Observables

tl;dr

- Observables represent a stream of future values.
- They will run a function as data arrives instead of once at the end.
- You tell the observable what it is waiting on.
- Then you use subscribe to tell it what to do when the data is updated
- There are some really useful operators that must be imported before they're used

What is an observable?

Promises are great, but there is room for improvement ...

- They only call the event when the whole process is complete
- They can't be interrupted

- Observables fix these problems

If a promise represents a future value, an observable represents a stream of future values

- Kind of like an array whose items arrive slowly.

```
[
{first:"Mal",last:"Reynolds",email:"capt@serenity.com"},
{first:"Zoë",last:"Washburne",email:"mate@serenity.com"},
{first:"Wash",last:"Washburne",email:"pilot@serenity.com"},
{first:"Jayne",last:"Cobb",email:"jcobb@serenity.com"},
{first:"Kaylee",last:"Frye",email:"mechanic@serenity.com"},
{first:"River",last:"Tam",email:"cargo@serenity.com"},
{first:"Derrial",last:"Book",email:"shepherd@serenity.com"}
]
```

tc39 is working on a standard for JavaScript

- Look here for the proposal and status:
`https://tc39.github.io/proposal-observable`

- Until it makes it to all browsers, we use a polyfill called rxjs

How to create and process an observable

You associate a function with an observable

```
const obs = Observable.create(
  observer => {
    setInterval(() => {
      observer.next(Math.random());
    }, 2000);
  }
);
```

.next(value) says to raise the Observable's event (aka. success handler) and provide *value* to it.

Promises can only respond to something running.

Observables won't let them run until you subscribe.

Observables are lazy

So how do you subscribe to an observable?

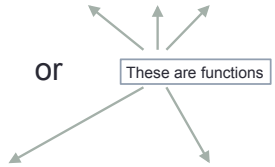
You provide a function to run

subscribe runs the function every time a new value arrives

```
obs.subscribe(v => {
  console.log("s fired", v);
  this.messages.push(v);
});
```

To handle exceptions

```
observable.subscribe(success, error, finally);
```




```
observable
.pipe(catchError(error)).subscribe(success)
```

That .catchError() is a pipeable operator.

Pipeable operators

aka. "lettable" operators

Operators enhance the capability of observables

Operator	Description
map(funcToConvert)	Converts each element
filter(predicate)	Allows some through, others not
first()	Only the first one in a series
last()	Only the last one
single()	Throws if >1 exist
skip(x)	Skip x of them
take(y)	Allow only y of them
debounce(timer)	Ignore for some time
... and many more!	

To use operators, put them in a pipe

```
Observable.pipe(...Operators);
```

```
// For example ...
observableOfPersons.pipe(
  filter(p => p.desc.includes(searchString)),
  map(p => `${p.first} ${p.last}`),
  skip(50),
  take(10)
).subscribe(nm => printFullname(nm));
```

We need to import operators

You can import each extension method like this:

```
import { map, skip, take, filter }
    from 'rxjs/operators';
```

Observables with Http

- Now you know how Observables work.
- Let's see how to apply them to Angular and Http
- Because, let's face it ...

Your main use for observables will be to process Ajax responses

You may want to convert your data to strongly-typed objects

```
this._http
.get("http://us.com/persons/123")
.pipe(
  map(res => <Person> res)
)
.subscribe(res => this.person = res)
```

".map()" says to convert the http response based on the function passed to it

"<Person>" says to cast the generic object as a Person

Remember, observables are lazy! Nothing happens until you subscribe.

tl;dr

- Observables represent a stream of future values.
- They will run a function as data arrives instead of once at the end.
- You tell the observable what it is waiting on.
- Then you use subscribe to tell it what to do when the data is updated
- There are some really useful operators that must be imported before they're used