**Section 13: Understanding Observables**

**Section 13: Lecture 158//Module Introduction**

1. In the last module about routing we had first contact with observables. Now, we will dive deeper into observables in this section. We have a separate course on observables so we will not dive that deep into them here.
2. Here we will understand what observables are and where we can use them for, and why angular uses them.
3. What is an observable?

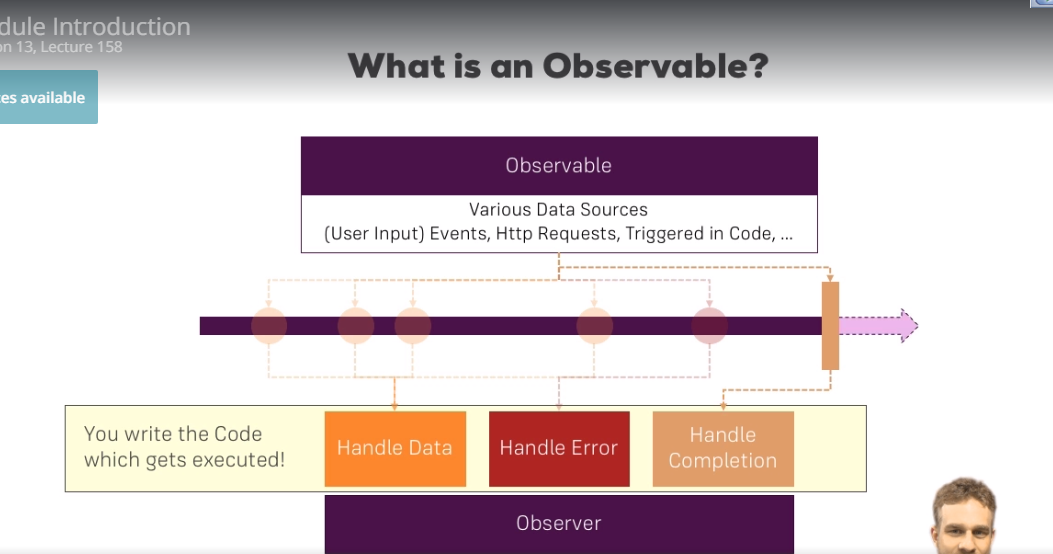
**An observable can be thought of as a data source**, now, in our angular project an observable basically is an object we import from a 3rd party package i.e. RXJS

1. The observable is implemented here such that it follows the observable pattern. So, we have an observable and we have an observer - in between we have a steam i.e. a timeline and on this timeline we have multiple events emitted by the observable or data packages you could say - emitted by observable depending on the data source of that observable.
2. So, an observable could emit data because you triggered it to do so – we can also do that programmatically i.e. it can be linked to a button - therefore when the button is clicked an event or a data package is emitted automatically or as the angular Http service does it - its connected to Http requests. So, when the response returns - the response is emitted as a data package.
3. There are dozens of other data sources too, we will have a look – where to find more soon.
4. So, as we observed - the other part is observer 🡪 this actually is your code, you could say. Its, the subscribe function we saw earlier or at least it has something to do with that.
5. Using observer, we have three ways of handling data packages

Handle Data

Handle error

Handle Completion

1. As shown above – by using observer we can handle normal data, we can handle the error or we can handle the completion of the observable. Because these are the 3 types of the data packages we can receive.
2. In these boxes our code gets executed, so, we can determine what shall we if we receive a new data package, what should happen if we receive an error , what should happen when the observable eventually completes - Note: An Observable doesn’t have to complete.
3. There are observables, for example, hooked up to normal button which neve completes – how would you know, when it completes?
4. Other observables such as http have a clear end and will complete eventually. Because once the response is there, what else should happen? - it’s done
5. This is how the observable pattern generally works and of course we use it to handle asynchronous tasks - because all these data sources here, user event triggered in your code or a HTTP request are asynchronous tasks - you don’t know when they will happen and you don’t know how long they will take. So if you execute your normal application code - you don’t want to wait for such an event and you don’t want to wait for such a HTTP requests, because that would block your program/block your logic.
6. Therefore we need methods for handling such an asynchronous tasks, historically you might have used callbacks and promises - it’s not necessarily bad to use them. Observable is just a different approach of handling that different alternative and angular embraces observables which is why I chose to explain these as angular uses them a lot.
7. Observables have one major advantage i.e. their operators which I will show later in this section too.
8. Back to this slide, we have got our observable and our observer with our free buckets or we can say with our free hooks where our different code is executed depending on the type of package we receive.
9. And then an observable may emit a couple of normal data packages, It might emit an error or it might get completed and the respective code is then executed.
10. 

**Section 13: Lecture 159// Analysing a Built-in Angular Observable**

1. This is a very simple application which we have prepared, which has 3 links. There is one home and 2 user links. If we click user one, then it would open at /user/1 and 2 at /user/2.
2. ID here in the routes gets updated depending on the route that we are using.
3. And of course this is one and the same user component we are using here in the background and we are using an observable. The parents observable Angular or ships within the router package to handle the change of these router parameters because just as a refresher angular doesn’t re-render the whole component just because the parameter changed.
4. Therefore, it uses observables to still give us a chance to react to this updated ID and this is what we do with the observable which angular offers us here now and in the last section we implemented this code and I think it works kind of intuitive because you can basically accept that route params – are some kind of strange asynchronous object and you subscribe to any events that might send you, in this case, updated parameters and then use simple handle these parameters here in those callback we pass to the Subscribe method – keep in mind that this is just argument passed to the subscribe method here this anonymous function where we handle the parameters and then we simply extract the ID in this case .
5. Now, as mentioned we are using observable here – so, which part of the slide, with this code you are basically be the sending part or the receiving part - while the receiving part ofcourse because here we handle the data.
6. The data is sent by as angular, we don’t do this. You could argue we kind of do it by implementing our links here, but that will only kind of trigger it.
7. The emitting of the data will be done by angular which recognizes that we clicked such a router link.
8. We will extract the data from this rattling directive and then will basically navigate us.
9. And at this point of time it will then also push this new data package through this parent observable.
10. So here in subscribe method this actually is our observer part to be precise and to stay in our X chase terms.
11. This is our subscriber – but subscriber here basically is the observer. Therefore, we could implement more callbacks then just one we only implement the handle the normal data call back but the subscribe method always takes the 3 arguments and our ID already shows in there.
12. We could also implement an anonymous function which gets executed in the case of error and one which gets executed if the observable gets completed.
13. Now on these routing observable here that’s not very practical, it doesn’t make much sense because the parents here will not fail and they will not complete. So, the first call here make the most sense.
14. We will soon dive into creating our own observables so that we can actually also see the other possible hooks.
15. We are subscribing to such an observable which wraps that data source being code in angular - which you could say emits a new parameter whenever we click a new link.
16. So, in the end our click is the triggering event but again angular sets up an observable and then pushes it to give us a new piece or a new data package and then we use it here in subscribe function with our subscriber; the subscriber basically just is this free method we pass through subscribe set up which is the Observer.
17. So that is how we use it already as angular does what happens behind the scenes.
18. Now, let’s implement our own observable to understand a bit better how it really works.

**Section 13: Lecture 160//Building and using a first sample Observable**

1. In the last lecture we had a look on this built in observable, now let’s build our own one we will do this in our home component, here in ngOnInit().
2. Now, there are tons of different ways of creating observable and we will soon see where to dive deeper and learn more. Let’s create a very simple observable.
3. So, our own observable here simply should emit some numbers as sending numbers to be precise and that in a fixed interval, so we will simply name it - myNumbers - and then to create it we need to import the observable.
4. Now, this is imported from rxjs and they are actually from rxjs/Observable; now if this observable object, we are not creating one yet, but there are a couple of helper methods which will create one for us for example - interval();
5. Now, interval is very simple method for creating a new observable, inside the method we will pass the number – and that would be the milliseconds it should wait for, when emitting data automatically, so it will do the data emitting automatically for us.
6. Here I will pass 1000 to emit the new piece of data every second and for that to work we need to import one other import i.e. import rxjs/Rx
7. If this import looks strange to you – this is because this package has its own logic of providing packages for import.
8. Chances are you don’t need to add imports too often because you are going to add imports too often because you are going to work with the observables provided by angular.
9. In general though especially the zeken import here is one you will need whenever you work with some of the Observable operators.
10. So, now if the observable is setup we can now subscribe to it. Now, keep in mind you can pass 3 arguments to subscribe a call back for handling normal data, a call back for handling error , a callback for handling the completion of the observable.
11. Now, this specific observable here will not fail and will not complete. We will soon see another example.
12. So, here we know that for the first callback for handling normal data we would get a number. And we can simply console log this number to see it.
13. So, if this in place, if I now save and we go to our running application lets open up the developer tools and navigate to the home page and there you see each second a new number is printed to the console. This is our own observable working here. **The data source is a normal timer** simply using a little utility function which is available in the rxjs package.
14. Now, why am I using such a utility function, because setting up an observable from scratch is a rather complex process but nonetheless let’s move on and dive deeper into observables.
15. [home.component.ts](http://home.component.ts):
16. import { Component, OnInit } from '@angular/core';
17. import { Observable } from 'rxjs/Observable';
18. import 'rxjs/Rx';
19. @Component({
20. selector: 'app-home',
21. templateUrl: './home.component.html',
22. styleUrls: ['./home.component.css']
23. })
24. export class HomeComponent implements OnInit {
25. constructor() { }
26. ngOnInit() {
27. const myNumbers = Observable.interval(1000);
28. myNumbers.subscribe(
29. (number: number) => {
30. console.log(number);
31. }
32. );
33. }
34. }

**Section 13: Lecture 161//Building and using a custom Observable from Scratch**

1. In the last lecture we created a simple observable here which would automatically add some numbers, now that we saw this let’s build one from scratch here. Here I want to create one observable which will fire after two seconds and after 4 seconds and which will also fail after let’s say 5 seconds and alternatively we will also observe how it works if it completes.
2. So, lets create a new observable – myObservable – maybe any name you like and here we will still use observable object, but now the create method, now create takes function as an argument - and this function should hold your asynchronous code. Let us see what we mean.
3. So, we will pass an ES6 arrow function in here and as we mentioned here, we will have our logic.
4. This function actually receives an argument an observer of type observer – don’t mistake here with the subscriber in the subscription function. The Observer we pass here will be our final observer, but we pass it to the function i.e. to this anonymous function which will make up our observable.
5. So, we will basically tell the observer when we will receive which data which we ofcourse need to do.
6. When we then subscribe to it and use the observer to react the data. The observer will know when to react because the observable tell it. Now, this is the general setup which is mostly hidden because you subscribe to an observable which already exists.
7. But if you build one from scratch you actually need need to build this bridge from observable to observer. So, let’s do it here so it becomes a bit clearer. So, in this callback function the body of it we will set a normal timeout because after 2 seconds I said I want to emit my first data package.
8. So, you are in this set a normal JavaScript function in the callback which will be executed after 2 seconds.
9. I want to use the observer we have available inside of this overall function here and call next method.
10. Next method emits a normal data package. That’s just something you kind of can keep in mind here if you build an observable from scratch; next just pushes the next data package and here you can pass any data you want. We will simply pass here the data ‘first package’
11. So a string. Now, I will copy that setTimeout function and create another one – this one should form a fire after 4 seconds. Here I want to execute or pass the second package, so, another string.
12. Note that we don’t nest these timeouts - so they are started at the same time, and after 5 seconds from the beginning it should fail.
13. So, instead of next I now can call error and a potential error message.
14. Now, this is our observable, now, as before lets subscribe to it; so, here I will set up the subscribe function on my own observable and now it will actually pass all 3 arguments.
15. The first argument being the callback which gets triggered on normal data package which doesn’t fail. So here I will get some data which I know will be a string because I’m the creator of the observable and we know that we will pass a string data here.
16. And ofcourse every observable you use even if it’s from a 3rd party provider from a third party package should somewhere tell you which kind of data you can expect in the documentation.
17. So, here we will simply log this data to the console. Now, I am going to replicate this the second call back will hold this error. And here therefore we will output the error, which will also be a string.
18. Now, I know the error will also be a string. So that is alright. In the third function we won’t receive any data because we can’t pass one here I will simply say completed - this won’t be trigged as of now. So, with this I do have all my inline anonymous functions here.
19. Which are the free functions I can actually provide and one thing I should do now? So that this actually works is - on the observer I have to define which kind of data it will emit so I know It will be a string here because this is the data I pass here all the time on next and on error.
20. So, with that in place save this and reload our page and watch what happens here in the console. After 2 seconds we see this, after 4 seconds we see this and after 5 seconds we see – this does not work.
21. So, this is the data our own observable emits and again we told it emit this because we run our code which run asynchronously wrapped in this function we passed to the observable when we created.
22. Now that is basically the same angular does for you when it gives you an observable. It sets this up behind the scenes.
23. But here you see how you can create an observable on your own. The data which we can see in the console is the data emitted by our own observable. Now just for completeness sake I will comment out the error case here and say after 5 seconds it actually is completed. So, I will call complete and I will also set another time out which after 6 seconds would emit a third package.
24. This function we passed to the observable when we created, now that is basically the same angular. Now, let’s complete the observable and let’s see if complete that here.
25. So, this reloaded first package second package and complete it and we’d never see the third package.
26. Even though I do call the next here a third time after 6 seconds this never arrives because before this actually gets fired – we complete our observable and therefore its done – its dead. We no longer use it.
27. [home.component.ts](http://home.component.ts)
28. import { Component, OnInit } from '@angular/core';
29. import { Observable } from 'rxjs/Observable';
30. import 'rxjs/Rx';
31. import { Observer } from 'rxjs/Observer';
32. @Component({
33. selector: 'app-home',
34. templateUrl: './home.component.html',
35. styleUrls: ['./home.component.css']
36. })
37. export class HomeComponent implements OnInit {
38. constructor() { }
39. ngOnInit() {
40. // const myNumbers = Observable.interval(1000);
41. // myNumbers.subscribe(
42. // (number: number) => {
43. // //console.log(number);
44. // }
45. // );
46. const myObservable = Observable.create((observer: Observer<String>)=>{
47. setTimeout(
48. () => {
49. observer.next('first package');
50. }
51. , 2000);
52. setTimeout(
53. () => {
54. observer.next('Second package');
55. }
56. , 4000);
57. setTimeout(
58. () => {
59. // observer.next('This does not work');
60. observer.complete();
61. }
62. , 5000);
63. setTimeout(
64. () => {
65. // observer.next('This does not work');
66. observer.next('third package');
67. }
68. , 6000);
69. });
70. myObservable.subscribe((data: string)=> {
71. console.log(data);
72. }, (error: string)=>{
73. console.log(error);
74. },()=>{
75. console.log('completed');
76. });
77. }
78. }

**Section 13: Lecture 162// Unsubscribe!**

1. In the last two lectures we saw how we can create our own observable and we created one which would one which would run infinitely.
2. We have the interval here and the one which is completed now. Now, what actually happens if we have the first case is infinitely running absorbable I will uncomment it in again and then again we will navigate away from this component.
3. But we do have a subscription which is active well let’s see what happens if I reload this.
4. If we reload this it will start counting and we will get the packages from the other observables and now we switch to page and it still accounts, even though I switched to page, even though the home component got destroyed in the background. This can be a severe problem because of your observable still has an active subscription. And the observable is not the observable which is completed automatically as disapproval isn’t.
5. Well then you’re creating a memory leak because you still have a subscription which you probably are not interested in anymore and still receive the data and handle that data and now imagine this case for a couple of observables or one hundreds of thousands of observables.
6. Therefore you should always make sure that you unsubscribe if you kind of leave the area where you handled this observable.
7. So in the example here with our component we should actually implement the onDestroy lifecycle hook.
8. In ngOnDestroy() we should unsubscribe the observable. I will actually store my subscriptions and not constants but in properties of this class.
9. So, my numbers observables subscription should be stored in one property and this will be of the type subscription.
10. And then the same for my custom observable with these two properties created I can now save my subscriptions
11. Now assign the observables to these properties, with the subscription stored we can go to ngOnDestroy hook and here I simply want to clear them, so I will call unsubscribe. And that’s the cool thing about this observable 3rd party package - it gives us this unsubscribe method on our subscriptions, so we can cancel them and so that we can make sure that we clean up our observables subscriptions.
12. With this in place let us go back to home, now we see in the console that the counter starts and once we navigate away then the counter stops. So, this successfully unsubscribes and cleans up the memory.
13. So, we definitely need to do this on our own observables and probably as a good practice on all observables, though we know that the observables built into angular clean themselves automatically but it is a good practice to clean themselves own your own.
14. You get into the habit and you make sure you don’t forget it on observables as there is no automatic clean-up is provided.
15. [home.component.ts](http://home.component.ts):
16. import { Component, OnInit, OnDestroy } from '@angular/core';
17. import { Observable } from 'rxjs/Observable';
18. import 'rxjs/Rx';
19. import { Observer } from 'rxjs/Observer';
20. import { Subscriber, Subscription } from 'rxjs/Rx';
21. @Component({
22. selector: 'app-home',
23. templateUrl: './home.component.html',
24. styleUrls: ['./home.component.css']
25. })
26. export class HomeComponent implements OnInit, OnDestroy {
27. numberObsSubscription: Subscription;
28. customObsSubscription: Subscription;
29. constructor() { }
30. ngOnInit() {
31. const myNumbers = Observable.interval(1000);
32. this.numberObsSubscription = myNumbers.subscribe(
33. (number: number) => {
34. console.log(number);
35. }
36. );
37. const myObservable = Observable.create((observer: Observer<String>)=>{
38. setTimeout(
39. () => {
40. observer.next('first package');
41. }
42. , 2000);
43. setTimeout(
44. () => {
45. observer.next('Second package');
46. }
47. , 4000);
48. setTimeout(
49. () => {
50. // observer.next('This does not work');
51. observer.complete();
52. }
53. , 5000);
54. setTimeout(
55. () => {
56. // observer.next('This does not work');
57. observer.next('third package');
58. }
59. , 6000);
60. });
61. this.customObsSubscription = myObservable.subscribe((data: string)=> {
62. console.log(data);
63. }, (error: string)=>{
64. console.log(error);
65. },()=>{
66. console.log('completed');
67. });
68. }
69. ngOnDestroy(){
70. this.customObsSubscription.unsubscribe();
71. this.numberObsSubscription.unsubscribe();
72. }
73. }

**Section 13: Lecture 163//Where to learn more**

1. In the last lectures we learnt to create and unsubscribe the observables. For more information one needs to refer the official documentation of the RxJS package.
2. [www.reactivex/rxjs/class](http://www.reactivex/rxjs/class)

**Section 13: Lecture 164//Using subjects to pass AND Listen to Data**

1. Here we will dive into the operators provided by the observables but also into another object and that is the – Subject.
2. A Subject is basically like an observable but it allows us to conveniently push it to emit a new data during your code which might a bit more what we are looking for.
3. So, let’s see how it works; we will create a new service for this; here we will create a new property userActivated which will be a new subject
4. This subject that we created here allows us to do some cool things; first of all we will provide the service in the app.module.ts.
5. Create a new click method in the .ts file and the inject the service in the constructor. We don’t need to provide the service here because we provided it in the app module.
6. A zup check is an observable and an observer at the same time. This is why we can conveniently call next method here and I can pass value – and here we will simply send the user id as an argument.
7. By doing this we are pushing a new data package which contains this id and now in my app component let’s say I want to display activated after user if the user was activated.
8. In the home component we will use the string interpolation to check if the user 1 actually went to the property which doesn’t exist yet.
9. So, we will add these properties to the app component and also add the string interpolation in the app.component.html.
10. Subject object is observable and observer at the same time. It is a kind of comparable to event emitter which ships with angular and actually event emitter is built on such a subject.
11. It is a good practice to use subject instead of the event emitter. So, next time if we need to implement the cross component communication we need to implement a subject.
12. Next is the method to push a new value and subscribe is used to consume it.
13. app.component.ts
14. import { Component, OnInit } from '@angular/core';
15. import { UsersService } from './user.service';
16. @Component({
17. selector: 'app-root',
18. templateUrl: './app.component.html',
19. styleUrls: ['./app.component.css']
20. })
21. export class AppComponent implements OnInit {
22. user1Activated = false;
23. user2Activated = false;
24. constructor(private usersService: UsersService){
25. }
26. ngOnInit(){
27. this.usersService.userActivated.subscribe(
28. (id: number)=>{
29. if(id===1){
30. this.user1Activated = true;
31. }else if (id === 2){
32. this.user2Activated = true;
33. }
34. }
35. );
36. }
37. }

14. app.component.html

<div class="container">

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<a routerLink="/">Home</a>

<a [routerLink]="['user', 1]" {{ user1Activated ? '(activated)' : '' }}>User 1</a>

<a [routerLink]="['user', 2]" {{ user2Activated ? '(activated)' : '' }}>User 2</a>

</div>

</div>

<hr>

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<router-outlet></router-outlet>

</div>

</div>

</div>

15. user.component.ts:

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

import { ActivatedRoute, Params } from '@angular/router';

import { UsersService } from '../user.service';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css']

})

export class UserComponent implements OnInit {

id: number;

constructor(private route: ActivatedRoute, private userService: UsersService) { }

ngOnInit() {

this.route.params

.subscribe(

(params: Params) => {

this.id = +params['id'];

}

);

}

onActivate(){

this.userService.userActivated.next(this.id);

}

}

16. user.component.html:

<p>User with <strong>ID {{ id }}</strong> was loaded</p>

<button class = "btn btn-primary" (click)="onActivate()">Activate!</button>

17. user.service.ts:

import { Subject } from "rxjs/Subject";

export class UsersService{

userActivated = new Subject();

}

`

1. app.module.ts:
2. import { BrowserModule } from '@angular/platform-browser';
3. import { NgModule } from '@angular/core';
4. import { AppComponent } from './app.component';
5. import { HomeComponent } from './home/home.component';
6. import { UserComponent } from './user/user.component';
7. import { AppRoutingModule } from './app-routing.module';
8. import { UsersService } from './user.service';
9. @NgModule({
10. declarations: [
11. AppComponent,
12. HomeComponent,
13. UserComponent
14. ],
15. imports: [
16. BrowserModule,
17. AppRoutingModule
18. ],
19. providers: [UsersService],
20. bootstrap: [AppComponent]
21. })
22. export class AppModule { }

**Section 13: Lecture 165//Understanding Observable Operators**

1. Subject is very helpful in creating cross component communication or your own event emitter – you can trigger it from anywhere in your code an subscribe to it from anywhere in your code.
2. Now, let’s look at some of the operators that RxJS offers you, we can find the list in the official documentation in the observable part.
3. What are operators?
4. These operators allow you to transform the data you receive to something else and still stay inside the observable world.
5. In the [home.component.ts](http://home.component.ts) were we subscribe to our numbers – here we can also chain an operator – so this is chain to the observable interval method.
6. The operators can be used on any observable; here we will use the map method; what does map do?
7. Map simply maps the data that we get to a new observable with any transformation of our choice, so map takes a function as an argument. In this function we will get the data and then it should return the transformed data.
8. [home.component.ts](http://home.component.ts):
9. import { Component, OnInit, OnDestroy } from '@angular/core';
10. import { Observable } from 'rxjs/Observable';
11. import 'rxjs/Rx';
12. import { Observer } from 'rxjs/Observer';
13. import { Subscriber, Subscription } from 'rxjs/Rx';
14. @Component({
15. selector: 'app-home',
16. templateUrl: './home.component.html',
17. styleUrls: ['./home.component.css']
18. })
19. export class HomeComponent implements OnInit, OnDestroy {
20. numberObsSubscription: Subscription;
21. customObsSubscription: Subscription;
22. constructor() { }
23. ngOnInit() {
24. const myNumbers = Observable.interval(1000)
25. .map(
26. (data: number)=>{
27. return data\*2;
28. }
29. );
30. this.numberObsSubscription = myNumbers.subscribe(
31. (number: number) => {
32. console.log(number);
33. }
34. );
35. const myObservable = Observable.create((observer: Observer<String>)=>{
36. setTimeout(
37. () => {
38. observer.next('first package');
39. }
40. , 2000);
41. setTimeout(
42. () => {
43. observer.next('Second package');
44. }
45. , 4000);
46. setTimeout(
47. () => {
48. // observer.next('This does not work');
49. observer.complete();
50. }
51. , 5000);
52. setTimeout(
53. () => {
54. // observer.next('This does not work');
55. observer.next('third package');
56. }
57. , 6000);
58. });
59. this.customObsSubscription = myObservable.subscribe((data: string)=> {
60. console.log(data);
61. }, (error: string)=>{
62. console.log(error);
63. },()=>{
64. console.log('completed');
65. });
66. }
67. ngOnDestroy(){
68. this.customObsSubscription.unsubscribe();
69. this.numberObsSubscription.unsubscribe();
70. }
71. }

**Section 13: Lecture 166//Wrap up**

1. This was only scratching the surface of the observables, if we go to official documentation there are a lot of features that we have not covered here, a lot of operators.

==========================END OF SECTION=========================================