* who.when(what)
* button.addEventListener(“click”, function callback () {})
  + who : button
  + when : click
  + what : callback function

This style of coding is known as “Imperative programming”, where everything is specified in a single programming statement and in-particular order (*who-when-what*)

Add. Notes to refer:

PubSub pattern old way, mediator pattern is more suitable for Observable

-------------------------------------------------------------------------------------------

Observable **achieves/de-couples** same functionality by it splits who, when and what across multiple statements

It calls “Who” with a new name - Observable,

It calls “When” with a new name - Data/Emitted Data/Event/ Emitted Event

It calls “What” with a new name - is implemented by Observer(internally it has sub-methods each of which is called as a sub event handlers, **Next()** will always be there, and additionally can have **error(),** **complete()**.

-------------------------------------------------------------------------------------------

In Reactive Programming, all the three (who, what, when) can be specified in any order across different program statements separated by other program statements, spread across the code/components/business logic.

-------------------------------------------------------------------------------------------

**Use case#1:**

Every time a event happens, directly do not trigger the event handler, mediate by using an observable in between which has a subscribed observer which has a next method. Every time event happens you trigger observable.next, which trigger observer.next inside which you write event handler logic

**Observable.fromEvent(who, when).subscribe(**

**() => {**

**what;**

**}**

**);**

Though it appears as an imperative style of programming, but one should notice who and when may be imported from other file/service/component.

**Use case#2:**

Similarly run some code after some time elapses(imperative programming -> setTimeOut())

Who = implicit context(Javascript runtime context)

When = timer event

What = some code/logic

**Observable.timer(1000ms).subscribe(**

**() => { what; }**

**);**

**setTimeOut**

**Observable.timer(3000ms, 1000ms).subscribe(**

**() => { what; }**

**);**

**setInterval(1000ms) inside setTimeOut(3000ms) implementations**

**Observable.timer(3000ms, 1000ms, 2).subscribe(**

**() => { what; }**

**);**

**setInterval(1000ms) inside setTimeOut(3000ms) for 2 times implementations**

**Use case#3:**

**Observable.timer(3000ms, 1000ms).pluck(2).subscribe(**

**() => { what; }**

**);**

Different syntax for Use case#2 last operation.

**‘pluck’** is one type of RxJs operator

**Use case #4:**

**Subject** is a special type of Observable which is also an Observer.

**let x: Subject<any> = new Subject<any>();**

**x.subscribe(**

**(emittedData: any) => what;**

**);**

**x.next(‘JAi Pathala Bhairavi’);**

Non-chained, non-complicated where Observable and Observer are merged into a single entity called **‘Subject’**

-------------------------------------------------------------------------------------------

In Angular component.html template DOM, you can bind @Input(s) to not just data variables/ getter/setters, but also you can bind @Input(s) to Observable streams.

**<div [innerText]=”someDataVariable”></div>**

But if you bind @Input(s) in template DOM to Observable streams, you need to use additional syntax

**<div [innerText]=”someStreamVariable$ | async”></div>**