
Introduction to RxJS

Day 01

RxJS is a library for reacting to events and data. At the core is the Observable.

There are other types like Observers, Schedulers, Subjects and operators. All of these functions handle asynchronous events as collections of data.

ReactiveX is a combination of the Observer and the Iterator patterns. The library is based on functional programming which when applied to collections, provides an elegant way of managing sequences of events.

Events such as key presses, mouse movements, touch gestures, timer events, remote API calls, or single whole numbers can be thought of as a stream of data. It is all data to be consumed. Streams in this way can be considered an array that is always changing..

Part 1 – Environment Setup (Node.js & HTML)

Part 2 – Warm Up Examples

Part 3 – Pipeable Operators

Part 4 – Special Observable - Subject

Part 5 – Schedulers

Part 6 – Schedulers Use Case

Part 7 – combine Latest()

Part 8 – merge(), mergeMap(), forkJoin(), concat(), zip()

Part 9 – Mouse Capture

Part 1 – Environment Setup (Node.js & HTML)

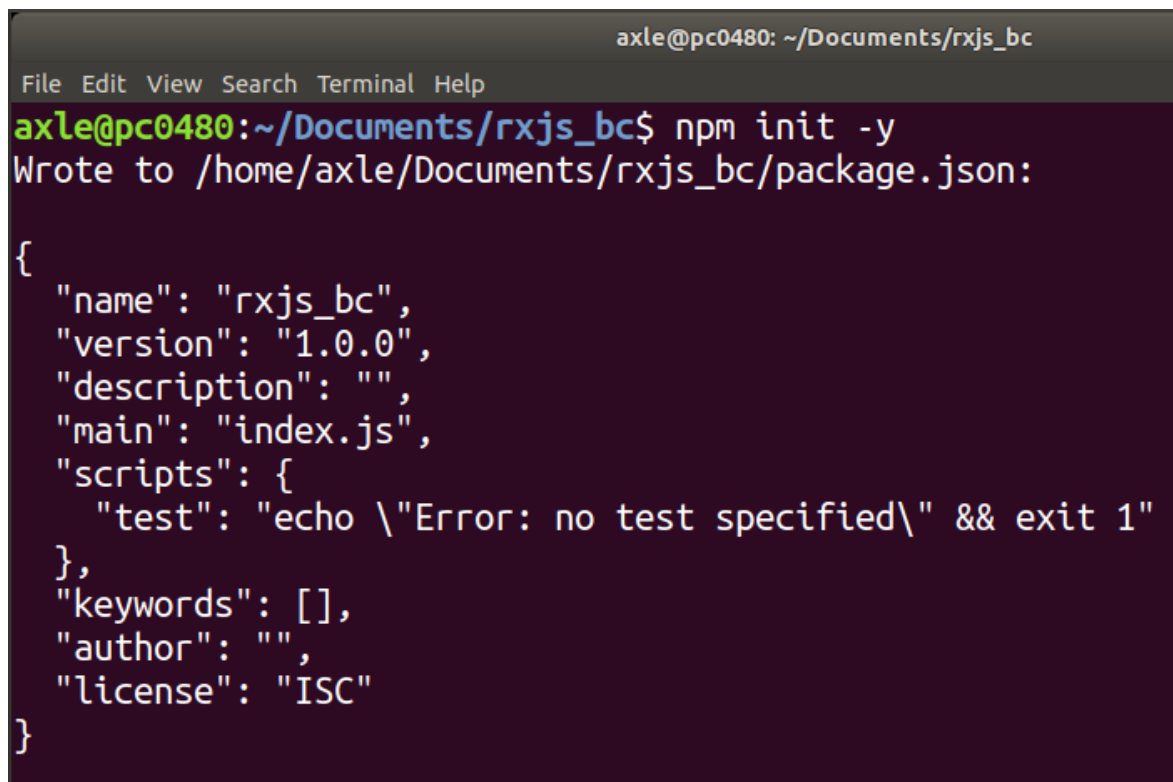
There will be two setup environments, **Node.js** and **HTML**. In this first set of instructions we will setup our environment on a Linux OS which already had Node installed. Later in the bootcamp we will use a **TypeScript** environment and we finish by building a boiler-plate Angular app.

Node.js Environment Setup on Linux

Choose or make a folder on your OS. From that folder run the following commands in order:

```
npm init -y  
npm i rxjs  
touch index.js
```

The first command will create a package.json file.



```
axle@pc0480: ~/Documents/rxjs_bc  
File Edit View Search Terminal Help  
axle@pc0480:~/Documents/rxjs_bc$ npm init -y  
Wrote to /home/axle/Documents/rxjs_bc/package.json:  
  
{  
  "name": "rxjs_bc",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

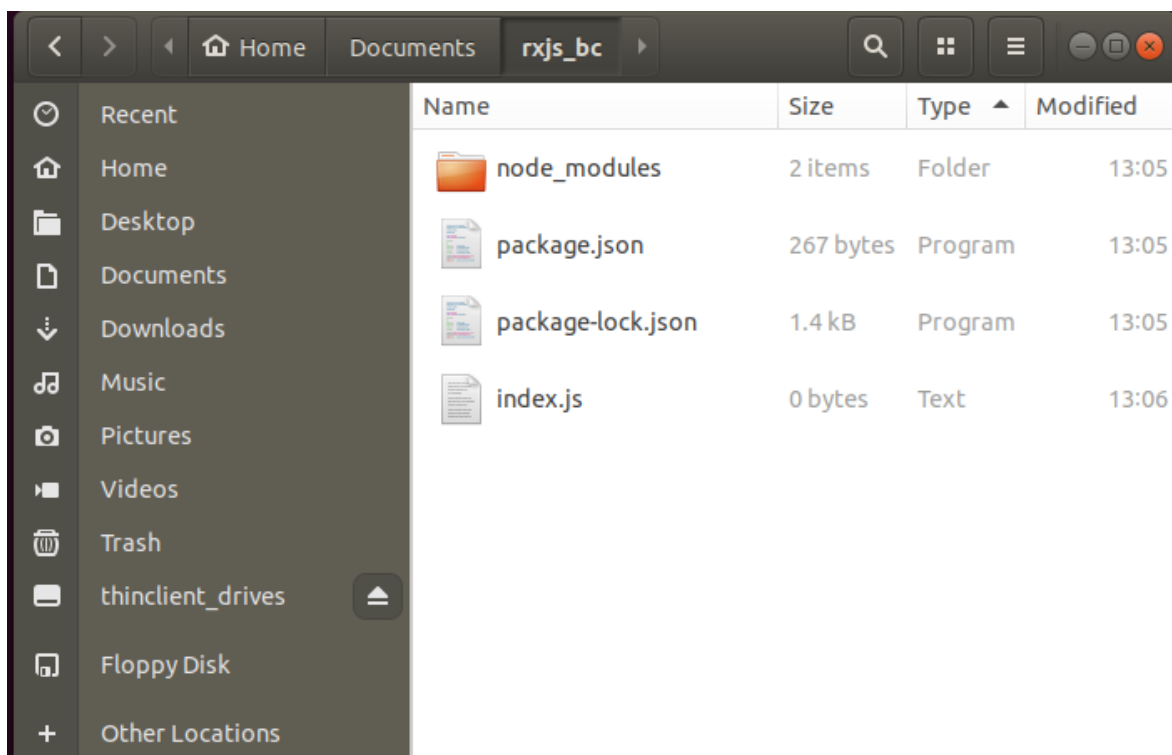
The second command will install the RxJS library and a node_modules folder will appear. The last will create a blank JavaScript file which we will use to enter and test our code.

```
axle@pc0480: ~/Documents/rxjs_bc
File Edit View Search Terminal Help
axle@pc0480:~/Documents/rxjs_bc$ npm i rxjs

added 2 packages, and audited 3 packages in 1s

found 0 vulnerabilities
axle@pc0480:~/Documents/rxjs_bc$ touch index.js
axle@pc0480:~/Documents/rxjs_bc$
```

This is the `rxjs_bc` folder as it appears after all the commands above were ran:



Open your `package.json` file and add this key/value pair: `"type": "module"`

Here is the entire `package.json` file on my Linux box:

```
{
  "name": "rxjs_bc",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [ ],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "rxjs": "^7.5.7"
  }
}
```

The highlighted line was inserted. Use any code editor for this.

Just to make sure everything works, enter the following eight lines of code into the `index.js` file:

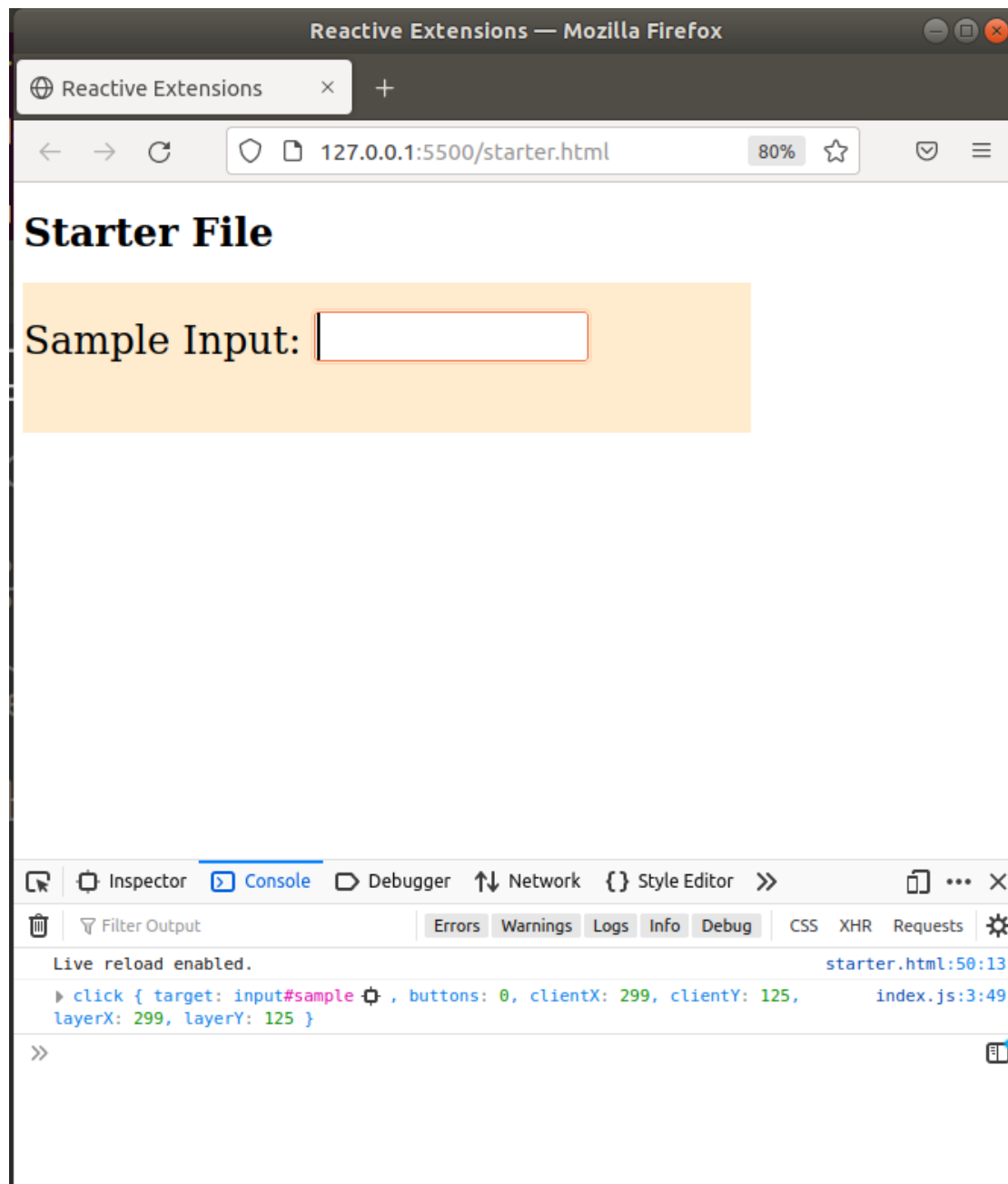
```
import { Observable } from 'rxjs';
//
const observable = new Observable(obs =>
  obs.next("Hello")
);
//
const observer = {
  next: aValue => console.log("We got " + aValue)
};
//
observable.subscribe(observer);
```

The code is explained in Appendix A.

In order to run the program, at a command prompt pointing to this file, type the command: `node index`

HTML Environment Setup

Inside of the GitHub repository for today, there is a zipped file called RxJS-HTML-Template.zip. Once you unzip that file, there is enough code to run and produce a result in the console window. Do not use the previous folder, just unzip this file and create a different folder. I just unzipped and renamed my folder to just RxJS-HTML.



The code is written to work with a **CDN**. The script that connects to the CDN is at the top of the HTML document. Once you have that script in place, then you would use the RxJS library by using the handle `rxjs`. Note, the HTML file [starter.html](#), should be ran through a web server. There is a server that can be installed inside of the VS Code IDE. The server is called **Live Server** and it was developed by Ritwick Dey.

All the JavaScript code is written in the [index.js](#) file. Line 2 uses the `fromEvent()` function to create an Observable that is stored in `myobs`. That `fromEvent()` takes two parameters, the source of the event (HTML Object) and the type of event you want to listen for, *click* in this case.

Line 3 subscribes to the observable and consumes the event. The event in this case is a click inside of the input box once the HTML file is loaded into the browser.



Part 2 – Warm Up Examples

In this section we take a look at creation Observables, classes from the library that create Observables automatically. There are several of them but `from` and `of` are the most popular.

1. From the **Node.js** introduction above you had this example. We will now use this example to demonstrate how `from` works:

```
import { Observable } from 'rxjs';
//
const observable = new Observable(obs =>
  obs.next("Hello")
);
//
const observer = {
  next: aValue => console.log("We got " + aValue)
};
//
observable.subscribe(observer);
```

In my case I'm using the [rxjs](#) [bc](#) folder for this example

2. First import `from` from the library:

```
import { from } from 'rxjs';  
//  
const observable = new Observable(obs =>  
  obs.next("Hello")  
);  
//  
const observer = {  
  next: aValue => console.log("We got " + aValue)  
};  
//  
observable.subscribe(observer);
```

3. The Observable or operator `from` will create its own Observable and return it, we just have to collect it somewhere:

```
import { from } from 'rxjs';  
//  
const fObs = from( );  
//  
const observer = {  
  next: aValue => console.log("We got " + aValue)  
};  
//  
observable.subscribe(observer);
```

4. The `from` operator can do several things but it works best with iterable objects like arrays, so let's supply a simple array:

```
import { from } from 'rxjs';  
//  
const fObs = from( ["a", "b", "c"] );  
//  
const observer = {  
  next: aValue => console.log("We got " + aValue)  
};  
//  
observable.subscribe(observer);
```


5. Since we changed the name of the Observable, we need to subscribe to it:

```
import { from } from 'rxjs';  
//  
const fObs = from( ["a","b","c"] );  
//  
const observer = {  
  next: aValue => console.log("We got " + aValue)  
};  
//  
fObs.subscribe(observer);
```

6. Now, just run the code and you should get the result shown below:



The screenshot shows a code editor with a file named `index.js`. The code in the editor is as follows:

```
1 import { from } from 'rxjs';  
2 //  
3 const fObs = from(["a","b","c"]);  
4 //  
5 const observer = {  
6   next: aValue => console.log("We got " + aValue)  
7 };  
8 //  
9 fObs.subscribe(observer);  
10
```

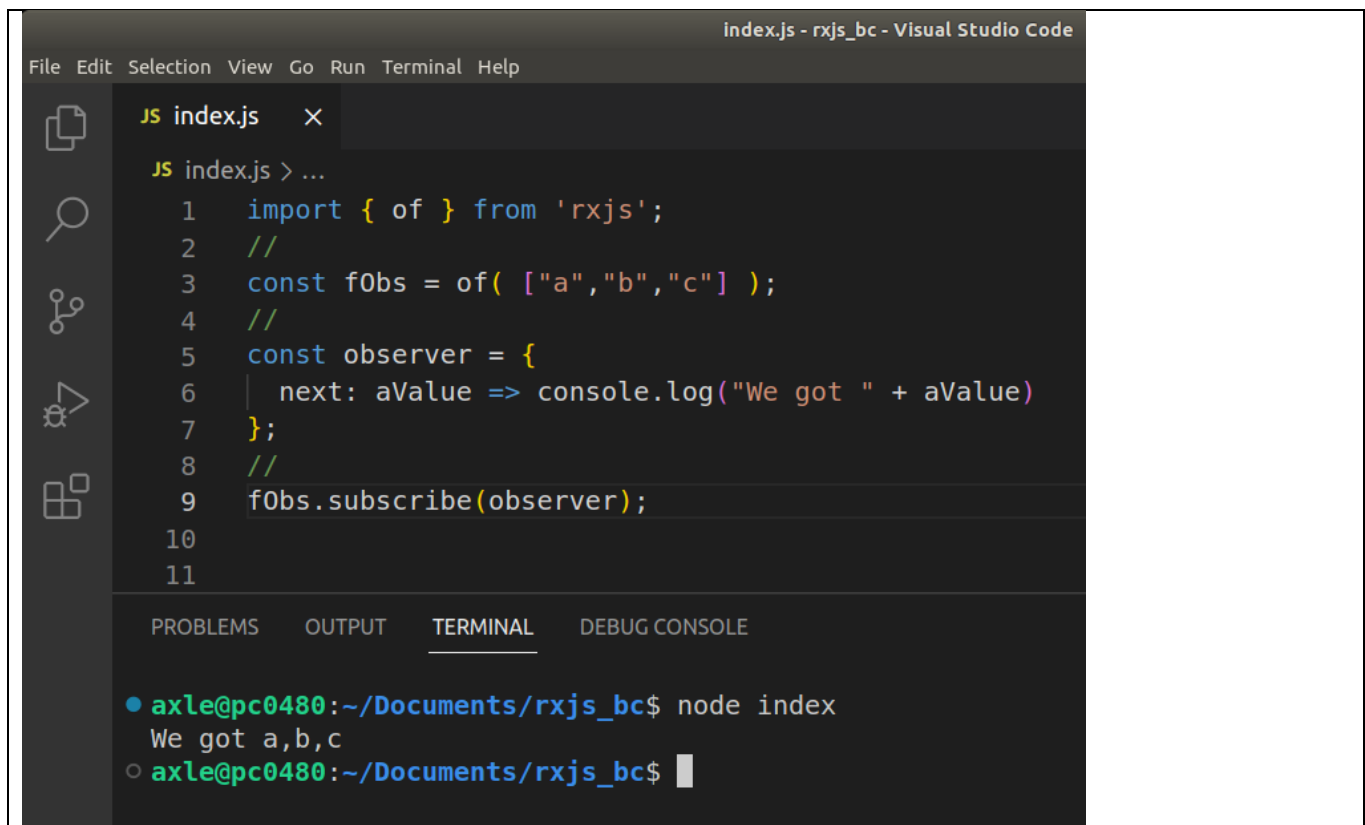
Below the code editor, the `TERMINAL` tab is active, showing the output of running `node index`:

```
● axle@pc0480:~/Documents/rxjs_bc$ node index  
We got a  
We got b  
We got c  
○ axle@pc0480:~/Documents/rxjs_bc$
```

7. The `of` operator is a creation operator, but it returns just one value:

```
import { of } from 'rxjs';
//
const fObs = from( ["a","b","c"] );
//
const observer = {
  next: aValue => console.log("We got " + aValue)
};
//
fObs.subscribe(observer);
```

8. This code simply returns the entire array as one object:



The screenshot shows the Visual Studio Code editor with a file named `index.js` containing the following code:

```
1 import { of } from 'rxjs';
2 //
3 const fObs = of( ["a","b","c"] );
4 //
5 const observer = {
6   next: aValue => console.log("We got " + aValue)
7 };
8 //
9 fObs.subscribe(observer);
10
11
```

The terminal at the bottom shows the command `node index` being executed, resulting in the output:

```
axle@pc0480:~/Documents/rxjs_bc$ node index
We got a,b,c
axle@pc0480:~/Documents/rxjs_bc$
```

The general rule is to use `from` when you want to work with individual array elements. Use the `of` operator when you have just one value **or** you want to work with an iterable object as just one object. The `of` operator can be made to work like the `from` operator if you employ the `spread` operator. Other creation operators include the `interval`, `timer` and `fromEvent` among others.



End of section



Part 3 – Pipeable Operators

A Pipeable Operator is just a function. It takes an input Observable and generates an output Observable. Usually something happens to the data once inside the pipe. Common examples include `filter`, `debounceTime`, `concatMap`, `reduce` and `tap`. The `pipe()` operator acts like a combiner of other operators. It provides a surface for other operators to perform on.

1. In order to prove the point, we will change the above example to use integers instead of letters, here is the first few lines of code:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
const fObs = from( [1,2,3,4] );
//
```

Also import the `map()` operator to transform the values

2. With these changes, remember `fObs` will represent an Observable. Since any observable can be pipeable, lets add in that function and pass the `map()` function into the `pipe()` operator:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
const fObs = from( [1,2,3,4] );
fObs.pipe( map( ) );
//
```

3. The `map()` operator will act on or manipulate each element of the array, so we need some variable to represent each element in the array:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
const fObs = from( [1,2,3,4] );
fObs.pipe( map( eachValue => eachValue ) );
```

For now, we just take each value in the array and simply return its original value

4. The `map()` operator returns a new observable, so lets capture this new object in a new variable so that we can use it later:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
const fObs = from( [1,2,3,4] );
const result = fObs.pipe( map( eachValue => eachValue ) );
```

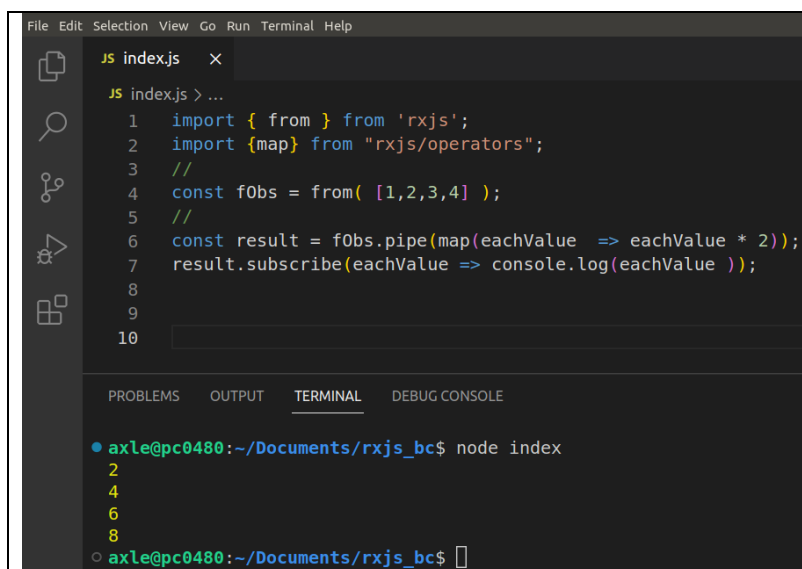
5. Now we can subscribe to `result`! `result` is the new transformed observable:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
const fObs = from( [1,2,3,4] );
const result = fObs.pipe(map(eachValue => eachValue));
result.subscribe(eachValue => console.log(eachValue ));
```

6. Of course all we are doing here is displaying the original values of the array, but usually something more important can be done to those values:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
const fObs = from( [1,2,3,4] );
const result = fObs.pipe(map(eachValue => eachValue * 2));
result.subscribe(eachValue => console.log(eachValue ));
```

7. This is the result on my system:



The screenshot shows a code editor with a file named `index.js`. The code defines an observable `fObs` from the array `[1,2,3,4]` and pipes it through the `map` operator, which multiplies each value by 2. The resulting observable `result` is subscribed to, logging each value to the console. The terminal output shows the values 2, 4, 6, and 8.

```
File Edit Selection View Go Run Terminal Help
JS index.js x
JS index.js > ...
1 import { from } from 'rxjs';
2 import { map } from "rxjs/operators";
3 //
4 const fObs = from( [1,2,3,4] );
5 //
6 const result = fObs.pipe(map(eachValue => eachValue * 2));
7 result.subscribe(eachValue => console.log(eachValue ));
8
9
10
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
axle@pc0480:~/Documents/rxjs_bc$ node index
2
4
6
8
axle@pc0480:~/Documents/rxjs_bc$
```

8. (Optional) it is possible to complete the above example is just one line of code, but it does become a bit unreadable:

```
import { from } from 'rxjs';
import { map } from "rxjs/operators";
//
from( [1,2,3,4] )
.pipe(map(eachValue => eachValue * 2))
.subscribe(eachValue => console.log(eachValue));
```

This is the infamous chaining technique and should be used once you are comfortable with the functions.

9. Finally for this section we will add two more operators, the `filter()` and `reduce()` operators, also add a couple more integers:

```
import { from } from 'rxjs';
import { map, filter, reduce } from "rxjs/operators";
//
const fObs = from( [1,2,3,4,5,6] );
const result = fObs.pipe(map(eachValue => eachValue * 2));
result.subscribe(eachValue => console.log(eachValue ));
```

10. To make the example interesting we will multiply each of the original values by three, then pull out all the even products using `filter()`. Finally we add all the even numbers using `reduce()`.

```
import { from } from 'rxjs';
import { map, filter, reduce } from "rxjs/operators";
//
const fObs = from( [1,2,3,4,5,6] );
//
const result = fObs.pipe(
  map(eachValue => eachValue * 3),
  filter(num => num % 2 === 0),
  reduce((startVal, currValue) => startVal + currValue)
);
//
result.subscribe(eachValue => console.log(eachValue ));
```

End of section

Part 4 – Special Observable - Subject

Within the RxJS library, there exist a special Observable called **Subject**. Well its also an Observer, it's a hybrid. The Subject has the special ability of multicasting, and each subscriber receives the same data.

You may continue using the files from the previous parts. Simply replace the code inside of the index.js file with the new code you see.

1. In this first example, the first two lines should be familiar by now. In line three we subscribe to the **Subject** and print it's value along with a prompt. Finally nothing will happen unless the last line is executed.

```
import { Subject } from 'rxjs';
const subject = new Subject();
subject.subscribe(
  value => console.log(`The data is ${value}.`)
);
subject.next('Skillsoft');
```

Notice that we subscribe to subject and we also call the next method of subject. Notice that subject and Subject are two different spelling.

2. What is nice about the Subject is that we can subscribe multiple times, we always get back the same data:

```
import { Subject } from 'rxjs';
const subject = new Subject();
subject.subscribe(
  value => console.log(`The data is ${value}.`)
);
subject.subscribe(
  value => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
```

Although the second subscription changed the look of the data, the data itself did not change, so the value did not change, only how it is being displayed.

3. We could also store each subscription:

```
import { Subject } from 'rxjs';
const subject = new Subject();
const s1 = subject.subscribe(
  value => console.log(`The data is ${value}.`)
);
const s2 = subject.subscribe(
  value => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
```

The output is the same, each subscription got the same value.

4. Lets now add a `next()` call in the middle of the code and change the last line slightly:

```
import { Subject } from 'rxjs';
const subject = new Subject();
const s1 = subject.subscribe(
  value => console.log(`The data is ${value}.`)
);
subject.next('Skillsoft rocks!');
const s2 = subject.subscribe(
  value => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft develop skills');
```

5. Here are the before and after results from the changes between #3 and #4 above:

```
axle@pc0480:~/Documents/rxjs_bc$ node index
The data is Skillsoft develop skills.
SKILLSOFT DEVELOP SKILLS, is the data.
axle@pc0480:~/Documents/rxjs_bc$ node index
The data is Skillsoft rocks!.
The data is Skillsoft develop skills.
SKILLSOFT DEVELOP SKILLS, is the data.
axle@pc0480:~/Documents/rxjs_bc$
```

By the time the JS interpreter gets to line 6 there is only one subscription, `s1`, so that is triggered. However by the time the interpreter gets to line 10 (last line), well there are now two subscriptions, `s1` and `s2`. So, line10 triggers **both** subscriptions. In other words 'Skillsoft Rocks' is only seen by `s1`. On the other hand 'Skillsoft develop skills' is seen by both `s1` and `s2`.

6. Lets add a `next()` method call before `s1` and another after the last `next()` method call:

```
import { Subject } from 'rxjs';
const subject = new Subject();
subject.next('Nice Courses');
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
subject.next('Skillsoft rocks!');
const s2 = subject.subscribe(
  value => console.log(`${value.toUpperCase()}, is the data.`)
);
//
subject.next('Skillsoft develop skills');
subject.next('Skillsoft has skills');
```

In this case 'Nice Courses' is completely ignored as there are no subscriptions when this line is executed. The rest is consistent with #5 above.

7. The results are consistent:

```
● axle@pc0480:~/Documents/rxjs_bc$ node index
The data is: Skillsoft rocks!.
The data is: Skillsoft develop skills.
SKILLSOFT DEVELOP SKILLS, is the data.
The data is: Skillsoft has skills.
SKILLSOFT HAS SKILLS, is the data.
```

Once the interpreter gets to line 11, there are two subscriptions. Then when line 12 is executed, there are still two subscriptions.

8. Observables are unicast, while **Subjects** are multicast. A **Subject** will maintain a list of subscribers, and therefore has some state. Use the **Subject** when you want all your subscribers to have the **same** data. There are two variations on the **Subject** construction. A **BehaviorSubject** will emit the last value to be sent to it. A **ReplaySubject** will give you a certain amount of data, for example the last three values. We will do more examples with **Subject** on Day02.

Part 5 – Schedulers

RxJS Schedulers have the ability to disrupt the flow of execution. It can also put certain execution into a different context, basically a different thread. At a deeper level, a scheduler can keep track of time. Note schedulers are used mainly by the library itself, there are not many use cases for it, in terms of everyday programming.

It is possible to achieve multithreading in JS. One way to do this is to use a scheduler. An operator like `observeOn()` will take a scheduler like `asyncScheduler` as a parameter. This forces the scheduler to complete it's tasks in a different thread! Different from the thread on which the Observable's `subscribe` method is called on.

The `observeOn()` operator spawns a different Scheduler (thread) that the Observable uses to emit data to any attached observers. In Part 5 section 7, if we did not use `observeOn()`, and pass it a scheduler, then all operations will utilize the same thread, and the benefit of using a scheduler will not be apparent.

1. Here is a piece of code you know very well:

```
import{ of } from "rxjs";
const obsOf = of("Hello", "from", "Skillsoft");
obsOf.subscribe(
  x => console.log(x)
);
```

This code should print the values in the `of()` operator vertically in a terminal window. Remember the `of()` operator exhibits synchronous behavior.

2. To really appreciate how this works, add a couple of log lines and run the program again:

```
import{of}from "rxjs";
const obsOf = of("Hello", "from", "Skillsoft");
console.log('Before subscribing...');
obsOf.subscribe(
  x => console.log(x)
);
console.log('after subscribing...');
```

You will see that both highlighted log lines run in the sequene we would expect, so in the same execution context. This is synchronous behavior. *Later in the course you will see how to use the `tap()` operator in combination with `pipe()` for situations like this.*

3. We can introduce a scheduler from the RxJS library by first importing it and then use it to schedule the values being emitted, we will also include the `observeOn()` operator which is a utility function:

```
import { of, asyncScheduler, observeOn } from "rxjs";
const obsOf = of("Hello", "from", "Skillsoft");
console.log('Before subscribing...');
obsOf.subscribe(
  x => console.log(x)
);
console.log('after subscribing...');
```

The `observeOn()` operator spawns a different Scheduler(thread), see Appendix A for more on this.

4. In order to use `observeOn()`, we need the `pipe()` operator. Once we have the `observeOn()` operator in place, we can then pass a scheduler into it:

```
import { of, asyncScheduler, observeOn } from "rxjs";
const obsOf = of("Hello", "from", "Skillsoft");
console.log('Before subscribing...');
obsOf.pipe(
  observeOn(asyncScheduler)
)
.subscribe(
  x => console.log(x)
);
console.log('after subscribing...');
```

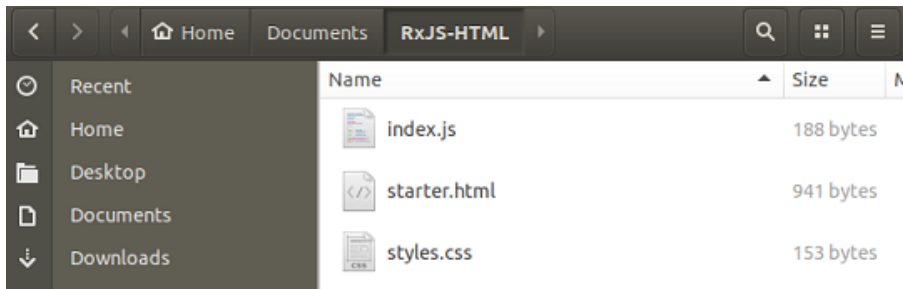
The `pipe()` operator must precede any of the other operators. It behaves like a container for other operators. The `subscribe()` operator always comes last.

5. Below is an image of how the output changed, notice that the two `console.log()` statements ran first, then the values were emitted. This indicates that there are two execution contexts in play:

```
axle@pc0480:~/Documents/rxjs_bc$ node index
Before subscribing...
after subscribing...
Hello
from
Skillsoft
axle@pc0480:~/Documents/rxjs_bc$
```

Part 6 – Schedulers Use Case

1. Let us now switch back to the RxJS-HTML folder. Remembe this was the the RxJS-HTML-Template.zip in unzipped form.



2. Start by adding a new `<div>` around line 20 of the starter.html file. Remove all the styles, if there:

```

        <input type="text" id="sample"></input>
        <!--button id="mybutton">C l i c k M e</button-->
    </div>
    <div id="pbar"></div>
</main>
<script type="module" src="index.js"></script>

```

3. Over in the css file, add a new style for that `div` so that we can see it better on the browser screen:

```

div {width: 600px;height: 100px;font-size: 2em; background-color:
blanchedalmond; padding-top: 24px;}
input {width: 220px;height: 36px;font-size: 1.2em;}
#pbar{
    height:10px;
    background:red;
}

```

4. This demo will feature the `animationFrameScheduler`. If this particular RxJS class is not present in this file, define it as below. We would need two more classes from the library so define them also:

```

const afs = rxjs.animationFrameScheduler;
const interval = rxjs.interval;
const take = rxjs.take;

```

Just replace the code currently in `index.js` with the code above, or comment out the previous code.

5. Obtain access to the `pbar` div:

```
const afs = rxjs.animationFrameScheduler;
const interval = rxjs.interval;
const take = rxjs.take;
let pBar = document.getElementById("pbar");
```

6. At this point, your browser should look like the image below:



7. Lets first see how the interval works, enter the following code in the JS file after the last line:

```
let pBar = document.getElementById("pbar");
interval(0)
  .pipe(take(4))
  .subscribe(x =>
    console.log(x)
  );
```

8. When you run the code above, you should see 0-3 print out in separate lines in the browser's console window (F12). We can now take advantage of this in a simple animation. The interval function takes a scheduler as a second argument, so change the code to the following and refresh the browser:

```
interval(0, afs)
  .pipe(take(4))
  .subscribe(x =>
    console.log(x)
  );
```

There should not be any changes. Remember `afs` was defined on line 1.

9. We could add a new style to the progress bar `<div>` using JS and implementing the interval count to increase the width of that div(`d1p6_asyncScheduler.zip`):

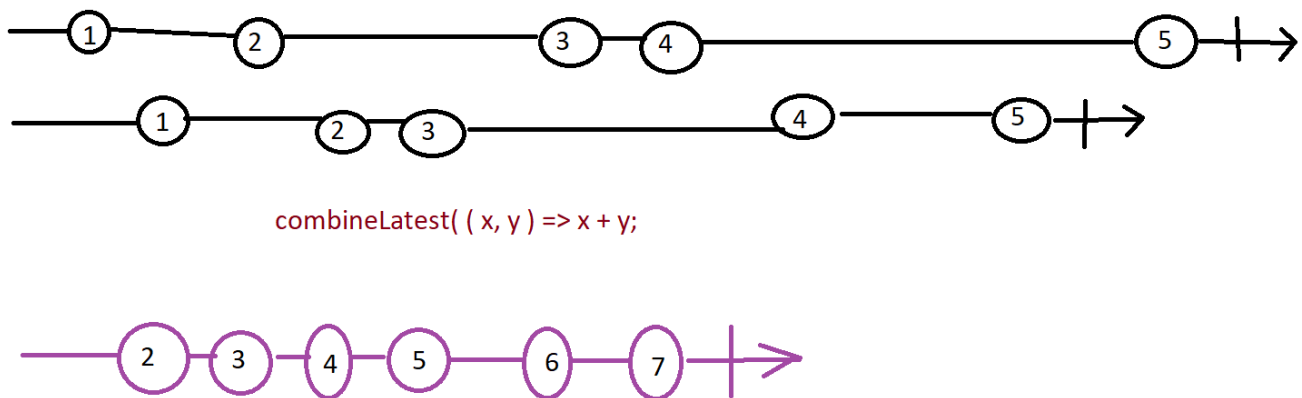
```
interval(0, afs)
  .pipe(take(600))
  .subscribe(x =>
    pBar.style.width = x+"px"
  );
```

Part 7 – combine Latest()

For this part, please have the [d1p7_combineLatest_starter.zip](#) file unzipped, renamed and ready for changes.

This operator works with two or more Observables. As each Observable emits a value, that value gets designated the latest value. The latest value from each of the Observables are then combined to create one Observable. In other words you will always get the latest value from each source.

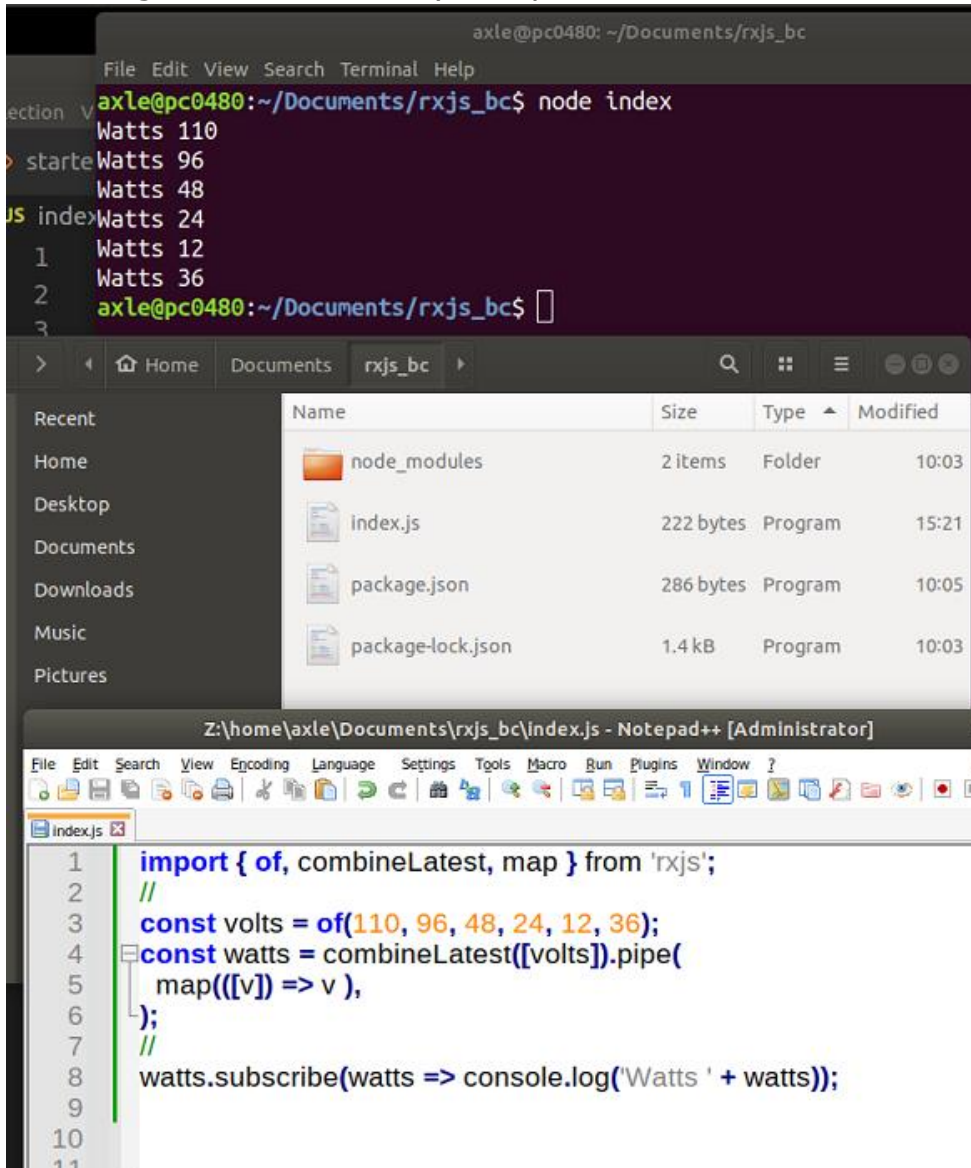
Usually a function is provided through which these combined values are passed.



1. For this first demo, return to your `rxjs_bc` folder. This code can be run in the `index.js` file and shown on the terminal window. Enter this code and run it to see the results:

```
import { of, combineLatest, map } from 'rxjs';
//
const volts = of(110, 96, 48, 24, 12, 36);
const watts = combineLatest([volts]).pipe(
  map(([v]) => v ),
);
//
watts.subscribe(watts => console.log('Watts ' + watts));
```

2. The image below shows my setup on the Linux box:



The top image is the result after running the code in `index.js` (last image). The folder setup is in the middle.

3. You may now close the command prompt window and `index.js`. Now switch to the starter file I referenced in #1 above. It is better to open this folder in VS Code. Look at `index.js`, there is already some code in this file. In fact from the VS Code IDE If you right-click on the `starter.html` file, you can open it with the Live Server app referred to in the **HTML Environment Setup** section above. If you click on each image you will see the name of the image in the console window. This data is being generated using the `fromEvent()` RxJS library operator.

4. From the starter folder, select index.js and begin by getting references to the two DDL boxes:

```
const horsepic$ = fromEvent(horsepic, 'click',
  (evt)=>evt.target.id);
const turtlepic$ = fromEvent(turtlepic, 'click',
  (evt)=>evt.target.id);
//reference user selections
const aname = document.getElementById("aname");
const bgcolour = document.getElementById("bgcolour");
//subscribe and log. This part will change in the bootcamp
```

Note: insert these lines above the last commented line in the code

5. At the moment, we have three observables for each of the three images. Although this could work moving forward, lets restructure this code to generate just one Observable and therefore just one image:

```
const bgcolour = document.getElementById("bgcolour");
//generate observable via click
const pic$ = fromEvent(
  [dynopic, horsepic, turtlepic], 'click', (evt)=>evt.target.id
);
//subscribe and log. This part will change in the bootcamp
pic$.subscribe(x=>console.log(x));
```

The output itself did not change, but the code structure has changed.

6. Now generate two more Observables via the user clicks, but based on the two DDL boxes:

```
[dynopic, horsepic, turtlepic], 'click', (evt)=>evt.target.id
);
//generate observable via change event
const aname$ = fromEvent(aname, 'change',
  (evt)=>evt.target.value);
const bgcolour$ = fromEvent(bgcolour, 'change',
  (evt)=>evt.target.value);
//subscribe and log. This part will change in the bootcamp
pic$.subscribe(x=>console.log(x));
```

Nothing will change at this point, just make sure there are no errors in the code

7. We can now apply the `combineLatest()` library function to see how it works in this situation:

```
const aname$ = fromEvent(aName, 'change',  
  (evt)=>evt.target.value);  
const bgcolour$ = fromEvent(bgColour, 'change',  
  (evt)=>evt.target.value);  
//apply combine latest  
const choices$ = rxjs.combineLatest(  
  pic$, aname$, bgcolour$  
);  
//subscribe and log. This part will change in the bootcamp  
pic$.subscribe(x=>console.log(x));
```

8. Change the subscription to reflect the changes:

```
const aname$ = fromEvent(aName, 'change',  
  (evt)=>evt.target.value);  
const bgcolour$ = fromEvent(bgColour, 'change',  
  (evt)=>evt.target.value);  
//apply combine latest  
const choices$ = rxjs.combineLatest(  
  pic$, aname$, bgcolour$  
);  
choices$.subscribe(  
  choiceArray => console.log(choiceArray)  
);
```

Note: you no longer need the `pic$.subscribe()` line, you can delete it or comment it.

9. Now when we test, we have to gather up all the values, then we get an array in the end. Once we have done that first selection of all three html objects, any changes will be reflected with a new output line in the developer console window:

Combine Latest:

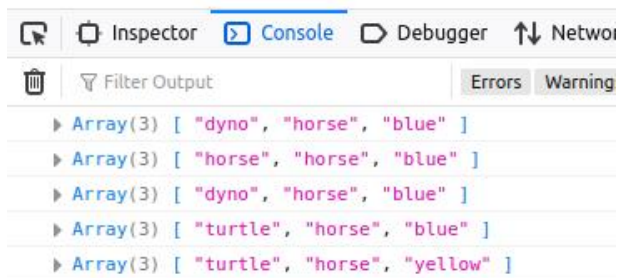


Animal Name

horse

Animal Background Color

yellow



10. We now need to return from `combinedLatest()` an image path (or url). First combine all the related Observables into one array, then pass each of their individual output to a function, via parameters of course. Once we have the arguments, we can build up an `` tag string:

```
//apply combine latest
const choices$ = rxjs.combineLatest(
  [pic$, aname$, bgcolour$],
  (pic, aname, bgcolour) => {
    return ``
  }
);
choices$.subscribe(
```

If you test now you will see the beginnings of an image path in the console window. It will have a `src`, `style` and `title`. I am using `title` instead of `alt` since *Firefox* will only react to `title`.

11. Of course at this point if we use this generated image path inside our `div` we should generate an image:

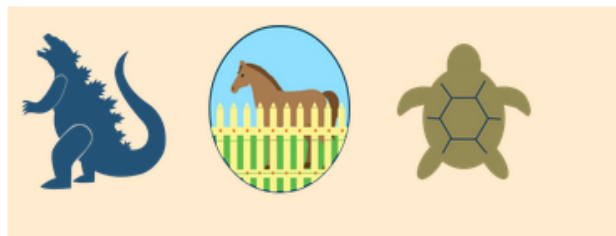
```

        title="${aname}">`
    }
};
//subscribe to the one choices$ observable, NOT the individual
ones
//build an image path/url
choices$.subscribe(
    choiceArray => document.getElementById("choice").innerHTML =
choiceArray
);

```

12. The result of the changes so far are shown below

Combine Latest:



Animal Name Animal Background Color

turtle blue



Notice that the logic is NOT correct. Obviously the image is a horse and not a turtle, but turtle was the last item to be clicked on, therefore the `combinedLatest()` feature is working. This is the point of the exercise.

13. You may have noticed that initially when the page loads, no image is shown and the two DDLs just default to their first options. Also you have to click on all three objects in order for something to happen: Lets instead, show an initial image and two values being applied to that image. After that, the user can change her selections:

```

        <select id="aname">
          <option value="dynosaur" selected>dynosaur</option>
          <option value="horse">horse</option>
          <option value="turtle">turtle</option>
        </select>
      </div>
      <div style="display: inline-block;margin-left: 40px;">
        <label for="bgcolour">Animal Background
        Color</label><br>
        <select id="bgcolour">
          <option value="blue" selected>blue</option>
          <option value="yellow">yellow</option>

```

Since *dinosaur* and *blue* are the first options, formalize this by making them the default selected on the HTML side

14. Add the `startWith()` operator from the RxJS library:

```

//get the basic operators
const fromEvent = rxjs.fromEvent;
const startWith = rxjs.startWith;

```

15. Restructure the code associated with the `fromEvent()` observable to add the `startWith()` operator via the `pipe()` method:

```

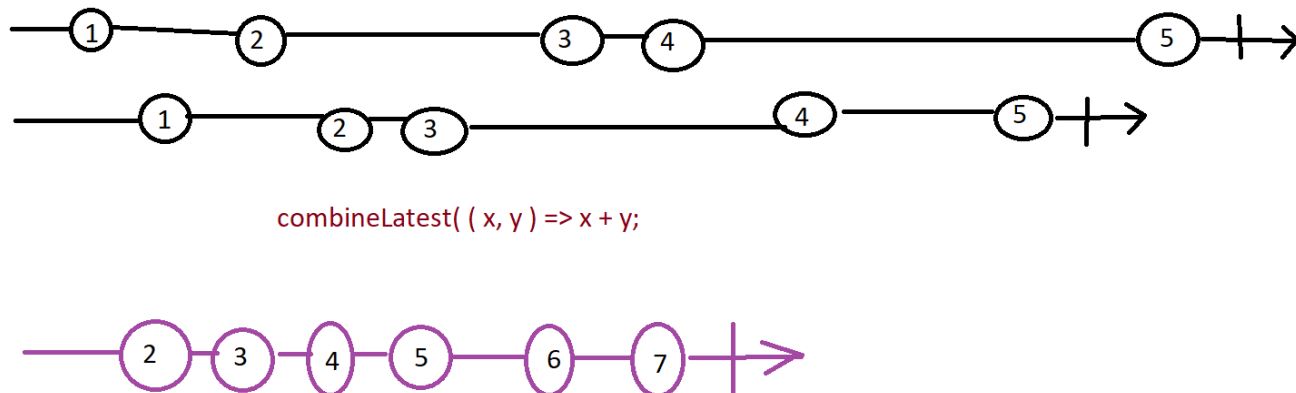
//generate observable via click
const pic$ = fromEvent(
  [dynopic, horsepics, turtlepics], 'click', (evt)=>evt.target.id
);
//generate observable via change event
const aname$ = fromEvent(
  aname, 'change', (evt)=>evt.target.value
).pipe(startWith('dyno'));
const bgcolour$ = fromEvent(
  bgcolour, 'change', (evt)=>evt.target.value
).pipe(startWith('blue'));
//apply combine latest

```

If you want, apply the same technique to the image to make it all work, so something like `.pipe(startWith('dinosaur'))`; Another operator `forkJoin()` is very similar to `combineLatest()`, just replace with `forkJoin()` to see differences.

Part 8 – merge(), mergeMap(), forkJoin(), concat(), zip()

With `merge()` it is possible to create a single observable from multiple ones. For this part, you may return to the Node based application.



1. Enter the following code and run it to see the results:

```
import { of, merge } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const merged = merge(volts, amps);
merged.subscribe(merged => console.log('Merged ' + merged));
```

As expected, this code will simply display all of the data from volts then amps in that order.

2. This is an interesting way to subscribe to any observable:

```
import { of, merge } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const merged = merge(volts, amps);
merged.subscribe(merged => console.log({merged}));
```

This should result in something like this: `{ merged: 110 } ...`

3. Here is an example of `concat()`, very similar to `merge()`:

```
import { of, concat } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const concated = concat(volts, amps);
concated.subscribe(concated => console.log({concated}));
```

4. The `forkJoin()` operator will wait until all observables complete, then emit the last value from each observable within `forkJoin()`:

```
import { of, forkJoin } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const forkJoined = forkJoin({volts, amps});
forkJoined.subscribe(forkJoined => console.log(forkJoined));
```

5. The `zip()` operator is considered a flattening operator just like `forkJoin()`. It works exactly like `forkJoin()` except it emits an *array* of the last values instead of an *object*:

```
import { of, zip } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const zipped = zip(volts, amps);
zipped.subscribe(zipped => console.log(zipped));
```

Notice that the data value of 24 was not included in the output, but we got more data than the previous operator.

6. The `mergeMap()` operator is considered a flattening operator just like `flatMap()`. We already know that we can take volts (or amps) in this case and just subscribe to its emitted values:

```
import { of, mergeMap, map } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
volts.subscribe(x => console.log(x));
```

7. If we just add a `pipe()` in the middle, nothing will change:

```
import { of, mergeMap, map } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
volts.pipe(v => v)
.subscribe(x => console.log(x));
```

But now we have positioned the code to accept the `mergeMap()` operator

8. Now we can pass `mergeMap()` into the `pipe()` function. This won't work all the way because we get only the object that `i` represents.

```
volts.pipe(
  mergeMap(v => amps.pipe(i => v * i))
)
.subscribe(x => console.log(x));
```

9. What we have to do instead is map the *amps* value, then we can pass that value to a function:

```
volts.pipe(
  mergeMap(v => amps.pipe(map(i => v * i)))
)
.subscribe(x => console.log(x));
```

10. Final code

```
import { of, mergeMap, map } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const watts = volts.pipe(
  mergeMap(v => amps.pipe(map(i => v * i)))
)
watts.subscribe(w => console.log(w));
```

For every emission of *amps*, that value will be multiplied by the first value of *volts*. So 110 will be multiplied by each *amp*. Then we move to 96 and that will then be multiplied by each *amp*.

11. (Bonus) adding a delay to `zip()`. You can add a time *delay* to the `zip()` operator in the following manner:

```
import { delay, of, zip } from 'rxjs';
const volts = of(110, 96, 48, 24);
const amps = of(4, 3, 6);
const zipped = zip(
  volts.pipe(delay(2000)),
  amps.pipe(delay(1000))
);
zipped.subscribe(zipped => console.log(zipped));
```

You can add a delay to just about any of these operators if you use the `pipe()` method. For example line 5 of the final code on part 10 above can be changed to:

```
mergeMap(v => amps.pipe(map(i => v * i), delay(3000)))
```

Part 9 – Mouse Capture

The RxJS functions `map()` and `pipe()` can be used outside of the context of an observable.

For this part, you may use any of the HTML templates.

1. With just three lines we can capture the x-coordinate of the mouse on a document:

```
const { fromEvent } = rxjs;
let down$ = fromEvent(document, 'mousedown');
down$.subscribe(x => console.log(x.clientX));
```

fromEvent() returns an Observable. That Observable emits data based on an event raised by some target object.

If you log just `x` you will see that it gives you a lot of detail about the mouse event, we just need the `clientX` property for this example. The above code is listening to `mousedown` events on the entire HTML document. As soon as that event happens, that event is emitted via the `down$` Observable.

2. Similarly we can get the x-coordinate of where the mouse-pressed event was released:

```
const { fromEvent } = rxjs;
let down$ = fromEvent(document, 'mousedown');
let up$ = fromEvent(document, 'mouseup');
down$.subscribe(x=>console.log(x.clientX));
up$.subscribe(x=>console.log(x.clientX));
```

The value where the mouse was pressed and released will be different if the mouse was moved while pressing down on the left button.

3. What I would like to do is create a function that will accept these observables and return to me both the x and y positions in a JSON object

```
function getCoords(mouseEvent) {
  return ({
    x:mouseEvent.clientX,
    y:mouseEvent.clientY
  })
};
```

Do not type in the code above it is just for illustration

4. We could perform the function above using RxJS library functions. Import both `map()` and `pipe()`. Both of these can be used outside of an observable:

```
const { fromEvent, map, pipe } = rxjs;
let down$ = fromEvent(document, 'mousedown');
let up$ = fromEvent(document, 'mouseup');
//down$.subscribe(x=>console.log(x.clientX, x.clientY));
//up$.subscribe(x=>console.log(x.clientX, x.clientY));
```

5. Now create a function that uses both `pipe()` and `map()`.

```
...
let up$ = fromEvent(document, 'mouseup');
down$.subscribe(x=>console.log(x.clientX, x.clientY));
const getCoords = pipe(
  map()
);
```

So, `pipe()` allows us to utilize any RxJS library function inside of it. With `map()` we can manipulate the data being passed down.

6. With this code in place we can now build our function inside of `map()`:

```
const getCoords = pipe(
  map(()=>{})
);
```

That function will accept the observable as an argument and return its X and Y positions via an object.

7. Now we can pass the `MouseEvent` into the `map()` operator. The function has to be configured to return an object, hence the `{}`. Finally all of this needs to be wrapped inside of parenthesis:

```
const getCoords = pipe(
  map((MouseEvent) => ( { } ) )
);
```

We don't want the `{}` to act like a function body, but instead as an object wrapper

8. We can now build our object that will be returned from this function:

```
const getCoords = pipe(
  map((MouseEvent)=>({
    x:MouseEvent.clientX,
    y:MouseEvent.clientY,
  })))
);
```

You can try to print the `down$` using this code: `console.log(getCoords(down$));`

9. We will use the `down$` Observable as an example. We can chain the `pipe()` operator to it and insert a function in the normal way:


```

        y:mouseEvent.clientY,
    )))
);
//
down$.pipe(evnt => {});

```

Here, `pipe()` takes a function, but that function has access to the Observable.

10. At this point, we can extract the parameter from the Observable and call it **coordinates**. This parameter can now be passed into `getCoords()` function we just wrote:

```

    )))
);
//
down$
.pipe(evnt => getCoords(evnt))

```

11. Now all we have to do is subscribe to the `down$` Observable and print the coordinates that was conditioned by the `getCoords()` function:

```

down$
.pipe(evnt=>getCoords(evnt))
.subscribe(coordinates=>console.log(coordinates));

```

Appendix A – merge vs concat

With **merge()** it is possible to create a single observable from multiple ones.

<pre>const interval1 = interval(2000) .pipe(take(4), map(x => `interval 1 : \${x}`)); const interval2 = interval(1000) .pipe(take(2), map(x => `interval 2 : \${x}`)); const interval3 = interval(500) .pipe(take(4), map(x => `interval 3 : \${x}`)); //merge but only show 2 observable output at once const merged = merge(interval1, interval2, interval3, 2); // merged.subscribe(data => console.log(`\${data}`));</pre>	<pre>const interval1 = interval(2000) .pipe(take(4), map(x => `interval 1 : \${x}`)); const interval2 = interval(1000) .pipe(take(2), map(x => `interval 2 : \${x}`)); const interval3 = interval(500) .pipe(take(4), map(x => `interval 3 : \${x}`)); // const concated = concat(interval1, interval2, interval3); // concated.subscribe(data => console.log(`\${data}`));</pre>
<p>With <i>concat</i> interval1 finishes all it's iterations. After that intervals 2 and 3 run. The Observables execute in sequence. However with <i>merge</i>, all Observables run as a single stream of output.</p>	
<pre>interval 2 : 0 interval 1 : 0 interval 2 : 1 interval 3 : 0 interval 3 : 1 interval 3 : 2 interval 1 : 1</pre>	<pre>interval 1 : 0 interval 1 : 1 interval 1 : 2 interval 1 : 3 interval 2 : 0 interval 2 : 1 interval 3 : 0</pre>

interval 3 : 3	interval 3 : 1
interval 1 : 2	interval 3 : 2
interval 1 : 3	interval 3 : 3

Appendix B – mergeMap and Events

Part 9 mouse capture is the basis of this more elaborate example of capturing and reporting mouse movement on a web page:

```
const { fromEvent, map, pipe, mergeMap, takeUntil } = rxjs;
//
const mousedown$ = fromEvent(document, 'mousedown')
const mouseup$ = fromEvent(document, 'mouseup');
const mousemove$ = fromEvent(document, 'mousemove');
//
mousedown$.pipe(
  mergeMap(_ => {
    return mousemove$.pipe(
      map(e => ({
        x: e.clientX,
        y: e.clientY
      })),
      takeUntil(mouseup$)
    )
  })
).subscribe(console.log);
```

Note: since we do not have a parameter to pass to mergeMap() we pass an underscore `_`. The mergeMap() operator can be used when you want to work with an inner observable and control that observable's subscriptions.

Appendix C – HTML Usage

For participants using just the HTML file, here is an example of how you would construct the example from Part5. This will be the [index.js](#) file:

```
const inputBox = document.getElementById('sample');  
  
const obsOf = rxjs.of("Hello", "from", "Skillsoft");  
  
console.log('Before subscribing...');  
  
obsOf  
  
.pipe(  
    rxjs.observeOn(rxjs.asyncScheduler)  
).subscribe(  
    x => console.log(x)  
);  
  
console.log('after subscribing...');
```

Specifically this is Part5 #4. Notice that I had to append *rxjs* in front of the name of the method. You do not have to do this for `pipe()`.