Cheatsheets / **Learn Intermediate JavaScript**

codecademy

Promises

States of a JavaScript Promise

A JavaScript `Promise` object can be in one of three states: `pending`, `resolved`, or `rejected`.

While the value is not yet available, the `Promise` stays in the `pending` state. Afterwards, it transitions to one of the two states: `resolved` or `rejected`.

A `resolved` promise stands for a successful completion. Due to errors, the promise may go in the `rejected` state.

In the example code block, if the `Promise` is on `resolved` state, the first parameter holding a callback function of the `.then()` method will print the resolved value. Otherwise, an alert will be shown.

```
const promise = new Promise((resolve,
reject) => {
  const res = true;
  // An asynchronous operation.
  if (res) {
    resolve('Resolved!');
  }
  else {
    reject(Error('Error'));
  }
});

promise.then((res) => console.log(res),
(err) => alert(err));
```

The `.catch()` method for handling rejection

The function passed as the second argument to a `.then()` method of a promise object is used when the promise is rejected. An alternative to this approach is to use the JavaScript `.catch()` method of the promise object. The information for the rejection is available to the handler supplied in the `.catch()` method.

```
const promise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    reject(Error('Promise Rejected
Unconditionally.'));
  }, 1000);
});

promise.then((res) => {
  console.log(value);
});

promise.catch((err) => {
  alert(err);
});
```

JavaScript Promise.all()

The JavaScript `Promise.all()` method can be used to execute multiple promises in parallel. The function accepts an array of promises as an argument. If all of the promises in the argument are resolved, the promise returned from `Promise.all()` will resolve to an array containing the resolved values of all the promises in the order of the initial array. Any rejection from the list of promises will cause the greater promise to be rejected.
In the code block, `3` and `2` will be printed respectively even though `promise1` will be resolved after `promise2`.

```
const promise1 = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve(3);
  }, 300);
});
const promise2 = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve(2);
  }, 200);
});

Promise.all([promise1,
promise2]).then((res) => {
  console.log(res[0]);
  console.log(res[1]);
});
```

Executor function of JavaScript Promise object

A JavaScript promise's executor function takes two functions as its arguments. The first parameter represents the function that should be called to resolve the promise and the other one is used when the promise should be rejected. A `Promise` object may use any one or both of them inside its executor function.
In the given example, the promise is always resolved unconditionally by the `resolve` function. The `reject` function could be used for a rejection.

```
const executorFn = (resolve, reject) => {
  resolve('Resolved!');
};

const promise = new Promise(executorFn);
```

.then() method of a JavaScript Promise object

The `.then()` method of a JavaScript `Promise` object can be used to get the eventual result (or error) of the asynchronous operation.

`.then()` accepts two function arguments. The first handler supplied to it will be called if the promise is resolved. The second one will be called if the promise is rejected.

```
const promise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve('Result');
  }, 200);
});

promise.then((res) => {
  console.log(res);
}, (err) => {
  alert(err);
});
```

`setTimeout()`

`setTimeout()` is an asynchronous JavaScript function that executes a code block or evaluates an expression through a callback function after a delay set in milliseconds.

```
const loginAlert = () =>{
  alert('Login');
};

setTimeout(loginAlert, 6000);
```

Avoiding nested `Promise` and `.then()`

In JavaScript, when performing multiple asynchronous operations in a sequence, promises should be composed by chaining multiple `.then()` methods. This is better practice than nesting.
Chaining helps streamline the development process because it makes the code more readable and easier to debug.

```
const promise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    resolve('*');
  }, 1000);
});

const twoStars = (star) => {
  return (star + star);
};

const oneDot = (star) => {
  return (star + '.');
};

const print = (val) => {
  console.log(val);
};

// Chaining them all together
promise.then(twoStars).then(oneDot).then(p
rint);
```

Creating a Javascript Promise object

An instance of a JavaScript `Promise` object is created using the `new` keyword.
The constructor of the `Promise` object takes a function, known as the *executor function*, as the argument. This function is responsible for resolving or rejecting the promise.

```
const executorFn = (resolve, reject) => {
  console.log('The executor function of
the promise!');
};

const promise = new Promise(executorFn);
```

The `Promise` Object

A `Promise` is an object that can be used to get the outcome of an asynchronous operation when that result is not instantly available.
Since JavaScript code runs in a non-blocking manner, promises become essential when we have to wait for some asynchronous operation without holding back the execution of the rest of the code.

Chaining multiple .then() methods

The `.then()` method returns a Promise, even if one or both of the handler functions are absent. Because of this, multiple `.then()` methods can be chained together. This is known as composition.
In the code block, a couple of `.then()` methods are chained together. Each method deals with the resolved value of their respective promises.

```
const promise = new Promise(resolve =>
setTimeout(() => resolve('dAlan'), 100));

promise.then(res => {
  return res === 'Alan' ?
Promise.resolve('Hey Alan!') :
Promise.reject('Who are you?')
}).then((res) => {
  console.log(res)
}, (err) => {
  alert(err)
});
```

Print Share