JavaScript Revisited

Sridhar Alagar

Callback - Motivation

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Creates a <script> tag and append it to the page

When invoked, it causes the script with given ‘src’ to start loading and run when complete

Callback - Motivation

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Will the code below work?

```
loadScript('./myscript.js');
// the script has "function sayHi() {…}"

sayHi();
```

Won't work. Script is taking some time to load…

Add a Callback

Use callback to do something after loading

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback();
  document.head.append(script);
}
```

```
loadScript(‘./myscript.js’, () => sayHi());
```

Callback Takeaway

A function that does something in the background should have a callback parameter where we put the function to run after it's complete

This is called a "callback-based" style of asynchronous programming

Callback in a callback

How can we load two scripts sequentially: the first one, and then the second one after it?

```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the first is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  }); //2nd callback ends
}); //1st callback ends
```

Callback in a callback in a callback

What if we want one more script…?

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...continue after all scripts are loaded
    });

  });

});
```

Handling Errors

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback…
  script.onerror = () => callback…

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

Handling Errors

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load
                                               error`));

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

"error-first" callback style

The first argument of the callback is reserved for an error if it occurs

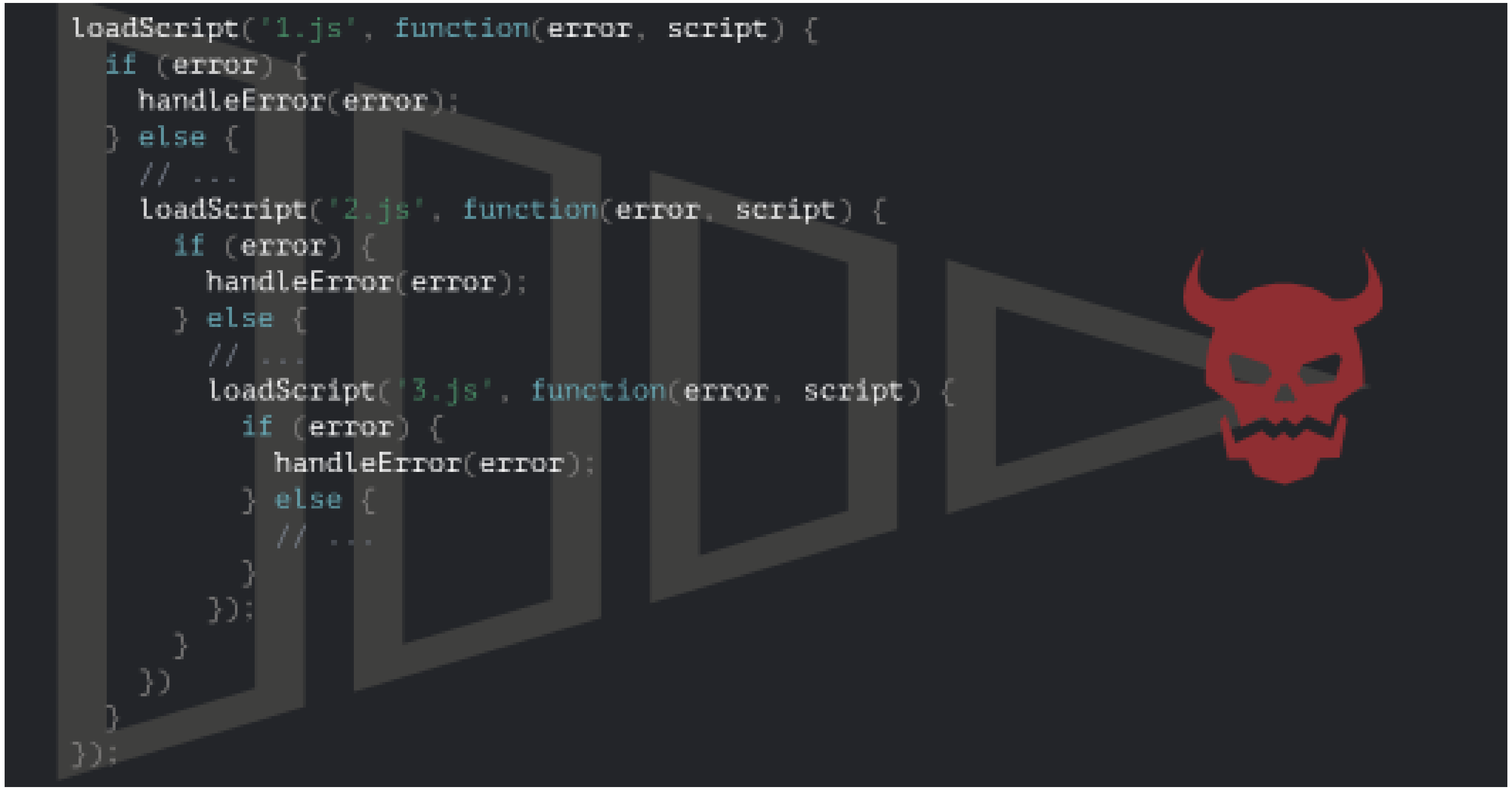
Nested callback with error handling

```
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 (*)
          }
        });
      }
    });
  }
});
```

Pyramid of doom or callback hell

As calls become more nested, the code becomes deeper and increasingly more difficult to manage

Especially, if we have code with loops, conditional statements…



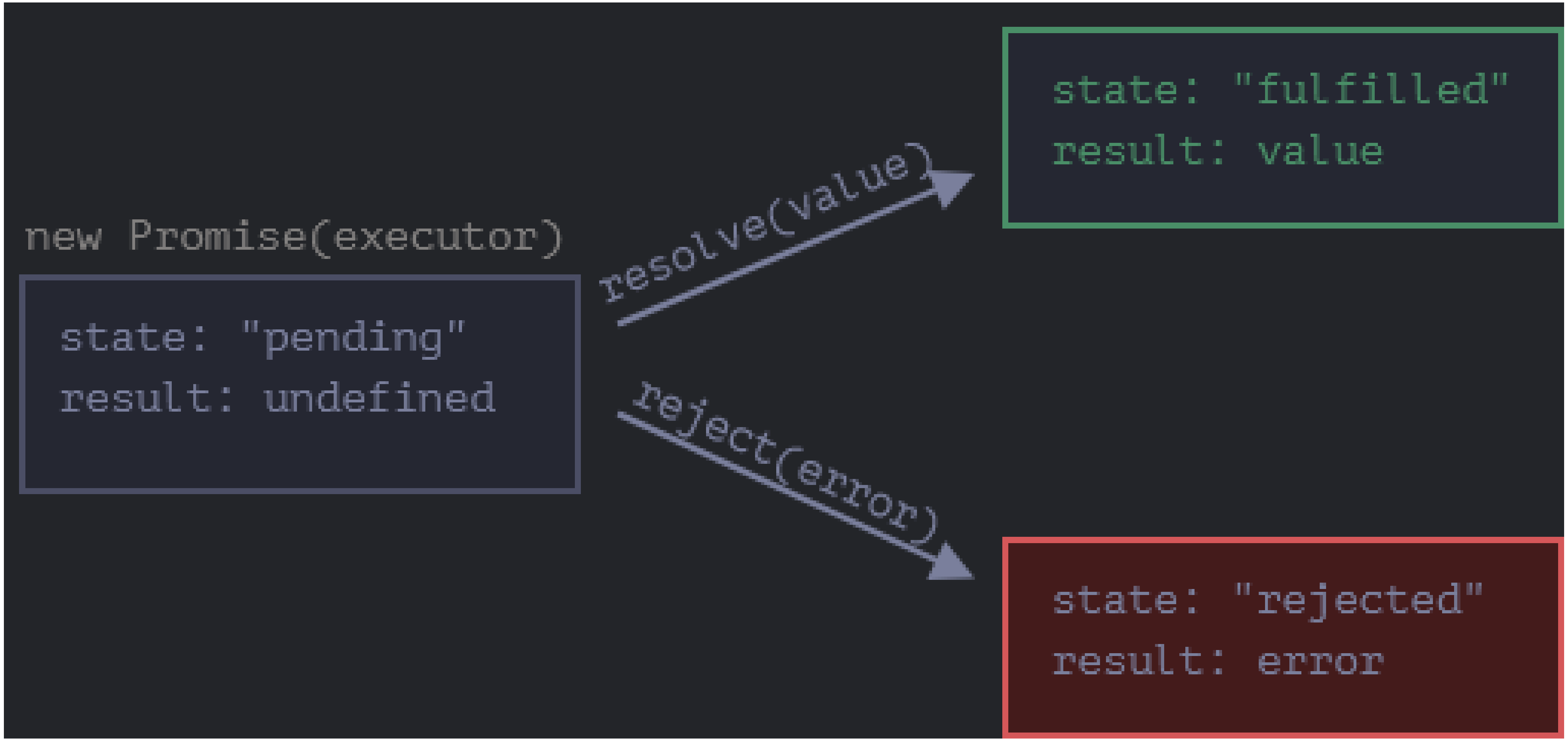
```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

A Promise to save us from doom

```
let promise = new Promise((resolve, reject) => {
  // executor (the producing) code
});
```

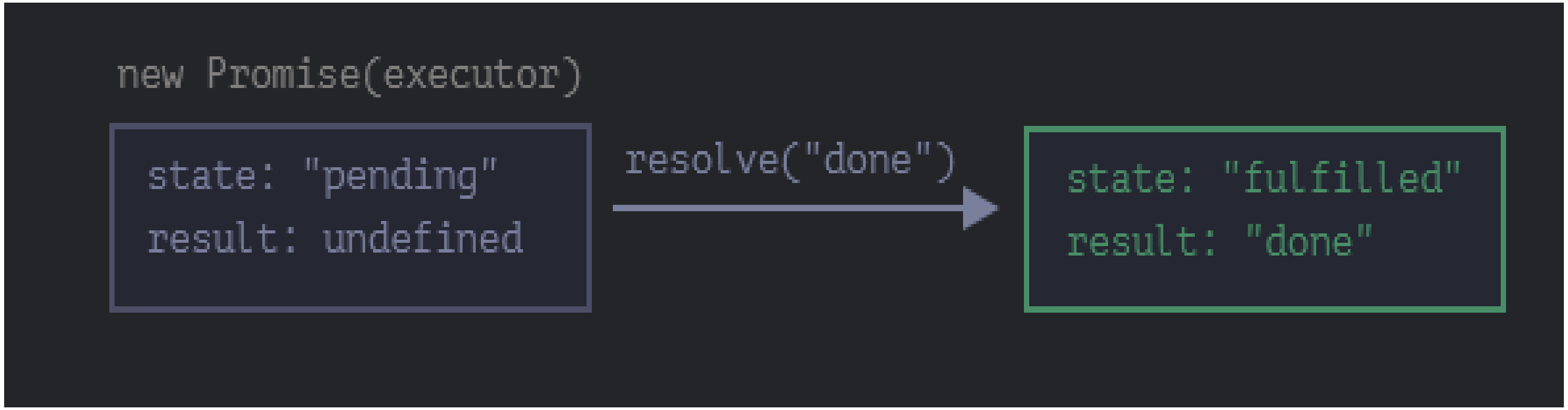
- When a new promise object is created, the function (executor) passed to the constructor is executed
- resolve, and reject are two callbacks provided by JavaScript
- Executor should call either of these callbacks
  - resolve(result) – if the job is finished successfully, with the value of result
  - reject(error) – in case of error, with 'error' object

Promise object has internal properties



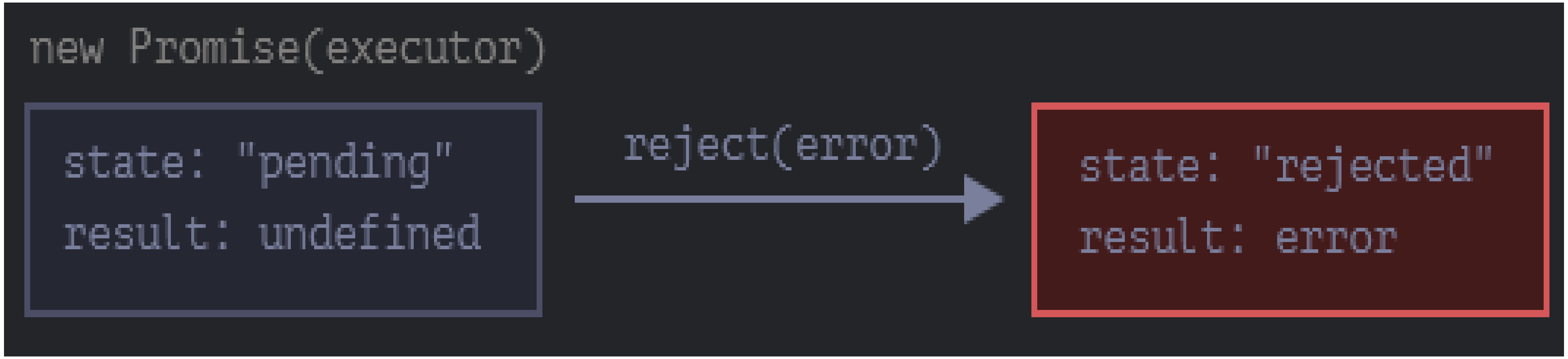
Promise usage

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```



Promise usage

```
let promise = new Promise(function(resolve, reject) {

  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```



Consumers: then, catch

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

```
// resolve runs the first function in .then
// reject runs the second function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

Consumers: catch

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

loadscript with promise

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

    script.onload = () => ?
    script.onerror = () => ?

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

loadscript with promise

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

loadscript usage

```
let promise = loadScript(“./myscript.js");


promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);


promise.then(script => alert('Another handler...'));
```

We can call ‘then’ on a promise as many times as we want

Whereas there can be only one callback

Promise chaining to prevent callback hell

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

}).then(function(result) { // result of 1st promise
            alert(result); // 1
            return result * 2;
// then returns a new promise object
}).then(function(result) { // return value of previous handler passed
            alert(result); // 2
            return result * 2;

}).then(function(result) {
            alert(result); // 4
            return result * 2;
});
```

Returning promises

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

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

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // waits for the previous promise to resolve

  alert(result); // 2

});
```

Chaining promisified ‘loadscript’

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
    one();
    two();
    three();
  });
```

Promise API – Promise.all

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert);
```

Async/Await

A special syntax to work with promises comfortably

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

```
function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Await

Works only inside an async function

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves

  alert(result); // "done!"
}

f();
```

Sources

1. JAVASCRIPT.INFO - Async