Observable

Observables are lazy Push collections of multiple values. They fill the missing spot in the following table:

	SINGLE	MULTIPLE
Pull	`Function`	`Iterator`
Push	`Promise`	`Observable`

Example. The following is an Observable that pushes the values `1`, `2`, `3` immediately (synchronously) when subscribed, and the value `4` after one second has passed since the subscribe call, then completes:

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

const observable = new Observable(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
```

To invoke the Observable and see these values, we need to *subscribe* to it:

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

const observable = new Observable(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});

console.log('just before subscribe');
observable.subscribe({
  next(x) { console.log('got value ' + x); },
  error(err) { console.error('something wrong occurred: ' + err); },
  complete() { console.log('done'); }
});
console.log('just after subscribe');
```

Which executes as such on the console:

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

Pull versus Push

Pull and *Push* are two different protocols that describe how a data *Producer* can communicate with a data *Consumer*.

What is Pull? In Pull systems, the Consumer determines when it receives data from the data

Producer. The Producer itself is unaware of when the data will be delivered to the Consumer.

Every JavaScript Function is a Pull system. The function is a Producer of data, and the code that calls the function is consuming it by "pulling" out a *single* return value from its call.

ES2015 introduced generator functions and iterators (`function*`), another type of Pull system. Code that calls `iterator.next()` is the Consumer, "pulling" out *multiple* values from the iterator (the Producer).

	PRODUCER	CONSUMER
Pull	**Passive:** produces data when requested.	**Active:** decides when data is requested.
Push	**Active:** produces data at its own pace.	**Passive:** reacts to received data.

What is Push? In Push systems, the Producer determines when to send data to the Consumer. The Consumer is unaware of when it will receive that data.

Promises are the most common type of Push system in JavaScript today. A Promise (the Producer) delivers a resolved value to registered callbacks (the Consumers), but unlike functions, it is the Promise which is in charge of determining precisely when that value is "pushed" to the callbacks.

RxJS introduces Observables, a new Push system for JavaScript. An Observable is a Producer of multiple values, "pushing" them to Observers (Consumers).

- A **Function** is a lazily evaluated computation that synchronously returns a single value on invocation.
- A **generator** is a lazily evaluated computation that synchronously returns zero to (potentially) infinite values on iteration.
- A **Promise** is a computation that may (or may not) eventually return a single value.
- An **Observable** is a lazily evaluated computation that can synchronously or asynchronously return zero to (potentially) infinite values from the time it's invoked onwards.

> *For more info about what to use when converting Observables to Promises, please refer to this guide.*

Observables as generalizations of functions

Contrary to popular claims, Observables are not like EventEmitters nor are they like Promises for multiple values. Observables *may act* like EventEmitters in some cases, namely when they are multicasted using RxJS Subjects, but usually they don't act like EventEmitters.

> *Observables are like functions with zero arguments, but generalize those to allow multiple values.*

Consider the following:

```
function foo() {
  console.log('Hello');
  return 42;
}

const x = foo.call(); // same as foo()
console.log(x);
const y = foo.call(); // same as foo()
console.log(y);
```

We expect to see as output:

```
"Hello"
42
"Hello"
42
```

You can write the same behavior above, but with Observables:

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

const foo = new Observable(subscriber => {
  console.log('Hello');
  subscriber.next(42);
});

foo.subscribe(x => {
  console.log(x);
});
foo.subscribe(y => {
  console.log(y);
});
```

And the output is the same:

```
"Hello"
42
"Hello"
42
```

This happens because both functions and Observables are lazy computations. If you don't call the function, the `console.log('Hello')` won't happen. Also with Observables, if you don't "call" it (with `subscribe`), the `console.log('Hello')` won't happen. Plus, "calling" or "subscribing" is an isolated operation: two function calls trigger two separate side effects, and two Observable subscribes trigger two separate side effects. As opposed to EventEmitters which share the side effects and have eager execution regardless of the existence of subscribers, Observables have no shared execution and are lazy.

> *Subscribing to an Observable is analogous to calling a Function.*

Some people claim that Observables are asynchronous. That is not true. If you surround a function call with logs, like this:

```
console.log('before');
console.log(foo.call());
console.log('after');
```

You will see the output:

```
"before"
"Hello"
42
"after"
```

And this is the same behavior with Observables:

```
console.log('before');
foo.subscribe(x => {
  console.log(x);
});
console.log('after');
```

And the output is:

```
"before"
"Hello"
42
"after"
```

Which proves the subscription of `foo` was entirely synchronous, just like a function.

> *Observables are able to deliver values either synchronously or asynchronously.*

What is the difference between an Observable and a function? **Observables can "return" multiple values over time**, something which functions cannot. You can't do this:

```
function foo() {
  console.log('Hello');
  return 42;
  return 100; // dead code. will never happen
}
```

Functions can only return one value. Observables, however, can do this:

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

const foo = new Observable(subscriber => {
  console.log('Hello');
  subscriber.next(42);
  subscriber.next(100); // "return" another value
  subscriber.next(200); // "return" yet another
});

console.log('before');
foo.subscribe(x => {
  console.log(x);
});
console.log('after');
```

With synchronous output:

```
"before"
"Hello"
42
100
200
"after"
```

But you can also "return" values asynchronously:

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

const foo = new Observable(subscriber => {
  console.log('Hello');
  subscriber.next(42);
  subscriber.next(100);
  subscriber.next(200);
  setTimeout(() => {
    subscriber.next(300); // happens asynchronously
  }, 1000);
});

console.log('before');
foo.subscribe(x => {
  console.log(x);
});
console.log('after');
```

With output:

```
"before"
"Hello"
42
100
200
"after"
300
```

Conclusion:

- `func.call()` means "*give me one value synchronously*"
- `observable.subscribe()` means "*give me any amount of values, either synchronously or asynchronously*"

Anatomy of an Observable

Observables are **created** using `new Observable` or a creation operator, are **subscribed** to with

an Observer, **execute** to deliver `next` / `error` / `complete` notifications to the Observer, and their execution may be **disposed**. These four aspects are all encoded in an Observable instance, but some of these aspects are related to other types, like Observer and Subscription.

Core Observable concerns:

- **Creating** Observables
- **Subscribing** to Observables
- **Executing** the Observable
- **Disposing** Observables

Creating Observables

The `Observable` constructor takes one argument: the `subscribe` function.

The following example creates an Observable to emit the string `'hi'` every second to a subscriber.

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

const observable = new Observable(function subscribe(subscriber) {
  const id = setInterval(() => {
    subscriber.next('hi')
  }, 1000);
});
```

> *Observables can be created with `new Observable`. Most commonly, observables are created using creation functions, like `of`, `from`, `interval`, etc.*

In the example above, the `subscribe` function is the most important piece to describe the Observable. Let's look at what subscribing means.

Subscribing to Observables

The Observable `observable` in the example can be *subscribed* to, like this:

```
observable.subscribe(x => console.log(x));
```

It is not a coincidence that `observable.subscribe` and `subscribe` in `new Observable(function subscribe(subscriber) {...})` have the same name. In the library, they are different, but for practical purposes you can consider them conceptually equal.

This shows how `subscribe` calls are not shared among multiple Observers of the same Observable. When calling `observable.subscribe` with an Observer, the function `subscribe` in `new Observable(function subscribe(subscriber) {...})` is run for that given subscriber. Each call to `observable.subscribe` triggers its own independent setup for that given subscriber.

> *Subscribing to an Observable is like calling a function, providing callbacks where the data will be delivered to.*

This is drastically different to event handler APIs like `addEventListener` / `removeEventListener`. With `observable.subscribe`, the given Observer is not registered as a listener in the Observable. The Observable does not even maintain a list of attached Observers.

A `subscribe` call is simply a way to start an "Observable execution" and deliver values or events to an Observer of that execution.

Executing Observables

The code inside `new Observable(function subscribe(subscriber) {...})` represents an "Observable execution", a lazy computation that only happens for each Observer that subscribes. The execution produces multiple values over time, either synchronously or asynchronously.

There are three types of values an Observable Execution can deliver:

- "Next" notification: sends a value such as a Number, a String, an Object, etc.
- "Error" notification: sends a JavaScript Error or exception.
- "Complete" notification: does not send a value.

"Next" notifications are the most important and most common type: they represent actual data being delivered to a subscriber. "Error" and "Complete" notifications may happen only once during the Observable Execution, and there can only be either one of them.

These constraints are expressed best in the so-called *Observable Grammar* or *Contract*, written as a regular expression:

```
next*(error|complete)?
```

> *In an Observable Execution, zero to infinite Next notifications may be delivered. If either an Error or Complete notification is delivered, then nothing else can be delivered afterwards.*

The following is an example of an Observable execution that delivers three Next notifications, then completes:

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

const observable = new Observable(function subscribe(subscriber) {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();
});
```

Observables strictly adhere to the Observable Contract, so the following code would not deliver the Next notification `4`:

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

const observable = new Observable(function subscribe(subscriber) {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();
  subscriber.next(4); // Is not delivered because it would violate the
contract
});
```

It is a good idea to wrap any code in `subscribe` with `try`/`catch` block that will deliver an Error notification if it catches an exception:

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

const observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(1);
    subscriber.next(2);
    subscriber.next(3);
    subscriber.complete();
  } catch (err) {
    subscriber.error(err); // delivers an error if it caught one
  }
});
```

Disposing Observable Executions

Because Observable Executions may be infinite, and it's common for an Observer to want to abort execution in finite time, we need an API for canceling an execution. Since each execution is exclusive to one Observer only, once the Observer is done receiving values, it has to have a way to stop the execution, in order to avoid wasting computation power or memory resources.

When `observable.subscribe` is called, the Observer gets attached to the newly created Observable execution. This call also returns an object, the `Subscription`:

```
const subscription = observable.subscribe(x => console.log(x));
```

The Subscription represents the ongoing execution, and has a minimal API which allows you to cancel that execution. Read more about the `Subscription` type here. With `subscription.unsubscribe()` you can cancel the ongoing execution:

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

const observable = from([10, 20, 30]);
const subscription = observable.subscribe(x => console.log(x));
// Later:
subscription.unsubscribe();
```

> *When you subscribe, you get back a Subscription, which represents the ongoing execution. Just call `unsubscribe()` to cancel the execution.*

Each Observable must define how to dispose resources of that execution when we create the Observable using `create()`. You can do that by returning a custom `unsubscribe` function from within `function subscribe()`.

For instance, this is how we clear an interval execution set with `setInterval`:

```
const observable = new Observable(function subscribe(subscriber) {
  // Keep track of the interval resource
  const intervalId = setInterval(() => {
    subscriber.next('hi');
  }, 1000);

  // Provide a way of canceling and disposing the interval resource
  return function unsubscribe() {
    clearInterval(intervalId);
  };
});
```

Just like `observable.subscribe` resembles `new Observable(function subscribe() {...})`, the `unsubscribe` we return from `subscribe` is conceptually equal to `subscription.unsubscribe`. In fact, if we remove the ReactiveX types surrounding these concepts, we're left with rather straightforward JavaScript.

```
function subscribe(subscriber) {
  const intervalId = setInterval(() => {
    subscriber.next('hi');
  }, 1000);

  return function unsubscribe() {
    clearInterval(intervalId);
  };
}

const unsubscribe = subscribe({next: (x) => console.log(x)});

// Later:
unsubscribe(); // dispose the resources
```

The reason why we use Rx types like Observable, Observer, and Subscription is to get safety (such as the Observable Contract) and composability with Operators.