Observable in JavaScript

Imagine you're waiting for a package to arrive at your doorstep. You don't know exactly when it will arrive, but you know it's on its way...

Avicsebooks

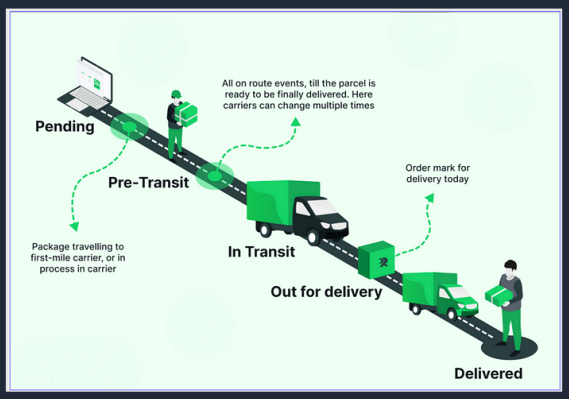
Follow

androidstudio · March 28, 2024 (Updated: March 29, 2024) · Free: Yes

Imagine you're waiting for a package to arrive at your doorstep. You don't know exactly when it will arrive, but you know it's on its way. You have two options for handling this situation:

1. **Promise Approach:** You could wait until the package arrives, and once it does, you'll get it. This is like using a promise. You make a request (to receive the package), and when it's ready, you get the result (the package).

2. Observable Approach: However, what if you want to track the progress of the package as it moves closer to your house? You might want to know when it's shipped from the warehouse, when it's out for delivery, and when it finally arrives. This is where observables come in. Instead of just waiting for the package to arrive, you're observing its journey every step of the way.



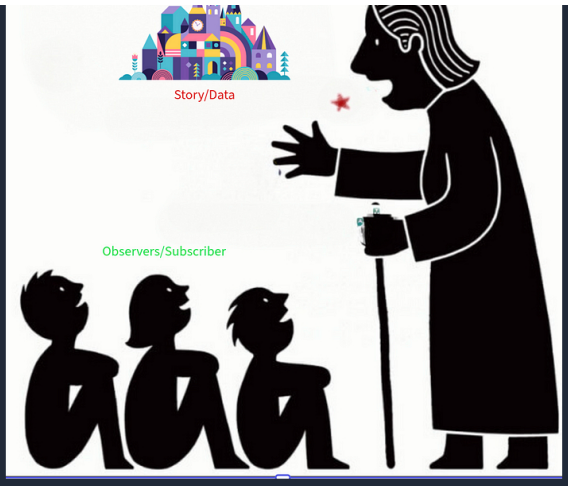
Observable Approach:

In JavaScript, observables work similarly. They allow you to track changes or data over time, rather than just waiting for a single result like with promises. With observables, you can observe a stream of events or data values and react to each one as they occur. It's like watching a movie where scenes (data or events) keep coming, and you can react to each scene in real-time.

So, in simpler terms, observables in JavaScript let you observe and react to a continuous stream of data or events, rather than waiting for a single outcome like with promises. They're useful when you need to handle ongoing processes or data streams, like user interactions, network requests, or data updates, and you want to respond to each event as it happens.

Requirement to be an Observable

To understand the basic requirements for an object to be an observable in JavaScript, let's break it down in layman's terms first:



Observer/Stream of Data / Subscriber

1. **Observable Object:** An observable object is like a storyteller who narrates a story (data or events) over time. It needs to have a way to start telling the story and a way for others to listen to the story as it unfolds.
2. **Start Telling the Story (Subscribe):** The storyteller (observable object) should have a method that allows others (subscribers) to start listening to the story. This method should provide a way for subscribers to react to each part of the story as it's told.
3. **Tell the Story (Emit Data/Events):** As the storyteller (observable object) narrates the story, it needs a way to emit different parts of the story (data or events) to its listeners (subscribers). Each emitted part of the story triggers a reaction from the listeners.
4. **Ability to Stop Listening (Unsubscribe):** Listeners (subscribers) should have the option to stop listening to the story if they're no longer interested. This is important for memory management and performance.

Now, let's dive into each requirement in more detail with examples:

Start Telling the Story (Subscribe):

- **Layman's Term:** The storyteller (observable object) should have a method called "subscribe" that allows listeners (subscribers) to start listening to the story.

story teller

- **Example:** Think of a radio station that broadcasts music. The radio station is the observable object, and listeners can subscribe to it by tuning in to a specific frequency on their radios.

Tell the Story (Emit Data/Events):

- **Layman's Term:** The storyteller (observable object) should have a way to emit different parts of the story (data or events) to its listeners (subscribers).

Emit Data

- **Example:** Continuing with the radio station analogy, as the radio station broadcasts music, it emits sound waves that carry the music to listeners'

radios. Each song played by the radio station is like an emitted part of the story.

Ability to Stop Listening (Unsubscribe):

- **Layman's Term:** Listeners (subscribers) should have the option to stop listening to the story if they're no longer interested.

unsubscribing

- **Example:** In our radio station analogy, listeners can turn off their radios or switch to a different frequency if they no longer want to listen to the music being broadcasted. This is similar to unsubscribing from an observable.

Here's a simple explanation of observables in JavaScript, along with an example implementation:

Observable Implementation in JavaScript

Copy

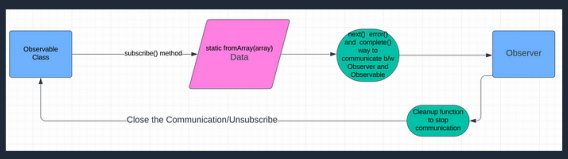
```
// Define an Observable class
class Observable {
  constructor(subscribe) {
    this._subscribe = subscribe;
  }
// Subscribe to the observable
  subscribe(observer) {
    return this._subscribe(observer);
  }
  // Create an observable from an array
  static fromArray(array) {
    return new Observable(observer => {
      array.forEach(item => observer.next(item));
      observer.complete();
    });
  }
}
// Example usage
const observable = new Observable(observer => {
  // Emit some values
  observer.next(1);
  observer.next(2);
  observer.next(3);
  // Emit an error
  // observer.error('Something went wrong');
  // Complete the observable
  observer.complete();
  // Cleanup function (optional)
  return () => {
    console.log('Observer unsubscribed');
  };
});
// Subscribe to the observable
const subscription = observable.subscribe({
  next: value => console.log('Received:', value),
  error: err => console.error('Error occurred:', err),
  complete: () => console.log('Observable completed')
});
// Unsubscribe (cleanup)
subscription.unsubscribe();
```

Explanation:

1. **Observable Class:** We define a class `Observable` representing an observable. It takes a `subscribe` function as a parameter in its constructor. This function will be called when someone subscribes to the observable.
2. **Subscribe Method:** The `subscribe` method is used to subscribe to the observable. It takes an `observer` object as a parameter. This object should have `next`, `error`, and `complete` methods to handle emitted values, errors, and the completion signal, respectively.
3. **Observer Object:** The `observer` object passed to the `subscribe` method defines how to handle emitted values, errors, and the completion signal. We've defined a simple observer object inline in the subscription.
4. **fromArray Static Method:** This static method of the `Observable` class creates

an observable from an array. It iterates over the array, emitting each item using `observer.next`, and then completes the observable using `observer.complete`.

5. **Example Usage:** We create an observable instance `observable` by passing a function to the `Observable` constructor. Inside this function, we emit some values using `observer.next`, emit an error using `observer.error` (commented out in the example), and finally complete the observable using `observer.complete`.
6. **Subscription:** We subscribe to the observable by calling its `subscribe` method. We pass an observer object defining how to handle emitted values, errors, and the completion signal.
7. **Unsubscribe (Cleanup):** Finally, we unsubscribe from the observable by calling `subscription.unsubscribe()`. If a cleanup function was returned from the `subscribe` function, it will be called at this point.



Explanation of each part.

This is a basic implementation of observables in JavaScript without using any third-party libraries. It helps understand the fundamental concepts of observables and how they can be used to represent asynchronous data streams. Keep in mind that this implementation is quite basic and lacks many features provided by libraries like RxJS.

What is the difference between Observer Pattern and observable in JS.

Observer Pattern:

- **Definition:** The Observer Pattern defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.
- **Key Components:**
- **Subject:** It maintains a list of observers, provides methods for adding and removing observers, and notifies observers of changes in state.
- **Observer:** It defines an interface for objects that should be notified of changes in the subject's state.
- **Example:**

Copy

```
// Subject
class Subject {
  constructor() {
    this.observers = [];
  }
addObserver(observer) {
    this.observers.push(observer);
  }
  removeObserver(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }
  notifyObservers() {
    this.observers.forEach(observer => observer.update());
  }
}
// Observer
class Observer {
  constructor(name) {
    this.name = name;
  }
  update() {
    console.log(`${this.name} has been notified of a change.`);
  }
}
// Example usage
const subject = new Subject();
const observer1 = new Observer("Observer 1");
const observer2 = new Observer("Observer 2");
subject.addObserver(observer1);
subject.addObserver(observer2);
subject.notifyObservers();
```

Observable (in libraries like RxJS):

- **Definition:** An Observable represents a stream of data or events over time. It can be subscribed to, and it emits values or notifications to its subscribers.
- **Key Components:**
- **Observable:** It represents the source of data or events.
- **Observer:** It consumes the values emitted by the Observable.
- **Example:**

Copy

```
// Example using RxJS
```

```
import { Observable } from 'rxjs';
// Create an Observable
const observable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.next('World');
  subscriber.complete();
});
// Subscribe to the Observable
observable.subscribe({
  next: value => console.log(value),
  complete: () => console.log('Observable completed')
});
```

In this example, when you subscribe to the Observable, you provide an object with functions to handle emitted values (`next`) and completion of the Observable (`complete`). This is different from the Observer Pattern, where observers are typically objects that implement a specific interface (`update` method).

In summary, the Observer Pattern is a design pattern for maintaining a one-to-many dependency between objects, while Observables, especially in libraries like RxJS, represent streams of data or events over time. Both are mechanisms for implementing reactive programming, but they differ in their implementation and usage patterns.

Difference between Observable from Rx.js and Mobx

RxJS and MobX are both libraries used in JavaScript for managing state and reacting to changes, but they have different purposes and approaches.

RxJS:

- **Purpose:** RxJS is a library for reactive programming using Observables. It provides a powerful set of tools for working with asynchronous data streams.
- **Key Concepts:**
- **Observable:** Represents a stream of data or events that can be observed over time. It can emit multiple values asynchronously.
- **Observer:** Consumes the values emitted by the Observable.
- **Operators:** Functions for manipulating the data emitted by Observables.
- **Example:**

Copy

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';
// Create an Observable that emits a value every second
const observable = interval(1000).pipe(
  take(5) // Take only 5 values
);
// Subscribe to the Observable
observable.subscribe(value => console.log(value));
```

In this example, `interval(1000)` creates an Observable that emits a value every second, and `take(5)` limits it to emitting only 5 values. The `subscribe` method is used to listen for these emitted values.

MobX:

- **Purpose:** MobX is a library for managing application state. It makes it simple to create reactive applications by automatically tracking the dependencies between state and the components that use it.
- **Key Concepts:**
- **Observable State:** Defines the state of the application, which can be observed by components. Changes to this state trigger reactions in dependent components.
- **Actions:** Functions that modify the state. MobX ensures that these modifications are tracked and reactions are triggered accordingly.
- **Reactions:** Side effects that occur in response to changes in observable state.
- **Example:**

Copy

```
import { observable, autorun } from 'mobx';
// Define observable state
const state = observable({
  count: 0
});
// Define a reaction
const disposer = autorun(() => {
  console.log('Count:', state.count);
});
// Update state
state.count++;
// Dispose the reaction
disposer();
```

In this example, `observable` is used to define an observable state property `count`. The `autorun` function creates a reaction that automatically runs whenever the `count` changes. When `state.count++` is called, the reaction logs the updated value of `count`.

Differences:

- **Purpose:** RxJS is primarily focused on handling asynchronous data streams, while MobX is focused on managing application state.
- **Approach:** RxJS uses the Observable pattern to represent streams of data, while MobX uses observable state to automatically track dependencies and trigger reactions.
- **Usage:** RxJS is commonly used for tasks like event handling, asynchronous operations, and data manipulation, while MobX is used for managing complex application state with minimal boilerplate.

In summary, RxJS and MobX serve different purposes in JavaScript development. RxJS is for handling asynchronous data streams, while MobX is for managing