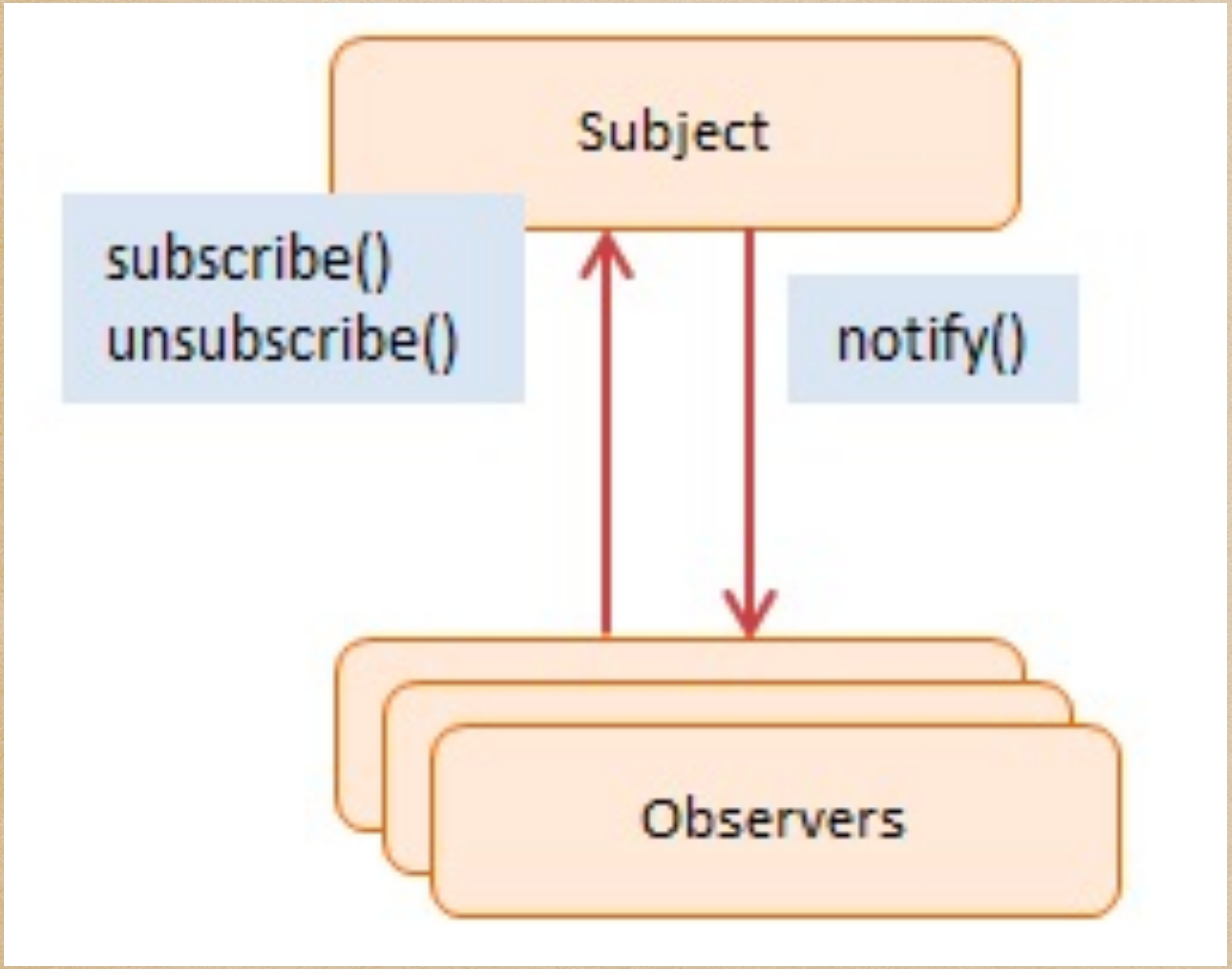
Observables in Angular

Yunqi Shen

Topics to be covered…

- The “Observer” Design Pattern
- Why We Need “Observables” (in Angular)
- Understanding the Key Terms
- Customized Usage
- Angular Modules that Implement Observables
- Rxjs that encapsulates Observables
- Observables vs. Promise

“Observer” Pattern



- Subject — Click
    - maintains list of observers. Any number of Observer objects may observe a Subject
    - implements an interface that lets observer objects subscribe or unsubscribe
    - sends a notification to its observers when its state changes
- Observers — clickHandler
    - has a function signature that can be invoked when Subject changes (i.e. event occurs)

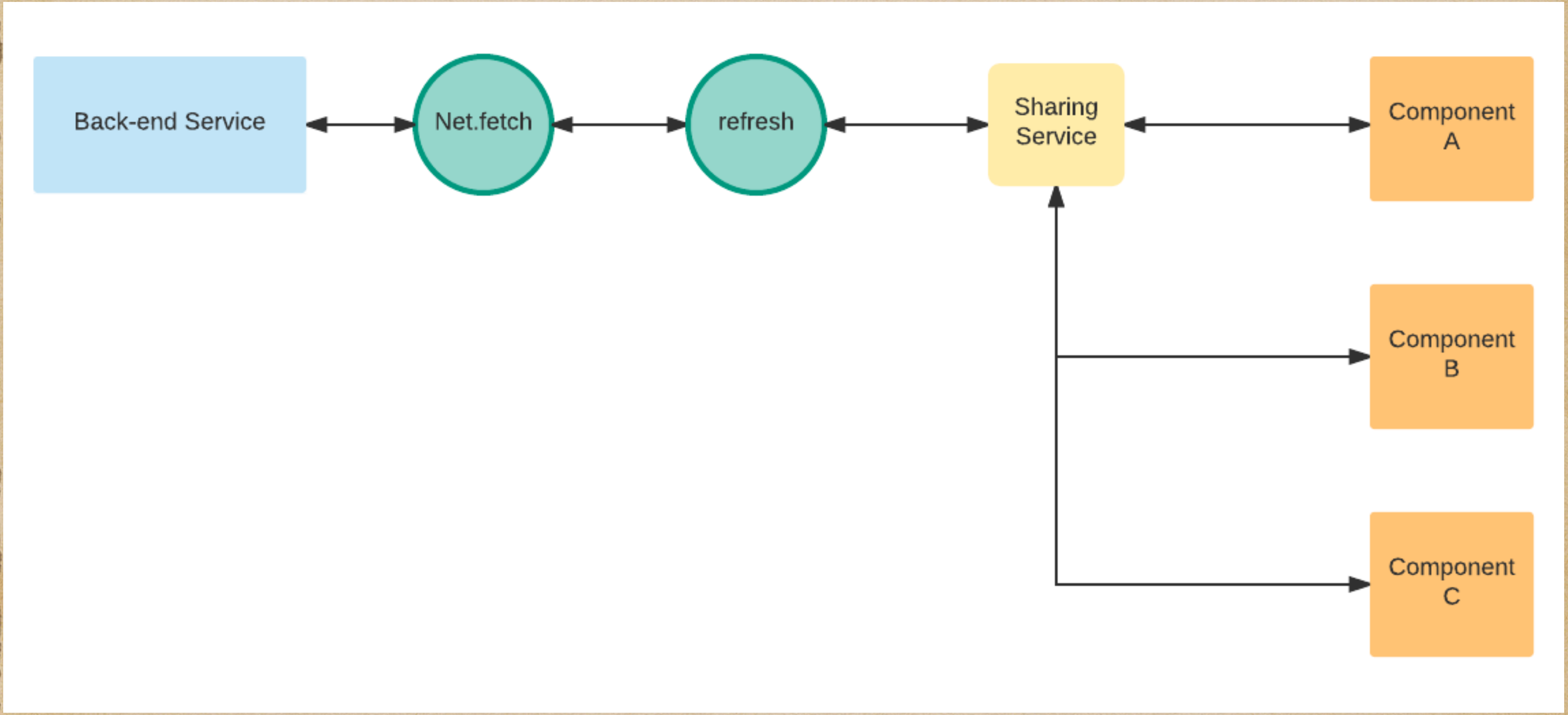
```javascript
function Click() {
    this.handlers = [];  // observers
}

Click.prototype = {

    subscribe: function(fn) {
        this.handlers.push(fn);
    },

    unsubscribe: function(fn) {
        this.handlers = this.handlers.filter(
            function(item) {
                if (item !== fn) {
                    return item;
                }
            }
        );
    },

    fire: function(o, thisObj) {
        var scope = thisObj || window;
        this.handlers.forEach(function(item) {
            item.call(scope, o);
        });
    }
}
```

```javascript
// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; }
        show: function() { alert(log); log = "";
    }
})();

function run() {

    var clickHandler = function(item) {
        log.add("fired: " + item);
    };

    var click = new Click();

    click.subscribe(clickHandler);
    click.fire('event #1');
    click.unsubscribe(clickHandler);
    click.fire('event #2');
    click.subscribe(clickHandler);
    click.fire('event #3');

    log.show();
}
```

Why “Observables”

- Observables provide support for passing messages between publishers and subscribers in an Angular application.
    - literals, messages, or events
- event handling, asynchronous programming, and handling multiple values
- keystrokes, an HTTP response, or an interval timer, the interface for listening to values and stopping listening is the same.

Why "Observables" in Angular



Data Flow w/o Observables

What is Observable

- Declarative
    - define a function for publishing values
    - not executed until a consumer subscribes
- `Observable()` constructor

Glossary

- Publisher
- Consumer/Subscriber
- Observer
- Stream
- Multicasting
- Error Handling
- Reactive Programming

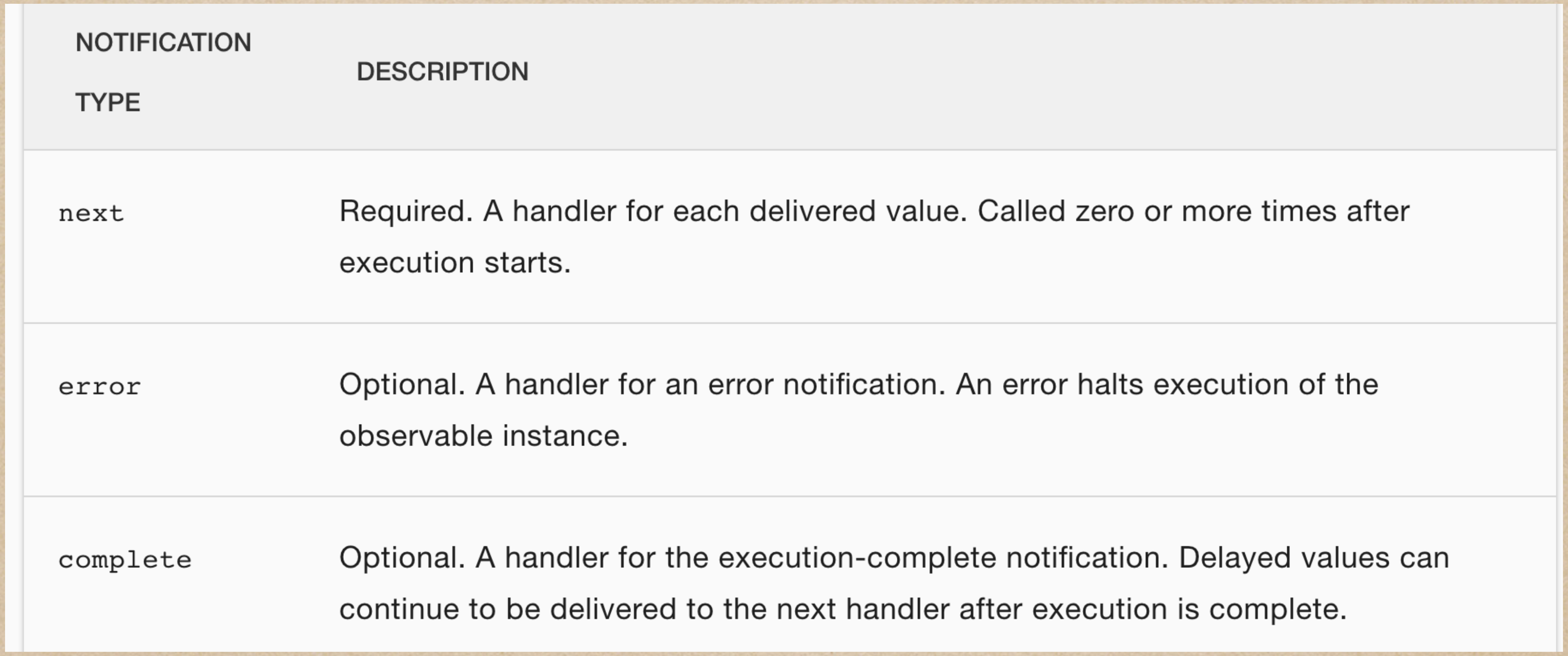
Basic Usage

- As a publisher, you create an Observable instance that defines a subscriber function. This is the function that is executed when a consumer calls the subscribe() method. The subscriber function defines how to obtain or generate values or messages to be published.
- To execute the observable you have created and begin receiving notifications, you call its subscribe() method, passing an observer. This is a JavaScript object that defines the handlers for the notifications you receive. The subscribe() call returns a Subscription object that has an unsubscribe() method, which you call to stop receiving notifications.

```
// 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;

  // Simple geolocation API check provides values to publish
  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

A handler for receiving observable notifications implements the Observer interface. It is an object that defines callback methods to handle the three types of notifications that an observable can send:



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.

An observer object can define any combination of these handlers. If you don't supply a handler for a notification type, the observer ignores notifications of that type.

Subscribing

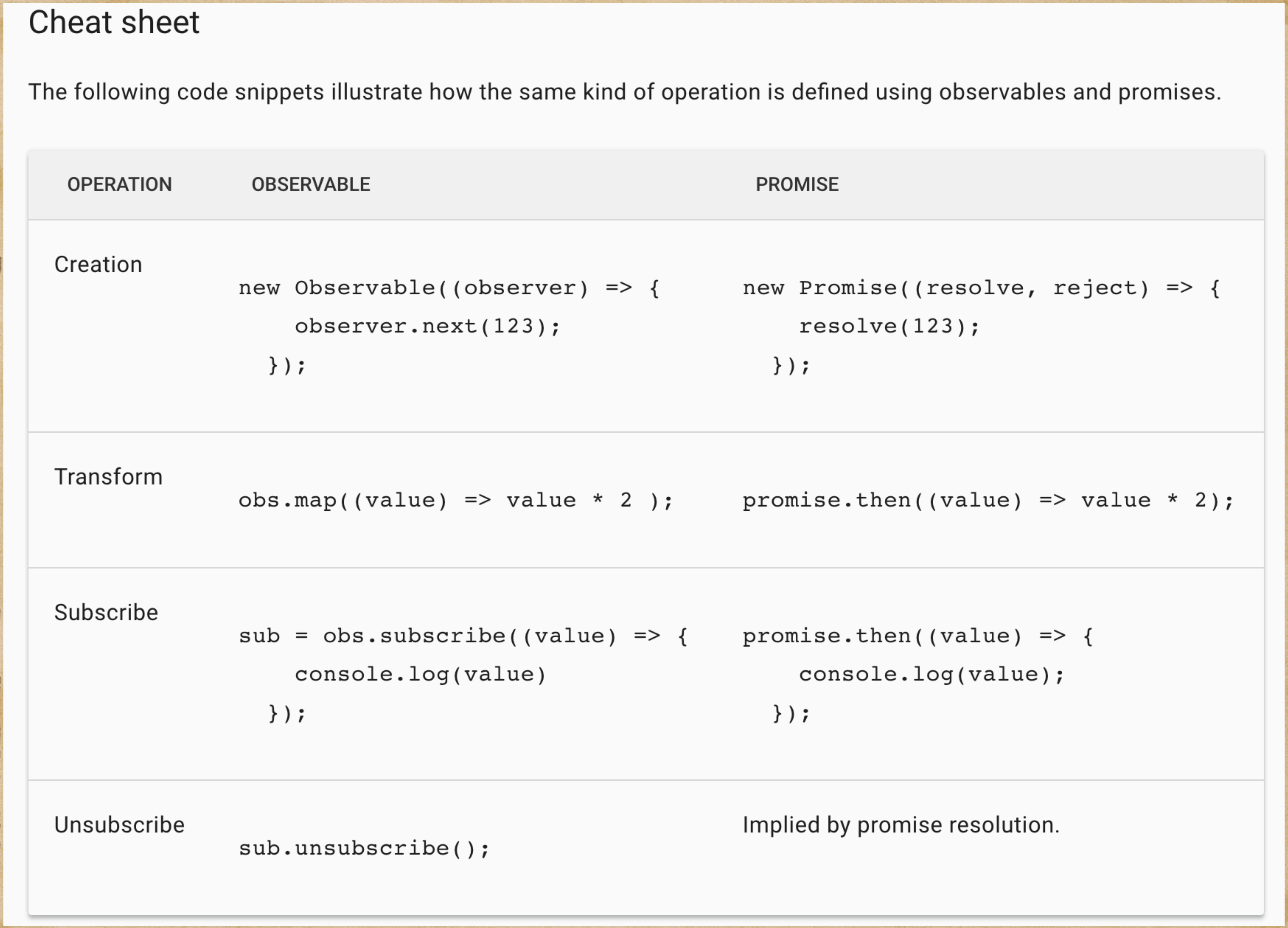
```
// Create simple observable that emits three values
const myObservable = of(1, 2, 3);

// 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
```

Inline Callback

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')
);
```

A next handler is required. The error and complete handlers are optional. Note that a next() function could receive, for instance, message strings, or event objects, numeric values, or structures, depending on context. Any type of value can be represented with an observable, and the values are published as a stream.



Cheat sheet

The following code snippets illustrate how the same kind of operation is defined using observables and promises.

OPERATION	OBSERVABLE	PROMISE

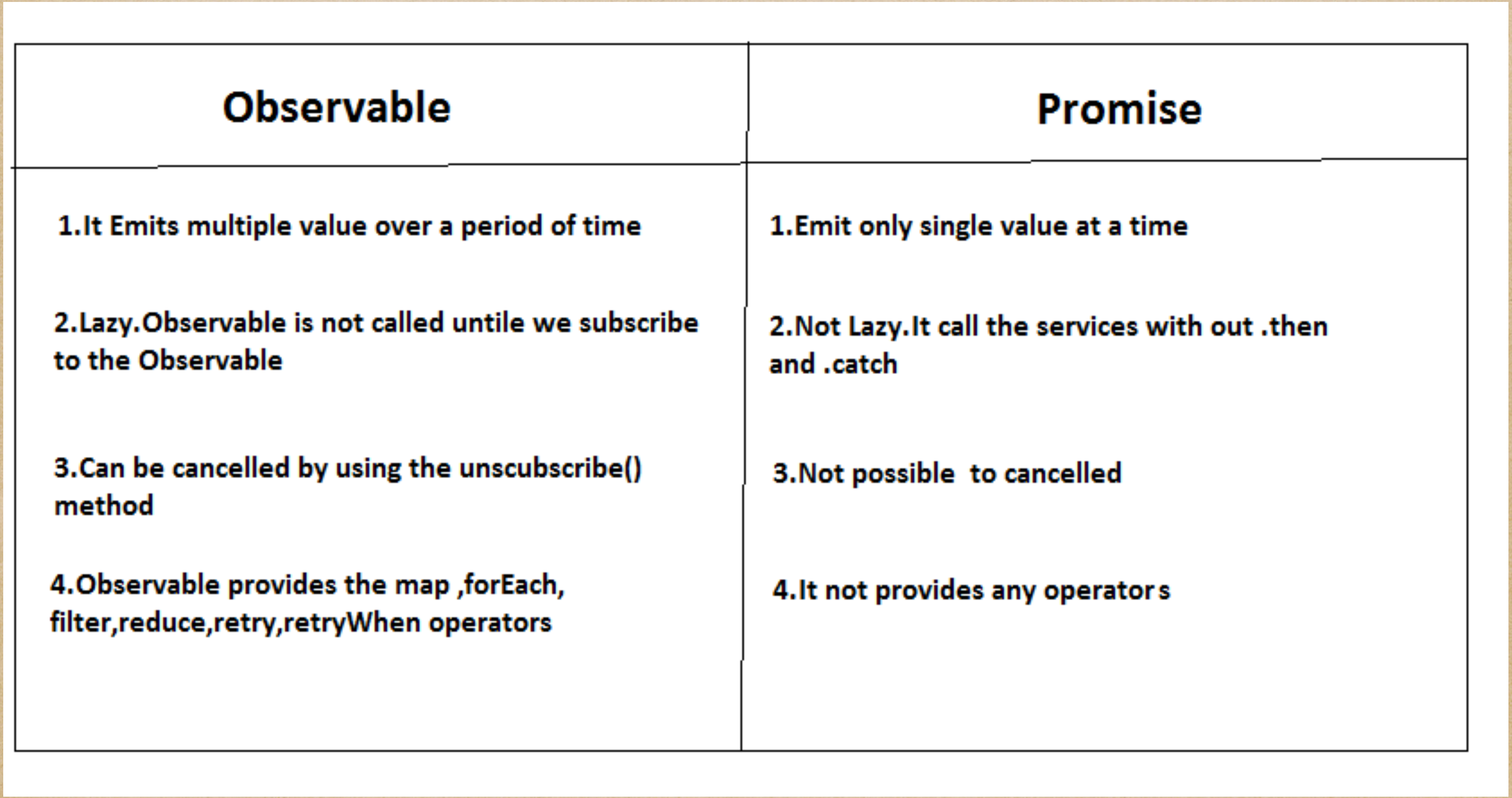
Creation	`new Observable((observer) => {`
	`    observer.next(123);`
	`  });`
		`new Promise((resolve, reject) => {`
		`    resolve(123);`
		`  });`

Transform	`obs.map((value) => value * 2 );`	`promise.then((value) => value * 2);`

Subscribe	`sub = obs.subscribe((value) => {`
	`    console.log(value)`
	`  });`
		`promise.then((value) => {`
		`    console.log(value);`
		`  });`

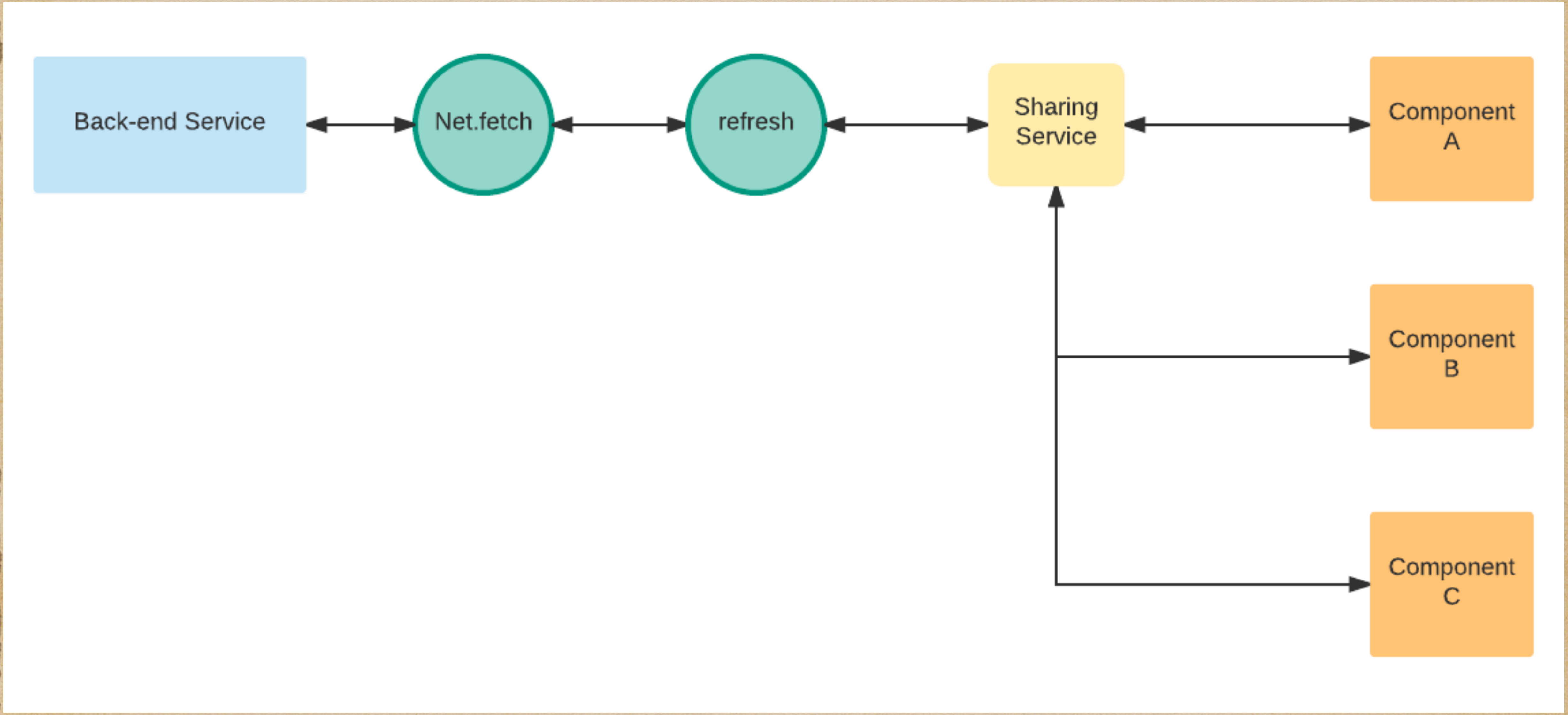
Unsubscribe	`sub.unsubscribe();`	Implied by promise resolution.

V.S. Promises



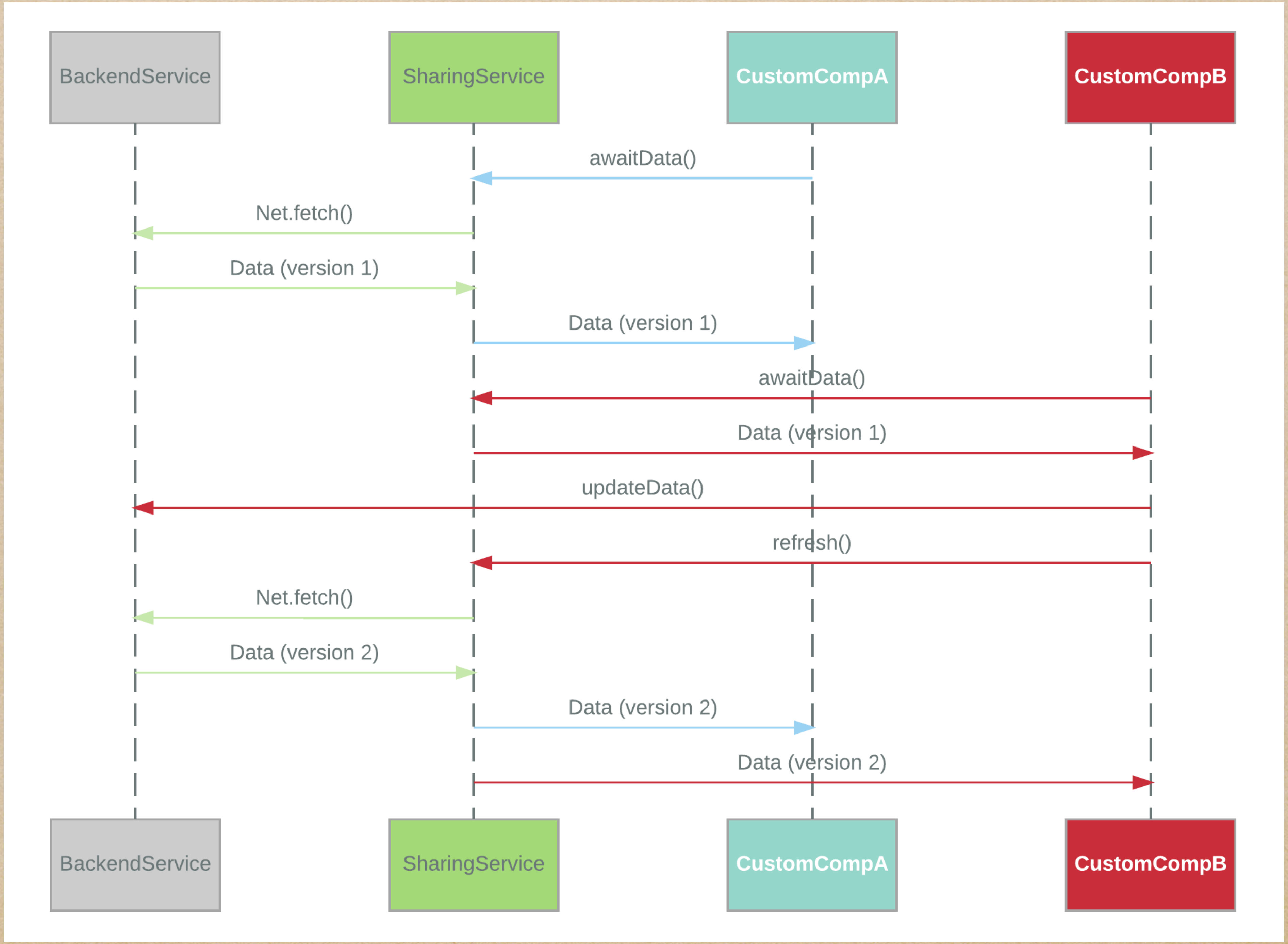
Observable	Promise
1.It Emits multiple value over a period of time	1.Emit only single value at a time
2.Lazy.Observable is not called untile we subscribe to the Observable	2.Not Lazy.It call the services with out .then and .catch
3.Can be cancelled by using the unscubscribe() method	3.Not possible to cancelled
4.Observable provides the map ,forEach, filter,reduce,retry,retryWhen operators	4.It not provides any operator s

Why "Observables"



Data Flow w/o Observables

Why "Observables"



Sequence Diagram w/ Observables

The RxJS Library

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

RxJS provides an implementation of the Observable type, which is needed until the type becomes part of the language and until browsers support it. The library also provides utility functions for creating and working with observables. These utility functions can be used for:

- 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

Create an observable from a promise

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

// Create an Observable out of a promise
const data = from(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'); }
});
```

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!`));
```

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();
  }
});
```

Create an observable that creates an AJAX request

```
import { ajax } from 'rxjs/ajax';

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

Observables in Angular

Angular makes use of observables as an interface to handle a variety of common asynchronous operations. For example:

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

EventEmitter

```
@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);
    }
  }
}
```

Angular provides EventEmitter that extends Observable, adding an emit() method so it can send arbitrary values. When you call emit(), it passes the emitted value to the next() method of any subscribed observer.
A good example of usage can be found on the EventEmitter documentation.
Here is the example component that listens for open and close events:

```
<zippy
(open)="onOpen($event)" (close)="onCl
ose($event)"></zippy>
```

HTTP Module

Angular's HttpClient returns observables from HTTP method calls. For instance, http.get('/api') returns an observable. This provides several advantages over promise-based HTTP APIs:

Observables do not mutate the server response (as can occur through chained .then() calls on promises). Instead, you can use a series of operators to transform values as needed.

HTTP requests are cancellable through the unsubscribe() method.

Requests can be configured to get progress event updates.

Failed requests can be retried easily.

Using 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)
  );
}
```

The AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes.
The following example binds the time observable to the component's view. The observable continuously updates the view with the current time.

Router events

```
import { Router, NavigationStart } from '@angular/router';
import { filter } from 'rxjs/operators';

@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable1Component implements OnInit {

  navStart: Observable<NavigationStart>;

  constructor(private router: Router) {
    // Create a new Observable that publishes only the NavigationStart event
    this.navStart = router.events.pipe(
      filter(evt => evt instanceof NavigationStart)
    ) as Observable<NavigationStart>;
  }

  ngOnInit() {
    this.navStart.subscribe(evt => console.log('Navigation Started!'));
  }
}
```

Router.events provides events as observables. You can use the filter() operator from RxJS to look for events of interest, and subscribe to them in order to make decisions based on the sequence of events in the navigation process.

ActivatedRoute

```
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable2Component implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) {}

   ngOnInit() {
     this.activatedRoute.url
       .subscribe(url => console.log('The URL changed to: ' + url));
   }
}

```

The ActivatedRoute is an injected router service that makes use of observables to get information about a route path and parameters. For example, ActivateRoute.url contains an observable that reports the route path or paths.

Reactive forms

```
import { FormGroup } from '@angular/forms';

@Component({
  selector: 'my-component',
  template: 'MyComponent Template'
})
export class MyComponent implements OnInit {
  nameChangeLog: string[] = [];
  heroForm: FormGroup;

   ngOnInit() {
     this.logNameChange();
   }
   logNameChange() {
     const nameControl = this.heroForm.get('name');
     nameControl.valueChanges.forEach(
       (value: string) => this.nameChangeLog.push(value)
     );
   }
}

```

Reactive forms have properties that use observables to monitor form control values.
The FormControlproperties valueChanges and statusChanges contain observables that raise change events.
Subscribing to an observable form-control property is a way of triggering application logic within the component class.

Reference

- https://angular.io/guide/observables
- https://angular-2-training-book.rangle.io/handout/observables/
- https://stackoverflow.com/questions/37364973/promise-vs-observable
- https://en.wikipedia.org/wiki/Reactive_programming
- https://www.dofactory.com/javascript/observer-design-pattern
- https://www.lucidchart.com/techblog/2016/11/08/angular-2-and-observables-data-sharing-in-a-multi-view-application/