TABLE OF CONTENTS

Hot vs Cold Observables

Understanding the nature of hot and cold Observables is a crucial part to master Observables. Before we try to explore the topic through some practical examples, let's read through the definition from the RxJS project itself:

> *Cold observables start running upon subscription, i.e., the observable sequence only starts pushing values to the observers when Subscribe is called. (…) This is different from hot observables such as mouse move events or stock tickers which are already producing values even before a subscription is active.*

Ok, so far so good. Let's see what that means in practice.

We start with a basic Observable that simply emits the number `1` . We make two independent subscriptions to the same Observable and print out the output with a prefix so that we can tell them apart.

```
let obs = Rx.Observable.create(observer => observer.next(1));
```

```
obs.subscribe(v => console.log("1st subscriber: " + v));
obs.subscribe(v => console.log("2nd subscriber: " + v));
```

When we run this code we'll see the following output in the console.

```
1st subscriber: 1
2nd subscriber: 1
```

Ok, cool. But the interesting question remains: is `obs` cold or hot? Let's forget for a moment that we know how `obs` was created and imagine we would have obtained a reference to `obs` by calling `getObservableFromSomewhere()` instead. If that was the case, we wouldn't be able to tell whether it's cold or hot. And that's one important thing to understand. It's not always possible from the subscriber side to know whether you are dealing with a cold or hot Observable.

If we turn back to the definition that we cited in the beginning and think about what makes an Observable cold or hot, we can read between the lines and notice that if `obs` was cold it should produce *fresh* values upon subscription. But the pity is, with a value such as `1` we can't easily tell whether it was created freshly upon subscription or not. So let's replace `1` with `Date.now()` and see what happens.

If we run that code again we'll notice that we actually get different output per subscription. Notice the change in the last digit.

```
1st subscriber: 1465990942935
2nd subscriber: 1465990942936
```

With this output it is clear that there must have been two calls to `observer.next(Date.now())`. In other words, the Observable *started producing* the values upon each subscription which makes it cold by definition.

Making Cold Observables Hot

Now that we know that our Observable is clearly cold, let’s try to warm it up a little.

```
let obs = Rx.Observable
            .create(observer => observer.next(Date.now()))
            .publish();

obs.subscribe(v => console.log("1st subscriber: " + v));
obs.subscribe(v => console.log("2nd subscriber: " + v));

obs.connect();
```

Let’s ignore the `publish` and `connect` operators for a moment and solely focus on the output.

```
1st subscriber: 1465994477014
2nd subscriber: 1465994477014
```

Clearly we can see that there must have been only one call to `observer.next(Date.now())` with this setup as the numbers are identical. So, is it hot now? Well, kind of. Let’s put it that way: it’s warmer than a cold one but colder than a really hot one. It’s hot in the sense that there’s no new *value producer/source* (the thing calling `observer.next(val)`) created upon subscription. But it’s cold in the sense that it doesn’t start producing values *before* the subscriptions exist.

We can fix that by moving the `connect` call before the subscriptions.

```
let obs = Rx.Observable
            .create(observer => observer.next(Date.now()))
            .publish();
```

```
obs.connect();

obs.subscribe(v => console.log("1st subscriber: " + v));
obs.subscribe(v => console.log("2nd subscriber: " + v));
```

However, when we run this code we don't see any output at all. And that's expected behaviour because now `obs` is really really hot as in *it produces values no matter if anyone listens or not*. So by the time that our two subscribers start listening the value has long been pumped through.

In order to be able to understand the purpose of `publish` better, let's use an Observable that will produce infinite values instead of just a single one.

```
let obs = Rx.Observable
            .interval(1000)
            .publish()
            .refCount();

obs.subscribe(v => console.log("1st subscriber:" + v));
setTimeout(()
  // delay for a little more than a second and then add second subscriber
  => obs.subscribe(v => console.log("2nd subscriber:" + v)), 1100);
```

The setup we use is slightly more complex from what we had before, so let's break it down.

- We use `interval(1000)` to create an Observable that emits every second with an increasing index value starting at `0` .
- We use `publish` to share the *value producer* across several subscriptions (one indicator of being hot!)

- We defer the second subscription by one second.

Let’s just ignore `refCount` for now as we’re going to explain it later.

We see the following output as we run the script.

```
1st subscriber:0
1st subscriber:1
2nd subscriber:1
1st subscriber:2
2nd subscriber:2
...
```

Clearly, we can see that the second subscriber is not starting over at `0`. But that would have been the case if we didn’t use `publish` to share the value producing source across subscribers.

Understanding `publish`, `refCount` and `connect`

But how hot is `obs` in this scenario? Let’s make it visible by defering both subscriptions by another 2 seconds. If it’s really hot we shouldn’t see the numbers `0` and `1` at all because they would have been emitted *before* we start to listen, right?

Let’s try out and run this code instead.

```
let obs = Rx.Observable
            .interval(1000)
            .publish()
            .refCount();
```

```
setTimeout(() => {
  // delay both subscriptions by 2 seconds
  obs.subscribe(v => console.log("1st subscriber:" + v));
  setTimeout(
    // delay for a little more than a second and then add second subscriber
    () => obs.subscribe(
          v => console.log("2nd subscriber:" + v)), 1100);

},2000);
```

Interestingly we see exactly the same output as in the previous experiment which means we are dealing with an Observable that is rather warm than really hot. And that's because of the way `refCount` works. The `publish` operator creates an `ConnectableObservable` which means it creates an Observable that shares one single subscription to the underlying source. However, the `publish` operator doesn't subscribe to the underlying source just yet. It's more like a gatekeeper that makes sure that subscriptions aren't made to the underlying source but to the `ConnectableObservable` instead.

It's the job of the `connect` operator to actually cause the `ConnectableObservable` to subscribe to the underlying source (the thing that produces values). In our case we're using `refCount` which is an operator that builds up on `connect` and causes the `ConnectableObservable` to subscribe to the underlying source as soon as there is a first subscriber and to unsubscribe from it as soon as there's no subscriber anymore. It simply keeps track of how many subscriptions are made to the `ConnectableObservable`.

And this explains why we see values starting at `0` with our last experiment. It's because the subscription to the underlying source is only made once that we have a first subscriber to the `ConnectableObservable`.

If we want `obs` to be truly hot, we have to call `connect` ourselves early on.

```
let obs = Rx.Observable
            .interval(1000)
            .publish();
obs.connect();

setTimeout(() => {
  obs.subscribe(v => console.log("1st subscriber:" + v));
  setTimeout(
    () => obs.subscribe(v => console.log("2nd subscriber:" + v)), 1000);

},2000);
```

Notice how we get values starting from `2` when we run this code.

```
1st subscriber:2
1st subscriber:3
2nd subscriber:3
```

This means we now have a truly hot Observable that produces values no matter if someone listens or not.

When to use what

As we've seen the question whether an Observable is hot or cold is everything but black and white. In fact there are several strategies how values may be pushed to subscribers that we didn't even touch on yet. In general we can say that we should be dealing with a hot Observable whenever we subscribe to something that is generating values no matter if someone is listening or not. When we subscribe to such a hot Observable, we don't see past values but only new ones that were generated after our subscription.

A typical example of a hot observable are `mousemove` events. The mouse moves happen regardless if someone is listening or not. When we start listening for them, we only get future events.

Cold Observables on the other hand are the lazy ones. They only start producing values when someone subscribes. But then again, it's really not a black and white thing. An iced cold Observable starts reproducing the values it emits independently with every new subscriber as we've seen in the examples above. But how should we call an Observable that only starts generating values as the first subscriber subscribes and then shares and reemits the exact same values to every new subscriber? Things get blurry and categorizing only by cold and hot doesn't really cut it for every possible use case.

As a rule of thumb, when you have a cold Observable and you want multiple subscribers to it, and you don't want them to cause regenerating the values but rather reusing existing values, you need to start thinking about `publish` and friends.

Caveat: Http with Observables

The `Http` service in Angular >= 2.x returns cold Observables and the implications may surprise us.

Consider this simple component that requests a `contacts.json` file from a server and renders the contacts as a list in the template.

```
...
@Component({
  ...
  template: `
    <ul>
      <li *ngFor="let contact of contacts | async">{{contact.name}}</li>
    </ul>
    `
  ...
})
export class AppComponent {
  contacts: Observable<Array<any>>;
  constructor (http: Http) {
    this.contacts = http.get('contacts.json')
                        .map(response => response.json().items);
  }
}
```

As we can see above, we're exposing an `Observable<Array<any>>` to the template and subscribe to it from the template using the `AsyncPipe`.

The `contacts.json` contains a simple object with an `items` property holding the collection of contacts.

```
{
  "items": [
    { "name": "John Conner" },
    { "name": "Arnold Schwarzenegger" }
  ]
}
```

Now the interesting question is: What happens if we'd subscribe to it twice by adding another list to our template?

```
1st List
<ul>
  <li *ngFor="let contact of contacts | async">{{contact.name}}</li>
</ul>
2st List
<ul>
  <li *ngFor="let contact of contacts | async">{{contact.name}}</li>
</ul>
```

If we run that code and take a look at the network tab of our browser, we'll notice that the browser fetches `contacts.json` twice! And that's in stark contrast to what we're used to when working with Promises for example. In fact, simply importing the `toPromise` operator and adding it as the last operator to our Observable causes the second request to vanish.

But let's not fall back to Promises just yet. With all our knowledge regarding cold and hot Observables we should be able to simply fix our Observable so that it shares one single subscription to the underlying source - the Observable that issues the HTTP call.

```
this.contacts = http.get('contacts.json')
                    .map(response => response.json().items)
                    .publish()
                    .refCount();
```

That should do it, right? Well, almost. We can see that there's no second request anymore. But there's still a big issue that makes the code behave quite differently from what we are used with Promises.

Consider we'd want the second list to show up after a delay of 500ms. We could change the code to this.

```
@Component({
  ...
  template: `
    1st List
    <ul>
      <li *ngFor="let contact of contacts | async">{{contact.name}}</li>
    </ul>
    2st List
    <ul>
      <li *ngFor="let contact of contacts2 | async">{{contact.name}}</li>
    </ul>
    `,
  ...
})
export class AppComponent {
  contacts: Observable<Array<any>>;
  contacts2: Observable<Array<any>>;
  constructor (http: Http) {
    this.contacts = http.get('contacts.json')
                       .map(response => response.json().items)
                       .publish()
                       .refCount();

    setTimeout(() => this.contacts2 = this.contacts, 500);
  }
}
```

Notice that we are still using our one and only Observable but we are reexposing it as `contacts2` after 500ms. But when we run the code, we notice that our second list isn't showing up!

If you think about it, it makes perfect sense. With the use of `publish` we caused the Observable to share a single subscription to the underlying source - we made it hot. But maybe we made it a little too hot. Now when we have a second subscriber starting to listen after 500ms it will only get notified about new values that arrive after its subscription. What we rather want is that new subscribers see exactly the old values that were already emitted earlier. Fortunately we can have exactly that behaviour by using `publishLast` instead of `publish`.

```
this.contacts = http.get('contacts.json')
                   .map(response => response.json().items)
                   .publishLast()
                   .refCount();
```

Now when we run that code we see our second list after 500ms but we still don’t see a second request. In other words, we created an Observable that emits upon subscription with the data that was fetched with the first request.

Useful shortcut

Since using publish and refCount together is such a useful pattern there is an operator that combines them, and it’s called `.share()`

```
this.contacts = http.get('contacts.json')
  .map(response => response.json().items)
  .share();
```

Whew! We came a long way. We hope this gives you a clearer picture of what the term hot vs cold actually means when it comes to Observables.