Understanding Observables

Module Introduction

- **Observables** are data sources
- Hails from **RxJS**
- **Observable Pattern**
  - We have an **observable** and an **observer**
  - Between them, there's a stream that contains potentially many data emissions from the **observable**
- **Observables** can be triggered **manually** (like a button click) or **automatically** (like an HTTP request)
- **Observers** handle data packages in three ways
  - **Handle Data**
  - **Handle Error**
  - **Handle Completion**
- **Observables** are used to handle **asynchronous** tasks

Analyzing Angular Observables

- **Observables** are constructs in which we **subscribe** to be informed about changes in data
  - We see thing with **routing** subscriptions as this maintains dynamic data comprehension

Getting Closer to the Core of Observables

- Observables aren't native to JavaScript or Typescript → They're added via a package named **RxJS**
- We must **subscribe** to observables to access their data
- Some observables finish after emitting one data value; others may emit values endlessly (even from other components)
  - The latter may lead to memory leaks
  - To squash the latter, we must **unsubscribe** to the observable
    - Create a **Subscription** object and assign your **subscribe** call to it

        - Destroy the **Subscription** object by **unsubscribing** to it in **ngOnDestroy**

Building a Custom Observable

- Import **Observable** from **rxjs**
- Create a new variable and assign to it **Observable.create(<anonymous-function>)**
- **create** takes an anonymous function (with parameter **overserver**) which performs its desired functionality in the body
- The anonymous function's body calls **observer.next()** to get the next value in the stream
    - **observer.error** to get the error
- **NOTE:** This approach is now deprecated → We should create **new Observable()**
    - Consult the **repo** for deprecated and modern implementations

Errors and Completion

- Similar to the **next** method, we have an **error** and **complete** methods
- **Error**
    - Inside of the **error** method, we should throw a **new Error** object that contains an error message
    - Encountering an error kills the observable
    - We can access the **error** response by giving and accessing **subscribe**'s **error** parameter
- **Complete**
    - This terminates the observable, ceasing all further emissions
    - This does not fire if you encounter an error
    - To react to the completion, we include an anonymous function after **data** and **error** in **subscription**
    - We don't need to unsubscribe from a **complete**d observable
- **NOTE:** This approach is now deprecated → **Subscribe** with multiple elements is deprecated
    - Consult the **repo** for deprecated and modern implementations

Observables and You!

- When you **subscribe** and establish your **handler functions**, RxJS merges them into one object and it (the observer) to the observable
- The observable responds with new data, errors, etc

Understanding Operators

- Observers get data from observables via subscriptions
- If we want to transform this data before receiving it, we can pass it through RxJS **operators**
  - **Subscribing** takes the result of these operators
- To use an operator, we must first apply the **pipe** method to the observable
  - From there, we populate the **pipe** method with our desired operators, each as their own argument
  - For instance, we can apply the **map** operator → Returns un/altered data
- We must subscribe to this piped data to use it

Subjects

- **Subjects** are another RxJS tool
- They're generic types in which you define what kinds of data are emitted
- Unlike **Observables**, you can call the **next** method from outside
- **Subject.next()**, similar to **EventEmitter**, emits events
- Used in conjection with **subscriptions** for **cross component communication**
- Using an **EventEmitter** requires **unsubscribe()**, while using **Subject** doesn't