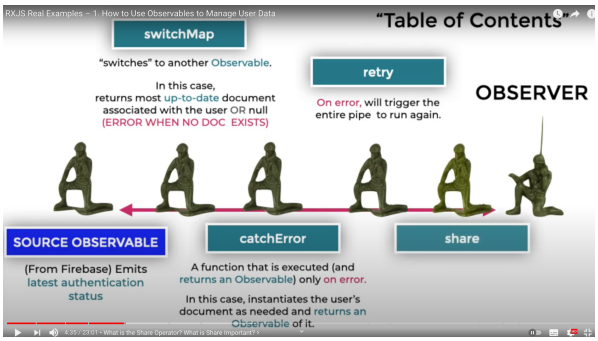
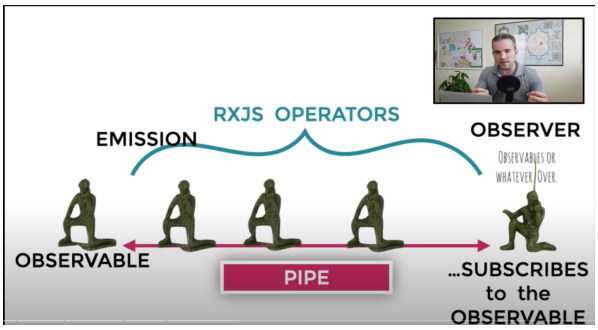
Observables (RxJS)

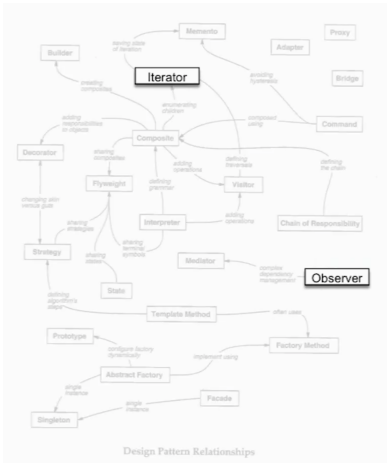
- Link to course
- Link to Interactive Code





Building Blocks:

- Observables utilize two design patterns: 1) Iterator 2) Observer



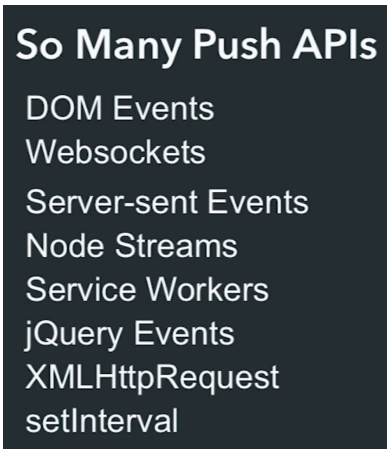
Observables can model...

- Events
- Async Server Requests
- Animations

Reactive Extensions

- Events as Streams
- Open Source (Apache2)
- Ported to many languages
  - C
  - .NET
  - JavaScript
  - Java (Netflix)
  - Objective-C

- Observables provide an unified interface to interact with all the commonly used 'push' APIs:

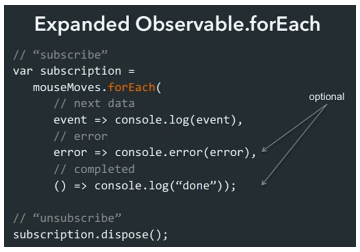


- Think of Observables as collections that arrive over time:

- With Observables, the consumer asks for data from the producer, not the other way around (i.e. we pull, NOT, not getting pushed at!) - this is the Iterator pattern in practice! With the Observer pattern, producer throws/pushes things to the consumer instead!

Observables:

- With an Observable the consumer (and NOT the producer) has the choice to unsubscribe and stop listening to the stream (i.e. `dispose` is the equivalent of `removeEventListener`):



Observable.forEach

```
// “subscribe”
var subscription =
   mouseMoves.forEach(console.log);

// “unsubscribe”
subscription.dispose();
```

Expanded Observable.forEach

```
// “subscribe”
var subscription =
   mouseMoves.forEach({
      onNext: event => console.log(event),
      // error
      onError: error => console.error(error),
      // completed
      onCompleted: () => console.log(“done”)
   });

// “unsubscribe”
subscription.dispose();
```

Observer

- Observables are simply Objects with a forEach method (which is how we get the data out of the stream) :

Converting Events to Observables

```
Observable.fromEvent = function(dom, eventName) {
   // returning Observable object
   return {
      forEach: function(observer) {
         var handler = (e) => observer.onNext(e);
         dom.addEventListener(eventName, handler);

         // returning Subscription object
         return {
            dispose: function() {
               dom.removeEventListener(eventName, handler);
            }
         };
      }
   };
}
```

- `fromEvent` creates a 'bridge' between the DOM API and the Observable API:

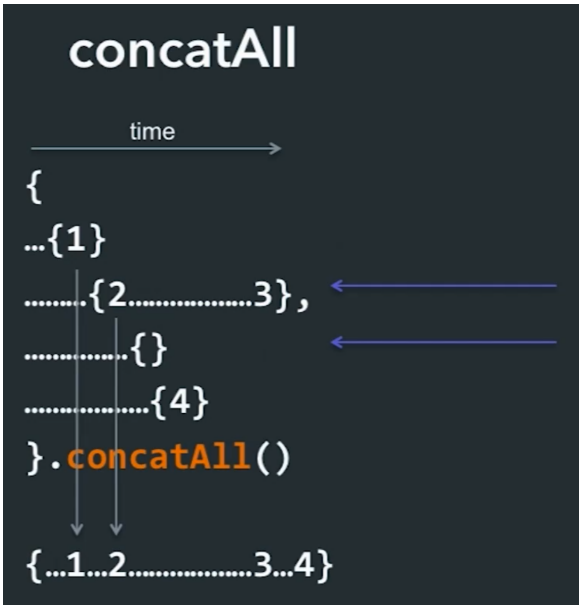
Events to Observables

```
var mouseMoves =
   Observable.
      fromEvent(element, “mousemove”);
```

- Observables do not hold any data in memory - they simply transform and pass it on to the consumer as soon as it's ready

- Almost all async concurrency problems (i.e. when we are dealing with an Observable of Observables and need to flatten the stream) can be solved with either of these 3 methods:

1. `concatAll` - this method keeps the same order (for the items coming out) as they were ordered in the original Observable they were part of when coming in:



Some Observables will not emit data values until we call `forEach` on them (called **"Cold Observable"**). However, not all Observables behave this way, for example: `mousemove` DOM event, which is an example of a **"Hot Observable"/"Hot Data Source"**, it emits regardless of if the consumer is listening to it or not! This is essentially a way to prevent race conditions as we control the order of data coming out of different sources.

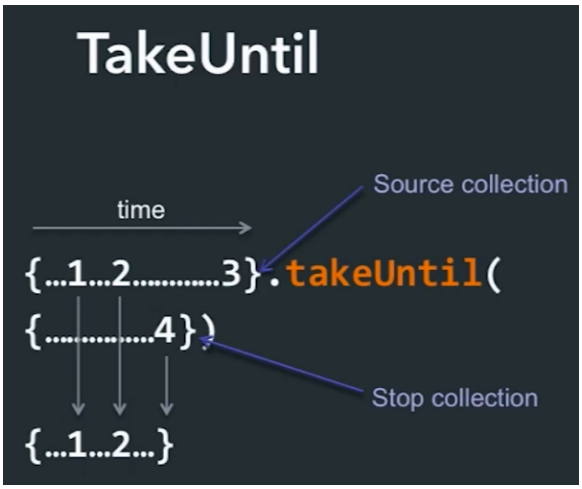
We should not call `concatAll` on an infinite stream (i.e. one that does not end like `mousemove` events-) as it will always be blocking the rest of Observables from delivering their data! Basically we are never getting to the end of the stream so there is no point to apply this method!

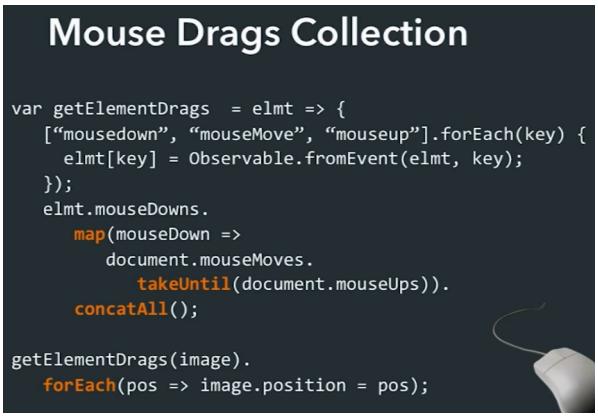
2. `takeUntil` - this method takes two Observables: a) Source Observable b) Stop Observable. This method returns a new Observable that is live while the source

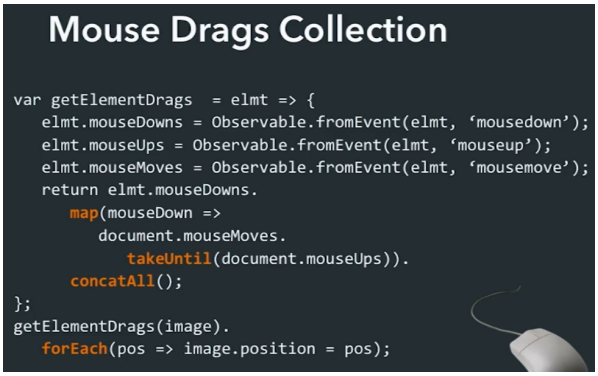
Observable keeps emitting, and stops once the stop Observable emits either an `onCompleted` or `onNext` .

So by using this method we are basically creating a (new) finite Observable from two infinite Observables! The big mental leap is that we do not need to unsubscribe from (e.g. DOM) events anymore, we can just stop asking for data whenever a condition is met and be done with such event streams exactly when we want them to.

The value from the stop collection is simply irrelevant as it is only used to indicate the end of the overall pipe.





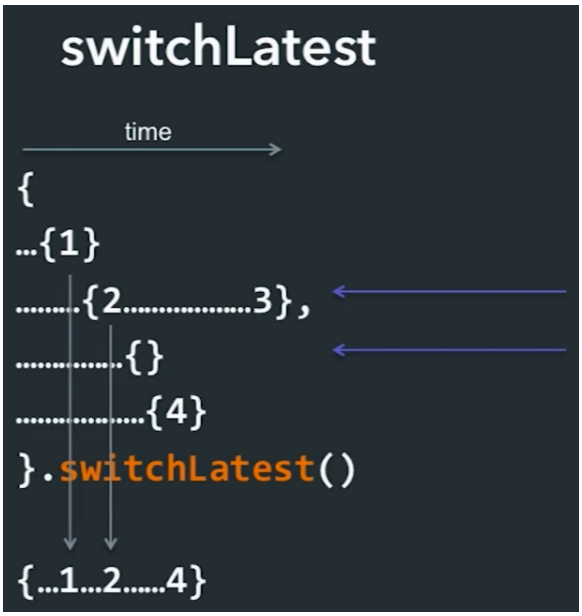


- Here we basically adopt from DOM events (mouse events) to Observables and create a single one from all the different mouse events. Then we start

consuming the data coming from this single Observable by `forEach` on it.

3. `mergeAll` - this differs from `concatAll` in that we do NOT wait for the current Observable we are `forEach` over to emit all its values before moving on to other (inner) Observables (i.e. we do NOT buffer inner/intermediate Observables anymore). In other words, we do NOT care about the order anymore.

4. `switchLatest` - this is perhaps the most common flattening pattern we use when dealing with user interfaces. With this strategy, while we are waiting on the inner/intermediate Observable to complete, if there is another inner Observable that comes along (does NOT matter if it actually has any data or not!), we simply stop waiting for the first one and ignore it, and move on to the next inner Observable - hence why '3' is no longer included in the end results. It actually calls `dispose` to get rid of the previous Observable.

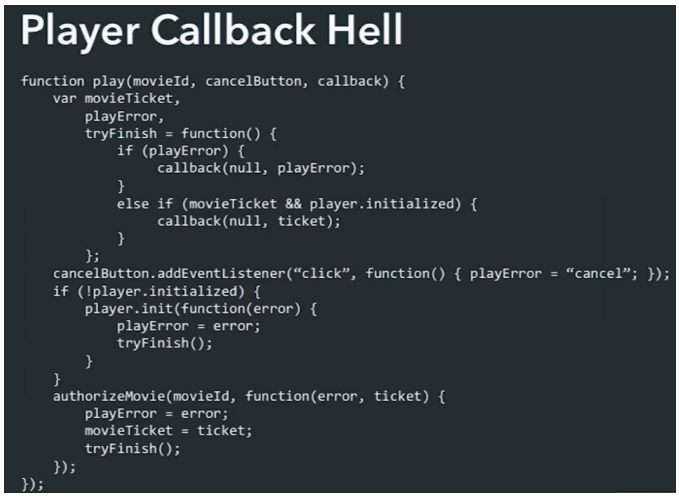


- A common async problem is demonstrated when we have an autocomplete box/search functionality: 1) race condition is an issue: user types 'abc', we send a network request to get results for that, but in the meantime user has kept typing and now wants results for 'abcdefg' instead! 2) we need debouncing/throttling on user input to avoid too many network requests:

Note: `retry` is used to avoid network failures - it is NOT concerning because Cold Observables are 'lazy', in the sense that they do not do anything unless we call `forEach` on them - which is when the action actually happens and the work gets done (in this case making a network request). It is a method on the Observable prototype.

Alternatively, we could use `takeLatest` to solve the above problem (a better approach actually) and simplify it to this:

- The biggest problem with Promises and their use for UIs is that we cannot cancel them! They are more useful for doing async stuff on the server side.
- We can turn this traditional example into a simpler one by using Observables:



Player with Observable

```
var authorizations =
   player.
      init().
      map(() =>
         playAttempts.
            map(movieId =>
               player.authorize(movieId).
                     catch(e => Observable.empty).
                     takeUntil(cancels)).
            concatAll())).
      concatAll();

authorizations.forEach(
   license => player.play(license),
   error => showDialog("Sorry, can't play right now."));
```

- We are dealing with a 3-dimensional Observable, because each of the `map` functions return another Observable, hence we need at least 2 merge strategies (done via `concatAll` here)
- The `catch` method here handles any potential issues, and just returns an empty Observable, so that we don't have to stop the whole stream of all other authorization calls (only the one that failed comes back empty!). This empty Observable immediately completes. This approach is equivalent to a try/catch block, where the catch is empty!
- The `init` method produces an Observable of one (i.e. it is NOT empty otherwise we could not call map on it!). This approach is commonly used to be able to apply the `map` in the first place.
-