Angular

III Lecture

Observables

Basic usage and terms

```
// Create an Observable that will start listening to geolocation updates
// when a consumer subscribes.
const locations = new Observable((observer) => {
  // Get the next and error callbacks. These will be passed in when
  // the consumer subscribes.
  const {next, error} = observer;
  let watchId;
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(next, error);
  } else {
    error('Geolocation not available');
  }
  // When the consumer unsubscribes, clean up data ready for next
subscription.
  return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
});
// Call subscribe() to start listening for updates.
const locationsSubscription = locations.subscribe({
  next(position) { console.log('Current Position: ', position); },
  error(msg) { console.log('Error Getting Location: ', msg); }
});
// Stop listening for location after 10 seconds
setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);
```

Defining observers

NOTIFICATION TYPE	DESCRIPTION
`next`	Required. A handler for each delivered value. Called zero or more times after execution starts.
`error`	Optional. A handler for an error notification. An error halts execution of the observable instance.
`complete`	Optional. A handler for the execution-complete notification. Delayed values can continue to be delivered to the next handler after execution is complete.

KindGeek School

Subscribing

`of(...items)` — Returns an Observable instance that synchronously delivers the values provided as arguments.

`from(iterable)` — Converts its argument to an Observable instance. This method is commonly used to convert an array to an observable.

Subscribe using observer

```
// Create simple observable that emits three values
const myObservable = of(1, 2, 3);

// Create observer object
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object
myObservable.subscribe(myObserver);
// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification
```

Subscribe with positional arguments

```
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

KindGeek School

Creating observables

```
// This function runs when subscribe() is called
function sequenceSubscriber(observer) {
  // synchronously deliver 1, 2, and 3, then complete
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // unsubscribe function doesn't need to do anything in this
  // because values are delivered synchronously
  return {unsubscribe() {}};
}

// Create a new Observable that will deliver the above sequence
const sequence = new Observable(sequenceSubscriber);

// execute the Observable and print the result of each notification
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// 1
// 2
// 3
// Finished sequence
```

KindGeek School

The RxJS library

- Converting existing code for async operations into observables
- Iterating through the values in a stream
- Mapping values to different types
- Filtering streams
- Composing multiple streams

KindGeek School

Create an observable from a promise

```
import { fromPromise } from 'rxjs';

// Create an Observable out of a promise
const data = fromPromise(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
 next(response) { console.log(response); },
 error(err) { console.error('Error: ' + err); },
 complete() { console.log('Completed'); }
});
```

KindGeek School

Create an observable from a counter

```
import { interval } from 'rxjs';

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

KindGeek School

Create an observable from an event

```
import { fromEvent } from 'rxjs';

const el = document.getElementById('my-element');

// Create an Observable that will publish mouse movements
const mouseMoves = fromEvent(el, 'mousemove');

// Subscribe to start listening for mouse-move events
const subscription = mouseMoves.subscribe((evt: MouseEvent) => {
  // Log coords of mouse movements
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

  // When the mouse is over the upper-left of the screen,
  // unsubscribe to stop listening for mouse movements
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
```

KindGeek School

Operators

Map operator

```
import { map } from 'rxjs/operators';

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);

squaredNums.subscribe(x => console.log(x));

// Logs
// 1
// 4
// 9
```

KindGeek School

Standalone pipe function

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

const nums = of(1, 2, 3, 4, 5);

// Create a function that accepts an Observable.
const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);

// Create an Observable that will run the filter and map functions
const squareOdd = squareOddVals(nums);

// Suscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));
```

KindGeek School

Observable.pipe function

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

const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );

// Subscribe to get values
squareOdd.subscribe(x => console.log(x));
```

Note that, for Angular apps, we prefer combining operators with pipes, rather than chaining. Chaining is used in many RxJS examples.

Common operators

AREA	OPERATORS
Creation	`from, fromPromise,fromEvent, of`
Combination	`combineLatest, concat, merge, startWith , withLatestFrom, zip`
Filtering	`debounceTime, distinctUntilChanged, filter, take, takeUntil`
Transformation	`bufferTime, concatMap, map, mergeMap, scan, switchMap`
Utility	`tap`
Multicasting	`share`

KindGeek School

Error handling

```
import { ajax } from 'rxjs/ajax';
import { map, catchError } from 'rxjs/operators';
// Return "response" from the API. If an error happens,
// return an empty array.
const apiData = ajax('/api/data').pipe(
  map(res => {
    if (!res.response) {
      throw new Error('Value expected!');
    }
    return res.response;
  }),
  catchError(err => of([]))
);

apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

KindGeek School

Retry failed observable

```
import { ajax } from 'rxjs/ajax';
import { map, retry, catchError } from 'rxjs/operators';

const apiData = ajax('/api/data').pipe(
  retry(3), // Retry up to 3 times before failing
  map(res => {
    if (!res.response) {
      throw new Error('Value expected!');
    }
    return res.response;
  }),
  catchError(err => of([]))
);

apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

Do not retry authentication requests, since these should only be initiated by user action. We don't want to lock out user accounts with repeated login requests that the user has not initiated.

KindGeek School

Naming conventions for observables

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html'
})
export class StopwatchComponent {

  stopwatchValue: number;
  stopwatchValue$: Observable<number>;

  start() {
    this.stopwatchValue$.subscribe(num =>
      this.stopwatchValue = num
    );
  }
}
```

KindGeek School

Naming conventions for observables

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html'
})
export class StopwatchComponent {

  stopwatchValue: number;
  stopwatchValue$: Observable<number>;

  start() {
    this.stopwatchValue$.subscribe(num =>
      this.stopwatchValue = num
    );
  }
}
```

Observables in Angular

- The EventEmitter class extends Observable.
- The HTTP module uses observables to handle AJAX requests and responses.
- The Router and Forms modules use observables to listen for and respond to user-input events.

KindGeek School

Event emitter

```
@Component({
  selector: 'zippy',
  template: `
  <div class="zippy">
    <div (click)="toggle()">Toggle</div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div>
  </div>`})

export class ZippyComponent {
  visible = true;
  @Output() open = new EventEmitter<any>();
  @Output() close = new EventEmitter<any>();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

KindGeek School

Async pipe

```
@Component({
  selector: 'async-observable-pipe',
  template: `<div><code>observable|async</code>:
       Time: {{ time | async }}</div>`
})
export class AsyncObservablePipeComponent {
  time = new Observable(observer =>
    setInterval(() => observer.next(new Date().toString()), 1000)
  );
}
```

Type-ahead suggestions

- Listen for data from an input.
- Trim the value (remove whitespace) and make sure it's a minimum length.
- Debounce (so as not to send off API requests for every keystroke, but instead wait for a break in keystrokes).
- Don't send a request if the value stays the same (rapidly hit a character, then backspace, for instance).
- Cancel ongoing AJAX requests if their results will be invalidated by the updated results.

```
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { map, filter, debounceTime, distinctUntilChanged, switchMap } from
'rxjs/operators';

const searchBox = document.getElementById('search-box');

const typeahead = fromEvent(searchBox, 'input').pipe(
  map((e: KeyboardEvent) => e.target.value),
  filter(text => text.length > 2),
  debounceTime(10),
  distinctUntilChanged(),
  switchMap(() => ajax('/api/endpoint'))
);

typeahead.subscribe(data => {
 // Handle the data from the API
});
```

KindGeek School

Observables compared to promises

Observables are often compared to promises. Here are some key differences:

- Observables are declarative; computation does not start until subscription. Promises execute immediately on creation. This makes observables useful for defining recipes that can be run whenever you need the result.
- Observables provide many values. Promises provide one. This makes observables useful for getting multiple values over time.
- Observables differentiate between chaining and subscription. Promises only have .then() clauses. This makes observables useful for creating complex transformation recipes to be used by other part of the system, without causing the work to be executed.
- Observables subscribe() is responsible for handling errors. Promises push errors to the child promises. This makes observables useful for centralized and predictable error handling.

Home task

Create search page.
As response return object with search text and timestamp.