Understanding Promises in JavaScript

An in-depth look at creating and handling Promises

Gokul N K
Jul 16, 2018 · 14 min read

I am making you a pinky promise that by the end of this post you will know JavaScript Promises better.

I have a kind of love-hate relationship with JavaScript. Nonetheless, it has always intrigued me. Having worked on Java and PHP for the last ten years, JavaScript seemed different but intriguing. I did not get to spend enough time on it and have been trying to make up for it lately.

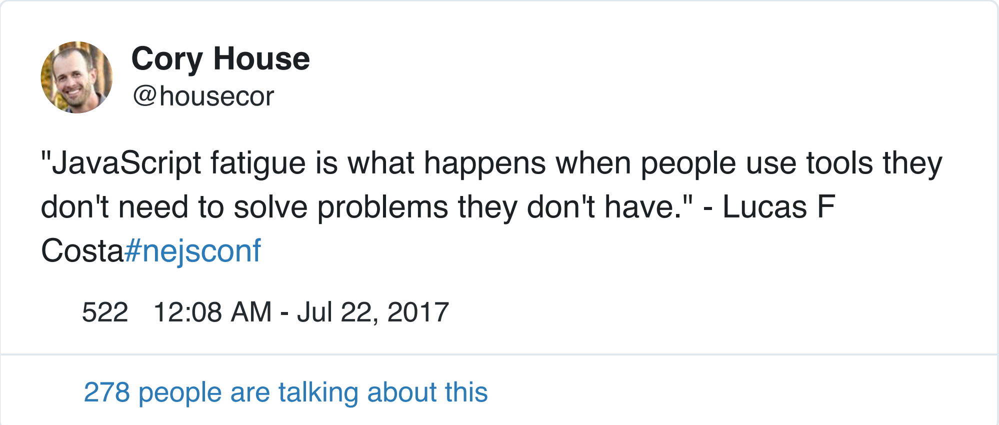
Promises was the first interesting topic that I came across. Time and again I have heard people saying that Promises "saves you from Callback hell". Well, that might be a pleasant side-effect, but there is more to Promises than that. This is what I have been able to figure out up until this point.

Note: This is going to be a long article. If you would like to highlight some parts you can use our extension http://bit.ly/highlights-extension.

. . .

Background

Working in JavaScript it can be a little frustrating at first. You'll hear people say that JavaScript is a synchronous programming language, then others will claim that it is asynchronous. You hear blocking code, non blocking code, event-driven design pattern, event life cycle, function stack, event queue, bubbling, polyfill, babel, angular, reactJS, vue JS and a lot of other tools and libraries. Fret not — you are not the first. There is a term for it: *JavaScript Fatigue.* This tweet captures it well:



Cory House
@housecor

"JavaScript fatigue is what happens when people use tools they don't need to solve problems they don't have." - Lucas F Costa#nejsconf

522 12:08 AM - Jul 22, 2017

278 people are talking about this

If you want further details about JavaScript fatigue you should check out the following article. There is a reason this post got 42k claps on Hackernoon :)

How it feels to learn JavaScript in 2016

No JavaScript frameworks were created during the writing of this article.

hackernoon.com

JavaScript is a *synchronous* programming language. But thanks to callback functions we can make it function like an *asynchronous* programming language.

Promises, in layman's terms

Promises in JavaScript are similar to the promises you make in real life, so let's look at promises in real life.

This is the dictionary definition of a promise:

promise : noun : Assurance that one will do something or that a particular thing will happen.

So what happens when someone makes a promise to you ?

1. A promise gives you an assurance that something will be done. Whether they (who made the promise) will do it themselves or they get it done by others is immaterial. They give you an assurance, based on which you can plan something.

2. A promise can either be kept or broken.

3. When a promise is kept you expect something out of that promise. You can make use of the output of a promise for your further actions or plans.

4. When a promise is broken, you want to know why the person who made the promise wasn't able to keep up his side of the bargain. Once you know the reason, and have a confirmation that the promise has been broken, you can plan what to do next.

5. When a promise is made to us all we have is an assurance. We can't act on it immediately. We can decide and formulate what needs to be done when the *promise is kept* (hence we have an expected outcome) or *broken* (we know the reason and hence we can plan a contingency).

6. There's a chance you don't back from the person who made the promise. For such a circumstance it is wise to set a deadline. For example, if the person doesn't come back to me within ten days I will consider them to have not kept their promise. Even if they come back to you 15 days later, it doesn't matter — you have already made alternate plans.

. . .

Promises in JavaScript

As a rule of thumb, I always read documentation from *MDN Web Docs* for JavaScript. Of all the resources available, I think they provide the most concise details. I read up the Promises page form MDSN Web Docs and played around with the code to get the hang of it.

There are two parts to understanding promises. *Creating promises* and *handling promises*. Although most of our code will cater to handling promises created by other libraries, gaining a complete understanding is important. Understanding the creation of promises will be increasingly important as you advance from the beginner stage.

. . .

Creating Promises

Let's look at the signature for creating a new promise:

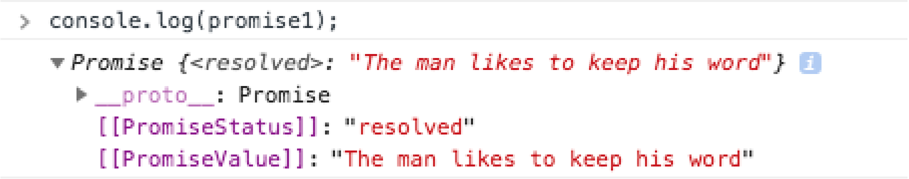
```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

The constructor accepts a function called `executor`. This `executor` function accepts two parameters: `resolve` and `reject`, which are in turn functions.

Promises are generally used for easier handling of asynchronous operations or blocking code, for example, file operations, API calls, DB calls, IO calls, etc. These asynchronous operations initiate inside the `executor` function. If the asynchronous operations are successful the expected result is returned by calling the `resolve` function. Similarly, if there was some unexpected error the reasons are passed on by calling the `reject` function.

Now we know how to create a promise. Let's create a simple promise to help our understanding.

```
var keepsHisWord;
keepsHisWord = true;
promise1 = new Promise(function(resolve, reject) {
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesnt want to keep his word");
  }
});
console.log(promise1);
```



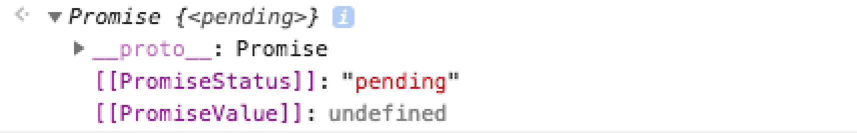
```
> console.log(promise1);
  ▼Promise {<resolved>: "The man likes to keep his word"} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: "The man likes to keep his word"
```

Every promise has a state and value

Since this promise gets resolved right away we will not be able to inspect the initial state of the promise. So let us just create a new promise that will take some time to resolve. The easiest way to do that is to use the `setTimeOut` function.

```
promise2 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve({
      message: "The man likes to keep his word",
      code: "aManKeepsHisWord"
    });
  }, 10 * 1000);
});
console.log(promise2);
```

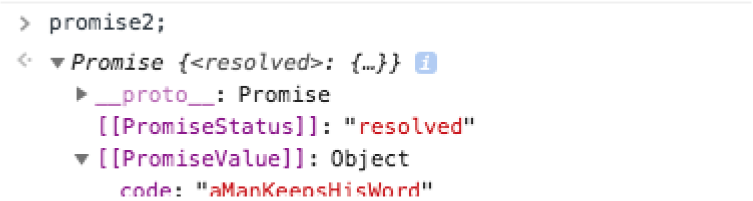
This code creates a promise that will resolve unconditionally after ten seconds. We can check out the state of the promise until it is resolved.



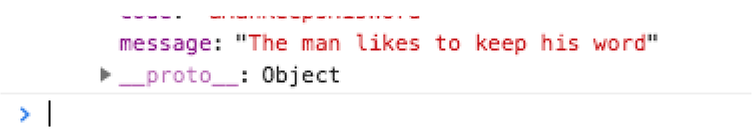
```
<· ▼Promise {<pending>} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "pending"
     [[PromiseValue]]: undefined
```

state of promise until it is resolved or rejected

Once the ten seconds are over the promise is resolved. Both `PromiseStatus` and `PromiseValue` are updated accordingly. As you can see, we updated the resolve function so that we can pass a JSON object instead of a simple string. This is to show that we can pass other values as well in the `resolve` function.



```
> promise2;
<· ▼Promise {<resolved>: {…}} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
    ▼[[PromiseValue]]: Object
       code: "aManKeepsHisWord"
```



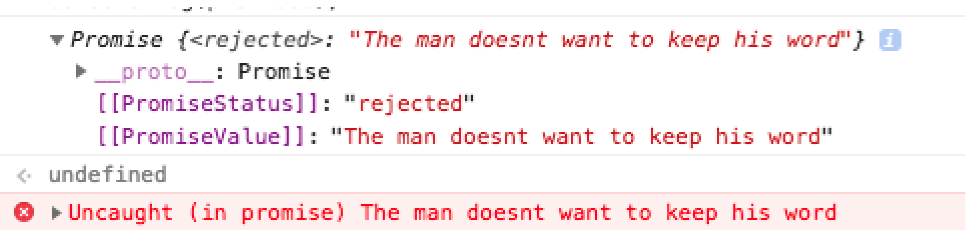
```
message: "The man likes to keep his word"
▶ __proto__: Object
> |
```

A promise that resolves after ten seconds with a JSON object as returned value

Now let's look at a promise that will be rejected. We just slightly modify promise one for this.

```
keepsHisWord = false;
promise3 = new Promise(function(resolve, reject) {
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesn't want to keep his word");
  }
});
console.log(promise3);
```

Since this creates an unhandled rejection, Chrome browser will show an error. You can ignore it for now — we'll get back to it later.



```
▼Promise {<rejected>: "The man doesnt want to keep his word"} i
  ▶__proto__: Promise
   [[PromiseStatus]]: "rejected"
   [[PromiseValue]]: "The man doesnt want to keep his word"
<· undefined
⊗ ▶Uncaught (in promise) The man doesnt want to keep his word
```

rejections in promises

As you can see, `PromiseStatus` can have three different values: `pending`, `resolved`, or `rejected`. When a promise is created, `PromiseStatus` is in the `pending` status — it will have `PromiseValue` `undefined` until the promise is either `resolved` or `rejected.` When a promise is in `resolved` or `rejected` states, a promise is said to be `settled`. So a promise generally transitions from the pending state to the settled state.

Now that we know how promises are created we can look at how we use or handle promises. This will go hand in hand with understanding the `Promise` object.

Understanding Promises Object

As per MDN documentation:

> *The `Promise` object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.*

`Promise` object has static methods and `prototype methods`.

Static methods in a `Promise` object can be applied independently, whereas the `prototype methods` needs to be applied to the instances of `Promise` object.

Remembering that both normal methods and prototypes all return a `Promise` makes it far easier to make sense of things.

. . .

Prototype Methods

We'll start with the `prototype methods`. There are three of them.

Just to reiterate: all these methods can be applied on an instance of `Promise` object and all these methods return a promise in turn. All of these methods assign handlers for different state transitions of a promise.

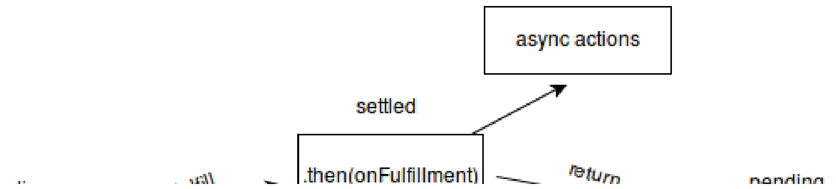
As we saw earlier when a `Promise` is created it is in `pending` state. One or more of the following three methods will be run when a promise is settled based on whether they are `fulfilled` or `rejected`:

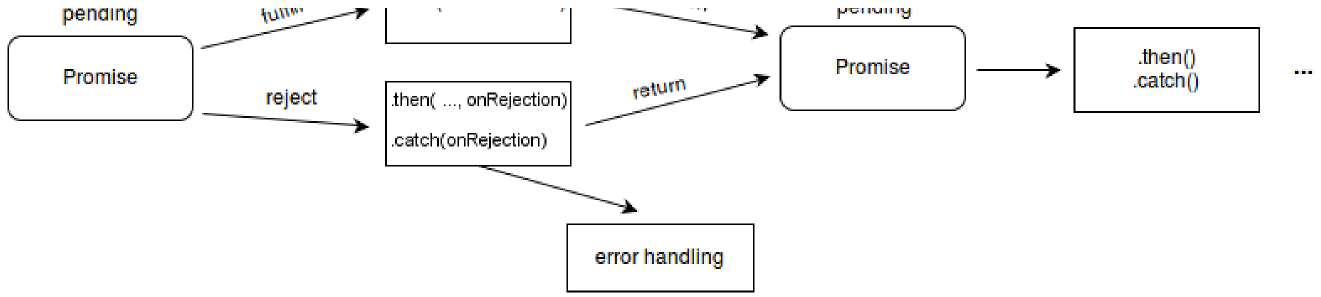
`Promise.prototype.catch(onRejected)`

`Promise.prototype.then(onFulfilled, onRejected)`

`Promise.prototype.finally(onFinally)`

The image below shows the flow for `.then` and `.catch` methods. Since they return a `Promise` they can be chained again, which is also shown in the image. If `.finally` is declared for a promise, it will be executed whenever a promise is `settled`, irrespective of whether it is fulfilled or rejected. As Konstantin Rouda pointed out, there is limited support for `.finally`, so please check before you use this.





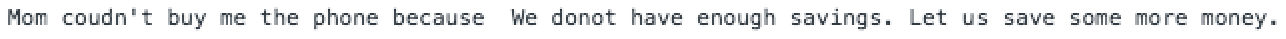
From : https://mdn.mozillademos.org/files/15911/promises.png

Here’s a little story. You’re a schoolkid and you ask your mom for a phone. She says “I promise to buy a phone at the end of the month.”

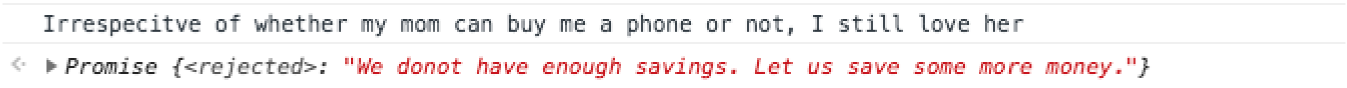
If this promise gets executed at the end of the month, let’s see what that promise would look like in JavaScript:

```
var momsPromise = new Promise(function(resolve, reject) {
  momsSavings = 20000;
  priceOfPhone = 60000;
  if (momsSavings > priceOfPhone) {
    resolve({
      brand: "iphone",
      model: "6s"
    });
  } else {
    reject("We donot have enough savings. Let us save some more
money.");
  }
});

momsPromise.then(function(value) {
  console.log("Hurray I got this phone as a gift ",
JSON.stringify(value));
});

momsPromise.catch(function(reason) {
  console.log("Mom coudn't buy me the phone because ", reason);
});

momsPromise.finally(function() {
  console.log(
    "Irrespecitve of whether my mom can buy me a phone or not, I
still love her"
  );
});
```

The output for this is:



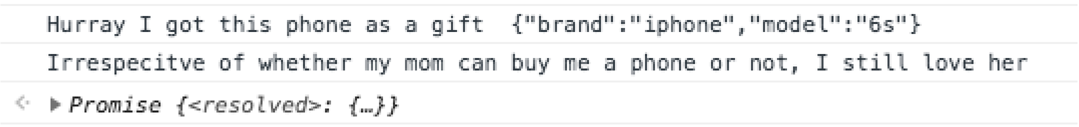
```
Mom coudn't buy me the phone because  We donot have enough savings. Let us save some more money.
```



```
Irrespecitve of whether my mom can buy me a phone or not, I still love her
<· ▶Promise {<rejected>: "We donot have enough savings. Let us save some more money."}
```

moms failed promise.

If we change the value of `momsSavings` to 200000 then mom will be able to gift the son. In this case, the output is:



```
Hurray I got this phone as a gift  {"brand":"iphone","model":"6s"}
Irrespecitve of whether my mom can buy me a phone or not, I still love her
<· ▶Promise {<resolved>: {…}}
```

mom keeps her promise.

Let's wear the hat of somebody who consumes this library. We're mocking the output and nature so that we can look at how to use `.then` and `.catch` effectively.

Since `.then` can assign both `onFulfilled, onRejected handlers` , instead of writing separate `.then` and `.catch` , we could have done the same with with `.then` . It would have looked like This:

```
momsPromise.then(
  function(value) {
    console.log("Hurray I got this phone as a gift ",
JSON.stringify(value));
  },
  function(reason) {
    console.log("Mom coudn't buy me the phone because ", reason);
  }
);
```

For the sake of the readability of the code, I think it is better to keep them separate.

To make sure that we can run all these samples, in browsers in general or Chrome specifically, I am ensuring that we have no external dependencies in our code samples. To better understand the later topics let's create a function that returns a promise, which will then be resolved or rejected at random, so that we can test out various scenarios.

To understand the concept of asynchronous functions let's introduce a random delay also into our function. Since we need random numbers let's first create a random function that returns a random number between x and y:

```
function getRandomNumber(start = 1, end = 10) {
  //works when both start,end are >=1 and end > start
  return parseInt(Math.random() * end) % (end-start+1) + start;
}
```

Next we create a function that returns a promise for us. Let's call for our function `promiseTRRARNOSG`, which is an alias for `promiseThatResolvesRandomlyAfterRandomNumnberOfSecondsGenerator`. This function will create a promise that resolves or rejects after a random number of seconds between two and ten. To randomise rejection and resolving we will create a random number between one and ten. If the random number generated is greater five we resolve the promise, else we reject it.

```
function getRandomNumber(start = 1, end = 10) {
  //works when both start and end are >=1
  return (parseInt(Math.random() * end) % (end - start + 1)) +
start;
}

var promiseTRRARNOSG =
(promiseThatResolvesRandomlyAfterRandomNumnberOfSecondsGenerator =
function() {
  return new Promise(function(resolve, reject) {
    let randomNumberOfSeconds = getRandomNumber(2, 10);
    setTimeout(function() {
      let randomiseResolving = getRandomNumber(1, 10);
      if (randomiseResolving > 5) {
        resolve({
          randomNumberOfSeconds: randomNumberOfSeconds,
          randomiseResolving: randomiseResolving
        });
      } else {
        reject({
          randomNumberOfSeconds: randomNumberOfSeconds,
          randomiseResolving: randomiseResolving
        });
      }
    }, randomNumberOfSeconds * 1000);
  });
});

var testProimse = promiseTRRARNOSG();
testProimse.then(function(value) {
  console.log("Value when promise is resolved : ", value);
});
testProimse.catch(function(reason) {
  console.log("Reason when promise is rejected : ", reason);
});
```

```
// Let us loop through and create ten different promises using the
function to see some variation. Some will be resolved and some will
be rejected.

for (i=1; i<=10; i++) {
  let promise = promiseTRRARNOSG();
  promise.then(function(value) {
    console.log("Value when promise is resolved : ", value);
  });
  promise.catch(function(reason) {
    console.log("Reason when promise is rejected : ", reason);
  });
}
```

Refresh the browser page and run the code in the console to see the different outputs for `resolve` and `reject` scenarios. Later we'll look at how to create multiple promises and check their outputs without having to do this.

. . .

Static Methods

There are four static methods in the `Promise` object.

The first two are helpers methods or shortcuts. They help you create resolved or rejected promises.

`Promise.reject(reason)` helps you create a rejected promise.

```
var promise3 = Promise.reject("Not interested");
promise3.then(function(value){
  console.log("This will not run as it is a resolved promise. The
resolved value is ", value);
});
promise3.catch(function(reason){
  console.log("This run as it is a rejected promise. The reason is
", reason);
});
```

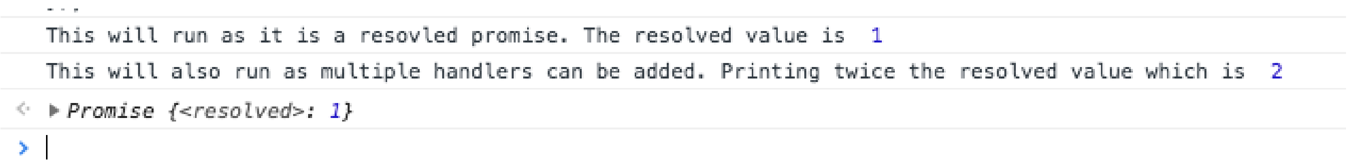
`Promise.resolve(value)` helps you create a resolved promise:

```
var promise4 = Promise.resolve(1);
promise4.then(function(value){
  console.log("This will run as it is a resovled promise. The
resolved value is ", value);
});
promise4.catch(function(reason){
  console.log("This will not run as it is a resolved promise",
reason);
});
```

Sidenote: a promise can have multiple handlers, so you can update the above code to this:

```
var promise4 = Promise.resolve(1);
promise4.then(function(value){
  console.log("This will run as it is a resovled promise. The
resolved value is ", value);
});
promise4.then(function(value){
  console.log("This will also run as multiple handlers can be added.
Printing twice the resolved value which is ", value * 2);
});
promise4.catch(function(reason){
  console.log("This will not run as it is a resolved promise",
reason);
});
```

And the output will look like this:



```
This will run as it is a resovled promise. The resolved value is  1
This will also run as multiple handlers can be added. Printing twice the resolved value which is  2
< ▶ Promise {<resolved>: 1}
> |
```

The next two methods helps you process a set of promises.

When you’re dealing with multiple promises it is better to first create an array of promises and then do the necessary action over the whole set of promises.

For understanding these methods we will not be able to use our handy `promiseTRRARNOSG` — it is too random. It’s better to have some deterministic promises so that we can understand the behaviour. Let’s create two functions. One that will resolve after n seconds and one that will reject after n seconds.

```
var promiseTRSANSG = (promiseThatResolvesAfterNSecondsGenerator =
function(
  n = 0
) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve({
        resolvedAfterNSeconds: n
      });
    }, n * 1000);
  });
});
var promiseTRJANSG = (promiseThatRejectsAfterNSecondsGenerator =
function(
  n = 0
) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      reject({
        rejectedAfterNSeconds: n
      });
    }, n * 1000);
  });
});
```

Now let's use these helper functions to understand `Promise.All`

· · ·

Promise.All

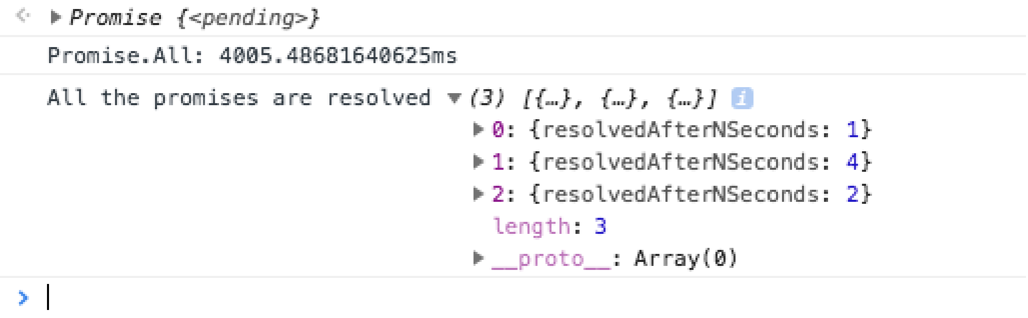
As per MDN documentation:

> *The* **Promise.all(iterable)** *method returns a single* `Promise` *that resolves when all of the promises in the* `iterable` *argument have resolved or when the iterable argument contains no promises. It rejects with the reason of the first promise that rejects.*

Case one

When all the promises are resolved. This is the most frequently used scenario:

```
console.time("Promise.All");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(1));
promisesArray.push(promiseTRSANSG(4));
promisesArray.push(promiseTRSANSG(2));
var handleAllPromises = Promise.all(promisesArray);
handleAllPromises.then(function(values) {
  console.timeEnd("Promise.All");
  console.log("All the promises are resolved", values);
});
handleAllPromises.catch(function(reason) {
  console.log("One of the promises failed with the following
reason", reason);
});
```



```
<· ▶ Promise {<pending>}
   Promise.All: 4005.48681640625ms
   All the promises are resolved ▼ (3) [{…}, {…}, {…}] ℹ
                                   ▶ 0: {resolvedAfterNSeconds: 1}
                                   ▶ 1: {resolvedAfterNSeconds: 4}
                                   ▶ 2: {resolvedAfterNSeconds: 2}
                                     length: 3
                                   ▶ __proto__: Array(0)
>  |
```

All promises resolved.

There are two important observations we need to make about this output.

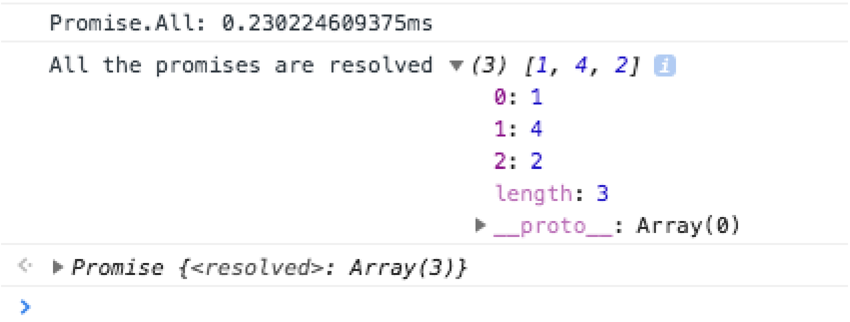
1: The third promise which takes 2 seconds finishes before the second promise which takes 4 seconds. But as you can see in the output, the order of the promises are maintained in the values.

2: I added a console timer to find out how long `Promise.All` takes. If the promises were executed in sequential it should have taken 1+4+2=7 seconds in total. But from our timer we saw that it only takes 4 seconds. This is a proof that all the promises were executed in parallel.

Case 2

When there are no promises. I think this is the least frequently used.

```
console.time("Promise.All");
var promisesArray = [];
promisesArray.push(1);
promisesArray.push(4);
promisesArray.push(2);
var handleAllPromises = Promise.all(promisesArray);
handleAllPromises.then(function(values) {
  console.timeEnd("Promise.All");
  console.log("All the promises are resolved", values);
});
handleAllPromises.catch(function(reason) {
  console.log("One of the promises failed with the following
reason", reason);
});
```



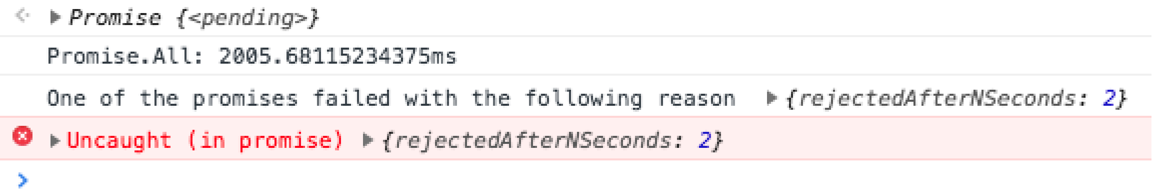
```
Promise.All: 0.230224609375ms
All the promises are resolved ▼(3) [1, 4, 2]
                                 0: 1
                                 1: 4
                                 2: 2
                                 length: 3
                               ▶__proto__: Array(0)
< ▶Promise {<resolved>: Array(3)}
>
```

Since there are no promises in the array the returning promise is resolved.

Case 3

It rejects with the reason of the first promise that rejects:

```
console.time("Promise.All");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(1));
promisesArray.push(promiseTRSANSG(5));
promisesArray.push(promiseTRSANSG(3));
promisesArray.push(promiseTRJANSG(2));
promisesArray.push(promiseTRSANSG(4));
var handleAllPromises = Promise.all(promisesArray);
handleAllPromises.then(function(values) {
  console.timeEnd("Promise.All");
  console.log("All the promises are resolved", values);
});
handleAllPromises.catch(function(reason) {
  console.timeEnd("Promise.All");
  console.log("One of the promises failed with the following reason
", reason);
});
```



```
< ▶Promise {<pending>}
Promise.All: 2005.68115234375ms
One of the promises failed with the following reason ▶{rejectedAfterNSeconds: 2}
⊗ ▶Uncaught (in promise) ▶{rejectedAfterNSeconds: 2}
>
```

Execution stopped after the first rejection

. . .

Promise.race

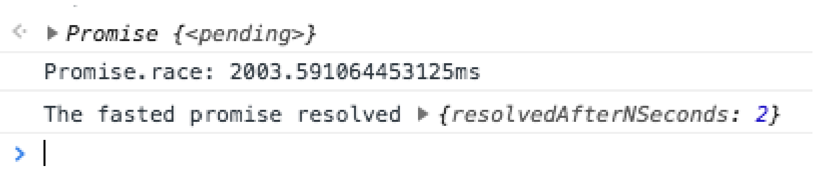
As per MDN documention

> *The `Promise.race(iterable)` method returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects, with the value or reason from that promise.*

Case 1

One of the promises resolves first:

```
console.time("Promise.race");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(4));
promisesArray.push(promiseTRSANSG(3));
promisesArray.push(promiseTRSANSG(2));
promisesArray.push(promiseTRJANSG(3));
promisesArray.push(promiseTRSANSG(4));
var promisesRace = Promise.race(promisesArray);
promisesRace.then(function(values) {
  console.timeEnd("Promise.race");
  console.log("The fasted promise resolved", values);
});
promisesRace.catch(function(reason) {
  console.timeEnd("Promise.race");
  console.log("The fastest promise rejected with the following
reason ", reason);
});
```



```
< ▸ Promise {<pending>}
  Promise.race: 2003.591064453125ms
  The fasted promise resolved ▸ {resolvedAfterNSeconds: 2}
> |
```

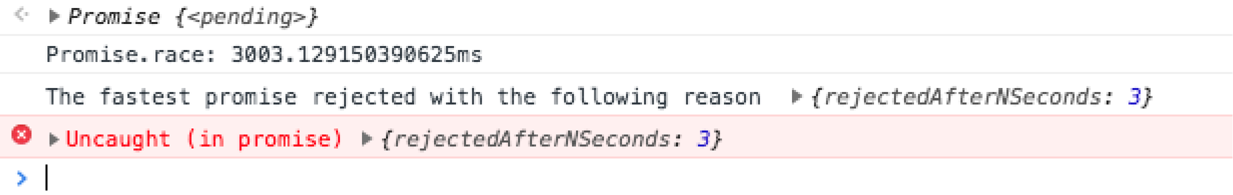
fastest resolution

All the promises are run in parallel. The third promise resolves in two seconds. As soon as this is done the promise returned by `Promise.race` is resolved.

Case 2

One of the promises rejects first.

```
console.time("Promise.race");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(4));
promisesArray.push(promiseTRSANSG(6));
promisesArray.push(promiseTRSANSG(5));
promisesArray.push(promiseTRJANSG(3));
promisesArray.push(promiseTRSANSG(4));
var promisesRace = Promise.race(promisesArray);
promisesRace.then(function(values) {
  console.timeEnd("Promise.race");
  console.log("The fasted promise resolved", values);
});
promisesRace.catch(function(reason) {
  console.timeEnd("Promise.race");
  console.log("The fastest promise rejected with the following
reason ", reason);
});
```



```
< ▶ Promise {<pending>}
  Promise.race: 3003.129150390625ms
  The fastest promise rejected with the following reason  ▶ {rejectedAfterNSeconds: 3}
⊗ ▶ Uncaught (in promise) ▶ {rejectedAfterNSeconds: 3}
> |
```

fastest rejection

All the promises are run in parallel. The fourth promise rejected in three seconds. As soon as this is done the promise returned by `Promise.race` is rejected.

I have written all the example methods so I can test out various scenarios and tests can be run in the browser itself. That‘s why you don’t see any API calls, file operations or database calls in the examples. While all of these are real-life examples, you need additional effort to set them up and test them.

Using the delay functions, on the other hand, gives you similar scenarios without the burden of additional setup. You can easily play around with the values to see and check out different scenarios. You can use the combination of `promiseTRJANSG` , `promiseTRSANSG` and `promiseTRRARNOSG` methods to simulate enough scenarios for a thorough understanding of promises.

Also, the use of `console.time` methods before and after relevant blocks will help us identify easily if the promises are run parallelly or sequentially. Let me know if you have any other interesting scenarios or if I have missed something. If you want all the code samples in a single place check out this gist.

```
var keepsHisWord;
keepsHisWord = true;
promise1 = new Promise(function(resolve, reject) {
  if (keepsHisWord) {
```

```javascript
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesnt want to keep his word");
  }
});
console.log(promise1);

promise2 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve({
      message: "The man likes to keep his word",
      code: "aManKeepsHisWord"
    });
  }, 10 * 1000);
});
console.log(promise2);

keepsHisWord = false;
promise3 = new Promise(function(resolve, reject) {
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesn't want to keep his word");
  }
});
console.log(promise3);

var momsPromise = new Promise(function(resolve, reject) {
  momsSavings = 20000;
  priceOfPhone = 60000;
  if (momsSavings > priceOfPhone) {
    resolve({
      brand: "iphone",
      model: "6s"
    });
  } else {
    reject("We donot have enough savings. Let us save some more money.");
  }
});
momsPromise.then(function(value) {
  console.log("Hurray I got this phone as a gift ", JSON.stringify(value));
});
momsPromise.catch(function(reason) {
  console.log("Mom coudn't buy me the phone because ", reason);
});
momsPromise.finally(function() {
  console.log(
    "Irrespecitve of whether my mom can buy me a phone or not, I still love her"
  );
});

momsPromise.then(
  function(value) {
    console.log("Hurray I got this phone as a gift ", JSON.stringify(value));
  },
  function(reason) {
    console.log("Mom coudn't buy me the phone because ", reason);
  }
);

function getRandomNumber(start = 1, end = 10) {
  //works when both start,end are >=1 and end > start
  return parseInt(Math.random() * end) % (end-start+1) + start;
}

function getRandomNumber(start = 1, end = 10) {
  //works when both start and end are >=1
  return (parseInt(Math.random() * end) % (end - start + 1)) + start;
}
var promiseTRRARNOSG = (promiseThatResolvesRandomlyAfterRandomNumnberOfSecondsGenerato
  return new Promise(function(resolve, reject) {
    let randomNumberOfSeconds = getRandomNumber(2, 10);
    setTimeout(function() {
      let randomiseResolving = getRandomNumber(1, 10);
      if (randomiseResolving > 5) {
        resolve({
          randomNumberOfSeconds: randomNumberOfSeconds,
          randomiseResolving: randomiseResolving
        });
      } else {
        reject({
          randomNumberOfSeconds: randomNumberOfSeconds,
          randomiseResolving: randomiseResolving
        });
      }
    }, randomNumberOfSeconds * 1000);
  });
});
var testProimse = promiseTRRARNOSG();
testProimse.then(function(value) {
  console.log("Value when promise is resolved : ", value);
});
testProimse.catch(function(reason) {
  console.log("Reason when promise is rejected : ", reason);
});
```

```
// Let us loop through and create ten different promises using the function to see son
for (i=1; i<=10; i++) {
  let promise = promiseTRRARNOSG();
  promise.then(function(value) {
    console.log("Value when promise is resolved : ", value);
  });
  promise.catch(function(reason) {
    console.log("Reason when promise is rejected : ", reason);
  });
}

var promise3 = Promise.reject("Not interested");
promise3.then(function(value){
  console.log("This will not run as it is a resolved promise. The resolved value is ",
});
promise3.catch(function(reason){
  console.log("This run as it is a rejected promise. The reason is ", reason);
});

var promise4 = Promise.resolve(1);
promise4.then(function(value){
  console.log("This will run as it is a resovled promise. The resolved value is ", val
});
promise4.catch(function(reason){
  console.log("This will not run as it is a resolved promise", reason);
});


var promise4 = Promise.resolve(1);
promise4.then(function(value){
  console.log("This will run as it is a resovled promise. The resolved value is ", val
});
promise4.then(function(value){
  console.log("This will also run as multiple handlers can be added. Printing twice th
});
promise4.catch(function(reason){
  console.log("This will not run as it is a resolved promise", reason);
});


var promiseTRSANSG = (promiseThatResolvesAfterNSecondsGenerator = function(
  n = 0
) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve({
        resolvedAfterNSeconds: n
      });
      });
    }, n * 1000);
  });
});
var promiseTRJANSG = (promiseThatRejectsAfterNSecondsGenerator = function(
  n = 0
) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      reject({
        rejectedAfterNSeconds: n
      });
    }, n * 1000);
  });
});

console.time("Promise.All");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(1));
promisesArray.push(promiseTRSANSG(4));
promisesArray.push(promiseTRSANSG(2));
var handleAllPromises = Promise.all(promisesArray);
handleAllPromises.then(function(values) {
  console.timeEnd("Promise.All");
  console.log("All the promises are resolved", values);
});
handleAllPromises.catch(function(reason) {
  console.log("One of the promises failed with the following reason", reason);
});

console.time("Promise.All");
var promisesArray = [];
promisesArray.push(1);
promisesArray.push(4);
promisesArray.push(2);
var handleAllPromises = Promise.all(promisesArray);
handleAllPromises.then(function(values) {
  console.timeEnd("Promise.All");
  console.log("All the promises are resolved", values);
});
handleAllPromises.catch(function(reason) {
  console.log("One of the promises failed with the following reason", reason);
});


console.time("Promise.All");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(1));
promisesArray.push(promiseTRSANSG(5));
promisesArray.push(promiseTRSANSG(3));
promisesArray.push(promiseTRJANSG(2));
promisesArray.push(promiseTRSANSG(4));
var handleAllPromises = Promise.all(promisesArray);
handleAllPromises.then(function(values) {
  console.timeEnd("Promise.All");
  console.log("All the promises are resolved", values);
});
handleAllPromises.catch(function(reason) {
  console.timeEnd("Promise.All");
  console.log("One of the promises failed with the following reason ", reason);
});


console.time("Promise.race");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(4));
promisesArray.push(promiseTRSANSG(3));
promisesArray.push(promiseTRSANSG(2));
promisesArray.push(promiseTRJANSG(3));
promisesArray.push(promiseTRSANSG(4));
var promisesRace = Promise.race(promisesArray);
promisesRace.then(function(values) {
  console.timeEnd("Promise.race");
  console.log("The fasted promise resolved", values);
});
promisesRace.catch(function(reason) {
  console.timeEnd("Promise.race");
  console.log("The fastest promise rejected with the following reason ", reason);
});


console.time("Promise.race");
var promisesArray = [];
promisesArray.push(promiseTRSANSG(4));
promisesArray.push(promiseTRSANSG(6));
promisesArray.push(promiseTRSANSG(5));
promisesArray.push(promiseTRJANSG(3));
promisesArray.push(promiseTRSANSG(4));
var promisesRace = Promise.race(promisesArray);
promisesRace.then(function(values) {
  console.timeEnd("Promise.race");
  console.log("The fasted promise resolved", values);
});
promisesRace.catch(function(reason) {
  console.timeEnd("Promise.race");
  console.log("The fastest promise rejected with the following reason ", reason);
});
```

1. Promise.prototype.timeout
2. Promise.some
3. Promise.promisify

We will discuss these in a separate post.

I will also be writing one more post about my learnings from async and await.

Before ending I would like to list all the rules of thumb rules I follow to keep my sanity around promises.

. . .

Rules of Thumb for Using Promises

1. Use promises whenever you are using async or blocking code.
2. `resolve` maps to `then` and `reject` maps to `catch` for all practical purposes.
3. Make sure to write both `.catch` and `.then` methods for all the promises.
4. If something needs to be done in both cases, use `.finally` .
5. We only get one shot at mutating each promise.
6. We can add multiple handlers to a single promise.
7. The return type of all the methods in `Promise` object, whether they are static methods or prototype methods, is again a `Promise`
8. In `Promise.all` the order of the promises is maintained in the values variable, irrespective of which promise was first resolved.

JavaScript Technology Promises Learning Programming