Promises

Introduction

Promises are used to handle asynchronous tasks in JavaScript. Managing them is easier when dealing with multiple asynchronous operations, where we'll get stuck in **callback hell**, created by callbacks which ultimately leads to unmanageable code.
Promises make error handling in JavaScript look like a piece of cake!

A Promise is in one of these states:

- **pending**: neither fulfilled nor rejected, in its initial state.
- **fulfilled**: implying that the operation went off without a hitch.
- **rejected**: implying that the operation was a failure.
- **settled:** either completed or rejected, but not pending

Creating Promises

We can create a promise using the **Promise** constructor:

```
var promise = new Promise(function(resolve, reject) {
    //do something
});
```

Promise constructor takes a **function** as a single argument, which takes two arguments, **resolve** and **reject**.
We will call **resolve** if everything inside the **function** goes **well**, else we call **reject**.

A pending promise can be fulfilled by providing a value or rejected by providing a reason (error). If any of these options occur, we must take the appropriate steps.

The methods **promise.then()** and **promise.catch()** are used to take any further action with a promise that becomes settled.

then():
When a promise is **resolved** or **rejected**, **then() is called**. Two functions are passed to the then() method. If the promise is resolved and a result is received, the **first function is called**. If the promise is **rejected** and an **error is returned**, the **second function is called**. (It's **optional** as the **catch() method is comparatively a better way to handle errors.**).

```
promise.then(function() {
    //handle success
}, function(error) {
    //handle error
});
```

catch():
catch() is called to handle the **errors**, i.e., when a promise is rejected or when some error has occurred during the execution.
catch() method takes one function as an argument, which is used to handle errors.

```
promise.catch(function(error){
    //handle error
    //catch() internally calls the error handler of then().
});
```

Given below is example of a rejected promise:

```
var promise = new Promise(function(resolve, reject) {
    var s1 = "Hi!";
    var s2 = "Hello";
    if (s1 == s2) {
        resolve();
    } else {
        reject();
    }
});
promise.then(function() {
    console.log('Matched!');
}).catch(function() {
    console.log("Not Matched!");
});
// Output: Not Matched!
```

Passing Parameters to Resolve/Reject

You can also pass parameters to resolve and reject!
For example, the above promise can also be written as:

```
var promise = new Promise(function(resolve, reject) {
    var s1 = "Hi!";
    var s2 = "Hello";
    if (s1 == s2) {
        resolve("matched!");
    } else {
        reject("not matched!");
    }
});
promise.then(function(successMsg) {
    console.log(successMsg);
}).catch(function(failureMsg) {
    console.log(failureMsg);
});
//Output: Not Matched!
```

Ideally, you should always wrap the promise within a function, which in turn will return the promise. This will increase the readability of your code and it'll also add meaning to it. For example:

```
    function equalOrNot() {
    var promise = new Promise(function(resolve, reject) {
        var s1 = "Hi!";
        var s2 = "Hi!";
        if (s1 == s2) {
            resolve("Matched!");
        } else {
            reject("Not Matched!");
        }
    });
    return promise;
}
```

```
equalOrNot().then(function(successMsg) {
    console.log(successMsg);
}).catch(function(failureMsg) {
    console.log(failureMsg);
});
//Output: Matched!
```

Chaining Requests

Multiple callback functions would create a callback hell that leads to unmanageable code. So many function calls will lead to confusion and it'll become harder for us to understand. We can use promises in chained requests to avoid callback hell and easily avoid callback hell.

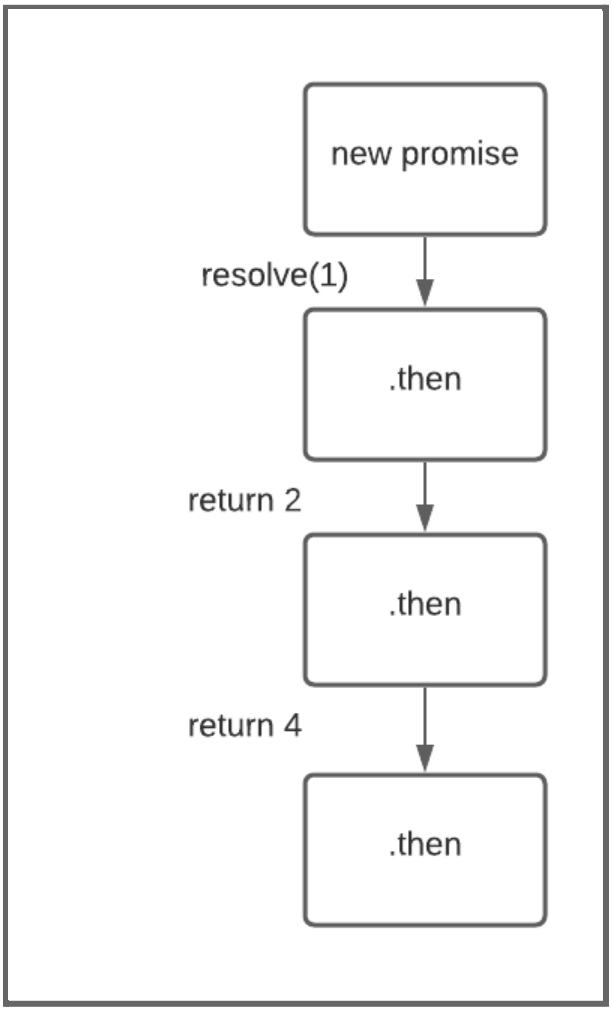
```
new Promise(function(resolve, reject) {
    setTimeout(() => resolve(1), 1000); // (A)
}).then(function(result) { // (B)
    alert(result); // 1
    return result * 2;
}).then(function(result) { // (C)
    alert(result); // 2
    return result * 2;
}).then(function(result) {
    alert(result); // 4
    return result * 2;
});
```

Here, the result is passed through the chain of .then() handlers.

The flow is given as:

1. The initial promise resolves in 1 second (A),
2. Then the .then() handler is called (B).
3. The value that it returns is passed to the next .then() handler (C)
4. ...and so on.

As the result is passed along the chain of handlers, we can see a sequence of alert calls: 1 → 2 → 4.
You can refer to the diagram below:



The whole thing works because the then() method returns a promise, which is used to call the next then() method (). When a handler returns a value, the promise becomes the result, and next then() method is called again.

This type of structure is much better because it's a lot more intuitive, logical, and easier to understand. Therefore we should use Promises in such cases.