JavaScript Programming

Tiep Phan – SSE – July 2018

NASH TECH
The Power to Innovate

Agenda

- Course overview.
- Course contents.
- Examination.

Course Overview

- Introduction to JavaScript.
- Why learn JavaScript?
- The fundamentals of JavaScript.
- JavaScript new features: ES2015 → ES2018.

Course Contents

- Section 1: Data type, Operators, Variables, Scope, Program Structure.
- Section 2: Function, Higher-Order Function, Object/Class, Array.
- Section 3: Asynchronous Programming, Events, Block scope variables, Constant.
- Section 4: Arrow functions, Default parameter value, Rest, Spread, Class, Module.
- Section 5: Promise, Async/Await.

Section 4

Arrow functions, Default parameter value, Rest, Spread, Class, Module

Functions

- A function is a JavaScript procedure—a set of statements that performs a task or calculates a value.
- To use a function, you must define it somewhere in the scope from which you wish to call it.

Functions

```
1 function sayHi(fname, lname) {
2     return 'Hi: ' + fname + ' ' + lname;
3 }
4
5 console.log(sayHi('Tiep', 'Phan'));
6
```

Functions

- The Three Roles of Functions in JavaScript:
  - Non-method function (normal function).
  - Constructor.
  - Method.

```
1 function HttpServer(host, port) {
2   this.host = host;
3   this.port = port;
4 }
5
6 HttpServer.prototype.print = function() {
7   console.log('Server running at ' + this.host + ':' + this.port);
8 }
9 // Constructor
10 var httpServer = new HttpServer('localhost', 80);
11
12 //method
13 httpServer.print();
14
```

Arrow Functions

- Clean format.
- `this` vs Arrow fn.

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);

const obj = {
  t: 50,
  lazyComp() {
    setTimeout(() => console.log(this.t), 1000);
  }
}
```

Arrow Functions

- Specifying parameters:

```
() => { /*CODE*/ } // no parameter
x => { /*CODE*/ } // one parameter, an identifier
(x, y) => { /*CODE*/ } // several parameters
```

Arrow Functions

- Specifying a body:

```
x => { return x * x } // block
x => x * x // expression, equivalent to previous line

// eg
const squares = [1, 2, 3]
   .map(function (x) { return x * x });

const squares = [1, 2, 3].map(x => x * x);
```

Arrow Functions

- **No line break after arrow function parameters**

```
const gx = (x, y) // SyntaxError
=> x + y;
const hx = (x, y) => // OK
x + y;
```

Arrow Functions

- **Returning object literals**

```
const gx = (x, y) => {x: x, y: y}; // SyntaxError
const hx = (x, y) => ({x: x, y: y}); // OK
```

Arrow Functions

Arrow functions versus normal functions

An arrow function is different from a normal function in only two ways:

- The following constructs are lexical:
  - **arguments**, **super**, **this**, **new.target**
- It can't be used as a constructor: Normal functions support **new** via the internal method [[Construct]] and the property prototype. Arrow functions have neither, which is why new (() => {}) throws an error.

Default Parameter Value

```
function getPost(perPage = 10) {
  console.log(`getPost with ${perPage} items`);
}
function getPost(perPage) {
  if (perPage === void 0) { perPage = 10; }
  console.log("getPost with " + perPage + " items");
}
```

Default Parameter Value

- **Referring to other parameters in default values**

```
function gx(x, y = x) {
  console.log(x, y);
}

gx(3)

gx(3, 5)
```

Destructuring Assignment

- The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Destructuring Assignment

```
const obj = { first: 'Jane', last: 'Doe' };
const { first: f, last: l } = obj;
// f = 'Jane'; l = 'Doe'

// { prop } is short for {prop: prop}
const { first, last } = obj;
// first = 'Jane'; last = 'Doe'
```

Destructuring Assignment

```
const [x, y] = ['a', 'b', 'c'];
// x = 'a'; y = 'b';


const [x, , y] = ['a', 'b', 'c'];
// x = 'a'; y = ?
```

Destructuring Assignment

```
const obj = {
  v: {
    g: 5
  }
};

const { v: { g } } = obj;
```

Destructuring Assignment

```
const [x, y = 10] = ['a'];
```

Rest Operators

```
function add(...args) {
  return args.reduce((acc, v) => acc + v);
}

add(2, 3, 4, 5)
//14
```

Rest Operators

```
const obj = {
   a: 5,
   b: 10,
   c: 35
};

const { b, ...bo } = obj;

const [x, ...y] = ['a', 'b', 'c', 'd'];
```

Spread Operators

```
const obj = {
  a: 5,
  b: 10,
  c: 35
};


const ba = {
  ...obj,
  b: 50
};
```

Spread Operators

```
const arr = ['a', 'b', 'c', 'd'];
const brr = ['g', 'h', 'i', ...arr];
```

ES2015 Class

```
class Shape {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  moveTo(x, y) {
    this.x = x;
    this.y = y;
  }
}

class Rect extends Shape {
  constructor(x, y, w, h) {
    super(x, y);
    this.w = w;
    this.h = h;
  }
}
```

ES2015 Class

- **Class declarations are not hoisted**

```
// OK
new Shape();
function Shape() {
}

// Error
new Shape();

class Shape {
}
```

ES2015 Class

- ***class declarations* vs *class expressions***

```
const Shape = class {
}
```

```
const Shape = class S {
}
```

ES2015 Class

- ***constructor, static methods, prototype methods***

```
class Shape {
  constructor(x, y) {
    this.moveTo(x, y)
  }
  moveTo(x, y) {
    this.x = x;
    this.y = y;
  }
  static create(x = 0, y = 0) {
    return new Shape(x, y)
  }
}
```

Module Pattern

```
const dataService = (function() {
  return {
    getPost: function() {
      console.log('GET POST');
    }
  };
})();
```

Module Pattern

```
const dataService = (function() {
  let privateData = 50;
  function updatePrivateData() {
    privateData += 5;
  }
  return {
    getPost: function(refresh = false) {
      console.log('GET POST');
      if (refresh) {
        updatePrivateData();
      }
    }
  };
})();
```

ES2015 Module

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export } from "module-name";
import { export as alias } from "module-name";
import { export1 , export2 } from "module-name";
import {
  export1 , export2 as alias2 , [...]
} from "module-name";
```

http://exploringjs.com/es6/ch_modules.html
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import

ES2015 Module

```
import defaultExport, {
  export [ , [...] ]
} from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
var promise = import(module-name);
```

http://exploringjs.com/es6/ch_modules.html
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import

ES2015 Module

You can’t nest import inside if statements, functions, etc.

```
if (true) {
  import { something } from './src/lib';
}
```

ES2015 Module

```
export { name1, name2, …, nameN };
export { variable1 as name1, variable2 as name2, …,
nameN };
export let name1, name2, …, nameN; // also var,
const
export let name1 = …, name2 = …, …, nameN; // also
var, const
export function FunctionName(){...}
export class ClassName {...}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

ES2015 Module

```
export default expression;
export default function (…) { … } // also class,
function*
export default function name1(…) { … } // also
class, function*
export { name1 as default, … };

export * from …;
export { name1, name2, …, nameN } from …;
export {
  import1 as name1, import2 as name2, …, nameN
} from …;
export { default } from …;
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

DEMO

NASH TECH
The Power to Innovate

Q&A

Section 5

Promise, Async/Await

ES2015 Promise

- Promises are an alternative to callbacks for delivering the results of an asynchronous computation.

```
const p = new Promise( /* executor */
function(resolve, reject) {
// statements
});
```

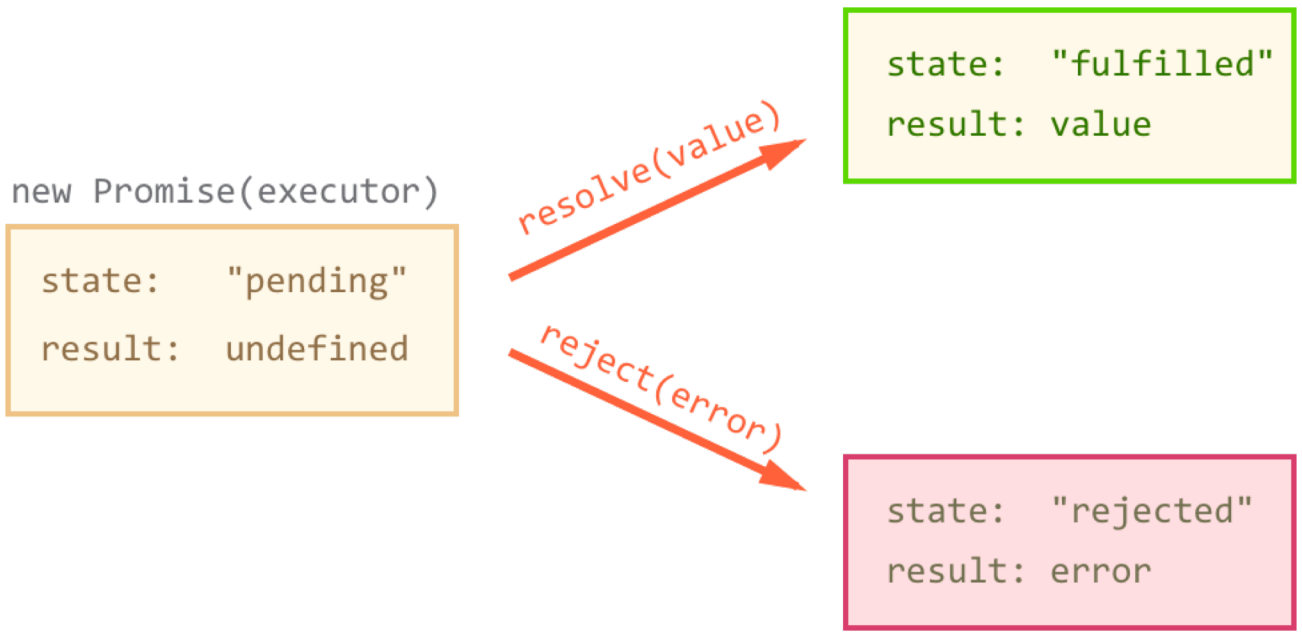
ES2015 Promise

- A Promise is always in one of three mutually exclusive states:
  - Before the result is ready, the Promise is *pending*.
  - If a result is available, the Promise is *fulfilled*.
  - If an error happened, the Promise is *rejected*.
- A Promise is *settled* if “things are done” (if it is either fulfilled or rejected).
- A Promise is settled exactly once and then remains unchanged.

ES2015 Promise

- *Promise reactions* are callbacks that you register with the Promise method then(), to be notified of a fulfillment or a rejection.
- A *thenable* is an object that has a Promise-style then() method. Whenever the API is only interested in being notified of settlements, it only demands thenables (e.g. the values returned from then() and catch(); or the values handed to Promise.all() and Promise.race()).

ES2015 Promise



ES2015 Promise

- Create a promise

```
const p = new Promise(
function (resolve, reject) { // (A)
  ...
  if (···) {
    resolve(value); // success
  } else {
    reject(reason); // failure
  }
});
```

ES2015 Promise

- Consuming a Promise

```
promise
  .then(
    value => { /* fulfillment */ },
    error => { /* rejection */ }
  )
  .catch(error => { /* rejection */ });
```

ES2015 Promise

```
function httpGet(url) {
  return new Promise(
    function (resolve, reject) {
      const request = new XMLHttpRequest();
      request.onload = function () {
        if (this.status === 200) {
          // Success
          resolve(this.response);
        } else {
          // Something went wrong (404 etc.)
          reject(new Error(this.statusText));
        }
      };
      request.onerror = function () {
      reject(new Error('XMLHttpRequest Error: '+ this.statusText));
    };
    request.open('GET', url);
    request.send();
  });
}
```

ES2015 Promise

```
httpGet(
'https://api.github.com/search/repositories?q=angular’
)
  .then(
    function (value) {
      console.log('Contents: ' + value);
    },
    function (reason) {
      console.error('Something went wrong', reason);
  });
```

ES2015 Promise

- **Other ways of creating Promises**
  - Promise.resolve()
  - Promise.reject()

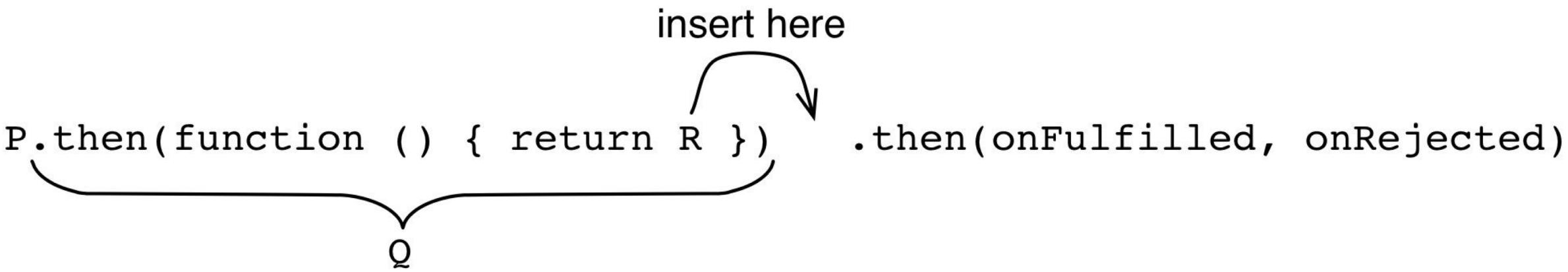
ES2015 Promise

- **Chaining Promises**

```
asyncFunc()
  .then(function (value1) {
    return 123;
  })
  .then(function (value2) {
    console.log(value2); // 123
  });
```

ES2015 Promise

- **Chaining Promises**



ES2015 Promise

- Promise anti-pattern: nested

```
asyncFunc1()
  .then(function (value1) {
    asyncFunc2()
      .then(function (value2) {
        ...
      });
  })
```

ES2015 Promise

- Promise anti-pattern: nested (fixed)

```
asyncFunc1()
  .then(function (value1) {
    return asyncFunc2();
  })
  .then(function (value2) {
    ...
  });
```

ES2015 Promise

- Promise anti-pattern: nested

```
function asyncFunc1() {
  return new Promise(function(resolve, reject) {
    asyncFunc2().then(function(data) {
      // extra work with data
      resolve(data);
    }).catch(reject);
  });
}
```

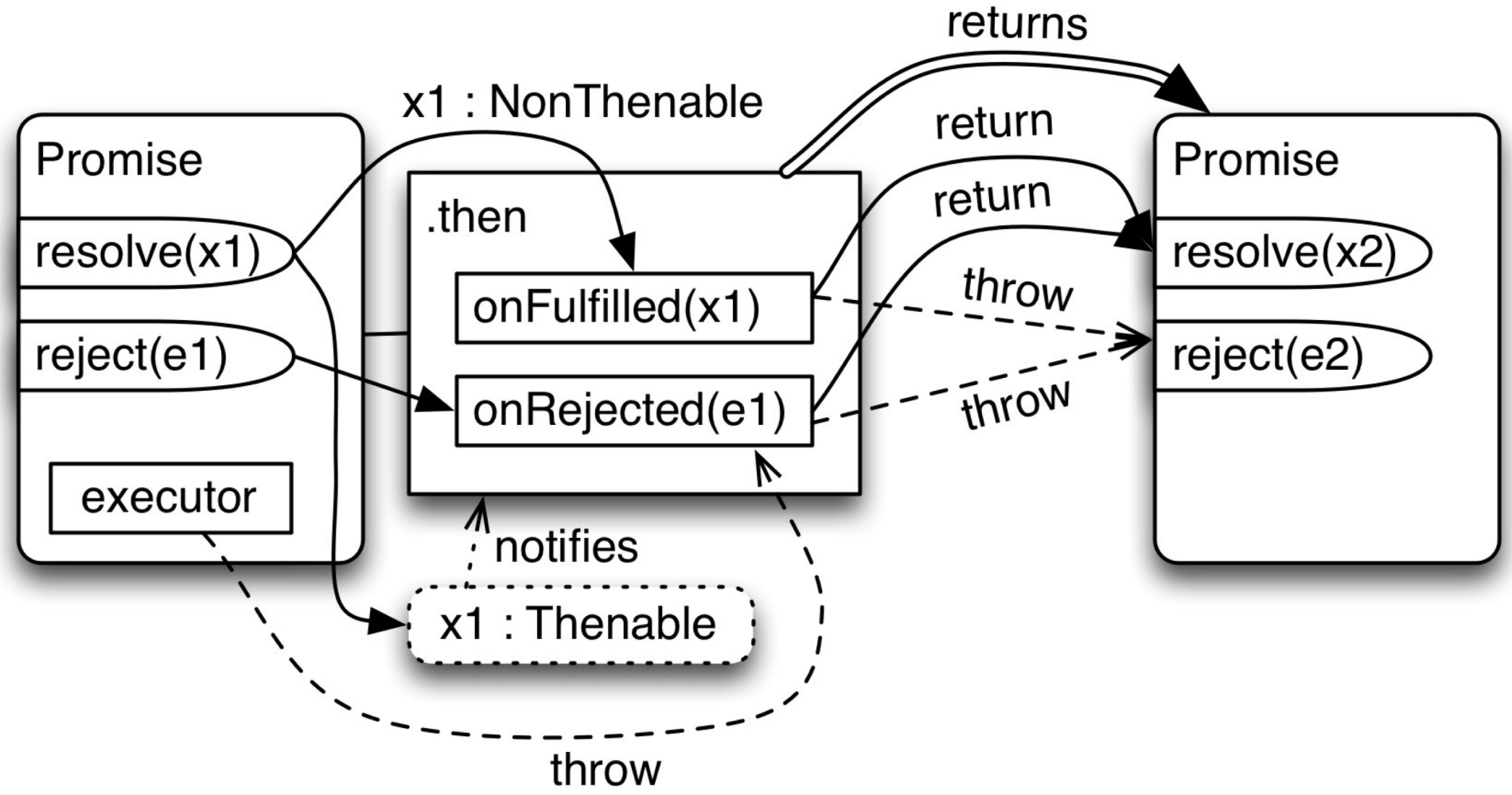
ES2015 Promise

- Promise anti-pattern: nested (fixed)

```
function asyncFunc1() {
  return asyncFunc2().then(function(data) {
    // extra work with data
    return data;
  });
}
```

ES2015 Promise

- then always return a promise



ES2015 Promise

- Promise.all
- Promise and the Event loop
- Promise only return single value.

```
const promise = new Promise(
  function(resolve, reject) {
    resolve("done");

    reject(new Error("…")); // ignored
    setTimeout(() => resolve("…")); // ignored
});
```

Async/Await

- Async functions: always return a promise

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

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

Async/Await

- Await: works only inside async functions

```
async function f() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });
  // wait till the promise resolves (*)
  const result = await promise;
  console.log(result); // "done!"
}

f();
```

Async/Await

- **Can’t use** await **in regular functions**

```
function f() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });
  // SyntaxError
  const result = await promise;
}

f();
```

Async/Await

- await **won’t work in the top-level code**

```
const response = await fetch(
  'https://api.github.com/search/repositories?q=angular’
);
const repos = await response.json();
```

Async/Await

- Async methods

```
class User {
  constructor(username) {
    this.username = username;
  }
  async getUser() {
    const response = await fetch(
    `https://api.github.com/search/users?q=${this.username}`
    );
    return await response.json();
  }
}

const u = new User('tiep');
u.getUser().then(res => console.log(res));
```

Async/Await

- Error handling

```
async function getUser(username) {
  try {
    const response = await fetch(
    `https://api.github.com/search/users?q=${username}`
    );
    return await response.json();
  } catch(e) {
    throw e;
  }
}
getUser('tiep’)
  .then(res => console.log(res))
  .catch(err => console.warn(err));
```

Async/Await

- Do not combine sync operations with async/await

```
let x = 0;
async function r5() {
  x += 1;
  console.log(x);
  return 5;
}

(async () => {
  x += await r5();
  console.log(x);
})();
```

Async/Await

- Do not combine sync operations with async/await

```
let x = 0;
async function r5() {
  x += 1;
  console.log(x);
  return 5;
}

(async () => {
  const y = await r5();
  x += y;
  console.log(x);
})();
```

Async/Await

- Too Sequential

```
async function getBooksAndAuthor(authorId) {
  const books = await fetchAllBook();
  const author = await fetchAuthorById(authorId);
  return {
    author,
    books: books.filter(book => book.authorId === authorId),
  };
}
```

Async/Await

- Too Sequential

```
async function getBooksAndAuthor(authorId) {
  const bookPromise = fetchAllBook();
  const authorPromise = fetchAuthorById(authorId);
  const books = await bookPromise;
  const author = await authorPromise;
  return {
    author,
    books: books.filter(book => book.authorId === authorId),
  };
}
```

DEMO

NASH TECH
The Power to Innovate

Q&A

References

- http://speakingjs.com/es5/index.html
- http://exploringjs.com/es6/index.html
- http://exploringjs.com/es2016-es2017/ch_async-functions.html
- http://exploringjs.com/es2018-es2019/toc.html
- http://eloquentjavascript.net/
- https://github.com/getify/You-Dont-Know-JS#titles

THANK YOU

www.nashtechglobal.com