Angular makes use of observables as an interface to handle a variety of common asynchronous operations. For example:

* The HTTP module uses observables to handle AJAX requests and responses
* The Router and Forms modules use observables to listen for and respond to user-input events

**What is Data Stream?**

* [Reactive programming (Rx)](https://gist.github.com/staltz/868e7e9bc2a7b8c1f754) is programming with asynchronous data streams.
* A **stream** is a data that arrives over a period of time.
* You are able to create data streams of anything, not just from click and hover events.
* Streams are cheap and **ubiquitous(existing or being everywhere at the same time)**, anything can be a stream: variables, user inputs, properties, caches, data structures, etc.
* A stream can be an input to another. Multiple streams can also be inputs to another stream. Two streams can be merged. You can filter a stream to get one that has only those events you are interested in. You can map data values from one stream to a new one.
* Stream can emit three different things: a **value**(of some type), an **error,**or a “**completed**” signal.
* We capture emitted events only asynchronously, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when ‘completed’ is emitted.
* The “listening” to the stream is called **subscribing**. The functions we are defining are **observers**. The stream is the subject (or “**observable**”) being observed.

**Note about Streams:**

* emit zero, one or more values of any time.
* can also emit errors.
* must emit the complete signal, when completed (finite streams).
* can be infinite, meaning they never complete

**Reactive Programming:**

* Reactive programming is all about creating the stream, emitting value, error or complete signals, manipulate, transfer or do something useful with the data streams.

**What is RxJS**

* The [RxJS](https://rxjs.dev/guide/overview" \t "_blank) (Reactive Extensions Library for JavaScript) is a javascript library, that allows us to work with asynchronous data streams.
* Angular uses the RxJS library heavily in its framework to implement Reactive Programming.
* Some of the examples where reactive programming used are:
* Reacting to an HTTP request in Angular
* Value changes / Status Changes in Angular Forms
* The Router and Forms modules use observables to listen for and respond to user-input events.
* You can define custom events that send observable output data from a child to a parent component.
* The HTTP module uses observables to handle AJAX requests and responses.
* The RxJs has two main players:

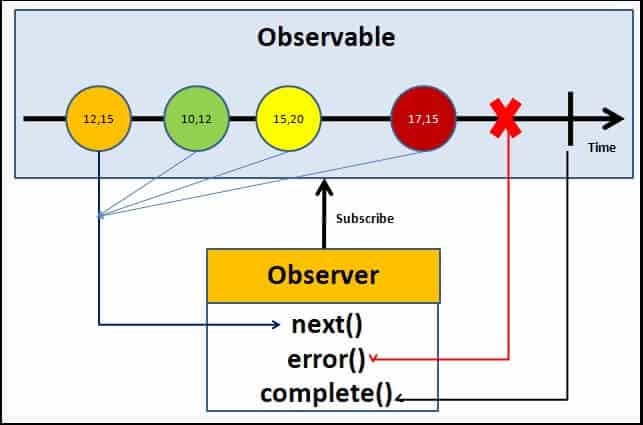
1. Observables
2. Observers(Subscribers)

# What is an Observable in Angular?

* An Observable is a lazily evaluated computation that can synchronously or asynchronously return zero to (potentially) infinite values from the time it’s invoked onwards.
* In other words, **Observable is a function that converts the** **ordinary stream of data** **into an** **observable stream of data**.
* Observable emits the **value from the stream** asynchronously. It emits the **complete** signals when the stream completes or an **error** signal if the stream errors out.
* Observables are declarative. You define an observable function just like any other variable. The observable starts to emit values only when **someone subscribes to it**. Subscribing to an Observable is analogous (similar) to calling a Function.

# Who are observers (subscribers)?

* The Observable on its own is useless unless someone consumes the value emitted by the observable. We call them observers or subscribers.
* The observers communicate with the Observable using callbacks.
* The observer must subscribe with the observable to receive the value from the observable. While subscribing, it optionally passes the three callbacks **next(), error() & complete()**.



* The observable starts emitting the value as soon as the observer or consumer subscribes to it.
* The observable invokes the next() callback whenever the value arrives in the stream. It passes the value as the argument to the next callback. If the error occurs, then the error() callback is invoked. It invokes the complete() callback when the stream completes.
* Observers/subscribers subscribe to Observables
* Observer registers three callbacks with the observable at the time of subscribing. i.e. next(), error() & complete()
* All three callbacks are optional
* The observer receives the data from the observer via the next() callback
* They also receive the errors and completion events from the Observable via the error() & complete() callbacks.

**Creating Observable:**

* There are few ways in which you can create observable in angular. Simplest is to use the Observable constructor. The observable constructor takes observer (or **subscriber**) as its argument. The subscriber will run when this observable’s subscribe() method executes.
* The following example creates an observable of a stream of numbers 1, 2, 3, 4, 5

export class HomeComponent implements OnInit {

  obs = new Observable((observer) => {

    console.log("Observable starts")

    observer.next("1")

    observer.next("2")

    observer.next("3")

    observer.next("4")

    observer.next("5")

    });

  constructor() { }

}

**Subscribing to an observable:**

* A subscribe call is simply a way to start an "Observable execution" and deliver values or events to an Observer of that execution. Subscribing to an Observable is like calling a function, providing callbacks where the data will be delivered to.

ngOnInit() {

    this.obs.subscribe(

      val => { console.log(val) }, //next callback

      error => { console.log("error") }, //error callback

      () => { console.log("Completed") } //complete callback

      )

  }

**Adding an interval:**

* We can add a timeout to insert a one second delay in each next() callback.

obs = new Observable(observer => {

console.log('Observable starts');

setTimeout(() => {

observer.next('1');

}, 1000);

setTimeout(() => {

observer.next('2');

}, 2000);

setTimeout(() => {

observer.next('3');

}, 3000);

setTimeout(() => {

observer.next('4');

}, 4000);

setTimeout(() => {

observer.next('5');

}, 5000);

});

**Unsubscribing from an Observable:**

* We need to unsubscribe to close the observable when we no longer require it. If not it may lead to memory leak & Performance degradation.
* To Unsubscribe from an observable, we need to call the Unsubscribe() method on the subscription. It will clean up all listeners and frees up the memory.
* To do that, first, create a variable to store the Subscription.

observer!:Subscription;

* Assign the subscription to the obs variable.

 ngOnInit() {

    this.observer=this.obs.subscribe(

      val => { console.log(val) }, //next callback

      error => { console.log("error") }, //error callback

      () => { console.log("Completed") } //complete callback

      )

}

* Call the unsubscribe() method in the [ngOnDestroy](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/" \t "_blank) method.

 ngOnDestroy(){

    this.observer.unsubscribe();

  }

**Building a Custom Observable:**

ngOnInit() {

   const customIntervalObservable=new Observable (obser=>

      {

        let count=0;

        setInterval(()=>{

          obser.next(count);

          count++;

        },1000)

      });

    customIntervalObservable.subscribe(data=>

      {

        console.log(data);

      });

  }

**Observable Operators:**

* The Operators are functions that operate on an Observable and return a new Observable.
* The power of observable comes from the [operators](https://angular.io/guide/rx-library#operators). You can use them to manipulate the incoming observable, filter it, merge it with another observable, alter the values or subscribe to another observable.
* You can also chain each operator one after the other using the pipe. Each operator in the chain gets the observable from the previous operator. It modifies it and creates a new observable, which becomes the input for the next observable.
* The following example shows the filter & map operators chained inside a pipe. The filter operator removes all data which is less than or equal to 2 and the map operator multiplies the value by 2.

The input stream is [1,2,3,4,5] , while the output is [6, 8, 10].

obs:Observable<number>=new Observable((observer)=>{

    observer.next(1);

    observer.next(2);

    observer.next(3);

    observer.next(4);

    observer.next(5);

    observer.complete();

  }

  );

  obs1=this.obs.pipe(filter(data=> data >2),

  map((val)=>{return val\*2})

  );

  constructor() { }

  ngOnInit() {

   this.observer=this.obs1.subscribe(

      val => { console.log(val) }, //next callback

      error => { console.log("error") }, //error callback

      () => { console.log("Completed") } //complete callback

      );

}

**Two communication protocols between data producer and data consumers:**

## Push model:

* In the push model, the most important is data producer. In this case, data producers decide when to send values to data consumers, and data consumers have no idea when data will come.
* This model is used in Promises, where the promise is a data producer, which is sending data to the callback. Callback doesn’t know when it will receive data, and it relay totally on the data producer.
* It’s similar to the Observables.

**Pull model:**

* Here, the most important is data consumer, and it decides when it wants to get data from the data producer. In this model, data producers have no decision power about delivering data.

**Subject:**

* The Subject is another type of Observable, and it allows value to be consumed by many Observers, not like in the normal Observable just by one. This means that Subjects are multicast, and Observables are unicast.
* Every Subject is an Observable, and it’s possible to subscribe to it, but the subscribe method doesn’t invoke the new execution. It just registers a new Observer to the list of Observers.
* Every Subject is an Observer, which means it has next, complete, and error methods. When you want to add new data to the Subject, you have to use the .next() method, then the value would be multicasted to all Observers.

ngOnInit() {

    const mySubject = new Subject();

mySubject.subscribe({

  next: (value) => console.log('First observer:' + value)

});

mySubject.subscribe({

  next: (value) => console.log('Second observer:' + value)

});

mySubject.next("hello");

mySubject.next(1);

mySubject.next(2);

mySubject.next("bye");

}