Promises

Everyone has their own claims about whether JavaScript is a synchronous programming language or asynchronous, blocking, or non-blocking code, but not everyone is sure about it (even I am not 🤭).

Let us try to get this thing clear and understand promises and how it works.

JavaScript is a synchronous programming language. However, callback functions enable us to transform it into an asynchronous programming language.

And promises are to help to get out of "callback hell" while dealing with the asynchronous code and do much more.

In simple terms, JavaScript promises are similar to the promises made in human life.

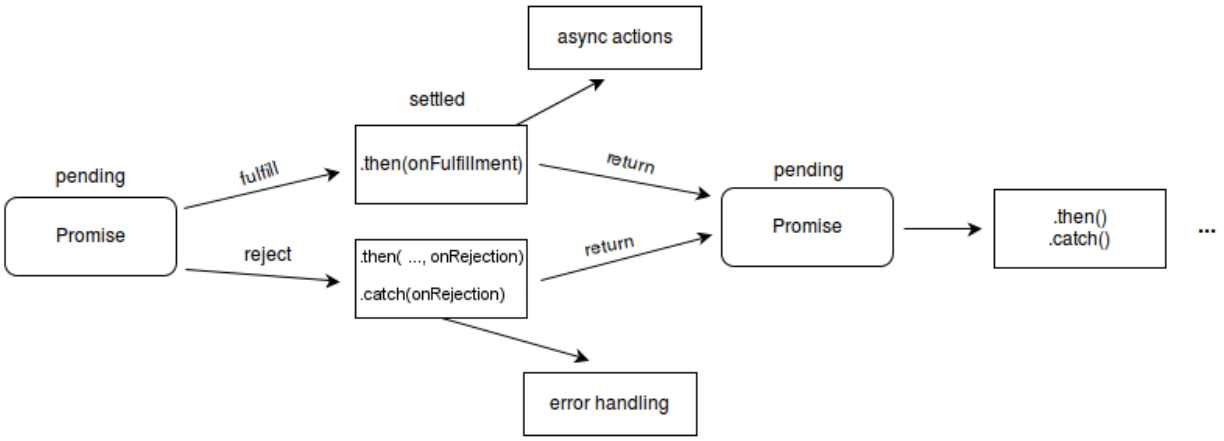
The dictionary definition of promises is –

> "Assurance that one will do something or that a particular thing will happen."

JavaScript promises also work in the same way.

- When a promise is created, there are only two outcomes to that promise.
- Either it will be fulfilled (resolved) or it will be rejected.
- By the time promises are not fulfilled or rejected, they will be in a pending state.
- Promises are fulfilled with a certain value, that value can be further processed (if the value also is a promise) or given back raw.

- Promises are rejected with the reason that caused them to be rejected.
- After either of the results, we can also perform the next set of operations.



Working of promises MDN reference

Anatomy of promise

```
const promise = new Promise((resolve, reject) => {
    // resolve or reject
});
```

Promise has three methods available to it (then, catch, & finally) that can be used once it is settled (resolved or rejected). Each method accepts a callback function that is invoked depending on the state of the promise.

- *then(onResolvedFn, onRejectedFn)* – This will be called either when the promise is rejected or resolved. Depending upon the state, appropriate callback functions will be invoked with the value.
- *catch(onRejectFn)* – This will be called when the promise is rejected with the reason.
- *finally(onFinallyFn)* – This will be called every time after then and catch.

```
Promise.prototype.then(onResolvedFn, onRejectedFn);
```

```
Promise.prototype.catch(onRejectedFn);

Promise.prototype.finally(onFinallyFn);
```

Working of promise

Create a promise that will resolve after 5 seconds.

```
const promise = new Promise((resolve, reject) => {
    // a promise that will resolve after
    // 5 second
    setTimeout(() => {
       resolve("Hello World!");
    }, 5000);
});
```

Initially, the promise will be in the pending state.

```
console.log(promise);

/*
Promise { : "pending" }
: "pending"
: Promise.prototype { ... }
*/
```

After 5 seconds, the state of the promise will be updated.

```
setTimeout(() => {
 console.log(promise);
}, 6000);

/*
Promise { : "fulfilled", : "Hello World!" }
: "fulfilled"
: "Hello World!"
: Promise.prototype { ... }
*/
```

We can assign the .then(*onResolvedFn*, *onRejectedFn*) method to the promise but the *onResolvedFn* callback function will be called only after the promise is resolved and will have the value.

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

// "Hello World!" // after the promise is resolved that is after 5 seconds
```

Thenable promises can be chained further.

```
promise
.then((val) => { return "ABC "+ val; })
.then((val) => { console.log(val); });

// "ABC Hello World!"
```

We can attach a finally block independently to the then, as well as catch, and it will be invoked at the end.

```
promise
.then((val) => { return "ABC "+ val; })
.then((val) => { console.log(val); })
.finally(() => { console.log("task done"); });

// "ABC Hello World!"
// "task done"
```

Similarly, let's say we reject a promise after 5 seconds, then we can either use the .then(null, onRejectedFn) or .catch(onRejectedFn).

```
const promise = new Promise((resolve, reject) => {
    // a promise that will reject after
    // 5 second
    setTimeout(() => {
       reject("Error 404");
    }, 5000);
});

promise.then(null, (error) => {
  console.error("Called from then method", error);
});
// "Called from then method" "Error 404"

promise.catch((error) => {
  console.error("Called from catch method", error);
});
```

```
// "Called from catch method" "Error 404"
```

As you can notice multiple handlers can be assigned on the same promise and .then() will execute in the order of assignment.

The catch block can also be extended further using .then().

```
promise.then(null, (error) => {
  return error;
}).then((val) => {
  console.log("I am chained from then", val);
});
// "I am chained from then" "Error 404"

promise.catch((error) => {
  return error;
}).then((val) => {
  console.log("I am chained from catch", val);
});
// "I am chained from catch" "Error 404"
```

And .finally() can be attached to both of these.

```
promise.then(null, (error) => {
  return error;
}).then((val) => {
  console.log("I am chained from then", val);
}).finally(() => {
  console.log(" Then block finally done");
});

promise.catch((error) => {
  return error;
}).then((val) => {
  console.log("I am chained from catch", val);
}).finally(() => {
  console.log(" Catch block finally done");
});

"I am chained from then" "Error 404"
```

```
"I am chained from catch" "Error 404"
" Then block finally done"
" Catch block finally done"
```

Notice the order of execution, the first error is handled in .then and then in .catch and then finally blocks of both are called in order.

Helper methods

The promise object has many static methods. Some are helper's methods while others help to process the promise better.

Promise.resolve(value) creates a resolved promise.

```
Promise
.resolve("I am resolved")
.then((val) => { console.log(val); });

// "I am resolved"
```

Similarly, Promise.reject(reason) creates a rejected promise.

```
Promise
.reject("I am throwing error")
.catch((error) => { console.error(error); });

// "I am throwing error"
```

Process methods

These methods help to process async task concurrency. We have covered each of them in the problems section.

- Promise.all()
- Promise.allSettled()
- Promise.any()
- Promise.race()

Async...await

This is new syntax introduced in ES6 that helps to process the promise better.

```
const promise = Promise.resolve("I am resolved");

async function example(promise){
  // promise is wrapped in a try-catch block
  // to handle it better
  try{
    const resp = await promise;
    console.log(resp);
  }catch(error){
    console.error(error);
  }finally{
    console.log("Task done");
  }
}

example(promise);

// "I am resolved"
// "Task done"
```

To use it we have to mark the function with the async keyword and then we can use the await keyword inside the async function.

The code is wrapped inside a try-catch-finally block for frictionless execution.

async keyword with different function declaration.

```
// fat arrow
const example = async () => {
  // await can be used
};

// assigning the function variable
const example = async function(){
  // await can be used
```

```
};
```

A function declared with the async keyword returns a promise.

```
const promise = Promise.resolve("I am resolved");

// fat arrow
const example = async (promise) => {
  // promise is wrapped in a try-catch block
  // to handle it better
  try{
    const resp = await promise;
    return resp;
  }catch(error){
    console.error(error);
  }finally{
    console.log("Task done");
  }
};

console.log(example(promise));
// Promise { : "fulfilled", : "I am resolved" }
// "Task done"

example(promise).then((val) => {
  console.log(val);
});

//"Task done"
//"I am resolved"
```

Notice the order of execution here, the try and finally block will be executed, thus content in the finally block is printed and the value returned is accessed in the .then that is why "Task done" is printed before "I am resolved".

Read more about promises on MDN.