Promises and promise chaining

Time before promise, callbacks

There are entities in JavaScript that let you schedule asynchronous tasks, tasks that you initiate first, then they will finish later.

Task like disk reading, fetching resources from network that will take arbitrary amount of time and you don't want your CPU to sit at that line of code waiting until it finishes. You would like to schedule it and then come back when the task finishes. Functions such as `setTimeout`, `setInterval`, or asynchronous file reading let you do that, **let you schedule an asynchronous task, they start, but finishes later, when the resources that they are waiting on are finished.**

Often time, after the task is finished, we want to use the result of the task immediately how would we do that? We cannot just add a function call right after the task because it will be scheduled later, and we don't know if it will finishes right after it starts.

Say `loadScript` will take 10 seconds to load, then calling `newFunction()` immediately after you started loading will be an error.

```
loadScript('myscript.js'); // contains newFunction(), but takes 10 seconds to load


newFunction(); // no such function
```

Introducing callbacks

A callback function is a function that will be run after the asynchronous task is finished. We can add a callback parameter to the `loadScript` function

```
function loadScript(src, callback) {
 let script = document.createElement('script');
   script.src = src;
   script.onload = () => callback(script); // Scripts are loaded asynchronously by the browser.
                      // We have to provide an empty arrow function because the
                      // event handler for onload calls a function with no parameter
                      // inside the body we will invoke the callback, when script is loaded
   document.head.append(script);
}
```

```
loadScript('myscript.js', (script) => console.load(`My script ${script} is loaded`));
```

Okay, then this guarantees that `"My script ${script} is loaded"` message to show up after `myscript.js` is loaded into the HTML. Now here comes the interesting part, what if I want to load a second script `myscript2.js` right after `myscript.js` is finished?

Easy, we can just add another `loadScript` function call inside the callback of the first script load.

```
loadScript('myscript.js', function(script) {
 console.log("First script loaded");

  loadScript('myscript2.js', function(script) {
     console.log("Second script loaded");
   });
 });
```

And what if there is a third script that I want to load only after `myscript2.js` finishes? We would just continue on nesting into the callbacks, and this is what is called the **Pyramid of Doom or callback hell.**

It gets worse if you added error handling into `loadScript` itself, what if the script that you are loading was unable to complete? It adds another layer or nesting like so:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });
```

```
        }
    });
    }
});
```

Solution

This can be partially alleviated by storing each step of script loading into a function on it's own, but then it creates a function that will only be called once. The better solution is to use `Promise`

Promise

An enhancement to callback functions!

The idea with `Promise` is "I promise a result to you at a later time". There are two components to a `Promise` object, the "Producing code" which is the code that can take some time. Then there is the "Consuming code" which is the code that is waiting for the producing code's result.

The "Producing code" is meant to take some long, it will be executed as an asynchronous function and the flow of the execution will move on to the next line of code. When the `Promise` is settled by either calling `resolve/reject` with a value, then "Consuming code" kicks in after it is settled and the callback function will be ran.

Producer code/Promise object creation

To create a new `Promise` object:

```
let promise = new Promise(function(res, rej) {
□// the producing code that will take some take
    // usually will be waiting for some work to be completed before resolving/rejecting
});
```

The `executor` or the "Producing code" after finishing with it's work will call either `resolve` if it was successful or `reject` if there was an error.

The `Promise` object returned by `new Promise` constructor have two internal properties

- `state` : Initially is `pending` then changes either to `fulfilled` when `resolve` is called or `rejected` when `reject` is called.
- `result` : Initially is `undefined`, then changes to `value` when `resolve(value)` is called or `error` when `reject(error)` is called. You usually reject with an `Error` even though you can technically reject with anything.

How executor is ran

```
let promise = new Promise((res, rej) => {
⬚// the function is executed automatically when promise is constructed
    setTimeout(() => resolve("done"), 1000);
});
```

1. The executor is called automatically and immediately by `new Promise` i.e. the producing work is immediately started
2. The executor will receive two arguments `resolve` and `reject`. These functions are pre-defined by the JavaScript engine, so you just need to use them.

After a second `resolve` is called with the value `"done"` and the state of `promise` is changed to `"fulfilled"` with result `"done"`.

What does (1) imply?

That means that if you have synchronous code for example a big for loop meant to print out 1 - 100,000,000 inside the `Promise` it will be executed synchronously before the promise moves onto the line after `Promise`. Unless you have `await` or waiting for other promises to resolve within this promise, the code will be executed synchronously.

Only one result

You can only call `resolve` or `reject` once. State change is final, any further calls to `resolve/resolve` are ignored.

Consumers

Now we talked about how "Producing code" work, we will look into how consumers uses the `Promise`. These are handlers that when the `Promise` resolves either a result or error, will be executed.

You register these handlers using `.then` and `.catch`.

then

```
promise.then(
⬚function(result) { /* handles successful result. Pass the value of the promise */ };
    function(error) { /* handles reject. Pass the error of the promise */ };
);
```

The first argument is the handler that will be run when `Promise` is resolved.

The second argument is the handler that will be run when `Promise` is rejected.

In the case that you only interested in the successful result, you can ignore writing the second function for handling the reject.

catch

If we are only interested in errors, then you can use `null` as the first argument:

```
.then(null, function(err) { /* error handling code */ });
```

Or the second way that you can do it is:

```
.catch(function(err) { /* error handling code */ });
```

They are equivalent.

Promise chaining

First you have to understand that the method `.then` will return a new `Promise` object that you can call `.then` again, and so on, and this is called Promise chaining.

Regardless of what you return, even if you return a primitive it will be wrapped in a `Promise` object with it's state resolved immediately. If you return a `Promise` object, then the next `.then` that you chained will only execute when that `Promise` you have returned is resolved.

In addition, even if you return nothing, `.then` will create a `Promise` object with `undefined` as it's result.

Wrong way of chaining

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

```
promise.then(function(result) {

  alert(result); // 1

  return result * 2;

});
```

This is the wrong way of chaining promises, these are registering three separate handlers listening on the same `promise` object that will all be executed simultaneously when `promise` is resolved.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

This is the correct way of chaining the promise handlers, the first `.then` will only execute when the promise object is resolved after one second.

The second `.then` will execute after the first `.then` has returned a resolved promise. Since the first `.then` returns a primitive it will execute immediately right after the first `.then` is executed. Then so on...

Returning promises

Inside `.then` like it was mentioned you can create and return a promise. In that case further handlers will wait until it settles, and then it will execute

```
new Promise(function(res, rej) {
 setTimeout(() => res(1), 1000);
}).then(function(val) {
 console.log(val);
    return new Promise((res, rej) => {
     setTimeout(() => res(val * 2), 1000);
 });
}).then(function(val) {
   console.log(val);
});
```

Here, the first `.then` will execute after one second, then the second `.then` will execute after another one second, because the first `.then` returns a new promise, the second handler must wait until that promise resolves before it can execute!

.catch for promise chaining

You can write one `.catch` to handle the entire promise chaining if error has occurred anywhere along the chain. This works because if one of the promises' state becomes rejected, then the `.then` will not execute and it will continue down the chain until it finds `.catch` or a `.then` with an error handler.

Promise handler have a implicit `try...catch`, if an exception happens it will get caught and treat as a rejection promise object.

```
new Promise((resolve, reject) => {
 throw new Error("Whoops");
}).catch(console.log);

// Equivalent to
new Promise((resolve, reject) => {
 reject(new Error("Whoops"));
}).catch(console.log);
```
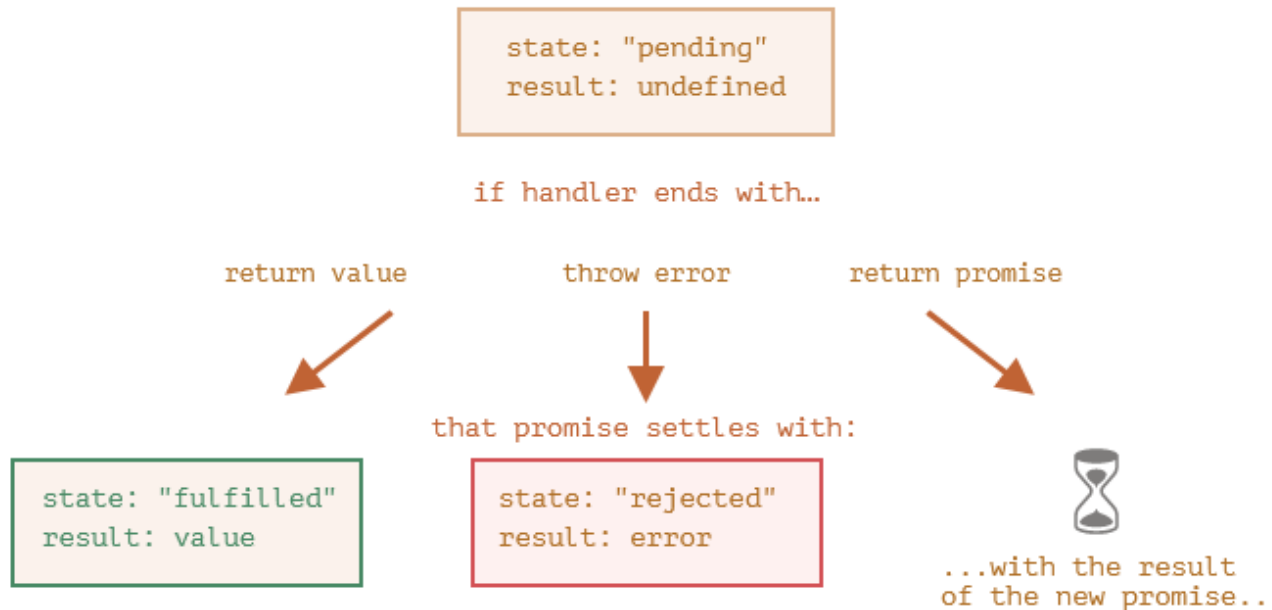
It also happens in `.then` chaining.

Good summary picture for promise chaining

the call of .then(handler) always returns a promise:

```
state: "pending"
result: undefined
```

if handler ends with...

return value   throw error   return promise

that promise settles with:

```
state: "fulfilled"    state: "rejected"
result: value         result: error
```

...with the result
of the new promise..

Solving loadScript with promise

Now let's revisit the pyramid of doom that we had with `loadScript`, how do we make it so that one script loads one after the other without callback hell?

Simple!

We make `loadScript` return a promise like so:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for ${src}`));

    document.head.append(script);
  });
}
```

The promise object will resolve when the script is loaded and reject when the script failed to load.

Now we can just chain `loadScript` right after one another because it returns a promise.

```
loadScript("myscript1.js")
.then(val => {
    return loadScript("myscript2.js"); // remember loadScript returns a promise
})
    .then(val => {
    return loadScript("myscript3.js"); // this will only run when the previous .then's promise
resolves, i.e. when myscript2.js is loaded
})
    .then(val => {
    one();
        two();
        three();
        // Now you can call the functions that were loaded from each script files
});
```

And it looks way nicer to the eye without callback hell.

How are code executed inside Promise?

If you write synchronous code inside the executor then it will be executed synchronously immediately, however, the handler will still be executed asynchronously.

However, if you write asynchronous code, meaning it will say wait for some task to be completed, then the flow of the code will move onto the next line of the code, and when the task is done then handler will be executed asynchronously as well.

Revision #7
Created 29 December 2022 03:50:15 by Tamarine
Updated 27 July 2023 01:53:21 by Tamarine