JavaScript Promises vs. RxJS Observables

Comparing JavaScript Asynchronous Programming Techniques

Daniel Weibel Follow
Sep 25, 2018 · 27 min read

A while ago I wrote an article about JavaScript promises and Node.js. In the present article I'm comparing the native JavaScript **promises**, that were introduced in **ES6**, with **observables**, that are provided by the **RxJS** library.

The focus is on highlighting the differences and similarities of promises and observables. The goal is to make it easier to understand observables if you already know promises (or vice versa). For this reason, I don't treat the RxJS operators in this article, because there exists nothing comparable to these operators for promises.

Note that this article is based on **RxJS 6**, the newest version of RxJS, that has been released on 24 April 2018.

Contents

Photo by Eric Perez on Unsplash.

Asynchronous Programming in JavaScript

First of all, let's recall what promises and observables are all about: handling asynchronous execution. There are different ways in JavaScript to create asynchronous code. The most important ones are the following:

- **Callbacks**
- **Promises**

- **Async/Await**
- **RxJS Observables**

Let's briefly introduce each of them.

Callbacks

This is the old-fashioned classical approach to asynchronous programming. You provide a function as an argument to another function that executes an asynchronous task. When the asynchronous task completes, the executing function calls your callback function.

The main disadvantage of this approach occurs when you have multiple chained asynchronous tasks, which requires you to define callback functions within callback functions within callback functions… This is called **callback hell**.

Promises

Promises have been introduced in **ES6** (2015) to allow for more readable asynchronous code than is possible with callbacks.

The main difference between callbacks and promises is that with callbacks you **tell the executing function** what to do when the asynchronous task completes, whereas with promises the executing function returns a special object to you (the promise) and then you **tell the promise** what to do when the asynchronous task completes.

In practice, this looks like this:

```
const promise = asyncFunc();

promise.then(result => {
    console.log(result);
});
```

That is, instead of providing a function reference as an argument to `asyncFunc` (as you would with callbacks), `asyncFunc` immediately returns a promise to you, and then you provide the action to be taken when the asynchronous task completes to this promise (through its `then` method).

Async/Await

Async/await has been introduced in **ES8** (2017). This technique should really be listed under *promises*, because it is just syntactic sugar for working with promises. However, it is a syntactic sugar that is really worth looking at.

Basically, you can declare a function to be `async`, which allows you to use the `await` keyword in the body of this function. The `await` keyword can be put in front of an expression that evaluates to a *promise*. The `await` keyword pauses the execution of the `async` function until the promise is resolved. When this happens, the entire `await` expression evaluates to the result value of the promise, and then the execution of the `async` function resumes.

Furthermore, the `async` function itself returns a promise as well that is resolved when the execution of the function body completes.

Let's see how this looks like in practice with the following example:

```
function asyncTask(i) {
    return new Promise(resolve => resolve(i + 1));
}

async function runAsyncTasks() {
    const res1 = await asyncTask(0);
    const res2 = await asyncTask(res1);
    const res3 = await asyncTask(res2);
    return "Everything done"
}

runAsyncTasks().then(result => console.log(result));
```

The `asyncTask` function implements an asynchronous task that takes an argument and returns a result. The function returns a promise that is resolved when the asynchronous task completes. There is nothing special about this function, it's just a normal function that returns a promise.

The `runAsyncTasks` function, on the other hand, is declared `async` so that the `await` keyword can be used in its body. This function calls `asyncTask` three times, and each time the argument must be the result of the preceding call to `asyncTask` (i.e. we are chaining three asynchronous tasks).

The first `await` keyword causes the execution of `runAsyncTasks` to stop until the promise returned by `asyncTask(0)` is resolved. The entire `await asyncTask(0)` expression then evaluates to the result value of the resolved promise and is assigned to `res1` . At this point, `asyncTask(res1)` is called, and the second `await` keyword causes execution of `runAsyncTasks` to stop again until the promise returned by `asyncTask(res1)` is resolved. This continues until eventually all the statements in the `runAsyncTasks` function body have been executed.

As mentioned, an `async` function returns a promise itself that is resolved with the function's return value when the execution of the function body completes. So, in other words, an `async` function is itself an asynchronous task (that typically manages the execution of other asynchronous tasks). This can be seen in the last line where we call the `then` function on the returned promise to print out the `async` function's return value.

If you add log statements after each assignment statement in the `runAsyncTasks` function , then the output will look like this:

```
1
2
3
Everything done
```

I claimed that async/await is just syntactic sugar for promises. If this is true, then we must be able to implement the above example with pure promises. Yes, we can, and this is how it would look like:

```
function asyncTask(i) {
    return new Promise(resolve => resolve(i + 1));
}


function runAsyncTasks() {
    return asyncTask(0)
        .then(res1 => { return asyncTask(res1); })
        .then(res2 => { return asyncTask(res2); })
        .then(res3 => { return "Everything done"; });
}

runAsyncTasks().then(result => console.log(result));
```

This code is equivalent to the the async/await version, and if you add appropriate log statements in the anonymous function bodies, then it produces the same output as the async/await version.

The one thing that has changed is the `runAsyncTasks` function. It is now a regular function (not `async`) and it uses `then` to chain the promises returned by `asyncTask` (instead of `await`).

I think it is needless to say that the async/await version is much more readable and easier to understand than the promise version. In fact, the main innovation of async/await is to allow to write asynchronous code with promises that "looks like" synchronous code.

RxJS Observables

To begin with, RxJS is the JavaScript implementation of the **ReactiveX** project. The ReactiveX project aims at providing an API for asynchronous programming for different programming languages.

The fundamental paradigm of ReactiveX is the Gang of Four **observer pattern** (ReactiveX even extends the observer pattern with completion and error notifications). Therefore, the central abstraction of all ReactiveX implementations is the **observable**. You can read more about the fundamental concepts of ReactiveX here.

The ReactiveX API is implemented in various programming languages. As mentioned, **RxJS** is the JavaScript implementation of ReactiveX. Besides that, there exist, for example, **RxJava** (Java), **RxKotlin** (Kotlin), **Rx.rb** (Ruby), **RxPY** (Python), **RxSwift** (Swift), **Rx.NET** (C#) implementations, and many more (see overview here).

That means if you understand observables in RxJS, then you also understand observables in RxJava or Rx.NET, or any other implementation, and you can use these libraries without having to learn new concepts.

So, now we know what RxJS is, but **what is an observable?** Let's try to characterise it along two dimensions and compare it with other known abstractions. The dimensions are **synchronicity/asynchronicity** and **single value/multiple values**.

Regarding an **observable**, we can say that the following is true:

- **Emits multiples values**
- **Emits its values asynchronously ("push")**

Let's contrast this with **promises**, that we just introduced in the previous subsection:

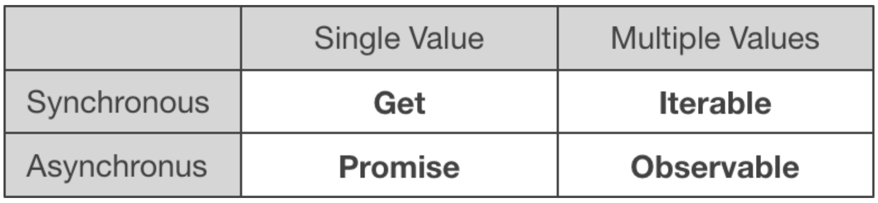
- **Emits a single value**
- **Emits its value asynchronously ("push")**

Finally, let's look at an **iterable**. This is an abstraction that exists in many programming languages and can be used to iterate through all the elements of a collection data structure, such as an array. For an iterable, the following holds true:

- **Emits multiple values**
- **Emits its values synchronously ("pull")**

Note that with *synchronous/pull* and *asynchronous/push* I mean the following: **synchronous/pull** means that the client code *requests* a value from the abstraction and *blocks* until this value is returned. **Asynchronous/push** means that the abstraction *notifies* the client code that a new value is being emitted, and the client code *handles* this notification.

If we arrange these abstractions along the dimensions graphically, we get the following picture (taken from ReactiveX):



	Single Value	Multiple Values
Synchronous	**Get**	**Iterable**
Asynchronus	**Promise**	**Observable**

Note that in some programming language promises are called futures (e.g. in Java).

Note that we didn't yet mention **Get**, but this stands just for a normal data access operation such as regular function call.

Looking at above picture, we could say that an *observable* is to an *iterable* what a *promise* is to a *get* operation. Or that a *promise* is like an *asynchronous get* operation whereas an *observable* is like an *asynchronous iterable*.

We could also say that the main difference between a promise and an observable is that a promise emits only a single value, whereas an observable emits multiple values.

But let's look at it in more detail. With a simple **get** operation, for example a function call, the calling code requests a single value and then waits or blocks until the function returns this value (the calling code *pulls* the value).

With a **promise**, on the other hand, the calling code also request a single value, but it doesn't block until the value is returned. It just kicks off the computation and then goes on with the execution of its own code. When the promise finishes the computation of the value, it emits the value to the calling code, which then handles the value (the value is *pushed* to the calling code).

Now, let's look at an **iterable**. In many programming languages we can create an iterable from a collection data structure, such as an array. The iterable typically has a `next` method, which returns the next unread value from the collection. The calling code can then repeatedly call `next` in order to read all the values of the collection. Each `next` call is basically a synchronous blocking *get* operation as explained above (the calling code repeatedly *pulls* values).

An **observable** takes the iterable to the asynchronous world. Like an iterable, an observable computes and emits a stream of values. However, unlike an iterable, with an observable the calling code does not synchronously *pull* each value, but the observable asynchronously *pushes* each value to the calling code, as soon as it is available. To this end, the calling code provides a handler function to the observable, which in RxJS is called `next` , and the observable then calls this function for each value that it computes.

The values that an observable emits can be anything: the elements of an array, the result of an HTTP request (it's OK if an observable emits just a single value, it doesn't always have to be multiple values), user input events, such as mouse clicks, etc. This makes observables very **flexible**. Furthermore, since an observable can also emit only a single value, an observable can do everything that a promise can do, but the reverse is not true.

In addition to this, ReactiveX observables provide a large number of so-called **operators**. These are functions that can be applied to an observable in order to modify the set of emitted values. Common categories of operators are *combinations*, *filters*, and *transformations*.

For example, there is a `map` operator that we could configure as follows: `map(value => 2 * value)` , and then we can apply this operator to an observable. The effect is that each value that the observable emits is multiplied by two before it is pushed to the calling code.

The complete list of operators in RxJS can be found here. However, in this article I don't treat operators, since the focus of this article is on a comparison of promises and observables, and promises don't have anything comparable to the observable operators.

The following is a short code example showing the creation and usage of an RxJS observable (the syntax for creating and using observables will be explained in the next section):

```
// Creation
const observable = new Observable(observer => {
  for (let i = 0; i < 3; i++) {
    observer.next(i);
  }
});

// Usage
observable.subscribe(value => console.log(value));
```

This concludes our overview of asynchronous programming techniques in JavaScript. We have seen that there **callbacks**, which are old-fashioned, **promises**, which can be used to obtain a single value asynchronously, **async/await**, which is syntactic sugar for promises, and RxJS **observables**, which can be used to obtain streams of values asynchronously.

In the next section we are going to highlight differences and similarities between promises and observables specifically.

Promises vs. Observables

In this section we compare promises and observables side by side and highlight their differences and similarities.

Note that if you want to run the following code examples that include observables, you have to install and import the RxJS library.

You can install RxJS as follows:

```
npm install --save rxjs
```

And you can import the `Observable` constructor (that's all you need for these examples) in your code files as follows:

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

However, if you use Node.js, you have to do the import in a different way as follows (because Node.js does not yet support the `import` statement):

```
const { Observable } = require('rxjs');
```

These import statements are omitted from all the following code snippets.

Creation

Let's look at how to create a promise versus how to create an observable. To keep it simple, we will for first neglect errors and only consider "successful" executions of promises and observables. We will introduce errors in the next subsection.

Note that there are two sides to both promises and observables: **creation** and **usage**. A promise/observable is an object that first of all needs to be *created* by someone. After it is created, it is typically passed to someone else who *uses* it. Creation defines the behaviour of a promise/observable and the values that are emitted, and usage defines the handling of these emitted values.

A typical use case is that promises/observables are *created* by API functions and returned to the user of the API. The user of the API then *uses* these promises/observables. So, if you use an API, you typically just *use* promises/observables, whereas if you're the author of an API, you also have to *create* promises/observables.

In the following, we will first look at *creation* of promises/observables, and we will look at their *usage* in a subsequent subsection.

Promises:

```
new Promise(executorFunc);

function executorFunc(resolve) {
    // Some code...
    resolve(value);
}
```

To create a promise, you call the `Promise` constructor passing it a so-called *executor function* as argument. The executor function is called by the system when the promise is created, and it is passed as argument a special `resolve` function (you can name this argument however you want, just remember that the *first* argument to the executor function is the resolve function, and you have to use it as such).

When you call the `resolve` function in the body of the executor function, the promise is transferred to *fulfilled* state and the value that you pass as argument to the `resolve` function is "emitted" (the promise is resolved).

This emitted value will then be used as the argument to the *onFulfilled* function that you pass as the first argument to the promise's `then` function on the *usage* side of a promise, as we will see later.

Observables:

```
new Observable(subscriberFunc);

function subscriberFunc(observer) {
    // Some code...
    observer.next(value);
}
```

To create an observable, you call the `Observable` constructor passing it a so-called *subscriber function* as argument. The subscriber function is called by the system whenever a new subscriber subscribes to the observable. The subscriber function gets as argument an *observer* object. This object has a method `next` , which when called, *emits* the value that you pass it as argument from the observable.

Note that after calling `next` , the subscriber function keeps running, and it can call `next` many more times. This is an important difference to promises, where after calling `resolve` the executor function is terminated. Promises can emit at most one value, whereas observables can emit any number of values.

Creation (With Error Handling)

The above examples didn't yet show the full capabilities of promises and observables. Errors may occur during the execution of a promise/observable, and both techniques provide means to indicate such errors to the code that "uses" them.

The following extends the above explanations with error handling capabilities.

Promises:

```
new Promise(executorFunc);


function executorFunc(resolve, reject) {
    // Some code...
    resolve(value);
    // Some code...
    reject(error);
}
```

The executor function that you pass to the `Promise` constructor gets actually a second argument, which is the `reject` function. The `reject` function is used to indicate an error in the promise execution. When you call it, the executor function is aborted and the promise is transferred to the *rejected* state.

On the usage side, this will cause the *onRejected* function (that you may pass to the `catch` method) to be executed.

Observables:

```
new Observable(subscriberFunc);


function subscriberFunc(observer) {
    // Some code...
    observer.next(value);
    // Some code...
    observer.error(error);
}
```

The observer object that is passed as argument to the subscriber function actually has one more method: the `error` method. Calling this method indicates an error to the subscriber of the observable.

Unlike `next`, calling the `error` method also terminates the subscriber function, and thus terminates the observable. This means that `error` can be called at most one time during the lifetime of an observable.

`next` and `error` are still not the entire truth. The observer object that is passed to the subscriber function has one more method: `complete`. Its usage is shown in the following:

```
new Observable(subscriberFunc);


function subscriberFunc(observer) {
    // Some code...
    observer.next(value);
    // If there is an error...
    observer.error(error);
    // If all successful...
    observer.complete();
}
```

The `complete` method is supposed to be called when an observable successfully “completes”. Completing means that there is no more work to, that is, all values have been emitted. Like the `error` method, the `complete` method terminates the execution of the subscriber function, which means that the `complete` method can be called at most one time during the lifetime of an observable.

Note that calling the `conplete` method of an observable execution is recommended, but not mandatory.

Usage

After having covered the *creation* of promises and observables, let’s now look at their *usage*. Using a promise or observable means “subscribing” to it, which in turn means registering a handler function with the promise or observable that will be invoked for each emitted value (one value for a promise, any number of values for an observable).

The registering of the handler function is done through a special method of the promise or observable object. These methods are, respectively:

- **Promise:** `then`
- **Observable:** `subscribe`

In the following we show the basic usage of these methods for both promises and observables. Again, we will consider the basic case neglecting error handling first, and will then add error handling in the next subsection.

Note that in the following code snippets we assume that a promise or observable object already exists. So, if you want to run the code, you must prepend it with a promise or observable creation statement, for example:

- `const promise = new Promise(/*...*/);`
- `const observable = new Observable(/*...*/)`

Promises:

```
promise.then(onFulfilled);

function onFulfilled(value) {
    // Do something with value...
}
```

Given a promise object, we call the `then` method of this object and pass it an *onFulfilled* function as argument. The *onFulfilled* function takes a single argument. This argument is the result value of the promise, that is, the value that has been passed to the `resolve` function inside the promise.

Observables:

Using an observable means subscribing to it, and this is done with the `subscribe` method of an observable. There are actually two equivalent ways to use the `subscribe` method. In the following we are going to present both of them:

Option 1:

```
observable.subscribe(nextFunc);
```

```
function nextFunc(value) {
    // Do something with value...
}
```

In this case, we call the `subscribe` method of an observable and pass it a *next* function as argument. This *next* function takes a single argument. This argument is the currently emitted value whenever the observable emits a value.

In other words, whenever the observable's internal subscriber function calls the `next` method, your *next* function is being called with the value that is passed to `next` (thus emitting the value from the observable to your handler function).

Option 2:

```
observable.subscribe({
    next: nextFunc
});

function nextFunc(value) {
    // Do something with value...
    console.log(value);
}
```

The second option might look a bit strange, but actually it shows better what's going on under the hoods.

In this case we call the `subscribe` not with a function as argument, but with an object. The object has a single property with a key called `next` and a function value. This function is nothing else than our good old *next* function from above.

Anything else stays the same, we just pass the *next* function inside an object rather than directly as an argument. But why would we wrap our handler function in an object before passing it to the `subscribe` method?

The object that can be passed to `subscribe` in this way is in fact an object that implements the `Observer` interface. Maybe you remember that when we created observables in the previous subsections, we used to define a subscriber function, and this subscriber function took a single argument that we called `observer`. In particular, we used code like this:

```
new Observable(subscriberFunc);

function subscriberFunc(observer) {
    // Some code...
    observer.next(value);
}
```

The `observer` argument of the subscriber function corresponds directly to the object that we pass to `subscribe` above (actually, the object passed to `subscribe` is first converted from type `Observer` to `Subscriber` before being passed to the subscriber function, and `Subscriber` implements the `Observer` interface).

So, with option 2, we already create an object that forms the basis of the actual object that will be passed into the subscriber function of the observable, whereas with option 1, we merely provide the functions that will be used as methods of this object.

Which of these two options to use is a matter of taste and coding style. Just note that if you use option 2, the object property key for the *next* function *must* mandatorily be called `next`. This is dictated by the `Observer` interface which this object is required to implement.

Usage (With Error Handling)

As before, we now extend the usage examples to include error handling. Error handling in this case means providing a special handler function that handles potential errors that are indicated by the promise or observable (in addition to the "regular" handler function that handles the "regular" values that are emitted from the promise or observable).

For both promises and observables, an error can be emitted in two cases:

1. The promise or observable implementation calls the `reject` function or `error` method, respectively (see above).
2. The promise or observable implementation throws an error with the `throw` keyword.

Let’s see how we can handle these types of errors for both promises and observables.

Promises:

There are actually two ways to handle an error emitted from a promise. The first uses a second argument to the `then` method, and the second uses method chaining. In the following we are going to present both of them.

Option 1 (second argument to `then`*):*

```
promise.then(onFulfilled, onRejected);

function onFulfilled(value) {
    // Do something with value...
}

function onRejected(error) {
    // Do something with error...
}
```

The `then` method of a promise takes a second function argument which is the *onRejected* function. This function is called when the promise’s executor function calls the `reject` function, or when the promise’s executor function throws an error with the `throw` keyword.

Providing an *onRejected* function allows you to handle such errors. If you don’t provide it, then errors may still occur, but they are not handled by your code.

Option 2 (method chaining):

The second option uses a chained **`catch`** method, and looks like this:

```
promise.then(onFulfilled).catch(onRejected);

function onFulfilled(value) {
    // Do something with value...
}

function onRejected(error) {
    // Do something with error...
}
```

That is, instead of providing both the *onFulfilled* and the *onRejected* function to the `then` method, we provide only the *onFulfilled* method to `then`, then call the `catch` method of the promise returned by `then`, and pass the *onRejected* function to this `catch` method. Note that in this case, the promise that we call `catch` on (and that is returned by `then`) is the same as the initial promise.

This second option using `catch` is actually more common than the first option. It makes use of the important **chaining** capability of promises. A discussion of promise chaining is beyond the scope of this article, but it is, for example, described here.

The important point to note about chaining is that `then` and `catch` always return a promise, which allows repeated calls of these methods in the same statement, as shown above. The returned promise is either the same promise as the previous one, or a new promise. The latter case allows nested asynchronous tasks (that would lead to callback hell if we used callbacks) to be handled “plainly” without any form of nesting. This is, by the way, one of the main advantages of promises over callbacks.

Another point that is good to know is that there is actually nothing special about `catch`. In fact, the `catch` method is just syntactic sugar for a certain invocation of the `then` method. In particular, calling `catch` with the *onRejected* function as it’s only argument is equivalent to calling `then` with an `undefined` first argument, and *onRejected* as its second argument.

Thus, the following two statements are equivalent:

```
promise.then(onFulfilled).catch(onRejected);
promise.then(onFulfilled).then(undefined, onRejected);
```

So, we can conceptually reduce chains of `then` and `catch` to pure chains of `then`, which makes it sometimes easier to reason about them.

Observables:

As already mentioned in the last subsection, there are two ways to call the `subscribe` method of an observable. One uses an object (implementing `Observer`) as argument, and the other uses functions as arguments.

In the following we will present both of these styles.

Option 1 (function arguments):

```
observable.subscribe(nextFunc, errorFunc);

function nextFunc(value) {
    // Do something with value...
}

function errorFunc(error) {
    // Do something with error...
}
```

The only difference to the case without error handling in the previous subsection is that we pass a second function argument to the `subscribe` method. This second argument is the *error* function which is called whenever the observable's subscriber function calls the `error` method of its passed observer argument, or throws an error with `throw` .

Option 2 (object argument):

```
observable.subscribe({
    next: nextFunc,
    error: errorFunc
});

function nextFunc(value) {
    // Do something with value...
}

function errorFunc(error) {
    // Do something with error...
}
```

The only difference to the case without error handling here again is the additional `error` property in the object that we pass to the `subscribe` method. The value of this property is the error handler function.

There is actually a third function that can be passed to the `subscribe` method: `complete` (we already mentioned it in a previous subsection). This function can be passed as either the third argument to `subscribe` (option 1), or as an additional property named `complete` in the observer object that is passed to `subscribe` (option 2).

Furthermore, the specification of each of these three functions is optional. If you don't provide it, then there will simply be no action executed on the corresponding events. All in all, this gives you the following ways to call `subscribe` :

1. **With function arguments:** one, two, or three functions.
2. **With an object argument:** object containing the optional function properties `next` , `error` , and `complete` .

Creation + Usage: Example

In this subsection, we apply all the concepts from the last subsections in a practical example that we implement both with promises and observables.

You can run these examples on any JavaScript engine. For the observable example, just remember to first install the RxJS library and add the appropriate `import` or `require` statement at the top of the source code file, as explained in the introduction of this section.

Promises:

```
// Creation
const promise = new Promise(executorFunc);


function executorFunc(resolve, reject) {
    const value = Math.random();
    if (value <= 1/3.0)
        resolve(value);
    else if (value <= 2/3.0)
        reject("Value <= 2/3 (reject)");
    else
        throw "Value > 2/3 (throw)"
}

// Usage
promise.then(onFulfilled).catch(onRejected);


function onFulfilled(value) {
    console.log("Got value: " + value);
}
```

```
function onRejected(error) {
    console.log("Caught error: " + error);
}
```

This code creates a promise that generates a random number between 0 and 1. If the number is less than or equal to 1/3, the promise is resolved with this value (the value is "emitted"). If the number is greater than 1/3 but less than or equal to 2/3, then the promise is rejected. Finally, if the number is greater than 2/3, an error is thrown with the JavaScript `throw` keyword.

There are three possible outputs of this program:

Output 1:

```
Got value: 0.2109261758959049
```

Output 2:

```
Caught error: Value <= 2/3 (reject)
```

Output 3:

```
Caught error: Value > 2/3 (throw)
```

Output 1 occurs when the promise is regularly resolved (with the `resolve` function). This causes the the `onFulfilled` handler function to be executed with the resolved value.

Output 2 occurs when the promise is explicitly rejected (with the `reject` function). This causes the `onRejected` handler function to be executed.

Finally, output 3 occurs when an error is thrown in the execution of the promise. As with the explicit rejection of a promise, this causes the `onRejected` handler function to be executed.

In the above code we use a relatively verbose syntax, because we use named functions. It is quite common to make use of anonymous functions, which makes the code more concise. In this respect, we could rewrite the above code equivalently as follows:

```
// Creation
const promise = new Promise((resolve, reject) => {
    const value = Math.random();
    if (value <= 1/3.0)
        resolve(value);
    else if (value <= 2/3.0)
        reject("Value <= 2/3 (reject)");
    else
        throw "Value > 2/3 (throw)"
});

// Usage
promise
    .then(value => console.log("Got value: " + value))
    .catch(error => console.log("Caught error: " + error));
```

Now let's implement the same example with observables.

Observables:

```
// Creation
const observable = new Observable(subscriberFunc);

function subscriberFunc(observer) {
    const value = Math.random();
    if (value <= 1/3.0)
        observer.next(value);
    else if (value <= 2/3.0)
        observer.error("Value <= 2/3 (error)");
    else
        throw "Value > 2/3 (throw)"
    observer.complete();
}

// Usage
observable.subscribe(nextFunc, errorFunc, completeFunc);

function nextFunc(value) {
    console.log("Got value: " + value);
}

function errorFunc(error) {
    console.log("Caught error: " + error);
}
```

```
function completeFunc() {
    console.log("Completed");
}
```

This is the same example as above for promises. If the random value is less than or equal to 1/3, the observable emits the value with the `next` method of the passed observer object. If the value is greater than 1/3 but less than or equal to 2/3, indicates an error with the `error` method of the observer object. Finally, if the value is greater than 2/3, it throws an error with the `throw` keyword. At the end of the subscriber function, the observer object's `complete` method is called.

This program also has three possible outputs:

Output 1:

```
Got value: 0.24198168409429077
Completed
```

Output 2:

```
Caught error: Value <= 2/3 (error)
```

Output 3:

```
Caught error: Value > 2/3 (throw)
```

Output 1 occurs when a regular value is emitted from the observable. It causes the `nextFunc` handler function to be executed. Because the observable's subscriber function also calls the `complete` at the very end of its body, the `completeFunc` handler function is also executed.

Output 2 occurs when the observable calls the observer object's `error` method. This causes the `errorFunc` handler function to be executed. Note that this also causes the execution of the observable's subscriber function to be aborted. Consequently, the `complete` method at the end of the subscriber function's body is not called, which means that also the `completeFunc` handler function is never executed. You can see this, because there is no `Completed` output line as in output 1.

Output 3 occurs if the observable's subscriber function throws an error with the `throw` keyword. It has the same effect as calling the `error` method, namely that the `errorFunc` handler function is executed, and that the execution of the observable's subscriber function is aborted (the `complete` method is not called).

As with the promise example, we can rewrite this example in an equivalent more concise notation:

```
// Creation
const observable = new Observable(observer => {
    const value = Math.random();
    if (value <= 1/3.0)
        observer.next(value);
    else if (value <= 2/3.0)
        observer.error("Value <= 2/3 (error)");
    else
        throw "Value > 2/3 (throw)"
    observer.complete();
});

// Usage
observable.subscribe({
    next(value) { console.log("Got value: " + value) },
    error(err) { console.log("Caught error: " + err) },
    complete() { console.log("Completed"); }
});
```

Note that here we use the alternative usage of the `subscribe` method that takes as argument a single object with the handler function as its properties. The alternative would be to use the `subscribe` method with three anonymous functions as argument, but having multiple anonymous functions in an argument list is often unwieldy and unreadable. However, both usages are perfectly equivalent and you can choose whichever you want.

So far we compared the creation and usage of promises and observables. In the remainder of this section, we are going to look at a set of additional differences between promises and observables.

Single Value vs. Multiple Values

- A promise can only emit a single value. After that it is in the *fulfilled* state and can only be used to query this value, but not to calculate and emit new values anymore.

- An observables can emit any number of values.

Promises:

```
const promise = new Promise(resolve => {
    resolve(1);
    resolve(2);
    resolve(3);
});

promise.then(result => console.log(result));
```

This prints:

```
1
```

Only the first call to `resolve` in the executor function is executed and resolves the promise with the value 1. After that, the promise transfers to “fulfilled” state and the result value doesn’t change anymore.

Observables:

```
const observable = new Observable(observer => {
    observer.next(1);
    observer.next(2);
    observer.next(3);
});

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

This prints:

```
1
2
3
```

Each call to `observer.next` in the subscriber function takes effect and causes a value to be emitted and the handler function to be executed.

Eager vs. Lazy

- Promises are **eager**: the executor function is called as soon as the promise is created.
- Observables are **lazy**: the subscriber function is only called when a client subscribes to the observable.

Promises:

```
const promise = new Promise(resolve => {
    console.log("- Executing");
    resolve();
});

console.log("- Subscribing");
promise.then(() => console.log("- Handling result"));
```

This prints:

```
- Executing
- Subscribing
- Handling result
```

As you can see, the executor function is already being executed before anyone subscribed to the promise.

The executor function would even be executed if no one at all subscribed to the promise. You can see this if you outcomment the last two lines: the output will still be `–Executing` .

Observables:

```
const observable = new Observable(observer => {
    console.log("- Executing");
    observer.next();
});

console.log("- Subscribing");
observable.subscribe(() => console.log("- Handling
result"));
```

This prints:

```
- Subscribing
- Executing
- Handling result
```

As you can see, the subscriber function is executed only after a subscription to the observable is created.

If you outcomment the last two lines, then there will be no output at all, because the subscriber function will never be executed.

Since observables are not executed when they are defined, but only when other code uses them, they are also called **declarative** (you *declare* an observable, but it is *executed* only when it is used).

Non-Cancellable vs. Cancellable

- Once you "subscribed" to a promise with `then`, then the handler function that you pass to `then` will be called, no matter what. You can't tell a promise to cancel calling the result handler function once the promise execution has been started.
- After subscribing to an observable with `subscribe`, you can cancel this subscription at any time by calling the `unsubscribe` method of the `Subscription` object returned by `subscribe`. In this case, the handler function that you passed to `subscribe` won't be called anymore.

Promises:

```
const promise = new Promise(resolve => {
    setTimeout(() => {
        console.log("Async task done");
        resolve();
    }, 2000);
});

promise.then(() => console.log("Handler"));

// Oops, can't prevent handler from being executed anymore.
```

This prints (after 2 seconds):

```
Async task done
Handler
```

Once we called `then`, there's no way that we can prevent the handler function that we passed to `then` from being called (even if we would have 2 seconds time to do so). So, after 2 seconds, when the promise is resolved, the handler is executed.

Observables:

```
const observable = new Observable(observer => {
    setTimeout(() => {
        console.log("Async task done");
        observer.next();
    }, 2000);
});

subscription = observable.subscribe(() =>
console.log("Handler"));
subscription.unsubscribe();
```

This prints (after 2 seconds):

```
Async task done
```

We subscribe to the observable, registering a handler function with it, but immediately after that we unsubscribe from the observable again. The effect is that after 2 seconds, when the observable would emit its value, our handler function is *not* called.

Note that `Async task done` is still printed. Unsubscribing by itself does not mean that any asynchronous task that the observable is executing is aborted. Unsubscribing just achieves that calls to `observer.next` (as well as to `observer.error` and `observer.complete`) in the subscriber function do not trigger calls to your handler functions. But everything else still runs as if you wouldn't have called `unsubscribe`.

Multicast vs. Unicast

- The executor function of a promise is executed exactly once (when the promise is created). This means, that all the calls to `then` on a given promise object just "tap" into the ongoing execution of the executor function and in the end get a copy of the result value. Therefore, promises perform **multicast**, because the same execution and result value is used for multiple "subscribers".
- The subscriber function of an observable is executed on each call to `subscribe` on this observable. Therefore, observables perform **unicast**, because there is a separate execution and result value for each subscriber.

Promises:

```
const promise = new Promise(resolve => {
    console.log("Executing...");
    resolve(Math.random());
});

promise.then(result => console.log(result));
promise.then(result => console.log(result));
```

This prints (for example):

```
Executing...
0.1951561731912439
0.1951561731912439
```

As you can see, the executor function is executed only once and the result value is shared between both `then` subscriptions.

Observables:

```
const observable = new Observable(observer => {
    console.log("Executing...");
    observer.next(Math.random());
});

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

This prints (for example):

```
Executing...
0.5884515904517829
Executing...
0.7974144930327094
```

As you can see, the subscriber function is executed separately for each subscriber, and each subscriber gets its own result value.

Asynchronous Handlers vs. Synchronous Handlers

- The handler functions of promises are executed **asynchronously**. That is, they are executed after all the code in the main program or the current function has been executed.
- The handler functions of observables are executed **synchronously**. That is, they are executed within the flow of the current function or the main program.

Promises:

```
console.log("- Creating promise");
const promise = new Promise(resolve => {
    console.log("- Promise running");
    resolve(1);
});

console.log("- Registering handler");
promise.then(result => console.log("- Handling result: " +
result));

console.log("- Exiting main");
```

This prints the following sequence of output messages:

```
- Creating promise
- Promise running
- Registering handler
- Exiting main
- Handling result: 1
```

First the promise is created, upon which it is directly executed (because promises are *eager*, see above). The promise is also immediately resolved. After that, we register a handler function with the promise by invoking its `then` method. At this point the promise is already resolved (i.e. it is in the *fulfilled* state), however, our handler function is **not** executed at this point. Rather, all the remaining code in the main program is first executed, and only *after* that our handler function is invoked.

The reason for this is that a promise fulfilment (or rejection) is handled as an **asynchronous** event. This means, when a promise is resolved (or rejected), the corresponding handler function is put as a separate item in the JavaScript event queue. This means that the handler is only executed after all the previous items in the event queue have been executed, and in our example there is one such previous item, which is the main program.

Observables:

```
console.log("- Creating observable");
const observable = new Observable(observer => {
    console.log("- Observable running");
    observer.next(1);
});

console.log("- Registering handler");
observable.subscribe(v => console.log("- Handling result: "
+ v));

console.log("- Exiting main");
```

This prints the following sequence of output messages:

```
- Creating observable
- Registering handler
- Observable running
- Handling result: 1
- Exiting main
```

First, the observable is created (but it is not yet executed, because observables are *lazy*, see above), and then we register a handler with it by calling the observable's `subscribe` method. At this time the observable starts running and immediately emits its first and only value. Now our handler function **is** executed, and finally the main program exits.

Unlike with promises, the handler function is run while the main program is still running. This is because the handler functions of observables are called **synchronously** within the currently executing code, and not as asynchronous events like the handler function of promises.

Conclusion

In this article we have first presented different asynchronous programming techniques in JavaScript, the most important ones of which are:

- **Callbacks**
- **Promises**
- **Async/Await**
- **RxJS Observables**

Then, we made a side-by-side comparison of promises and observables. In particular, we highlighted differences and similarities for the following aspects:

- **Creation**
- **Usage**
- **Single Value vs. Multiple Values**
- **Eager vs. Lazy**
- **Non-Cancellable vs. Cancellable**
- **Multicast vs. Unicast**
- **Asynchronous Handlers vs. Synchronous Handlers**

References

- **https://www.academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/**
- **https://medium.com/@mpodlasin/promises-vs-observables-4c123c51fe13**

- **http://reactivex.io/intro.html**