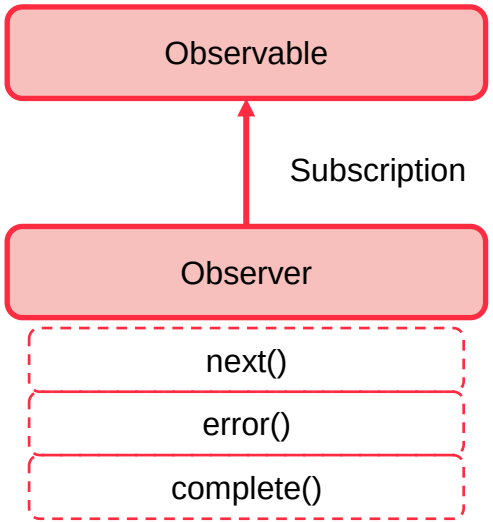
Observable

- Part of RxJS primitive type
- Stream of asynchrounous values pushed to the observer
- An observer can **subscribe** to an observable
- An **operator** can be applied to transform data before reaching observers
- **UNICAST** : each subscribed Observer owns an independent execution of the Observable



Observer

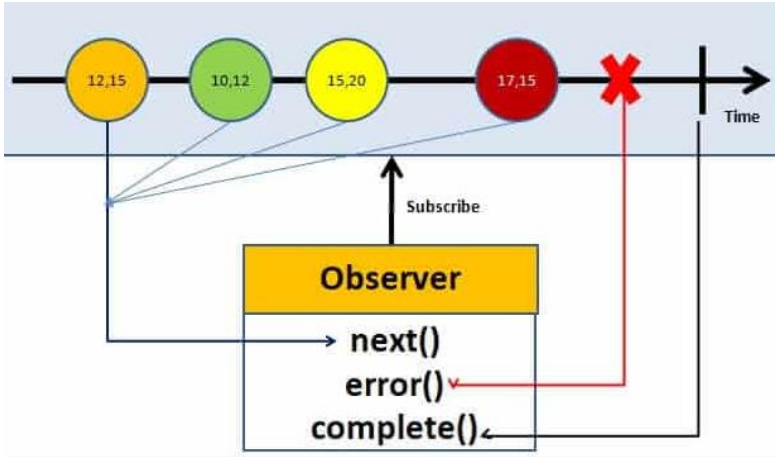
- An object which **subscribes** to observable changes
- It has three methods:
  - `next()`: called when observable emit the next value
  - `error()`: called when an error is thrown
  - `complete()`: called when observable complete (finish)



Observer (Example)

```
//Create an observable
var observable = of(1,2,3)

//subscribe to observable
observable.subscribe({
   next: (value)=>{
      console.log(value)
   },
   error: (err)=>{
     console.log(err)
   },
   complete: () => {
     //Do something when completed
   }
})
```



Observable vs Promise

- **Observable**
  - Can handle multiple values
  - Use .subscribe()
  - Can be disposed/cancelled
  - Can be chained using operators
  - next, error, completed
- **Promise**
  - One time use
  - Use .then()
  - Cannot be disposed
  - Cannot be chained
  - resolve, reject

Observable vs Promise

- Observables can deal better with situations like
  - Avoid doing things multiple times
  - Out-of-order responses
  - Time-based logic
  - Cancellation

- You can convert an observable to promise by calling **.toPromise()**
- You can create an observable from a promise by calling **.fromPromise()**

- In Angular, HTTP calls returns **Observables** and NOT promises

RxJS types: Subject

- Subject extends from the base class Observable
- Can **multicast** to many observers
- Keeps a registry of many listeners (observers)
- Every subject is both Observable and Observer
  - `next()`: feed data to registered observers
  - `error()`: throw an error
  - `complete()`: close the subject

RxJS types: Subject

```
const subject = new Subject<number>();

subject.subscribe({
   next: (v) => console.log(`observerA: ${v}`),
});

subject.next(1);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});

subject.next(2);

// Logs:
// observerA: 1
// observerA: 2
// observerB: 2
```

RxJS types: BehaviorSubject

- Variant of Subject, extends Subject class
- Stores the latest value emitted
- When an observer subscribes, it will immediately receive the latest emitted value (the current value)
- Can be initialized with an initial value before any subscription

RxJS types: BehaviorSubject

```
const subject = new BehaviorSubject<number>(0);

subject.subscribe({
next: (v) => console.log(`observerA: ${v}`),
});

subject.next(1);

subject.subscribe({
next: (v) => console.log(`observerB: ${v}`),
});

subject.next(2);

// Logs:
// observerA: 0
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

Observable, Subject, BehaviorSubject

Observable

- Creates copy of data
- Unicast
- Uni-directional
- Observers get upcoming events only

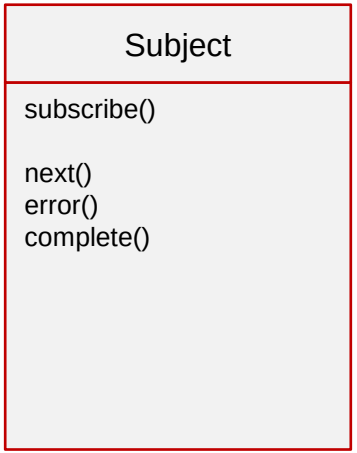
Subject

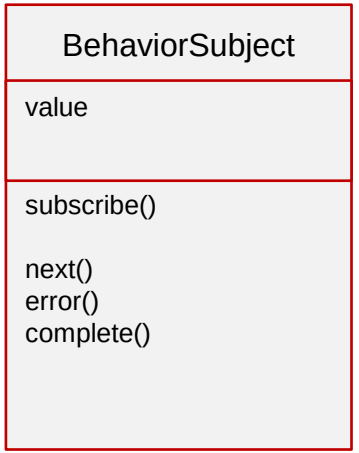
- Shared data
- Multicast
- Bi-directional
- Observers get upcoming events only

BehaviorSubject

- Shared data
- Multicast
- Bi-directional
- Observer get previous and upcoming events
- Set initial value

Observable, Subject, BehaviorSubject





RxJS Operators

- Operators are functions that operate on observable and return a new observable
- RxJS operators allows for complex asynchronous code to be easily composed
- To chain operators we use the method .**pipe()** on observables
- Most used operators : **map, filter, delay, merge, concat, catchError**

RxJS Operators: map

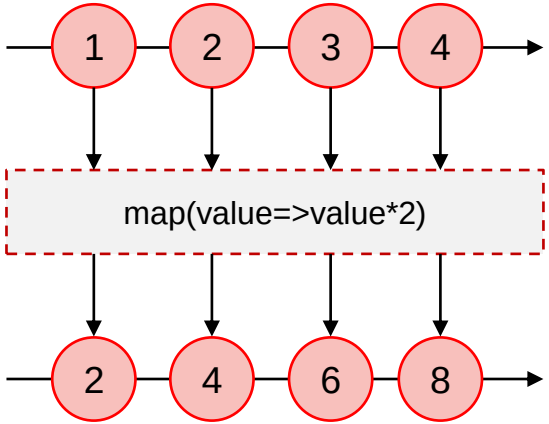
- Apply projection with each value from source observable

Example:

```
let observable = of(1,2,3,4);

observable.pipe(
   map(value=>{
      return value*2
   })
).subscribe(value=>{
   console.log("Value:" value)
})

//logs
//Value: 2
//Value: 4
//Value: 6
//Value: 8
```



RxJS Operators: filter

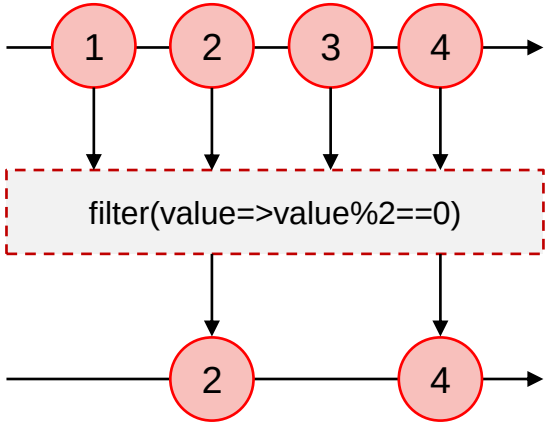
- Emit only values that pass a given condition

Example:

```
let observable = of(1,2,3,4);

observable.pipe(
   filter(value=>{
      return value%2==0
   })
).subscribe(value=>{
   console.log("Value:" value)
})

//logs
//Value: 2
//Value: 4
```



RxJS Operators: delay

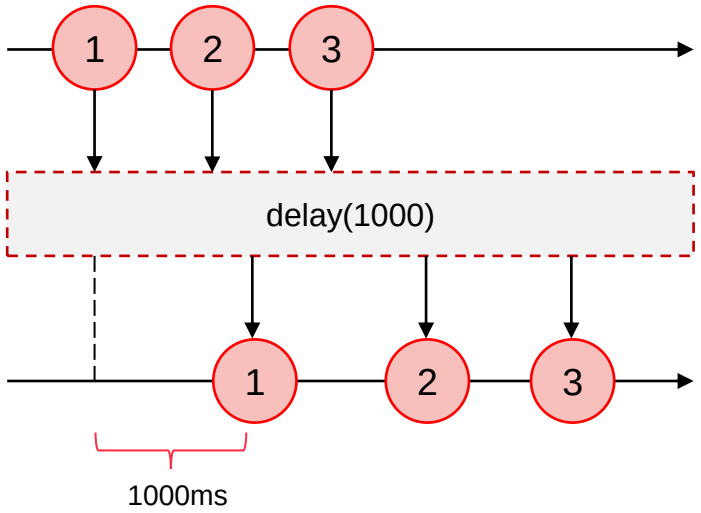
- Delay emitted values by a given time

Example:

```
let observable = of(1,2,3);

observable.pipe(
   delay(1000)
).subscribe(value=>{
   console.log("Value:" value)
})

//logs (After 1second)
//Value: 1
//Value: 2
//Value: 3
```

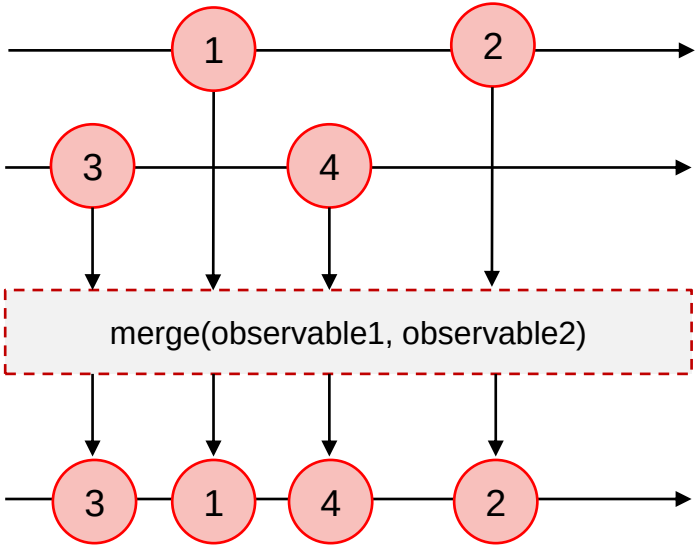


RxJS Operators: merge

- Turn multiple observables into one
  - Order is **NOT** **important**

Example:

```
merge(observable1, observable2).subscribe(value=>{
            console.log("Value:" value)
})
```

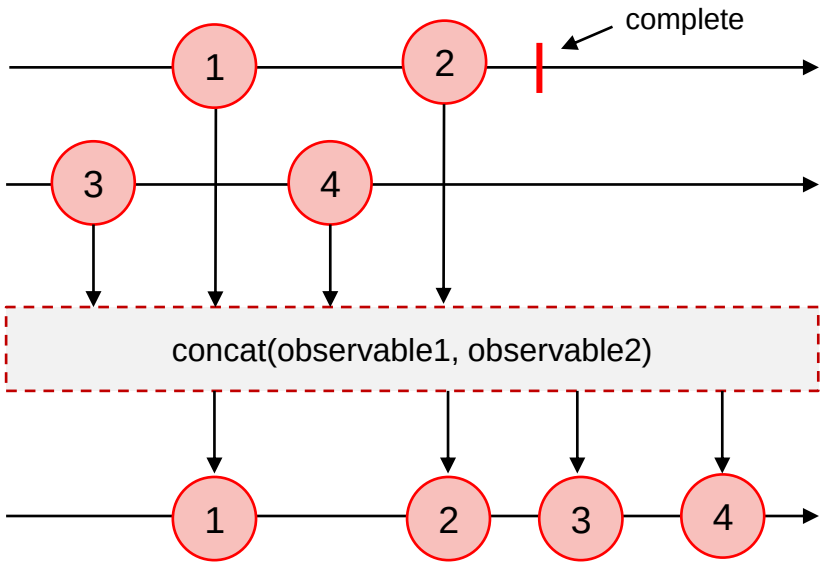


RxJS Operators: concat

- Turn multiple observables into one
  - Order is **important**

Example:

```
concat(observable1, observable2)
.subscribe(value=>{
            console.log("Value:" value)
})
```

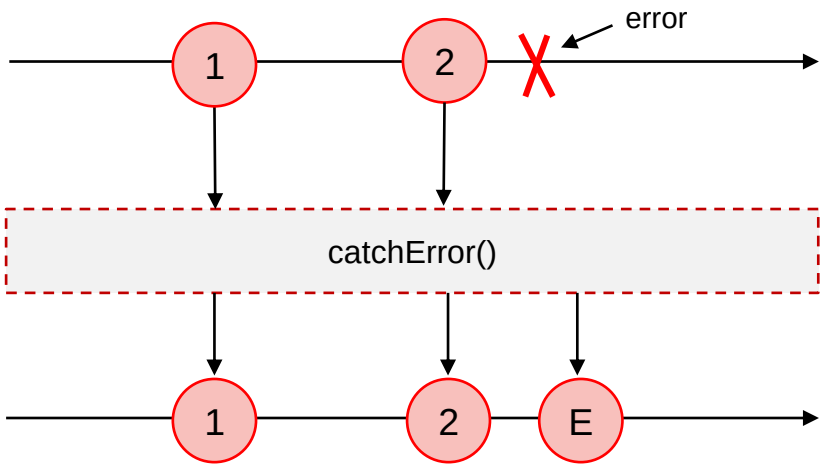


RxJS Operators: catchError

- Handle errors in an observable sequence
  - Must return an observable

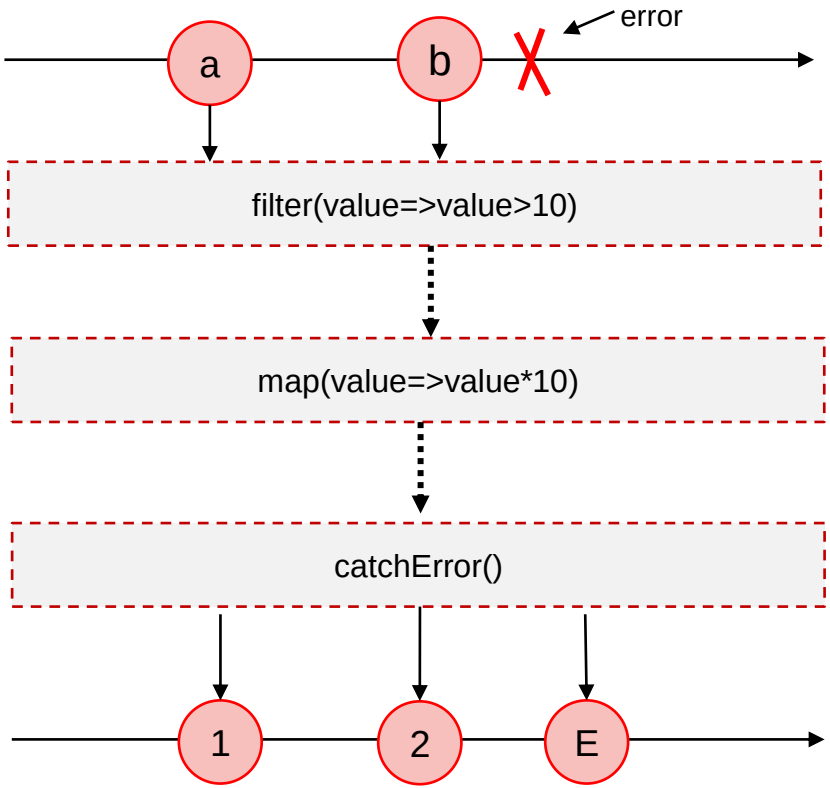
Example:

```
observable.pipe(
    catchError(err=>{
        return of("Error happened")
    })
).subscribe(value=>{
    console.log("Value:" value)
})
```



Chaining RxJS Operators

```
of(10,20,30).pipe(
    filter(value=>value>10),
    map(value=>value*10),
    …,
    catchError(err=>{
        return of("Error happened")
    })
).subscribe(value=>{
    console.log("Value:" value)
})
```



Other RxJS Operators

- **Combination:** concat, merge, withLatestFrom, combineAll
- **Conditional:** every, iif
- **Creation:** of, from
- **Filtering:** debounceTime, filter, find, first, last, take, takeUntil
- **Transformation:** map, reduce, concatMap
- **Error handling**: catch, catchError, retry, retryWhen
- **Utility**: tap, delay, finalize

LAB 9

Work with observables