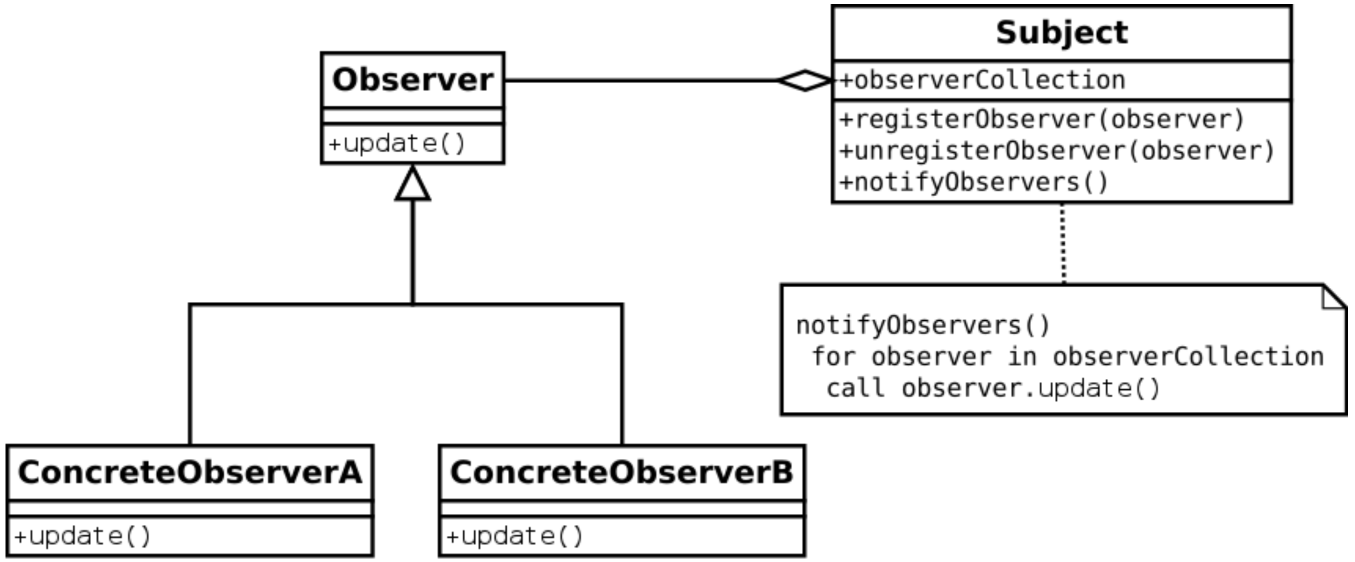
RxJS anatomy

AND MULTICASTING OPERATORS

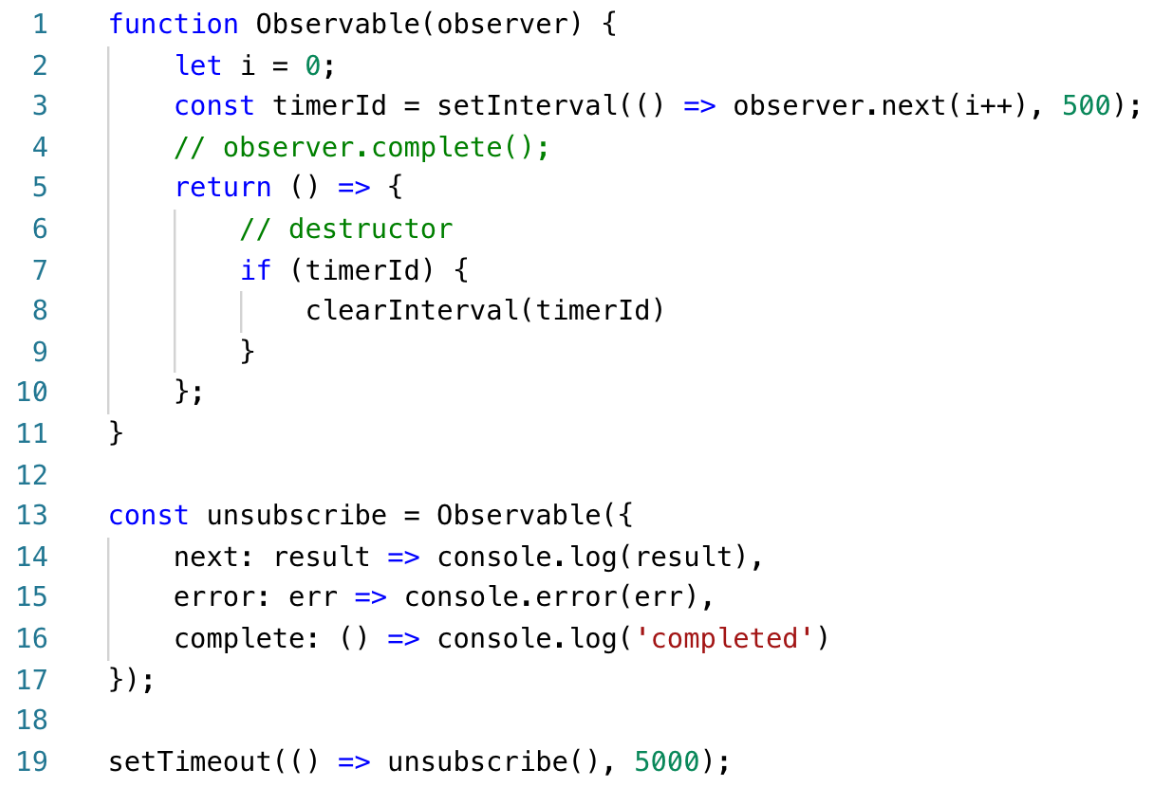
Observer pattern

- ***Observer*** is an interface introducing next, error and complete callback functions.
- ***Subject*** is a class extending Observable base class and implementing Observer interface.
- ***Observable*** is a class implementing subscribe functions.



Observable oversimplified

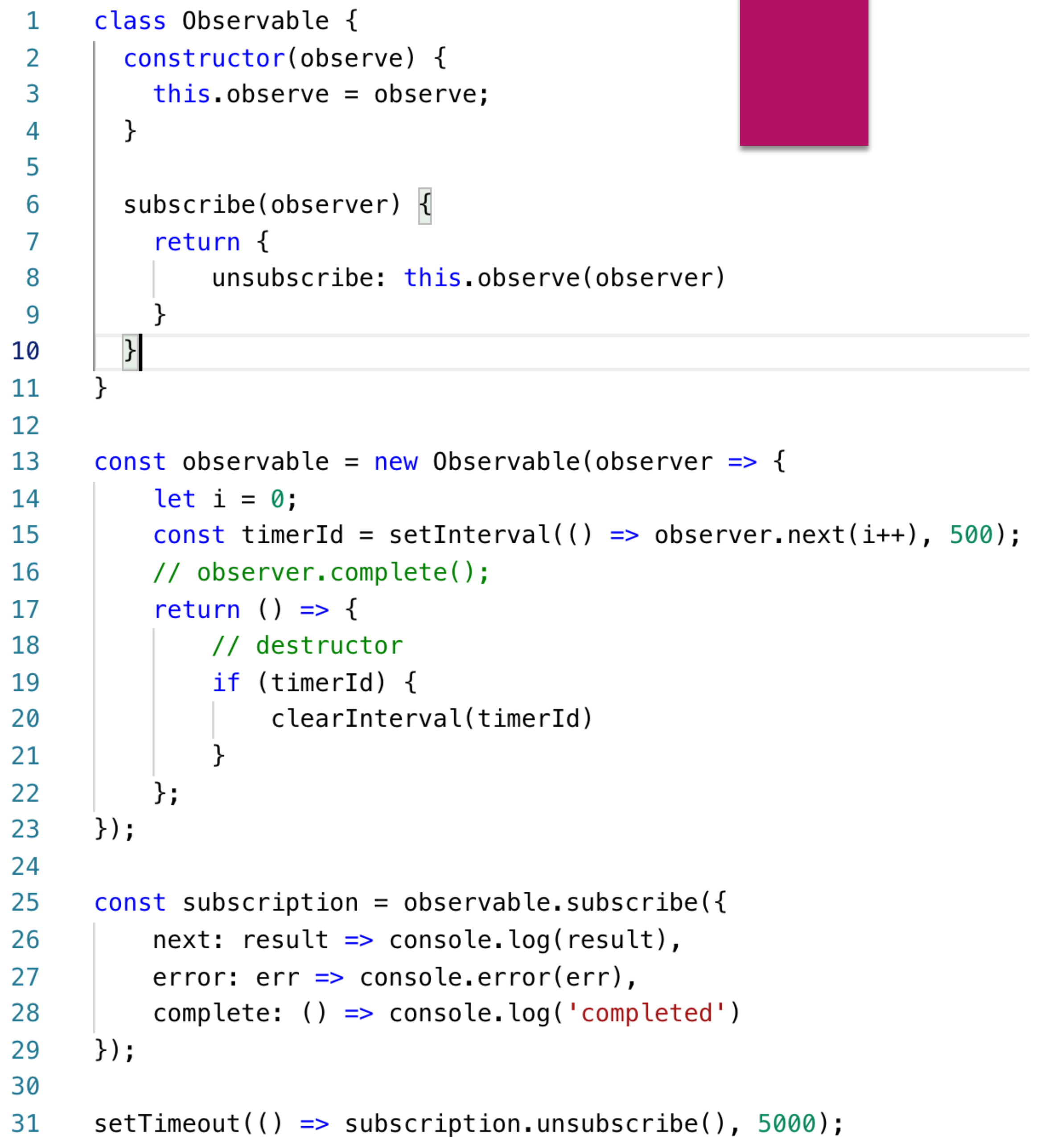
Cold



```javascript
function Observable(observer) {
    let i = 0;
    const timerId = setInterval(() => observer.next(i++), 500);
    // observer.complete();
    return () => {
        // destructor
        if (timerId) {
            clearInterval(timerId)
        }
    };
}

const unsubscribe = Observable({
    next: result => console.log(result),
    error: err => console.error(err),
    complete: () => console.log('completed')
});

setTimeout(() => unsubscribe(), 5000);
```



```javascript
class Observable {
  constructor(observe) {
    this.observe = observe;
  }

  subscribe(observer) {
    return {
        unsubscribe: this.observe(observer)
    }
  }
}

const observable = new Observable(observer => {
    let i = 0;
    const timerId = setInterval(() => observer.next(i++), 500);
    // observer.complete();
    return () => {
        // destructor
        if (timerId) {
            clearInterval(timerId)
        }
    };
});

const subscription = observable.subscribe({
    next: result => console.log(result),
    error: err => console.error(err),
    complete: () => console.log('completed')
});

setTimeout(() => subscription.unsubscribe(), 5000);
```

Subject oversimplified

Hot

```
const subject = new Subject();

subject.subscribe({
    next: result => console.log(result),
    error: err => console.error(err),
    complete: () => console.log('completed')
});

let i = 0;
const timerId = setInterval(() => subject.next(i++), 500);

setTimeout(() => {
    clearInterval(timerId);
    subject.complete();
}, 5000);
```

```
class Subject {
    constructor() {
        this.observers = [];
    }

    next(value) {
        for (let i = 0; i < this.observers.length; i++) {
            this.observers[i].next(value);
        }
    }

    error(err) {
        for (let i = 0; i < this.observers.length; i++) {
            this.observers[i].error(err);
        }
        this.observers = [];
    }

    complete() {
        for (let i = 0; i < this.observers.length; i++) {
            this.observers[i].complete();
        }
        this.observers = [];
    }

    subscribe(observer) {
        this.observers.push(observer);
        return () => {
            const subscriberIndex = this.observers.indexOf(observer);
            if (subscriberIndex > -1) {
                this.observers.splice(subscriberIndex, 1);
            }
        };
    }
}
```

Operators

- *Operator* is a function that takes a source observable and returns a new one to chain.
- Operator function is immutable and obeys to functional programming paradigm.

```
function map(predicate) {
    return (source) => new Observable((observer) => {
        return source.subscribe({
            next: value => observer.next(predicate(value)),
            error: err => observer.error(err),
            complete: () => observer.complete()
        });
    });
}
```

How to “warm up” a cold observable

- An observable is “cold” if it is created through constructor, static function create or creation operators like of, interval, range…
- To make a source Observable “hot”, observers should subscribe to a new Subject resubscribed to a source Observable.
- Or multicast operator can be used due to have this logic pipeable.

```
const subject = new Subject();

const connectedSubscription = observable.subscribe({
    next: result => subject.next(result),
    error: err => subject.error(err),
    complete: () => subject.complete()
});
```

```
const subject = new Subject();

const connectedSubscription = observable.subscribe(subject);
```

```
const multicasted = observable.pipe(
    multicast(() => new Subject())
);
const connectedSubscription = multicasted.connect();
```

Multicast operators: *publish…, share…*

Connectable observable in output

- *publish* is a simple multicast operator wrapper with simple Subject under the hood.
- *publishReplay* is the same wrapper, but with ReplaySubject under the hood.
- *publishLast* is the wrapper with AsyncSubject under the hood.
- *publishBehavior* is the wrapper with BehaviorSubject.

Use refCount for autoconnection

- *share* is a simple multicast operator wrapper with simple Subject and chained with refCount operator. So it is the same as obs.pipe(publish(), refCount())
- *shareReplay* is the same wrapper, but with ReplaySubject under the hood. The same as obs.pipe(publishReplay(), refCount())

Share operators use cases

- In Angular app with AsyncPipe: an Angular HttpClient request returns a “cold” observable, so any time an observer subscribes to it, a new http request is proceeded. Using multiple async pipes with such observables in one template can lead useless duplicate http requests. To avoid this we can chain the observable with share operator (make it “hot”), that means it will broadcast the same result from http request for multiple subscribers.
- In case if we use AsyncPipe inside dynamically rendered templates (inside *ngIf or *ngFor) even if we use share operator we can encounter with the same duplicate http requests due to async pipe subscribes during appearing in DOM and unsubscribes on disappearing. In the case we can use shareReplay(1). It will replay the latest cached result in the underlying ReplaySubject to a new subscribers.

Publish operators use cases

- A good example is imaging we have external timer library which starts a new timer counter for every single subscriber, so it is cold. And we need to display a result of this timer counter in multiple places reactively. And one of the requirements is an ability to control when a timer should start and stop counting. In this case we can use publish operator to make the observable “hot” (multicasted) and control the timer startup with connect() function of ConnectableObservable operator output.

Links

- Learning Observable By Building Observable
- Hot vs Cold Observables
- RxJS: multicast's Secret
- Angular Async Pipes—Beware the share
- RxJS open source
- C# 3.0 Design Patterns by Judith Bishop
- Design Patterns Elements of Reusable Object-Oriented Software by Gang Of Four