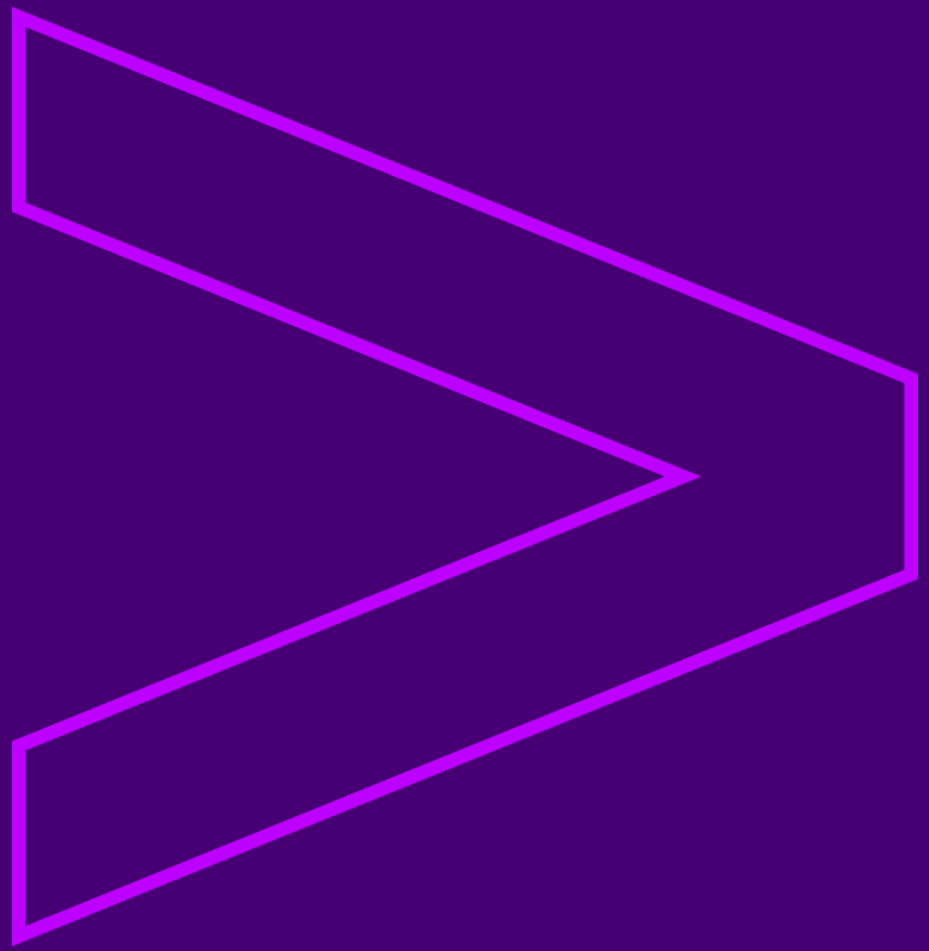
ANGULAR –

RXJS & OBSERVABLES



accenture

MODULE OBJECTIVES

At the end of this module, you should be able to:

- Understand the importance of Reactive Programming
- Use RxJS library to handle asynchronous data streams
- Write complex asynchronous code using Observables
- Apply RxJS operators to create, transform and filter Observables

AGENDA

- Reactive Programming
- Introduction to Streams
- Introduction to RxJS library
- Asynchronous code using Observables
- RxJS operators

Why Reactive Programming?

- Have you seen modern web apps that automatically trigger backend save operation while filling a form?
- Have you noticed Twitter showing other related accounts to follow once you follow a handle?
- Have you liked a page in Facebook that gets reflected in real time to other connected users?
- How to create such apps that handle real-time events and provide highly interactive experience to the users?
- Does the current imperative programming style provide tools to create such reactive systems?

accenture

Imperative Vs Reactive Programming

Imperative	Reactive
A = 1 B = 2 C = A + B B = 4 Result A = 1 B = 4 **C = 3**	A = 1 B = 2 C = A + B B = 4 Result A = 1 B = 4 **C = 5**

```
add ( A , B ) {
  return A + B;
}

C = add(1, 2);
```

The Reactive Manifesto

Responsive	Resilient	Elastic	Message driven
• React in a timely manner	• React to failure	• React to load	• React to events

accenture

Reactive Programming

- Reactive Programming is programming paradigm that deals with asynchronous data streams
- It is all about handling events and data flow
- It allows us to create a data stream, listen to that stream and react accordingly

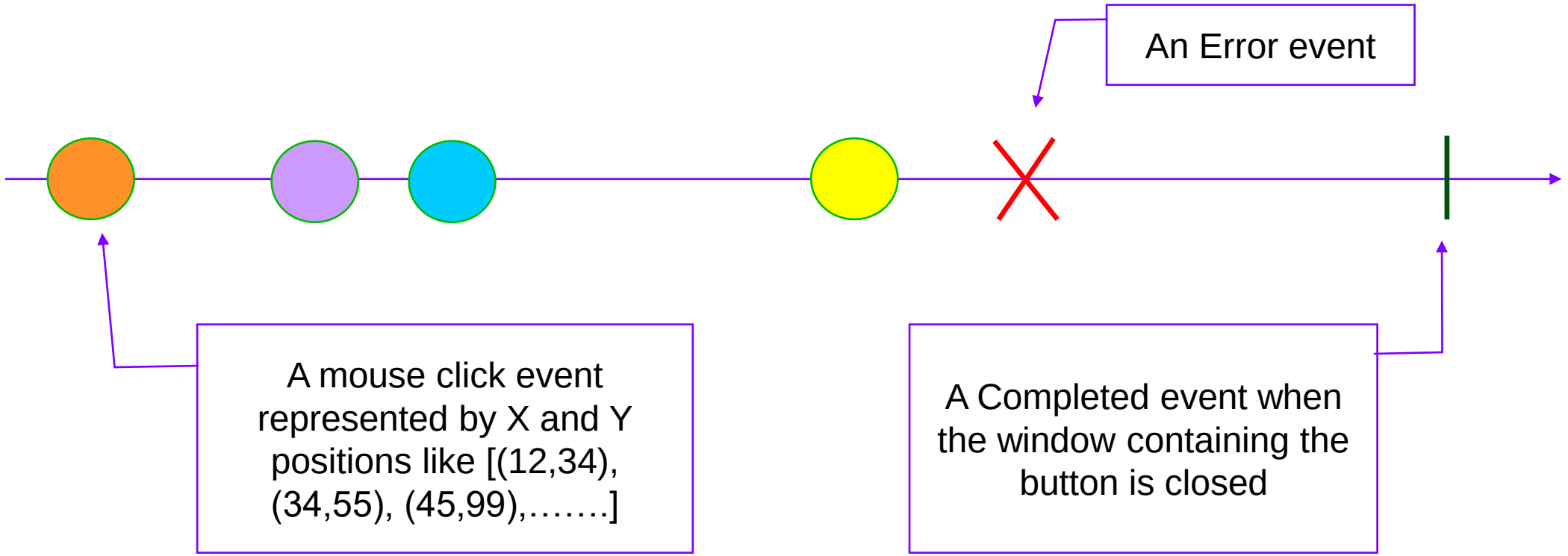
What is a Stream?

- A Stream is a sequence with ongoing events ordered in time
- Think of it like items on a conveyor belt processed as it flows over time
- A Stream can emit a **value**, an **error**, or a **completed** signal
- Ex) A stream of click events, a stream of keystrokes, a stream of API responses, a stream of Twitter feeds, a stream to represent a user filling in a form etc

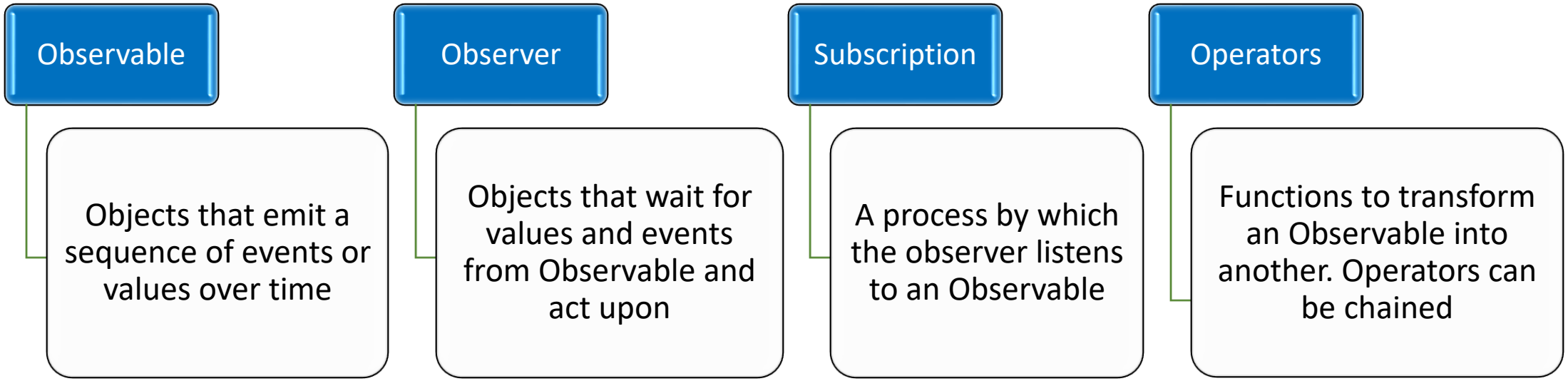
accenture

Example

- A stream based on user clicks of a button

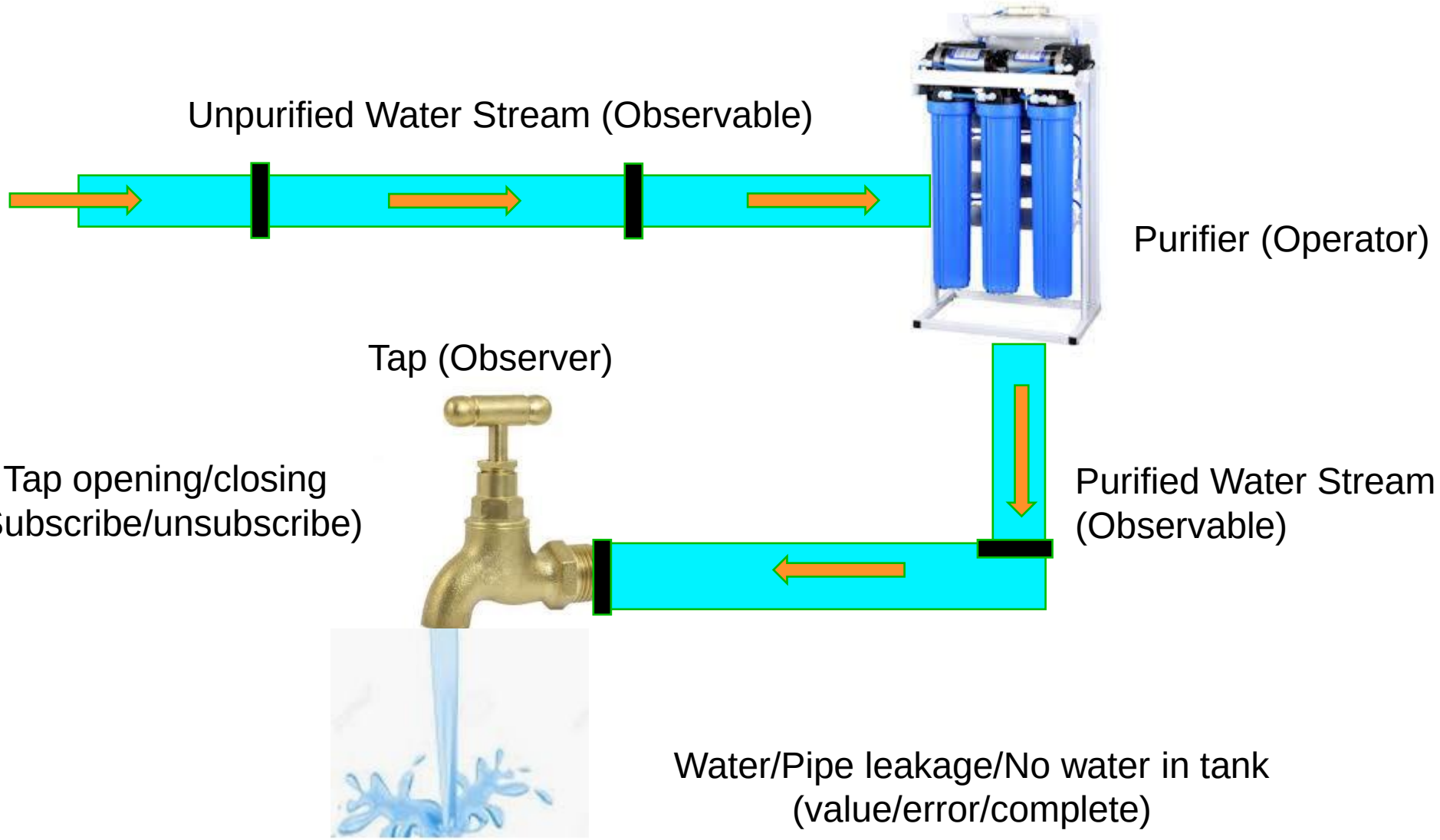


Building blocks of Reactive Programming



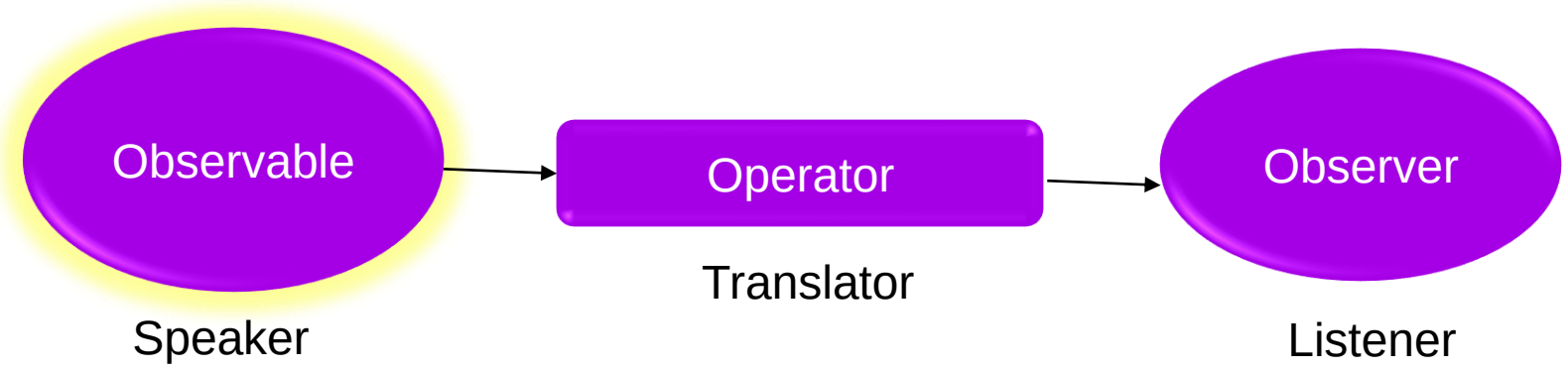
accenture

Real world analogy



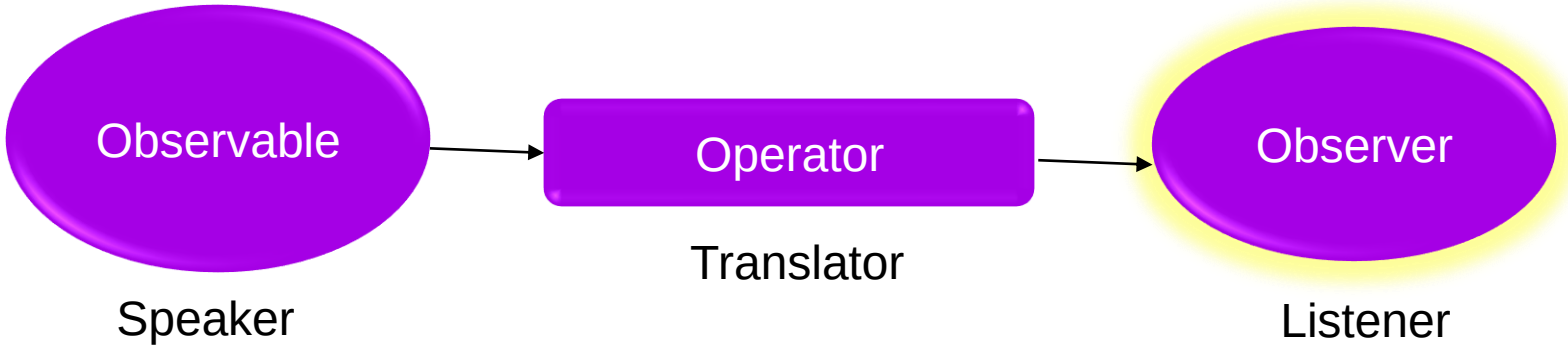
Observable

- Observable is the data source/stream
- It can emit just one value (like http request) or multiple values (like keystrokes or mouse movements)
- It can be *infinite* or *finite*. If *finite*, it emits completion event. It can also emit errors
- Observables are not yet a built-in feature of JavaScript but there is a proposal to add in ECMAScript



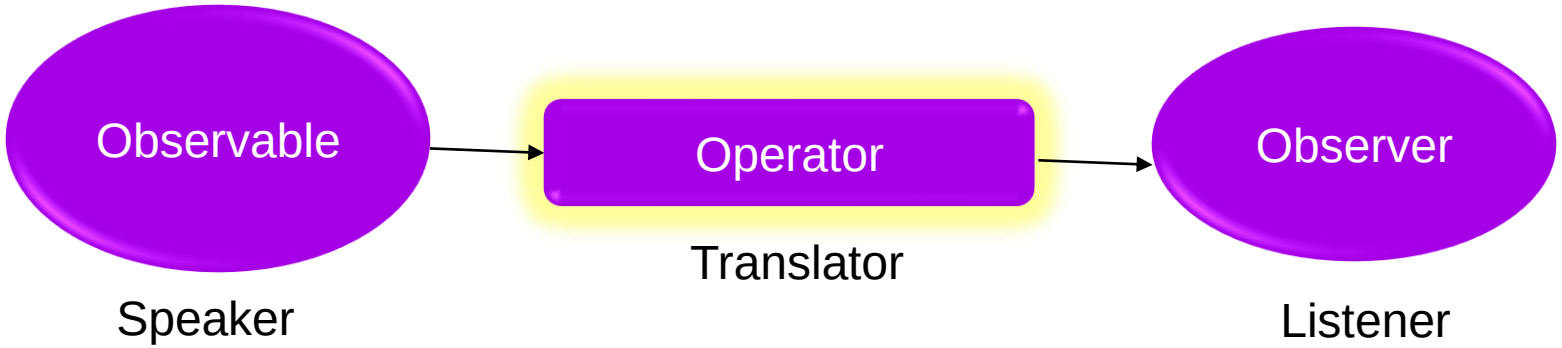
Observers/Subscribers

- Observers subscribe to the Observable and consume the emitted data stream
- Whenever the Observable emits the data, all registered subscribers will receive the data
- They also receive the error and completion events from the Observable



Operators

- Functions that transform the behavior of an existing observable stream in some way.
- When an operator is called on an observable, it returns a modified observable,
- We can chain multiple operators to transform an original stream to make it emit what we need



accenture

RXJS

Learning and Knowledge Management

Reactive Extensions (ReactiveX)

- Reactive extensions is a set of API that brings Reactive programing to different programming languages
  - Rx.NET, RxJava, RxScala, RxCpp, RxJS and so on

- Check http://reactivex.io/ and click on “Choose your platform”

- **RxJS** is the JavaScript implementation of the ReactiveX API

RxJS

- RxJS stands for **R**eactive E**x**tensions for **J**ava**S**cript
- An open source library for reactive programming that makes use of Observables
- It provides an implementation of the Observable type
- Makes it easy to write asynchronous and event-based code
- Contains utility functions for creating and working with Observables

Setting up RxJS

- Installing RxJS

```
npm install rxjs
```

Note: No need to install manually from Angular 6+ as it is now a mandatory dependency in angular-cli projects

- Importing the entire functionality of RxJS

```
import * as rxjs from 'rxjs';

rxjs.of(1, 2, 3);
```

- Importing only what we require from rxjs and rxjs/operators

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

import { map } from 'rxjs/operators';

of(1,2,3).pipe(map(x => x * x));
```

accenture

1. Creating Observables

- We can create a new Observable using **creational operators** imported from the rxjs library
  - From one value or an array of values (from, of)
  - From an event (fromEvent)
  - From timers (interval, range)
  - From multiple source observables (zip, merge, concat)
  - Etc

Example : A simple observable that emits 3 values

```
import { of } from 'rxjs';
const myObservable$ = of(1, 2, 3);
```

By practice, variables holding Observable are named with $ as suffix

accenture

2. Subscribing to Observables

- An Observable instance will begin publishing values only when someone subscribes to it.

- subscribe() method on the Observable instance takes 3 callbacks
  - "**Next**" callback: receives a value such as Number, String, Object, etc. (*Mandatory*)
  - "**Error**" callback: receives a JavaScript Error or exception (*Optional*)
  - "**Complete**" callback: no argument function called on complete notification (*Optional*)

Example : Calling the subscribe method with 3 callbacks

```
myObservable$.subscribe(x => console.log('Got a next value: ' + x),
                        err => console.error('Got an error: ' + err),
                        () => console.log('Got a complete notification') );
```

3. Pipeable Operators

- Pure functions that take an Observable as input and returns another Observable
- Imported from 'rxjs/operators' and applied using pipe() method of Observable
- Classification
  - Transformation operators (map, scan, switchMap etc)
  - Filtering operators (filter, distinct, skip, debounce etc)
  - Error handling operators (catchError, retry etc)
  - Utility operators (tap, delay etc)

Example : Using map operator to transform the stream

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

const myObservable$ =of(10, 20, 30);
myObservable$
  .pipe(map(num => num*num ))
  .subscribe(x => console.log(x))   // 100, 400, 900
```

4. Unsubscribing to Observable

- Calling unsubscribe() will stop listening to streams
- Unsubscribing is essential to avoid memory leaks

Example : Calling unsubscribe to cancel

```
const subscription = myObservable$.subscribe()

subscription.unsubscribe()
```

accenture

SAMPLE CODE

Learning and Knowledge Management

Examples – RxJS Operators (1/6)

Example : observable from an array

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

from([1,2,3])
.subscribe(x => console.log(x))  // 1, 2, 3
```

Example : observable from mouse click event

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

const myclicks$ = fromEvent(document, 'click')

myclicks$.subscribe(click => console.log(click))

// MouseEvent object logged to console every time a click occurs on the document
```

accenture

Examples – RxJS Operators (2/6)

Example : observable using of

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

const myObservable$ = of(1, 2, 3);  // emits the arguments and then completes.

// An alternate method - Create an observer object with 3 callbacks and pass to the subscribe function

const myObserver = {
  next: x => console.log('Next value got by Observer : ' + x),
  error: err => console.error('Error got by Observer : ' + err),
  complete: () => console.log('Completed signal got by Observer'),
};

myObservable$.subscribe(myObserver);
```

Predict the Output

accenture

Examples – RxJS Operators (3/6)

Example : observable from an interval and range

```
import { interval, range } from 'rxjs';

// An Observable that emits sequential numbers every specified interval of time

let numbers$ = interval(1000);
numbers$.subscribe(x => console.log(x));

// Observable that emits a range of numbers

let range$ = range(1, 10);
range$.subscribe(x => console.log(x));
```

accenture

Examples – RxJS Operators (4/6)

Example : Using pipe to chain filter and map

```
import { of } from 'rxjs';
import { filter, map } from 'rxjs/operators';

const squareOdd$ = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );

squareOdd$.subscribe(console.log);
// 1, 9, 25
```

accenture

Examples – RxJS Operators (5/6)

Example : Using merge to create an Observable that concurrently emits all values from the input Observables

```
import { fromEvent. interval } from 'rxjs';
import { merge } from 'rxjs/operators';

let myclicks$ = fromEvent(document, 'click');
let timer$ = interval(1000);
let clicksOrTimer$ = merge(myclicks$, timer$);
clicksOrTimer$.subscribe(x => console.log(x));

// timer will emit ascending values, one every second to console
// click logs the MouseEvent everytime the "document" is clicked
// The two streams are merged
```

accenture

Examples – RxJS Operators (6/6)

Example : Using catchError to catch errors on the observable by returning a new observable or throwing an error

```
import { of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

of(1, 2, 3, 4, 5).pipe(
    map(n => {
                if (n === 4) {
                    throw 'four!';
      }
     return n;
    }),
    catchError(err => of('I', 'II', 'III', 'IV', 'V')),
  )
  .subscribe(x => console.log(x));
  // 1, 2, 3, I, II, III, IV, V
```

MODULE SUMMARY

Now, you should be able to:

- Understand the importance of Reactive Programming
- Use RxJS library to handle asynchronous data streams
- Write complex asynchronous code using Observables
- Apply RxJS operators to create, transform and filter Observables

QUESTIONS

THANK YOU