What is an Observable

Notes on Observables and Promises in Angular

1. **Understanding Observables**

- **Definition**: Observables are used to handle asynchronous data in Angular applications. They are provided by the RxJS library, not natively part of JavaScript.
- **Asynchronous Data Handling**: Observables are ideal for dealing with data that arrives over time or in chunks, such as data from HTTP requests or real-time data streams.

An Observable is a wrapper around asynchronous data. We use an observable to handle asynchronous data.

- **Comparison with Promises**:
  - **Promises**: Handle single events or data pieces. They resolve with either a result or an error but do not handle multiple events or data streams.
  - **Observables**: Can handle multiple values over time. They are suited for scenarios where data is received in a stream, such as video streaming or live data feeds.

Promises returns Single value.

2. **Asynchronous vs Synchronous Operations**

- **Synchronous Operations**: Code execution is line-by-line, blocking the execution until the current task completes.
    - **Example**: An HTTP request in synchronous code will block subsequent code until the response is received.
- **Asynchronous Operations**: Code execution continues without waiting for the current task to complete. This avoids blocking the main thread.
    - **Example**: An asynchronous HTTP request allows the next line of code to execute immediately, even if the response has not yet arrived.

What is synchronous programming

JavaScript is a single-threaded programming language.

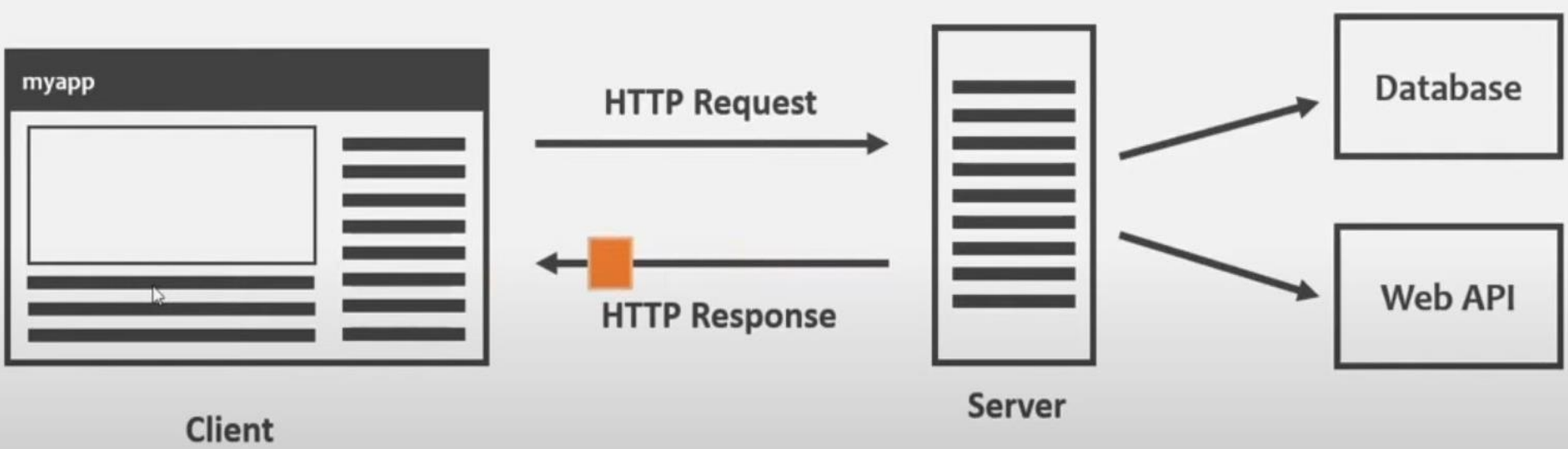
- The code is executed line by line one after the other in the order in which they are defined.
- Synchronous code is blocking in nature.

What is asynchronous programming

Asynchronous code does not execute in single-threaded. It gets executed in the background.

- Asynchronous code is blocking in nature.

What is data streaming?



3. Streaming of Data

- **Definition**: Streaming involves sending data in small chunks rather than waiting to send the entire data at once.
- **Example**: Streaming a video file in chunks allows a user to start watching before the entire file is downloaded, unlike downloading the whole file before playback.

Promise vs Observable

- A promise cannot handle stream of asynchronous data. It always returns a single value. On the other hand, we can use observables to handle stream of asynchronous data. It can return multiple values.
- A promise will certainly return a data even if no code is using that data. Whereas an observable will return a data only if someone is going to use that data.
- A promise is native to JavaScript program. Whereas observable is not native to JavaScript and it is provided by "**RxJS**" library.

4. Promises vs Observables

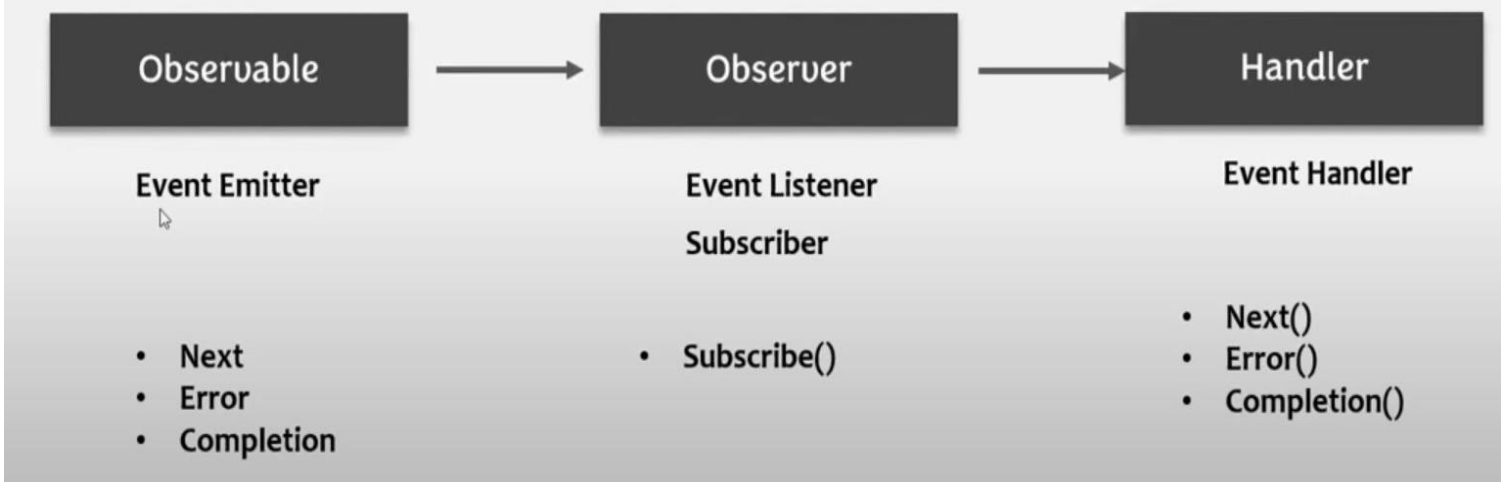
- **Promises**:
  - Resolve once with either data or an error.
  - Not suitable for handling multiple data pieces over time.
  - Example: An HTTP request returns a single response or error.

- **Observables**:
  - Can emit multiple values over time.
  - Only provide data if there is a subscriber to use it.
  - Example: Observing a real-time data stream where new data points arrive over time.
- **Native vs Library**:
  - Promises are built into JavaScript.
  - Observables are part of the RxJS library, which provides additional methods and functionality.

What is an Observable?

An 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.

Observable Pattern



5. **Observable Pattern**
   - **Components**:
     - **Observable (Event Emitter)**: Emits events or data.
     - **Observer (Subscriber)**: Listens and reacts to the emitted events.
   - **Event Types**:
     - **Next Event**: Regular data.
     - **Error Event**: Error occurrences.
     - **Completion Event**: Stream has completed.

- **Example**: In a video streaming app, the observable could emit chunks of video data as they are received, while the observer handles each chunk to display the video.

6. RxJS Library

- **Purpose**: Provides utilities for working with asynchronous data streams and observables.
- **Capabilities**: Offers methods to create, transform, and manage observables.

Next Steps

- **Practical Understanding**: In upcoming lectures, learn to create and work with observables using the RxJS library in Angular applications.

RxJS (Reactive Extensions for JavaScript) is a library for composing asynchronous and event-based programs using **Observables**. It provides a powerful way to handle asynchronous data streams, making it easier to manage events, API calls, timers, and more.

Creating & Using an Observable

Observable in RxJS

Definition:

An **Observable** is a core part of the RxJS library, used to handle asynchronous data streams. It is based on the Observer design pattern. Observables emit data over time, and Observers can subscribe to them to receive and react to that data.

What is an Observable?

An 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.

Key Concepts:

1. **Observable**: Emits data (such as a data stream) over time.
2. **Observer (Subscriber)**: Receives the data emitted by the Observable by subscribing to it.
3. **Subscription**: The process by which an Observer subscribes to an Observable to start receiving data.

RxJS Main Players

Example:

Creating and subscribing to an Observable in Angular:

1. **Creating an Observable:**

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


// Create an observable
const myObservable = new Observable((observer) => {
    observer.next([1, 2, 3, 4, 5]);  // Emit an array
});
```

2. Subscribing to an Observable:

typescript

```typescript
myObservable.subscribe((value) => {
    console.log(value);  // Output: [1, 2, 3, 4, 5]
});
```

```typescript
export class ObservablesComponent {

  // using new Keyword I can create this Observable & for emitting data use next method
  myObservable = new Observable((val) => {
    val.next([1, 2, 3, 4, 5]);
  })

  //We can use the data emitted by observable only if there is an subscriber

  // this observer is responsible to handle the data emitted by observable
  // The subscribe is an observer
  Handler = this.myObservable.subscribe((val) => {
    console.log(val);
  })
```

Observer and Observable Example

Definition:

- **Observable**: Emits values using the `next()` method.
- **Observer**: Subscribes to the Observable and handles the emitted data using a callback (e.g., in the `subscribe()` method).

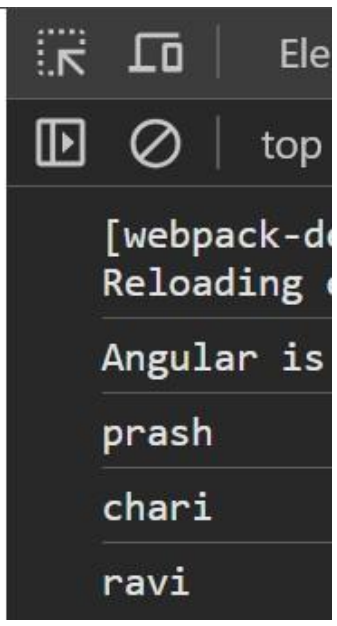
Example:

1. Creating an Observable that Streams Data:

typescript Copy code

```typescript
const myObservable = new Observable((observer) => {
    setTimeout(() => observer.next(1), 1000);  // Emit 1 after 1 second
    setTimeout(() => observer.next(2), 2000);  // Emit 2 after 2 seconds
    setTimeout(() => observer.next(3), 3000);  // Emit 3 after 3 seconds
});
```

observables works!



Subscribing to the Observable is an observer

2. Subscribing to the Observable:

typescript

```typescript
myObservable.subscribe((value) => {
    console.log(value);  // Output: 1, 2, 3 (with 1-second intervals)
});
```

```
////////////////////////////EXAMPLE 2 ////////////////
myObservable2 = new Observable((val) => {
  setTimeout(() => {
    val.next("prash")
  }, 1000)

  setTimeout(() => {
    val.next("ravi")
  }, 2000)

  setTimeout(() => {
    val.next("chari")
  }, 3000)
})

result = this.myObservable2.subscribe((val) => {
  console.log(val);
})
```

Observable Methods

- **next()**: Emits a value.
- **subscribe()**: Allows the Observer to subscribe to the Observable and handle the emitted values.

Callback Functions in `subscribe()`

- **next()**: Handles data emitted by the Observable.
- **error()**: Handles errors emitted by the Observable.
- **complete()**: Handles the completion of the Observable's data stream.

Example of Handling All Callbacks:

typescript

```typescript
myObservable.subscribe({
  next(value) { console.log(value); },
  error(err) { console.log('Error:', err); },
  complete() { console.log('Data stream complete'); }
});
```

```typescript
@Component({
  selector: 'app-observables',
  templateUrl: './observables.component.html',
  styleUrls: ['./observables.component.css']
})
export class OBSERVABLESComponent {

  data: any[] = [];

}
```

app.component.html

app > app.component.html > div.container > button

Go to component

```html
<div class="container">
  <h2>Observables in Angular</h2>
  <div class="data-list" *ngFor="let x of data">
    {{ x }}
  </div>

  <button>Get Data</button>
</div>
```

Observables in Angular

Get Data

Let’s create an Observable?

```
data: any[] = [];


//1. CREATE AN OBSERVABLE

//Observable
myObservable = new Observable((observer) => {
  observer.next([1, 2, 3, 4, 5]);
});
```

```
GetAsyncData(){

  //Observer
  //next, error, complete
  this.myObservable.subscribe((val: any) => {
    this.data = val;
  });
}
```

Observables in Angular

1

2

3

4

5

Get Data

Setting time out for values

```
//1. CREATE AN OBSERVABLE

//Observable
myObservable = new Observable((observer) => {
  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);
});
```

```
GetAsyncData(){

  //Observer
  //next, error, complete
  this.myObservable.subscribe((val: any) => {
    this.data.push(val);
  });
```

Error & Completion in Observable

Observable in RxJS

An **Observable** is a tool used to handle asynchronous data in RxJS. It can emit multiple values over time, handle errors, and signal completion.

Key Methods:

1. `next()` - Emits data values.
2. `error()` - Emits an error signal.
3. `complete()` - Emits a completion signal, indicating all data has been emitted.

Example 1: Basic Observable Emission

```
const myObservable = new Observable((observer) => {
  observer.next(1); // Emits value 1
  observer.next(2); // Emits value 2
  observer.complete(); // Signals completion
});
myObservable.subscribe({
  next: (value) => console.log(value), // Handles emitted values
  complete: () => console.log('Completed') // Handles completion
});
```

```
  Observable3 = new Observable((val) => {
    val.next("123445"),
      // val.error(new Error("You have Entered Wrong Value")),
      val.next([3, 4, 5, 6, 7]),
      val.complete()
  })

  res = this.Observable3.subscribe({
    next: (val) => { console.log(val) },
    complete: () => { console.log('Completed') }
  })
}
```

Handling Errors with Observable

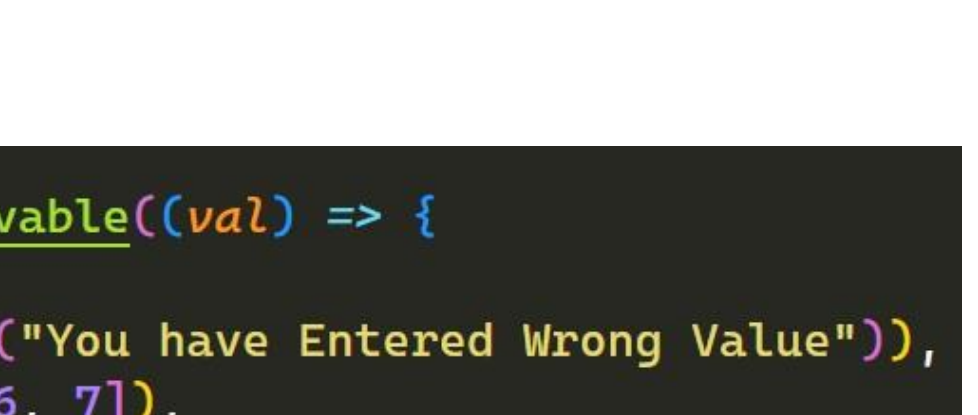
- **Error Emission**: If an observable encounters an error, it stops emitting further values.
- You can handle errors by passing an error callback to the `subscribe` method.

Example 2: Emitting and Handling Errors

```
const myObservable = new Observable((observer) => {
  observer.next(1); // Emits 1
  observer.next(2); // Emits 2
  observer.error(new Error('Something went wrong!')); // Emits error
});

myObservable.subscribe({
  next: (value) => console.log(value),
  error: (err) => console.error(err.message) // Handles error
});
```

```
  Observable3 = new Observable((val) => {
    val.next("123445"),
      val.error(new Error("You have Entered Wrong Value")),
      val.next([3, 4, 5, 6, 7]),
      val.complete()
  })

  res = this.Observable3.subscribe({
    next: (val) => { console.log(val) },
    error: (err) => { console.log(err) },
    complete: () => { console.log('Completed') }
  })
}
```

observables works!



Emitting Completion Signal

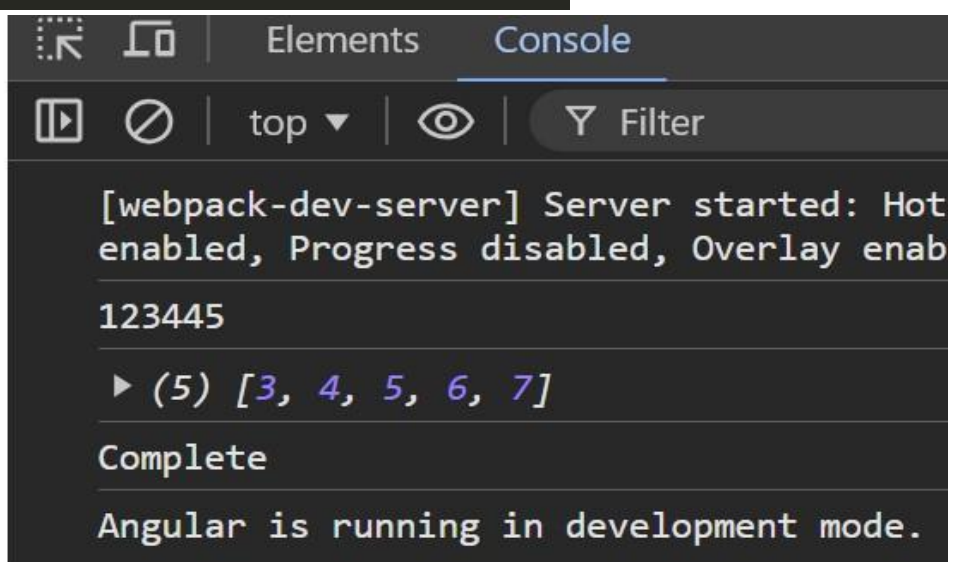
- Once the `complete()` method is called, no further values can be emitted by the observable.
- You can handle the completion by passing a third callback to the `subscribe` method.

Example 3: Handling Completion Signal

```
const myObservable = new Observable((observer) => {
  observer.next(1);
  observer.next(2);
  observer.complete(); // Emits completion
});

myObservable.subscribe({
  next: (value) => console.log(value),
  complete: () => alert('All data streamed!') // Handles completion
});
```

```
51
52  Observable3 = new Observable((val) => {
53    val.next("123445"),
54      val.error(new Error("You have Entered Wrong Value")),
55      val.next([3, 4, 5, 6, 7]),
56      val.complete()
57  })
58
59  res = this.Observable3.subscribe({
60    //Here, Keys : we can use normal functions for values
61    next: function (val) {
62      console.log(val);
63    },
64
65    // or arrow functions
66    error: (err) => { console.log(err) },
67
68    complete: function () {
69      console.log("Complete")
70    }
71  })
72  }
```

observables works!



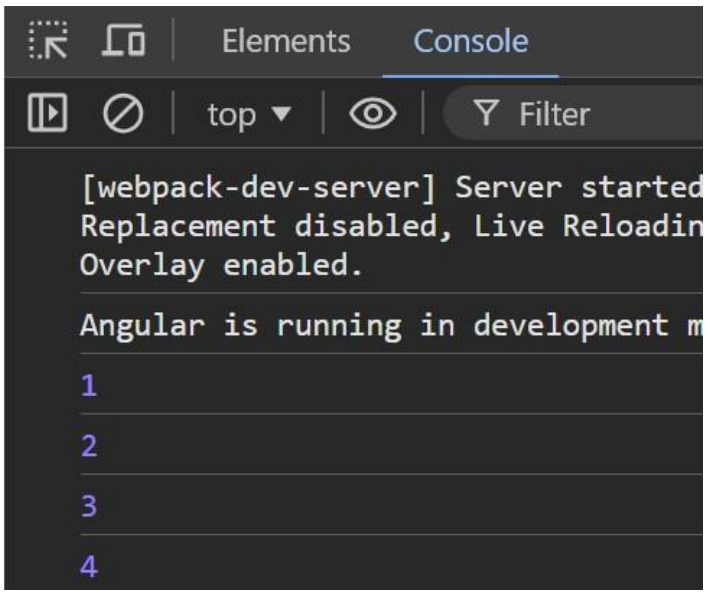
Key Notes:

- Once an observable emits an **error**, it cannot emit further values or the completion signal.
- Once the **completion signal** is emitted, the observable cannot emit any more data.

This structure provides robust handling for asynchronous data streams in Angular and RxJS-based applications.

1
2
3
4

Get Data



observables.component.ts U

AllConcepts > src > app > observables > observables.component.ts > ...

```ts
Empty: any[] = [];

observable4 = new Observable((val) => {

  setTimeout(() => {
    val.next(1)
  }, 1000),

    setTimeout(() => {
      val.next(2)
    }, 2000),

    setTimeout(() => {
      val.next(3)
    }, 3000)
})
```

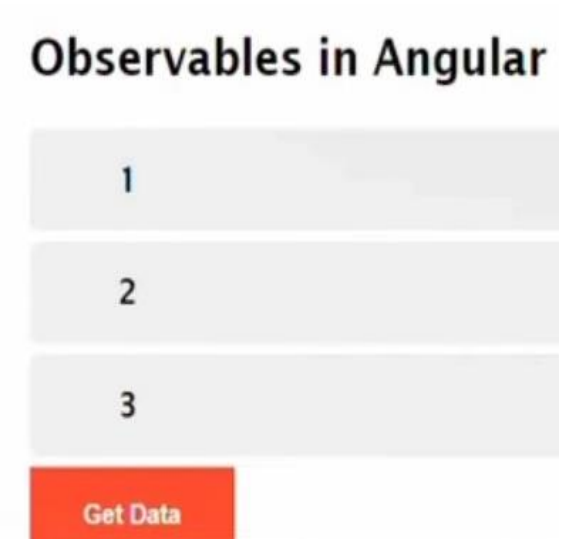
```ts
getData() {
  this.observable4.subscribe((val) => {
    console.log(val);
    this.Empty.push(val);
  })
}
```

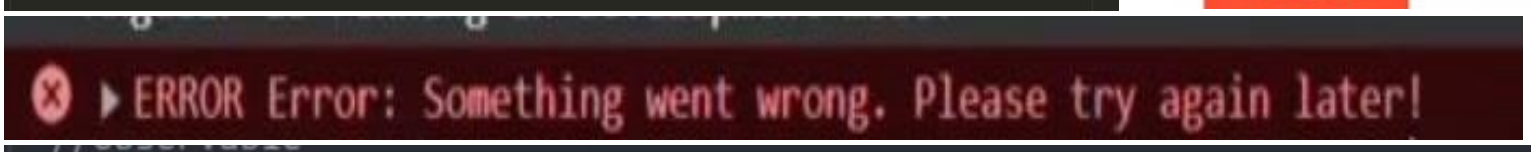
observables.component.html U

AllConcepts > src > app > observables > observables.component.html > div.container

Go to component

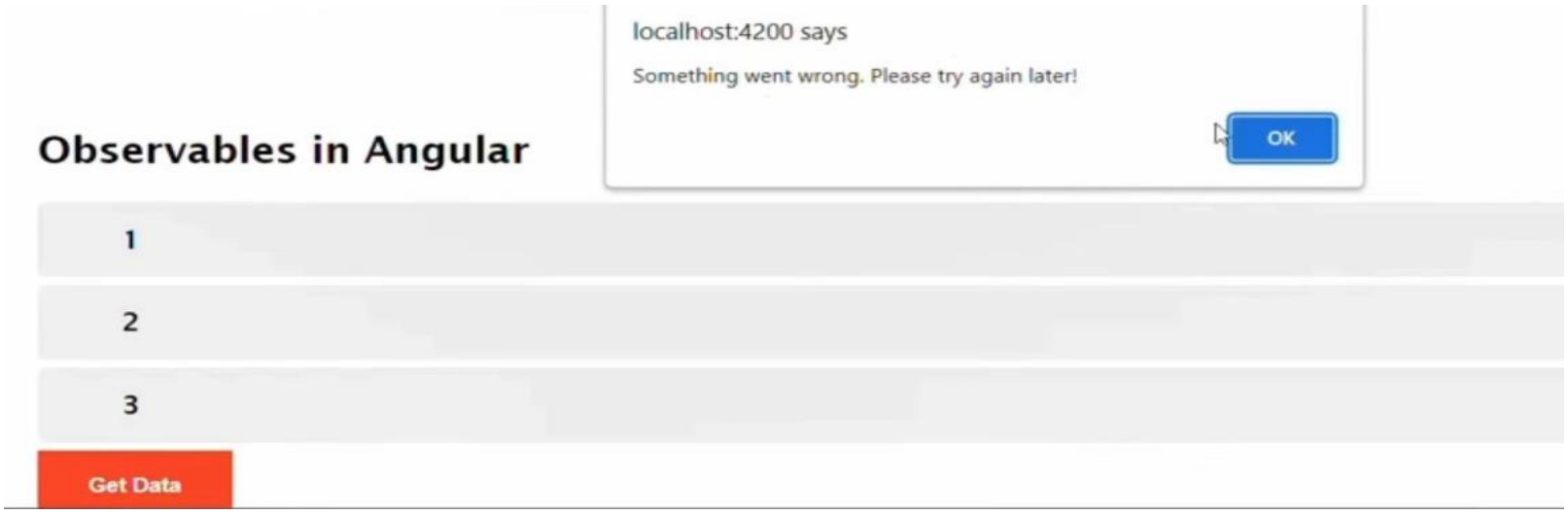
```html
<div class="container" style="text-align:center">

  <div *ngFor="let data of Empty">

    {{data}}

  </div>

  <button (click)="getData()">Get Data</button>
</div>
```






```ts
myObservable = new Observable((observer) => {
  setTimeout(() => { observer.next(1) }, 1000);
  setTimeout(() => { observer.next(2) }, 2000);
  setTimeout(() => { observer.next(3) }, 3000);
  setTimeout(() => { observer.error(new Error('Something went wrong. Please try again later
  setTimeout(() => { observer.next(4) }, 4000);
  setTimeout(() => { observer.next(5) }, 5000);
});
```

```ts
GetAsyncData(){

  //Observer
  //next, error, complete
  this.myObservable.subscribe((val: any) => {
    this.data.push(val);
  },
  (err) => {
    alert(err.message);
  });
```

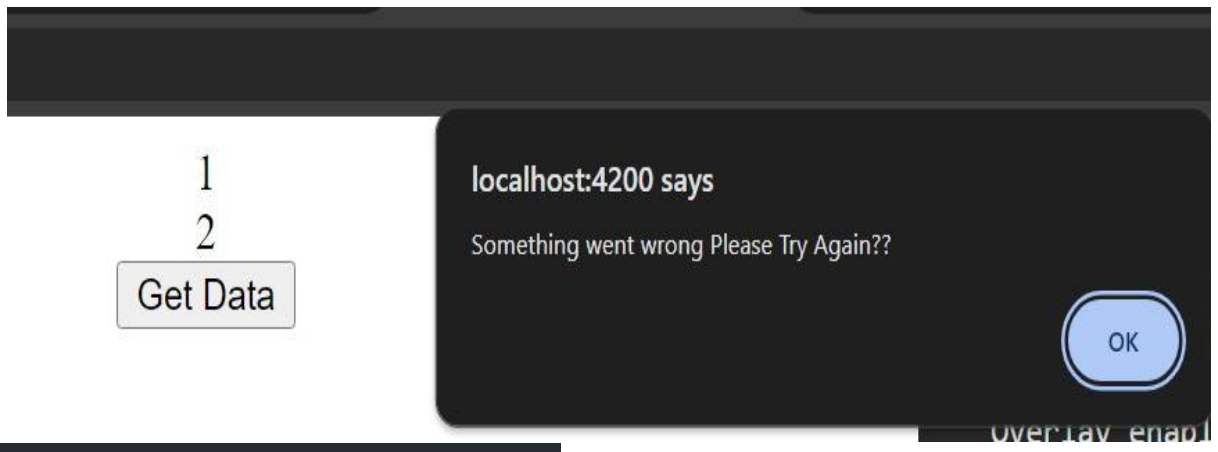



```
ables.component.ts U
ts > src > app > observables > observables.component.ts > ...
Empty: any[] = [];

observable4 = new Observable((val) => {

  setTimeout(() => {
    val.next(1)
  }, 1000),
    setTimeout(() => {
      val.error(new Error("Something went wrong Please Try Again??"));
    }, 3000),

    setTimeout(() => {
      val.next(2)
    }, 2000)
```

```
92
93    getData() {
94      this.observable4.subscribe((val) => {
95        this.Empty.push(val);
96      },
97        (err) => {
98          alert(err.message);
99        });
100     }
101   }
```

```
Go to component
<div class="container"
style="text-align:center">

  <div *ngFor="let data of Empty">

    {{data}}

  </div>

  <button (click)="getData()">Get Data</button>
</div>
```

```
myObservable = new Observable((observer) => {
  setTimeout(() => { observer.next(1) }, 1000);
  setTimeout(() => { observer.next(2) }, 2000);
  setTimeout(() => { observer.next(3) }, 3000);
  //setTimeout(() => { observer.error(new Error('Something went wrong. Please try again late
  setTimeout(() => { observer.next(4) }, 4000);
  setTimeout(() => { observer.next(5) }, 5000);
  setTimeout(() => { observer.complete() }, 6000);
});
```




```
GetAsyncData(){

  //Observer
  //next, error, complete
  this.myObservable.subscribe((val: any) => {
    this.data.push(val);
  },
  (err) => {
    alert(err.message);
  },
  () => {
    alert('All the data is streamed!')
  });
```

After complete signal has been emitted after 3 second,
Then fourth, fifth and second will not execute.

```
myObservable = new Observable((observer) => {
  setTimeout(() => { observer.next(1) }, 1000);
  setTimeout(() => { observer.next(2) }, 2000);
  setTimeout(() => { observer.next(3) }, 3000);
  //setTimeout(() => { observer.error(new Error('Something went wrong. Please try again late
  setTimeout(() => { observer.next(4) }, 4000);
  setTimeout(() => { observer.next(5) }, 5000);
  setTimeout(() => { observer.complete() }, 3000);
});
```

```
GetAsyncData(){

  //Observer
  //next, error, complete
  this.myObservable.subscribe((val: any) => {
    this.data.push(val);
  },
  (err) => {
    alert(err.message);
  },
  () => {
    alert('All the data is streamed!')
  });
```

of & from Operator

Observables in Angular

1

2

3

Get Data

Above way of declaring call back functions is deprecated so use this way.

```
this.myObservable.subscribe({
  next: (val: any) => {
    this.data.push(val);
  },
  error(err){
    alert(err.message);
  },
  complete(){
    alert('All the data is streamed!')
  }
})
```

`off` Operator

- **Definition**: The `off` operator in RxJS creates an observable from the arguments passed to it. You can pass multiple arguments, and each one is emitted separately, one after the other. After all arguments are emitted, a "complete" signal is automatically sent.
- **Example**:

```
import { of } from 'rxjs';

// Create an observable from arrays
const array1 = [1, 2, 3];
const array2 = ['a', 'b', 'c'];
const observable = of(array1, array2);

observable.subscribe({
  next: (value) => console.log(value),   // Each array is emitted as is
  complete: () => console.log('Complete') // "Complete" signal is emitted
});
```

- **Output**: First `array1` is emitted, then `array2`. After all data is emitted, the complete signal is sent.

- **Key Points**:
  - Emits each argument passed to it one by one.
  - Can handle multiple arguments, including arrays, strings, numbers, etc.
  - Arrays are emitted as a single object, not broken into elements.
  - Automatically sends a complete signal after all emissions.

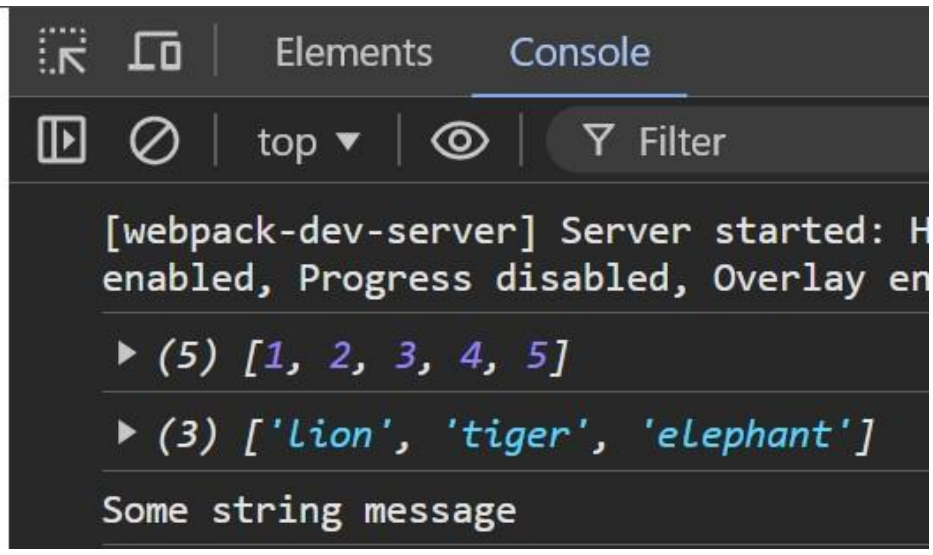
The of Operator

The `of` operator creates an observable from the arguments that we pass into it. You can pass any number of arguments to `of` operator.

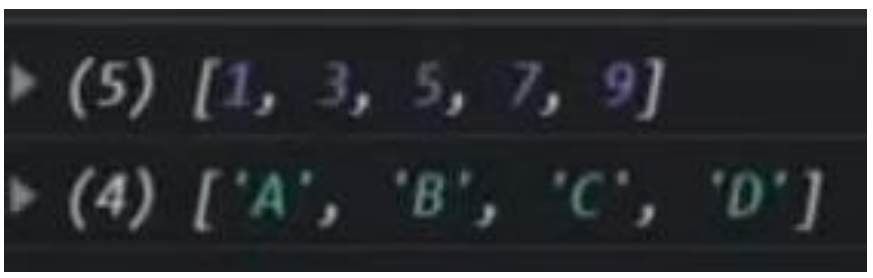
Each argument is emitted separately one after the other. It send the complete signal at the end.

```
export class OperatorsComponent {

  // Create an Observable using OF Operator

  array1 = [1, 2, 3, 4, 5];
  array2 = ['lion', 'tiger', 'elephant'];

  observable = of(this.array1, this.array2, "Some string message");

  ngOnInit() {
    this.observable.subscribe({
      next: function (val) {
        console.log(val)
      }
    })
```

of operator from RxJS works!



Here, arrays passed & values are not displaying arrays only



`from` Operator

- **Definition**: The `from` operator creates an observable from an iterable (e.g., arrays, strings). Unlike `off`, it breaks down the iterable and emits each item separately.
- **Example**:

```
import { from } from 'rxjs';

const array = [1, 2, 3];
const observable = from(array);

observable.subscribe({
  next: (value) => console.log(value),  // Each element is emitted individually
  complete: () => console.log('Complete') // "Complete" signal is emitted
});
```

The `from` Operator

The from operator takes a single argument which can be iterated over and convers it into an observable.

- **Output**: Each element of the array is emitted individually, and after all elements are emitted, the complete signal is sent.

- **Key Points**:
  - Takes a single iterable as an argument (array, string, etc.).
  - Emits each element of the iterable individually, unlike `off`, which emits the whole iterable at once.
  - Automatically sends a complete signal once all elements are emitted.

Converting Promise to Observable Using `from`

- **Definition**: The `from` operator can also convert promises into observables.
- **Example**:

PROMISE EXAMPLE

```
promises = new Promise((resolve, reject) => {
  resolve("Promise Returning a Value");
  reject("Error Occured");
}).then((val) => {
  console.log(val)
}).catch((err) => {
  console.log(err);
})
```

```
Promise Returning a Value
```

```
const promiseData = new Promise((resolve) => {
  resolve([10, 20, 30]);
});

const observableFromPromise = from(promiseData);

observableFromPromise.subscribe({
  next: (value) => console.log(value), // Emits the resolved array
  complete: () => console.log('Complete')
});
```

- **Output**: The promise resolves, and the array `[10, 20, 30]` is emitted as the observable.

- **Key Points**:
  - Converts promises into observables.
  - Emits the resolved value of the promise once the promise is fulfilled.

From operator Takes Only One Argument & iterables
Only like arrays or strings.

```
data: any[] = [];
arr1 = [1, 2, 3, 4, 5];
arr2 = [6, 7, 8];

// myObservable = of(this.arr1, this.arr2, 'A', 'B', 'C', 30);
myObservable = from(this.arr1);
```

```
getAsyncData() {

  this.myObservable.subscribe({
    next: (val) => {
      this.data.push(val);
    },

    error(err) {
      alert(err);
    },

    complete() {
      alert("All stages are completed")
    },
  })
```

Observables

1

2

3

4

5

Get Data

```
Promise Returning a Value
undefined
```

We can convert an promise into an Observable easily with from method

```
const promiseData = new Promise((resolve, reject) => {
    resolve([10, 20, 30, 40, 50]);
});


const myObservable = from(promiseData);
```

```
// Create an Observable using FROM Operator
promises = new Promise((resolve, reject) => {
  resolve("Promise Returning a Value");
  reject("Error Occured");
}).then((val) => {
  console.log(val)
}).catch((err) => {
  console.log(err);
})
```

```
// Converting an promise to observable from FROM operator

observable6 = from(this.promises);

ngOnInit() {
  this.observable6.subscribe((val) => {
    console.log(val)
})
```

fromEvent Operator

Notes on RxJS `fromEvent` Operator

Definition:

The `fromEvent` operator in RxJS allows you to create an observable from a DOM event. This enables handling events such as clicks, key presses, or any other DOM interaction in a reactive programming style.

Steps to Create an Observable from a DOM Event:

1. **Get Access to the DOM Element:**
   - Use Angular's `@ViewChild` decorator to get a reference to the DOM element you want to observe.
   - Example:

```typescript
@ViewChild('createbtn') createBtn: ElementRef;
```

2. **Create the Observable:**
   - Use the `fromEvent` operator to create an observable from the event you want to listen to (e.g., `click`).
   - Pass two arguments:
     - The target element (e.g., a button).
     - The event to listen for (e.g., `click`).
   - Example:

```typescript
import { fromEvent } from 'rxjs';

const observable = fromEvent(this.createBtn.nativeElement, 'click');
```

3. **Subscribe to the Observable:**

   - Use the `subscribe()` method to respond when the event occurs.
   - Example:

```typescript
observable.subscribe((event) => {
  console.log(event);
});
```

4. **Create and Display Dynamic Content:**

   - Handle the event in the `subscribe()` callback by manipulating the DOM or updating the UI.
   - Example: Dynamically add a new `div` with incrementing text every time the button is clicked.

```
showItem(value: number) {
  const div = document.createElement('div');
  div.innerText = `Item ${value}`;
  document.getElementById('container').appendChild(div);
}

observable.subscribe(() => {
  this.showItem(++count);
});
```

5. **Lifecycle Hook:**

   - Ensure the observable is set up after the view is initialized by calling the method in `ngAfterViewInit`.
   - Example:

```typescript
ngAfterViewInit() {
  this.setupClickObservable();
}
```

```typescript
import { Component, ViewChild, ElementRef, AfterViewInit } from '@angular/core';
import { fromEvent } from 'rxjs';

@Component({
  selector: 'app-from-event',
  templateUrl: './from-event.component.html'
})
export class FromEventComponent implements AfterViewInit {
  @ViewChild('createbtn') createBtn: ElementRef;
  count = 0;

ngAfterViewInit() {
  this.setupClickObservable();
}

setupClickObservable() {
  const observable = fromEvent(this.createBtn.nativeElement, 'click');
  observable.subscribe(() => {
    this.showItem(++this.count);
  });
}
```

```typescript
  showItem(value: number) {
    const div = document.createElement('div');
    div.innerText = `Item ${value}`;
    document.getElementById('container').appendChild(div);
  }
}
```

Key Points:

- The `fromEvent` operator is useful for turning user interactions into observable streams.
- You can handle dynamic UI updates by subscribing to the observable and performing DOM manipulations.
- This is a simple example; real-world applications may involve more complex scenarios such as handling multiple events or combining multiple streams.

EXAMPLE:

```
  buttonClicked() {
    return fromEvent(this.createBtn.nativeElement, 'click');
  }
```

In the Angular code you shared, you're using `fromEvent` from RxJS to listen for button click events. Here's a brief explanation of what's happening:

HTML (`from-event.component.html`):

You have a button with a `#createbtn` template reference variable.

```
<div class="container">
  <h2>From Event</h2>

  <div class="data-list">
    <button #createbtn>Create New Item</button>
  </div>
</div>
```

TypeScript (`from-event.component.ts`):

- You're using Angular's `@ViewChild` decorator to reference the `createbtn` element from the template.
- After the view has been initialized (`ngAfterViewInit` lifecycle hook), you're subscribing to the `click` event on this button using `fromEvent`.

```
import { AfterViewInit, Component, ElementRef, ViewChild } from '@angular/core';
import { fromEvent } from 'rxjs';

@Component({
  selector: 'app-from-event',
  templateUrl: './from-event.component.html',
  styleUrls: ['./from-event.component.css']
})
export class FromEventComponent implements AfterViewInit {
```

```
@ViewChild('createbtn') createBtn: ElementRef;

createBtnObservable: any;

buttonClicked() {
  this.createBtnObservable = fromEvent(this.createBtn.nativeElement, 'click').subscribe(
    console.log(data);
  });
}

ngAfterViewInit() {
  this.buttonClicked();
}
```

Key Points:

- `@ViewChild('createbtn')`: This grabs the button element once the view is initialized.
- `fromEvent(this.createBtn.nativeElement, 'click')`: This creates an observable for the `click` event on the button.
- **Subscription**: In `buttonClicked`, you subscribe to the observable and log event details when the button is clicked.

The `ngAfterViewInit` ensures that the button's reference is available when you try to attach the event listener. This approach leverages the reactive style of RxJS for handling events in Angular.

Let me know if you need further clarifications!

```
Error: src/app/RXJS/from-event/from-event.component.ts:11:27 - error
TS2564: Property 'createBtn' has no initializer and is not definitely
assigned in the constructor.

11   @ViewChild('createbtn') createBtn: ElementRef;
                             ~~~~~~~~~
```

The error you're encountering, `TS2564: Property 'createBtn' has no initializer and is not definitely assigned in the constructor`, is due to strict type checking introduced by TypeScript's `strictPropertyInitialization` setting. This setting ensures that class properties are initialized before they are accessed.

To resolve this issue, you can explicitly tell TypeScript that the `createBtn` property might be `undefined` initially, and will be assigned later by Angular's `@ViewChild` mechanism. You can do this by marking the property with the `!` non-null assertion operator.

Solution

Modify the `createBtn` declaration as follows:

typescript

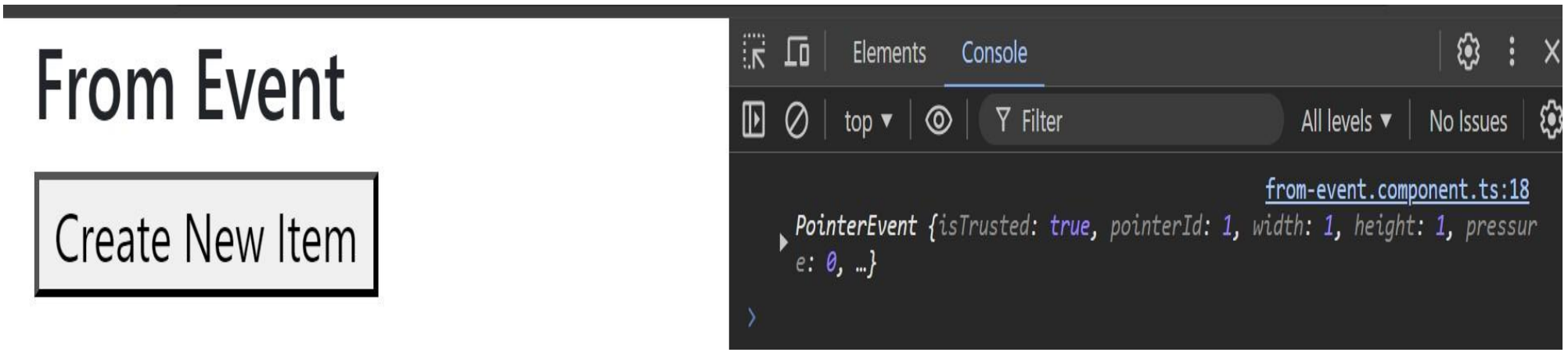
```typescript
@ViewChild('createbtn') createBtn!: ElementRef;
```

```typescript
@ViewChild('createbtn') createBtn!: ElementRef;  // Non-null assertion operator added
```

Explanation:

- `createBtn!`: The `!` tells TypeScript that you are sure the `createBtn` will be initialized before it's accessed, which happens after Angular initializes the view.

This should resolve the error. Let me know if it works!



Map & Filter Operator

The map Operator

The map operator is used to transform data emitted by a source observable in some way.

Map Operator

- **Definition**: The `map` operator is used to transform data emitted by a source observable. It modifies each value emitted by the observable based on a provided logic.
- **Use Case**: You want to multiply every emitted value by a certain number.
- **Example**:

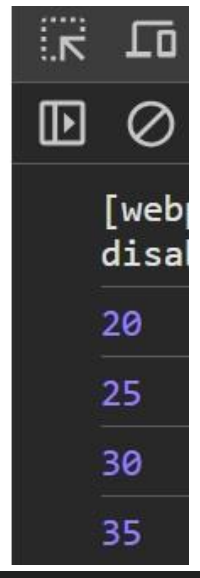
```
import { from } from 'rxjs';
import { map } from 'rxjs/operators';

const myObservable = from([2, 4, 6, 8, 10]);
const transformedOBS = myObservable.pipe(
  map(value => value * 5)
);

transformedOBS.subscribe(data => console.log(data));
// Output: 10, 20, 30, 40, 50
```

```
//////////////////////////// MAP ////////////////////////////

observable3 = from([4, 5, 6, 7]);

TransformedObs = this.observable3.pipe(map(val => val * 5))

ngOnInit() {
  this.TransformedObs.subscribe((val) => {
    console.log(val);
  })
}
```

The `filter` Operator

The `filter` operator is used to filter data emitted by a source observable based on a given condition.

MAP Operator => Transforming Values



Filter Operator

- **Definition**: The `filter` operator is used to filter the emitted values of an observable based on a condition. Only values that meet the specified condition will pass through.
- **Use Case**: You want to filter only values that are divisible by 4.
- **Example**:

```
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

const transformedOBS = from([10, 20, 30, 40, 50, 60]);
const filteredOBS = transformedOBS.pipe(
  filter(value => value % 4 === 0)
);

filteredOBS.subscribe(data => console.log(data));
// Output: 20, 40, 60
```

Pipe Method

- **Definition**: The `pipe` method is used to chain multiple operators together. Each operator processes the result from the previous one in the order defined.
- **Example** (Chaining map and filter operators):

```
//////////////////////////// FILTER ////////////////////////////
observable3 = from([4, 5, 6, 7]);

TransformedObs = this.observable3.pipe(filter(val => val > 5))

ngOnInit() {
  this.TransformedObs.subscribe((val) => {
    console.log(val);
  })
}
```

FILTER Operator => Display Values which met the condition

```
[we
dis
6
7
```

```
const myObservable = from([2, 4, 6, 8, 10, 12]);
const filteredOBS = myObservable.pipe(
  map(value => value * 5),          // Step 1: Multiply by 5
  filter(value => value % 4 === 0) // Step 2: Filter values divisible by 4
);

filteredOBS.subscribe(data => console.log(data));
// Output: 20, 40, 60
```

Example:

map-filter.component.ts U map-filter.component.html U

Complete_Course > angular-ekart > src > app > RXJS > map-filter > map-filter.component.html > ...

Go to component

```html
<div class="item" *ngFor="let x of data">
  {{x}}
</div>
<button (click)="getAsyncData()">Get Data</button>
```

```
data: any[] = [];
```

```
myObservable = from([2, 4, 6, 8, 10, 12]);

// Transform values by multiplying each by 5
transformedObs = this.myObservable.pipe(
  map((val) => val * 5)
); // Result: [10, 20, 30, 40, 50, 60]
```

```
// Filter values that are divisible by 4
filteredObs = this.transformedObs.pipe(
  filter((val) => val % 4 === 0)
); // Result: [20, 40, 60]
```

```
getAsyncData() {
  this.filteredObs.subscribe((val) => {
    this.data.push(val);
  })
}
```

Chaining Both methods use Coma(,) instead (.) as in Js

```
// transformedObs 10, 20, 30, 40, 50, 60
transformedObs = this.myObservable.pipe(
  map((val) => {
    return val * 5;
  }),
  filter((val, i) => {
    return val % 4 === 0;
  })
);
```

When to Use Subjects:

Subjects are used when we need to communicate between **sibling components** or **unrelated components** that don't have a parent-child relationship.

Example Explanation:

Subjects In RxJS

A `subject` is a special type of observable that allows values to be muticasted to many observers. Subjects are like EventEmitters.

RxJS Subjects in Angular

Definition:

A **Subject** in RxJS is a special type of **Observable** that allows values to be **multicasted** to multiple observers. It acts both as an **Observable** and **Observer**, meaning it can emit values (as an observable) and can also subscribe to other observables (as an observer).

- **Multicasting**: Multiple observers can listen to a subject's emitted data.
- **Cross-Component Communication**: Subjects make cross-component communication simple and effective.

```
  // Method to emit task
  emitTask(task: string) {
    this.taskCreated.emit(task);
  }
}
```

1. **Component Setup**:
    - There are two sibling components: `NewTaskComponent` (for input) and `ShowTaskComponent` (for displaying tasks).
    - The goal is to allow the user to input a task in `NewTaskComponent`, and display that task in `ShowTaskComponent`.

2. **Without Subjects (Using Event Emitters)**:
    - A service is created to manage data transfer.
    - In `NewTaskComponent`, the task is emitted using an `EventEmitter` via a service.
    - `ShowTaskComponent` subscribes to the emitted task and updates its list when a new task is created.

Here's an example of how you can achieve communication between two Angular components (`NewTaskComponent` and `ShowTaskComponent`) using **Event Emitters** without using **Subjects**. The `TaskService` will manage data transfer via an `EventEmitter`.

1. TaskService:

Create a service to manage the task emission and subscription using `EventEmitter`.

```
// task.service.ts
import { Injectable, EventEmitter } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class TaskService {
  taskCreated = new EventEmitter<string>(); // EventEmitter for task communication
```

2. NewTaskComponent:

This component is responsible for emitting a new task using the service's `EventEmitter`.

```
// new-task.component.ts
import { Component } from '@angular/core';
import { TaskService } from '../task.service';

@Component({
  selector: 'app-new-task',
  template: `
    <input [(ngModel)]="newTask" placeholder="Enter task" />
    <button (click)="addTask()">Add Task</button>
  `,
```

```
export class NewTaskComponent {
  newTask: string = '';

  constructor(private taskService: TaskService) {}

  addTask() {
    if (this.newTask) {
      this.taskService.emitTask(this.newTask); // Emit new task through service
      this.newTask = ''; // Clear input
    }
  }
}
```

3. ShowTaskComponent:

This component will subscribe to the emitted task and display it in a list.

```
// show-task.component.ts
import { Component, OnInit } from '@angular/core';
import { TaskService } from '../task.service';

@Component({
  selector: 'app-show-task',
  template: `
    <ul>
      <li *ngFor="let task of tasks">{{ task }}</li>
    </ul>
  `,
```

```
export class ShowTaskComponent implements OnInit {
  tasks: string[] = [];

  constructor(private taskService: TaskService) {}

  ngOnInit() {
    // Subscribe to the task created event
    this.taskService.taskCreated.subscribe((task: string) => {
      this.tasks.push(task); // Add new task to the list
    });
```

4. AppModule:

Ensure that both components and `TaskService` are declared and provided.

```
@NgModule({
  declarations: [AppComponent, NewTaskComponent, ShowTaskComponent],
  imports: [BrowserModule, FormsModule],
  providers: [TaskService],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

5. **AppComponent** (Main component template):

Add the `NewTaskComponent` and `ShowTaskComponent` in the main app component.

html

```
<!-- app.component.html -->
<h1>Task Manager</h1>
<app-new-task></app-new-task>
<app-show-task></app-show-task>
```

Explanation:

1. **TaskService**: Manages task emission using an `EventEmitter`.
2. **NewTaskComponent**: Emits a new task using the `TaskService` when the user adds a task.
3. **ShowTaskComponent**: Subscribes to the emitted task and adds it to a list for display.

Result:

- When a user adds a task in the `NewTaskComponent`, it is emitted via `EventEmitter` in the service.
- The `ShowTaskComponent` listens for emitted tasks and displays them in a list.

With Subjects:

To replace the `EventEmitter` with a `Subject`, follow these steps:

1. Create Subject in the Service:

In the service, instead of an `EventEmitter`, use a `Subject` to broadcast changes:

```
import { Subject } from 'rxjs';

export class TaskService {
  createTask = new Subject<string>();  // Emits task data as a string

  onCreateTask(value: string) {
    this.createTask.next(value);  // Emits the value using 'next'
  }
```

2. Emit Value Using the Subject:

In `NewTaskComponent`, call the service's `onCreateTask` method when the button is clicked:

```
import { TaskService } from './services/task.service';

export class NewTaskComponent {
  constructor(private taskService: TaskService) {}

  onCreateTask() {
    this.taskService.onCreateTask(this.newTask);  // Pass input task to service
```

3. Subscribe to Subject in the Component:

In `ShowTaskComponent`, subscribe to the `createTask` subject in `ngOnInit` to listen for new tasks:

```
import { TaskService } from './services/task.service';

export class ShowTaskComponent implements OnInit {
  tasks: string[] = ['Task 1', 'Task 2', 'Task 3'];

  constructor(private taskService: TaskService) {}

  ngOnInit() {
    this.taskService.createTask.subscribe((task: string) => {
      this.tasks.push(task);  // Add new task to the list
    });
  }
}
```

Key Differences:

- **EventEmitter** vs **Subject**: EventEmitter is for emitting events, while a Subject can also multicast values to multiple subscribers.
- **next()**: Used in a subject to emit data, similar to how `emit()` is used in `EventEmitter`.

Advantages of Subjects:

- **Efficient multicasting**: Subjects allow emitting data that can be consumed by multiple subscribers, unlike regular observables.
- **Simplified cross-component communication**: Facilitates data sharing between unrelated or sibling components without complicated setups.

Observable vs Subjects

Difference between Observables and Subjects in RxJS

1. **Definition**:
    - **Observable**: It is a function that produces values over time and can be subscribed to by observers. Observables are unicast, meaning they are executed independently for each observer. Each subscriber gets its own execution of the observable.
    - **Subject**: It is a special type of observable that allows multicasting (i.e., multiple observers can share the same execution). It can act as both an observable and an observer.

2. **Key Differences**:
    - **Unicast vs Multicast**:
        - **Observable**: Each subscriber has its own execution, i.e., each observer gets a unique set of data produced by the observable.
        - **Subject**: All subscribers receive the same execution, i.e., a subject shares a single execution with all its observers.

- **Producer**:
  - **Observable**: It has a single producer, and it starts emitting values only when subscribed.
  - **Subject**: It acts as both a producer and a consumer. You can manually push data into a subject which will be emitted to all its subscribers.

3. **Examples**:

   Observable Example:

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

const observable = new Observable((subscriber) => {
    subscriber.next('Hello');
    subscriber.next('World');
    subscriber.complete();
});

observable.subscribe(value => console.log('Subscriber 1:', value));
observable.subscribe(value => console.log('Subscriber 2:', value));
```

Output:

yaml Copy code

```
Subscriber 1: Hello
Subscriber 1: World
Subscriber 2: Hello
Subscriber 2: World
```

- Here, both `Subscriber 1` and `Subscriber 2` get their own independent execution of the observable.

Subject Example:

```
import { Subject } from 'rxjs';

const subject = new Subject();

subject.subscribe(value => console.log('Subscriber 1:', value));
subject.subscribe(value => console.log('Subscriber 2:', value));

subject.next('Hello');
subject.next('World');
```

Output:

yaml Copy c

```
Subscriber 1: Hello
Subscriber 2: Hello
Subscriber 1: World
Subscriber 2: World
```

- Here, both `Subscriber 1` and `Subscriber 2` get the same data from the subject as it multicasts the data to all its subscribers.

4. **Use Cases:**
    - **Observable**: Used when you want each subscriber to get their own independent stream of data. Example: API calls where each request should be handled separately.
    - **Subject**: Used for multicasting data, like for sharing state between components or cross-component communication. Example: Broadcasting events to multiple listeners in Angular services.

Conclusion:

- Use **Observables** when each subscriber should have its own execution of the data stream.
- Use **Subjects** when you want to share a single execution across multiple subscribers.

Subject in RxJS

A `subject` is a special type of observable that allows values to be muticasted to many observers. Subjects are like EventEmitters.

Observable vs Subject

A `subject` is **multicast.** On the other hand an Observable is **unicast.**

```
export class SubjectComponent implements OnInit {
  ngOnInit() {
    let obs = new Observable((observer) => {
      observer.next(Math.random());
    });
   // Subscriber 1
   obs.subscribe((data) => console.log(data)); // Subscriber 1

   // Subscriber 2
   obs.subscribe((data) => console.log(data)); // Subscriber 2
 }
```

Observable has generated two different value for same Observable

```
0.7893787237240497

0.9854200030126947
```

But in subjects Both Observers are getting Same value

```ts
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';

@Component({
  selector: 'app-subject',
  templateUrl: './subject.component.html',
  styleUrls: ['./subject.component.css']
})
```

Explanation:

- `Subject`: A special type of Observable that allows multicasting values to multiple observers.
- **Subscribers**: Two subscribers are created for the subject. They will log the emitted values to the console.
- `next` **Method**: The `next` method is called to emit a random number to all subscribers.

```ts
export class SubjectComponent implements OnInit {
  private subject: Subject<number> = new Subject<number>();

  ngOnInit() {
    // Subscriber 1
    this.subject.subscribe((data) => {
      console.log(data); // Logs data from Subscriber 1
    });

    // Subscriber 2
    this.subject.subscribe((data) => {
      console.log(data); // Logs data from Subscriber 2
    });

    // Emit a random number
    this.subject.next(Math.random());
  }
```

```
ngOnInit(){
  //let obs = new Observable((observer) => {observer.next(Math.random())})

  const subject = new Subject();

  //Subscriber 1
  subject.subscribe((data) => {console.log(data)})

  //Subscriber 2
  subject.subscribe((data) => {console.log(data)})

  subject.next(Math.random());
```

```
ngOnInit() {
  // let obs = new Observable((observer) => observer.next(Math.random()));
  const subject = new Subject();

  // Subscriber 1
  subject.subscribe((data) => {
    console.log(data);
  });
// Subscriber 2
subject.subscribe((data) => console.log(data));

subject.next(Math.random()); // Emits a random number
```

Explanation of Key Points:

- `ngOnInit`: This lifecycle method is called when the component is initialized.
- `const subject = new Subject();`: Creates a new instance of a Subject.
- **Subscribers**: Two subscriptions are made to the subject, each logging the emitted data.
- `subject.next(Math.random());`: Emits a random number, which both subscribers will receive.

This setup will log the random number emitted by the subject to the console twice, once for each subscriber.

```
0.743910224324017
0.743910224324017
```

EXAMPLE 2:

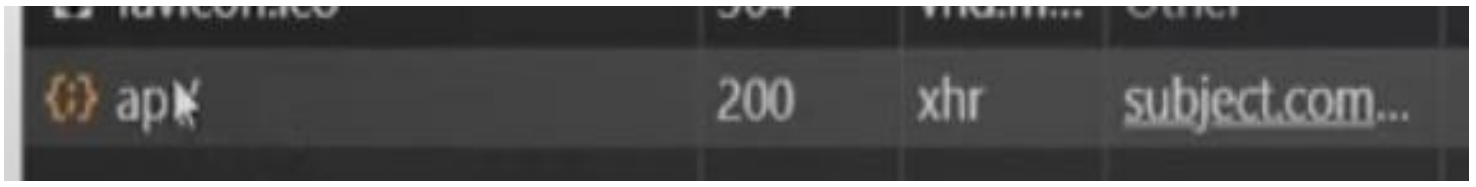
import ajax from RXJS to make Http Requests

```
import { Component, OnInit } from '@angular/core';
import { Observable, Subject } from 'rxjs';
import { ajax } from 'rxjs/ajax';
```

```
//////////////////////////HTTP REQUESTS USING AJAX MEthod

/**This ajax using get method &
 * this ajax returns an Observable by fetching data
 * we are storing that  observable in data variable
 */
let data = ajax("https://randomuser.me/api/");

/** let's subscribe to data observable */
data.subscribe((val) => {
  console.log(val);
});
```

```
                                                    obs-vssub.component.ts:43
▶ AjaxResponse {originalEvent: ProgressEvent, xhr: XMLHttpRequ
  est, request: {…}, type: 'download_load', status: 200, …}
```

```
// AJAX CALL
const subject = new Subject();
const data = ajax('https://randomuser.me/api/');

subject.subscribe((res) => console.log(res));
subject.subscribe((res) => console.log(res));
subject.subscribe((res) => console.log(res));

data.subscribe(subject);
```



Observables vs. Subjects in RxJS

Definitions:

- Observable:
    - An **Observable** is a data source that can emit values over time to subscribers. It is **unicast**, meaning each subscriber receives a separate copy of the data stream.

Subject:

- A **Subject** is a special type of Observable that allows values to be multicasted to many observers. It is **multicast**, meaning all subscribers receive the same emitted value.

Key Differences:

1. **Unicast vs. Multicast:**
    - **Observable:** Each subscriber gets its own independent execution. For example, if two subscribers listen to the same Observable emitting random numbers, they may receive different values.
    - **Subject:** All subscribers receive the same value emitted by the Subject. If two subscribers listen to a Subject, they will both get the same emitted value.

Example Implementation:

1. Using Observables:

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-example',
  template: `<h2>Observable Example</h2>`
})
```

```ts
export class ExampleComponent implements OnInit {
  ngOnInit() {
    const observable = new Observable((observer) => {
      observer.next(Math.random()); // Emit a random number
    });

    // Two subscribers
    observable.subscribe(data => console.log('Subscriber 1:', data));
    observable.subscribe(data => console.log('Subscriber 2:', data));
  }
```

- **Output:** Two different random numbers will be logged for each subscriber.

2. Using Subjects:

```ts
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';

@Component({
  selector: 'app-subject-example',
  template: `<h2>Subject Example</h2>`
})
export class SubjectExampleComponent implements OnInit {
  ngOnInit() {
    const subject = new Subject<number>();

    // Two subscribers
    subject.subscribe(data => console.log('Subscriber 1:', data));
    subject.subscribe(data => console.log('Subscriber 2:', data));

    // Emit a random number
    subject.next(Math.random());
  }
}
```

- **Output:** The API call is made once, and the same response is shared among all subscribers.

Conclusion:

- Use **Observables** when each subscriber needs a separate execution context or different values.
- Use **Subjects** when you want to multicast values to multiple subscribers and share the same emitted values.

- **Output:** The same random number will be logged for both subscribers.

Data Consumer and Provider:

- **Observable:** Acts only as a data provider. It emits data that can be subscribed to.
- **Subject:** Can act as both a data provider and a data consumer. It can emit values and also consume data from other Observables.

Example of Subject as a Data Consumer:

```
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
import { ajax } from 'rxjs/ajax';

@Component({
  selector: 'app-api-example',
  template: `<h2>API Example with Subject</h2>`
})
```

```
export class ApiExampleComponent implements OnInit {
  ngOnInit() {
    const subject = new Subject<any>();

    // Subscribe to the subject
    subject.subscribe(data => console.log('API Response:', data));

    // Making an API call
    const apiObservable = ajax('https://randomuser.me/api/');
    apiObservable.subscribe(subject); // Subject consumes data from API Observable
```

Behaviour Subjects

Notes on Behavior Subject in RxJS

Definition

- **Behavior Subject**: A special type of subject that holds an initial value and emits that value to new subscribers if no new value has been emitted.

Key Differences from Normal Subject

1. **Initial Value**: A Behavior Subject can be initialized with a value that is emitted immediately upon subscription.
2. **Last Emitted Value**: If a new subscriber subscribes after values have been emitted, it receives the last emitted value instead of an undefined value.

Example Implementation

1. **Creating a Behavior Subject**

```javascript
import { BehaviorSubject } from 'rxjs';

// Create a Behavior Subject with an initial value of 100
const behaviorSubject = new BehaviorSubject(100);
```

2. **Subscribing to the Behavior Subject**

```javascript
behaviorSubject.subscribe(value => console.log('Subscriber 1:', value));
behaviorSubject.subscribe(value => console.log('Subscriber 2:', value));
```

3. Emitting Values

```javascript
behaviorSubject.next(2020); // Emit new value
```

- **Output**: Both Subscriber 1 and Subscriber 2 will log:

```yaml
Subscriber 1: 100
Subscriber 2: 100
Subscriber 1: 2020
Subscriber 2: 2020
```

4. Adding a New Subscriber

```javascript
behaviorSubject.subscribe(value => console.log('Subscriber 3:', value));
```

- **Output**: Subscriber 3 will log:

```yaml
Subscriber 3: 2020
```

5. Further Emission

```javascript
behaviorSubject.next(2023);
```

- **Output:**

```yaml
Subscriber 1: 2023
Subscriber 2: 2023
Subscriber 3: 2023
```

Summary

- **Behavior Subject** is useful for maintaining state and sharing values across components in Angular applications.
- Initial values are beneficial for ensuring subscribers always have meaningful data upon subscription.

Subscriber 1, Subscriber 2, Subscriber 3

```
// Create a BehaviorSubject with an initial value of 100
const subject = new BehaviorSubject<number>(100);

// Subscribers
subject.subscribe((data) => console.log("Subscriber 1: " + data));
subject.subscribe((data) => console.log("Subscriber 2: " + data));
subject.next(2020);
subject.subscribe((data) => console.log("Subscriber 3: " + data));
subject.next(2023);
```

RxJS (Reactive Extensions for JavaScript) is a library for composing asynchronous and event-based programs using **Observables**. It provides a powerful way to handle asynchronous data streams, making it easier to manage events, API calls, timers, and more.

observables works!