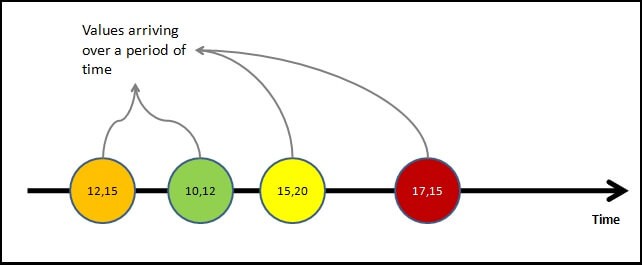
**What is a data stream?**

A **data stream** is the data that arrives over some time. The stream of data can be anything. Like variables, user inputs, properties, caches, data structures, and even failures, etc

Consider the example of a sequence of x and y positions of mouse click events. Assume that the user has clicked on the locations (12, 15), (10, 12), (15, 20), and (17, 15) in that order.

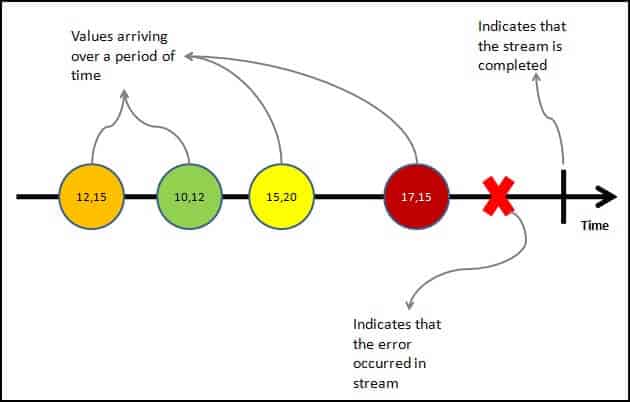
The following diagram shows how the values arrive over a period of time. As you can see, the stream emits the values as they happen, i.e., asynchronously.

mouse click events as data streams

Value is not the only thing that streams emit. The stream may complete as the user closes the window or app. Or an error may happen, resulting in the stream’s closure. At any point in time, the stream may emit the following three things.

**Value:** i.e., the next value in the stream  
**Complete**: The stream has ended  
**Error**: The error has stopped the stream.

The following diagram shows all three possibilities in a stream

mouse click events as data streams with emit error and complete events

As said earlier the stream of data can be anything. For Example

* Mouse click or Mouse hover events with x & y positions
* Keyboard events like keyup, keydown, keypress, etc
* Form events like value changes etc
* Data that arrives after an HTTP request
* User Notifications
* Measurements from any sensor

Important Points regarding streams can

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

Now we have understood what a data stream is, let us look at what is Reactive Programming is

**Reactive Programming**

Reactive programming is about creating the stream, emitting value, error, or complete signals, manipulating, transferring, or doing something useful with the data streams.

This is where the RxJs come into the picture.

[The introduction to Reactive Programming you’ve been missing](https://gist.github.com/staltz/868e7e9bc2a7b8c1f754) gives you a very nice introduction to Reactive Programming. Also, refer to [Introduction to Rx](http://introtorx.com/)

**What is RxJS**

The [RxJS](https://rxjs.dev/guide/overview) (Reactive Extensions Library for JavaScript) is a Javascript library that allows us to work with asynchronous data streams.

**Observable in Angular**

Angular uses the RxJS library heavily in its framework to implement Reactive Programming. Some of the examples where reactive programming is used are

* Reacting to an [HTTP request in Angular](https://www.tektutorialshub.com/angular/angular-httpclient/)
* [Value changes](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) / [Status Changes](https://www.tektutorialshub.com/angular/statuschanges-in-angular-forms/) 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. Observable
2. Observers ( Subscribers)

**What is an Observable in Angular**

Observable is a function that converts the **ordinary data stream** into an **observable one**. You can think of Observable as a wrapper around the **ordinary data stream**.

**An observable stream** or simple 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.

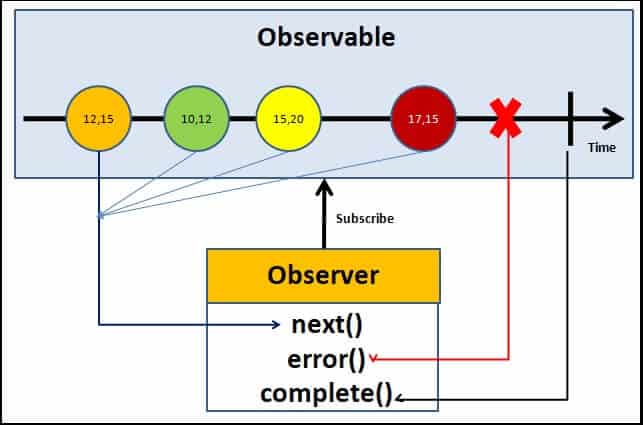
Observables are declarative. You define an observable function just like any other variable. The observable starts to emit values only when **someone subscribes to it**.

**Who are the observers (subscribers)**

The Observable is only useful if 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 to the observable to receive the value from the observable. While subscribing, it optionally passes the three callbacks. next(), error() & complete()

Angular Observable Tutorial on how observable and observers communicates with callbacks

The observable emits 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.
* The 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

Observables in Angular, powered by the RxJS library, provide a powerful mechanism for handling asynchronous operations and managing streams of data. They work on a publish-subscribe model, where an "Observable" emits values over time, and "Observers" (or subscribers) listen to and react to those values.

Here's how they work:

* **Observable Definition:** An Observable is created to define a source of values. This could be anything from HTTP requests, user input events (like clicks or form changes), or custom data streams. The Observable itself is "cold" by default, meaning it won't produce values until someone subscribes to it.

TypeScript

import { Observable } from 'rxjs';  
  
 const myObservable = new Observable(subscriber => {  
 *// Emit values asynchronously*  
 subscriber.next('First value');  
 setTimeout(() => {  
 subscriber.next('Second value after 1 second');  
 subscriber.complete(); *// Signal completion*  
 }, 1000);  
 });

* **Subscription:** To receive values from an Observable, an Observer must "subscribe" to it. The subscribe() method takes up to three callback functions:
  + next(value): Called for each emitted value.
  + error(err): Called if an error occurs.
  + complete(): Called when the Observable finishes emitting values.

TypeScript

myObservable.subscribe({  
 next: value => console.log(value),  
 error: err => console.error('Error:', err),  
 complete: () => console.log('Observable completed')  
 });

* **Value Emission:**

Once subscribed, the Observable's internal logic executes, and it starts emitting values to its subscribers using the next() method.

* **Error Handling and Completion:**

Observables can handle errors gracefully using the error() callback and signal completion using the complete() callback. After complete() or error() is called, the Observable will no longer emit values.

* **Unsubscription:**

It is crucial to unsubscribe from Observables when they are no longer needed to prevent memory leaks, especially with long-lived or "hot" Observables (e.g., event listeners). The subscribe() method returns a Subscription object, which has an unsubscribe() method.

TypeScript

const subscription = myObservable.subscribe(*/\* ... \*/*);  
 *// Later, when no longer needed:*  
 subscription.unsubscribe();

* **Operators:** RxJS provides a rich set of operators that allow for powerful transformations, filtering, and combining of Observable streams. These operators return new Observables, enabling a declarative and functional approach to data manipulation. Examples include map, filter, debounceTime, merge, etc.

TypeScript

import { of } from 'rxjs';  
 import { map } from 'rxjs/operators';  
  
 of(1, 2, 3)  
 .pipe(map(value => value \* 2))  
 .subscribe(doubledValue => console.log(doubledValue)); // Outputs: 2, 4, 6