Observables:

* In the routing section, we learned about using params, queryParams and fragments in logic to navigate to dynamic URLs and to fetch data from the URLs. In order to get the data, we subscribed to few Angular provided observables for each of these.
* Now we will learn a bit more about them. Angular uses a 3rd party API named rxjs to work with observables. These emit a response when data changes occur due to events, user-interaction, http requests, etc.
* There are 3 possible responses - resultant data, error and completion. We write our custom logic to handle these responses.
* Observables can quickly introduce memory leaks if not cleaned up. We can do this by unsubscribing from it in the ngOnDestroy function. All custom observables have to be cleaned up. Angular takes care of its own observables.
* The simplest way to write a custom observable is using the interval function in rxjs. This works similar to setInterval. It takes the time in milliseconds as an argument. It outputs the count everytime it is triggered.

Custom Observable:

* To create a custom observable, we need to import Observable from rxjs and call its create() method (This is deprecated. We use new Observable() now).
* This method takes a function with an observer as an argument. The Observer has a lot functions with itself but we will primarily user the next, error and complete.
* The next function will take the value that we need to pass to the actual observer as a parameter. The error function takes an Error object as a parameter. The complete function doesn’t take any parameters.

Operators:

* In many cases, we don’t need the raw data as is. We need to perform some level filtering or data transformation before we use the data. Using our current approach, we can do this in the subscribe function.
* If the transformation is complex, a more elegant way of doing this is using Operators. We can apply operators to the data by using the pipe() function. This function takes multiple comma-separated operators that transform the data and execute them in the order they are added.
* One such operator is the map operator, imported from rxjs/operators. This takes each entry outputted by the observable and transforms it according to our logic. filter is another operator which is used to filter out data based on a condition.

Subject:

* A Subject works just like an event emitter but is *optimized for inter-component event communication only*. The code is pretty much the same, except we use Subject (imported from rxjs) instead of EventEmitter.
* Here, instead of emit, we call the next function to actually trigger the event and pass data.
* Just like an observable, we have to unsubscribe on navigation.