Javascript promises

In web development, asynchronous programming is notorious for being a challenging topic.

An *asynchronous operation* is one that allows the computer to “move on” to other tasks while waiting for the asynchronous operation to complete. Asynchronous programming means that time-consuming operations don’t have to bring everything else in our programs to a halt.

JavaScript handles asynchronicity using the `Promise` object, introduced with ES6.

What is a promise?

Promises are objects that represent the eventual outcome of an asynchronous operation. A `Promise` object can be in one of three states:

- **Pending**: The initial state— the operation has not completed yet.
- **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request's promise might resolve with a JSON object as its value.
- **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an `Error` of some kind.

We refer to a promise as *settled* if it is no longer pending— it is either fulfilled or rejected.

Constructing a Promise

Let's construct a promise! To create a new `Promise` object, we use the `new` keyword and the `Promise` constructor method:

```
const executorFunction = (resolve, reject) => { };
const myFirstPromise = new Promise(executorFunction);
```

The `Promise` constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the `resolve()` and `reject()` functions. The `resolve()` and `reject()` functions aren't defined by the programmer. When the `Promise` constructor runs, JavaScript will pass **its own** `resolve()` and `reject()` functions into the executor function.

- `resolve` is a function with one argument. Under the hood, if invoked, `resolve()` will change the promise's status from `pending` to `fulfilled`, and the promise's resolved value will be set to the argument passed into `resolve()`.
- `reject` is a function that takes a reason or error as an argument. Under the hood, if invoked, `reject()` will change the promise's status from `pending` to `rejected`, and the promise's rejection reason will be set to the argument passed into `reject()`.

Example

```
const executorFunction = (resolve, reject) => {
 if (someCondition) {
     resolve('I resolved!');
 } else {
     reject('I rejected!');
 }
}
const myFirstPromise = new Promise(executorFunction);
```

setTimeout() function

Knowing how to construct a promise is useful, but most of the time, knowing how to *consume*, or use, promises will be key. Rather than constructing promises, you'll be handling `Promise` objects returned to you as the result of an asynchronous operation. These promises will start off pending but settle eventually.

Let's look at how we'll be using `setTimeout()` to construct asynchronous promises:

```
const returnPromiseFunction = () => {
  return new Promise((resolve, reject) => {
    setTimeout(( ) => {resolve('I resolved!')}, 1000);
  });
};

const prom = returnPromiseFunction();
```

Consuming Promises

The initial state of an asynchronous promise is `pending`, but we have a guarantee that it will settle. How do we tell the computer what should happen then? Promise objects come with an aptly named `.then()` method. It allows us to say, "I have a promise, when it settles, **then** here's what I want to happen…"

`.then()` is a higher-order function— it takes two callback functions as arguments. We refer to these callbacks as *handlers*. When the promise settles, the appropriate handler will be invoked with that settled value.

- The first handler, sometimes called `onFulfilled`, is a *success handler*, and it should contain the logic for the promise resolving.
- The second handler, sometimes called `onRejected`, is a *failure handler*, and it should contain the logic for the promise rejecting.

Examples

```
const prom = new Promise((resolve, reject) => {
  resolve('Yay!');
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

prom.then(handleSuccess); // Prints: 'Yay!'
```

```
let prom = new Promise((resolve, reject) => {
  let num = Math.random();
  if (num < .5 ){
    resolve('Yay!');
  } else {
    reject('Ohhh noooo!');
  }
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

const handleFailure = (rejectionReason) => {
  console.log(rejectionReason);
};

prom.then(handleSuccess, handleFailure);
```

Using catch() with promises

Remember, `.then()` will return a promise with the same settled value as the promise it was called on if no appropriate handler was provided. This implementation allows us to separate our resolved logic from our rejected logic.

The `.catch()` function takes only one argument, `onRejected`. In the case of a rejected promise, this failure handler will be invoked with the reason for rejection. Using `.catch()` accomplishes the same thing as using a `.then()` with only a failure handler.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Chaining multiple promises

One common pattern we'll see with asynchronous programming is multiple operations which depend on each other to execute or that must be executed in a certain order. We might make one request to a database and use the data returned to us to make another request and so on.

```
firstPromiseFunction()
.then((firstResolveVal) => {
  return secondPromiseFunction(firstResolveVal);
})
.then((secondResolveVal) => {
  console.log(secondResolveVal);
});
```

Using Promise.all()

When done correctly, promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters. What if we're dealing with multiple promises, but we don't care about the order?

To maximize efficiency we should use *concurrency*, multiple asynchronous operations happening together. With promises, we can do this with the function `Promise.all()`.

`Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

- If every promise in the argument array resolves, the single promise returned from `Promise.all()` will resolve with an array containing the resolve value from each promise in the argument array.
- If any promise from the argument array rejects, the single promise returned from `Promise.all()` will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

Example

```
let myPromises = Promise.all([returnsPromOne(),
returnsPromTwo(), returnsPromThree()]);

myPromises
  .then((arrayOfValues) => {
    console.log(arrayOfValues);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```