Institute of Data

Advanced JavaScript

- Variable scope, **closure**
- Function object, **NFE**
- Scheduling: **setTimeout and setInterval**
- **Decorators** and forwarding, call/apply
- Function **binding**
- Property **getters and setters**
- Prototypal **inheritance**
- Function.prototype
- **Native** prototype
- Classes
- **Error** handling
- **Promises**, async/await

Variable scope, closure

Lexical Environment

In JavaScript, every running **function, code block** `{...}`, and the **script** as a whole have an **internal (hidden)** associated **object** known as the **Lexical Environment**.

The **Lexical Environment object** consists of **two** parts:

- **Environment Record**, an **object** that stores all **local variables** as its **properties** (and some other information like the **value** of `this`).
- A **reference** to the **outer lexical environment**, the one associated with the **outer code**.

Variable scope, closure

Lexical Environment

A **variable** is just a **property** of the special **internal object**, **Environment Record**. **To get or change a variable** means **to get or change a property of that object**.

```
var x = 10;
function foo(){ var y = 20;}
globalEnvironment = {
  environmentRecord: {
    x: 10
  },
  outer: null // no parent environment
};
fooEnvironment = {
  environmentRecord: {
    y: 20
  },
  outer: globalEnvironment
};
```

Vriable scope, closure

Closure

A closure is a function that **remembers** its **outer variables** and can **access** them.

Garbage collection

Usually, a **Lexical Environment** is **removed** from memory with all the variables **after the function call finishes**. It is because there are **no references** to it. As any JavaScript **object**, it is only **kept in memory** while it is **reachable**.

```
function f() {
  const value = 123;
  return function() {
    alert(value);
  }
}
const g = f(); // g.[[Environment]] stores a reference to the Lexical Environment of the corresponding f() call
```

Function object, NFE

In JavaScript, functions are objects(callable **"action objects"**). We can not only **call** them, but also treat them as **objects**, eg **add/remove** properties, **pass by reference** etc.

`name` property.

In the specification, this feature is called a **"contextual name"**. If the function does not provide one, then in an assignment it is figured out from the **context**.

```
function sayHi() { alert("Hi"); }
alert(sayHi.name); // sayHi

const sayHi = function() { alert("Hi"); };
alert(sayHi.name); // sayHi
```

Function object, NFE

`length` property.

It returns the **number** of function **parameters**, the **rest parameters** `...` are not counted.

```
function f1(a) {}
function f2(a, b) {}

function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2

alert(many.length); // 2
```

Function object, NFE

Custom property.

We can also add properties of our own.

```
function sayHi() {
  alert("Hi");
  // let's count how many times we run
  sayHi.counter++;
}
sayHi.counter = 0; // initial value

sayHi(); // Hi
sayHi(); // Hi

alert( `Called ${sayHi.counter} times` ); // Called 2 times
```

Function object, NFE

Named Function Expression

Named Function Expression, or **NFE**, is a term for Function **Expressions** that have a name.

- An **ordinary** Function Expression.

```
const sayHi = function(who) {
  alert(`Hello, ${who}`);
}
```

- A Function Expression **with a name**.

```
const sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

Function object, NFE

NFE Use Case

It allows the function to **reference itself internally**, it is **not visible outside** of the function.

- Non-working example **without NFE**.

```
const sayHi = function(who) {
 if (who) {
   alert(`Hello, ${who}`);
 } else {
   sayHi("Guest"); // Error: sayHi is not a function
 }
};
const welcome = sayHi;
sayHi = null;
welcome(); // Error, the nested sayHi call doesn't work any more!
```

Function object, NFE

NFE Use Case

- Working example **with NFE**.

```
const sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // Now all fine
  }
};

const welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (nested call works)
```

Function object, NFE

The "new Function" syntax

The function is created with the **arguments** `arg1...argN` and the given `functionBody`.

```
const func = new Function ([arg1, arg2, ...argN], functionBody);

//example
const sum = new Function('a', 'b', 'return a + b');
alert( sum(1, 2) ); // 3
```

`new Function` allows to turn any **string into a function**.

We can receive a **new function** from a **server** and then **execute** it.

```
const str = ... receive the code from a server dynamically ...

const func = new Function(str);
func();
```

Function object, NFE

The "new Function" closure

Usually, a function remembers where it was born in the special property `[[Environment]]`. It references the `Lexical Environment` from where it is **created**.

But when a function is created using **new Function**, its `[[Environment]]` is set to **reference not the current Lexical Environment**, but the **global one**.

```
function getFunc() {
  const value = "test";
  const func = new Function('alert(value)');
  return func;
}
getFunc()(); // error: value is not defined
```

Scheduling

`setTimeout`

It allows us to run a function once **after the interval of time**.

- Syntax

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

- `func|code` **Function** or a **string** of code to execute.
- `delay`: the **delay** before run, in **milliseconds**, by default **0**.
- `arg1, arg2…`: **arguments** for the function.

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Scheduling

`clearTimeout`

It allows us to cancel a `timeId` (**time identifier**), returned by the call of `setTimeout`.

- Syntax

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

- In a **browser**, the timer identifier is a **number**. In **Node.js**, It returns a **timer object** with additional methods.

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier
clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

Scheduling

`setInterval`

- The `setInterval` method has the same syntax as `setTimeout`

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

- It is unlike **setTimeout**, which runs the function only once, it runs the function **regularly** after the given **interval of time**.

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Scheduling

Nested setTimeout

There are two ways of running something **regularly**. **One** is `setInterval`. The **other** one is a **nested setTimeout**.

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

Scheduling

Zero delay setTimeout

There is a special use case: `setTimeout(func, 0)`, or just `setTimeout(func)`.

This schedules the **execution** of `func` as soon as possible. But the scheduler will invoke it only **after** the currently executing script is **complete**.

In the browser, there is a **limitation** of how often nested timers can run. The HTML5 standard claims: "after **five nested timers**, the interval is forced to be at least **4 milliseconds**.".

Decorators and forwarding, call/apply

Decoractors

Decorator is a **wrapper** around a **function** that **alters** its behavior. The main job is still **carried** out by the **function**. It can be seen as **"features"** or **"aspects"** that can be added to a function. We can add one or add many **without changing** its code!

Use case: Transparent caching

If the function(**a pure function**) is called **often**, we may want to **cache** (remember) the results to **avoid** spending **extra-time on recalculations**.

Decorators and forwarding, call/apply

Decoractors

Use case: Transparent caching

```
function slow(x) {
  // there can be a heavy CPU-intensive job here
  return x;
}
function cachingDecorator(func) {
  const cache = new Map();
  return function(x) {
    if (cache.has(x)) {    // if there's such key in cache
      return cache.get(x); // read the result from it
    }
    let result = func.call(this, x);
    cache.set(x, result);  // and cache (remember) the result
    return result;
  };
}
slow = cachingDecorator(slow);
alert( slow(1) ); // slow(1) is cached and the result returned
alert( "Again: " + slow(1) ); // slow(1) result returned from cache
alert( slow(2) ); // slow(2) is cached and the result returned
alert( "Again: " + slow(2) ); // slow(2) result returned from cache
```

Decorators and forwarding, call/apply

`func.apply` and `func.call`

- Syntax

```
func.call(context, ...args);
func.apply(context, args);
```

- They are used to bind the context.
  - The spread syntax `...` allows to pass **iterable args** as the list to call.
  - The `apply` accepts only **array-like** args.
- For objects that are both **iterable and array-like**, such as a **real array**, we can use any of them, but `apply` will probably be faster, because JS **engine optimizes** it.

Decorators and forwarding, call/apply

call forwarding

It is a way to **pass all arguments** along with the **context** to another function

```
let wrapper = function(func) {
  return function() {
    return func.apply(this, arguments);
  }
};

//let wrapper = (func) => (...args) => func.apply(this, args);  // with arrow function

const sum = (a, b) => a + b;

const sumWrapper = wrapper(sum);

sumWrapper(1, 2); //3
```

Decorators and forwarding, call/apply

Borrowing a method

In a normal function, `arguments` object is both **iterable and array-like**, but **not** a real array.

```
function hash() {
  alert( arguments.join() ); // Error: arguments.join is not a function
}
hash(1, 2);
```

After borrowing a **method** from an **Array**.

```
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}
hash(1, 2);
```

Function binding

Issue of losing `this`

Sometimes, the function is a **method** from an **object**, the **"context"** (`this`) is lost.

```
cosnt user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

Function binding

Solution 1 for missing `this`.

we can **wrap** it **inside** a function.

```
const user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);

//Or arrow function
//setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Function binding

Solution 2 for missing `this`.

we can `bind` the **"context"** into the **function**.

- Basic syntax

```
const boundFunc = func.bind(context);
```

- The fix with `bind`

```
const user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};
const sayHi = user.sayHi.bind(user);
setTimeout(sayHi, 1000); // Hello, John!
```

Function binding

Partial functions

- Full syntax of `bind`

```
const bound = func.bind(context, [arg1], [arg2], ...);
```

- It does not only bind `this`, but also `arguments`. It allows to **bind context** as `this` and **starting arguments** of the function.

```
function mul(a, b) {
  return a * b;
}
const triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
```

Function binding

Partial functions

- Going partial **without context**

```
function partial(func, ...argsBound) {
  return function(...args) {
    return func.call(this, ...argsBound, ...args);
  }
}
const user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}!`);
  }
};

user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello"); // [10:00] John: Hello!
```

Property getters and setters

Two kinds of object properties:

- The **first** kind is **data properties**. We already know how to work with them. All properties that we have been using until now were data properties.
- The **second** kind is **accessor properties**. They are essentially functions that execute on **getting** and **setting** a value, but **look like regular properties** to an external code.

Property getters and setters

`get` and `set` methods

- **Accessor properties** are represented by **getter** and **setter** methods. In an object literal they are **denoted** by `get` and `set`.

```
// an object literal
const foo = { bar: 1 };

//its getter and setter
const foo = {
 get bar() {
    return this._bar || 1;
 },
 set bar(value) {
  this._bar = value;
  }
}
```

Property getters and setters

Property descriptors

- The **shared keys** for **both** of **data descriptor** and **accessor descriptor**.
  - `configurable`: It is `true` if the type of this property **descriptor** may be **changed** and if the property may be **deleted** from the corresponding object.
  - `enumerable`: It is `true` if and **only** if this property shows up during **enumeration** of the properties on the corresponding object.
- Optional Keys for **Data descriptor**
  - `value`: The **value** associated with the **property**, which can be any valid JavaScript value (number, object, function, etc)
  - `writable`: It is **true** if the **value** associated with the property may be **changed** with an **assignment operator**

Property getters and setters

Property descriptors

- Optional Keys for **Accessor descriptor**
  - `get`: A **function** which serves as a **getter** for the **property**, or **undefined** if there is **no** getter. When the the **property** or **inherited property** is **accessed**, this function is called **without arguments**, the **return value** will be used as **the value of the property**.
  - `set`: A **function** which serves as a **setter** for the **property**, or **undefined** if there is **no** setter. When the property is **assigned**, this function is **called** with **one argument**.
- If a descriptor has **neither** `value`, `writable`, **nor** `get`, `set` keys, it is treated as a **data descriptor**. If a descriptor has **both** of `[value or writable]` **and** `[get or set]` keys, It will throw an **error**.

Property getters and setters

Use cases

- **Accessor Descriptor** in **Function constructor**

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  Object.defineProperty(this, "age", {
     get() {
        return new Date().getFullYear() - this.birthday.getFullYear();
     }
  });
}
const john = new User("John", new Date(1992, 6, 1));

john.age      // 29
```

Property getters and setters

Use cases

- **Accessor Property** in **ES6 class**

```
class User {
   constructor(name, birthday) {
     this.name = name;
     this.birthday = birthday;
   }
   get age() {
      return new Date().getFullYear() - this.birthday.getFullYear();
   }
}
const john = new User("John", new Date(1992, 6, 1));

john.age //29
```

Prototypal inheritance

`[[Prototype]]` property

In JavaScript, objects have a **special hidden** property `[[Prototype]]` (as named in the specification), that is either `null` or **references another object**. That object is called **"a prototype"**.

- `__proto__` is used to set `[[Prototype]]` for object

```
const animal = {
  eats: true
};
const rabbit = {
  jumps: true
};
rabbit.__proto__ = animal; // sets rabbit.[[Prototype]] = animal
```

Prototypal inheritance

- `for..in` loop for **inherited properties**.

```
const animal = {
  eats: true
};

const rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys only returns own keys
alert(Object.keys(rabbit)); // jumps

// for..in loops over both own and inherited keys
for(let prop in rabbit) alert(prop); // jumps, then eats
```

F.prototype

`F.prototype` here means a regular **property** named `prototype` on `F`, which is a function with **first capitalized letter** as its name **conventionally**. If `F.prototype` is an **object**, then the `new` operator uses it to set `[[Prototype]]` for the **new object**.

```
const animal = {
 eats: true
};
function Rabbit(name) {
 this.name = name;
}

Rabbit.prototype = animal;
const rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal
alert( rabbit.eats ); // true
```

F.prototype

Default F.prototype, constructor property

The **default** `prototype` is an object with the **only property** **constructor** that points back to the **function itself**.

```
function Rabbit() {}

/* default prototype
  Rabbit.prototype = { constructor: Rabbit };
*/

const rabbit = new Rabbit(); // inherits from {constructor: Rabbit}

alert(rabbit.constructor === Rabbit); // true (from prototype)
```

Native prototypes

The `prototype` **property** is widely used by the core of JavaScript itself. All **built-in constructor functions** use it.

- `Object.prototype`

```
const obj = {};
obj.__proto__ === Object.prototype; // true
obj.toString === obj.__proto__.toString; //true
obj.toString === Object.prototype.toString; //true
```

- Other **built-in** prototypes: `Array`, `Date`, `Function` and **etc** also keep **methods** in **prototypes**.

```
const arr = [1, 2, 3];
// it inherits from Array.prototype?
arr.__proto__ === Array.prototype; // true
```

Native prototypes

Changing native prototypes

Native prototypes can be **modified**. **BUT**, **BUT**, **BUT** It is **not** recommended to do so unless it is **polyfilling**.

```
String.prototype.show = function() {
  alert(this);
};
"BOOM!".show(); // BOOM!

// polyfilling for String.prototype.string
if (!String.prototype.repeat) { // if there's no such method
  String.prototype.repeat = function(n) {
    return new Array(n + 1).join(this);
  };
}
alert( "La".repeat(3) ); // LaLaLa
```

Native prototypes

Borrowing from prototypes

Some methods of native prototypes are often **borrowed**.

```
const obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};
obj.join = Array.prototype.join;
// or make it look like more native
Object.defineProperty(obj.__proto__, 'join', {
  value: function join() {
    return Array.prototype.join.apply(obj, arguments)
  }
});
obj.join(','); // Hello,world!
```

Prototype methods

Objects without `__proto__`

The `__proto__` is considered outdated and somewhat **deprecated**, the modern methods are:

- `Object.create(proto, [descriptors])`: creates an **empty** object with given **proto** as `[[Prototype]]` and optional **property descriptors**.

```
const animal = {
  eats: true
};
const rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});
rabbit.jumps; // true
```

Prototype methods

Objects without `__proto__`

- `Object.getPrototypeOf(obj)`: returns the `[[Prototype]]` of **obj**.
- `Object.setPrototypeOf(obj, proto)`: sets the `[[Prototype]]` of **obj** to **proto**.

```
const animal = {
  eats: true
};

// create a new object with animal as a prototype
const rabbit = Object.create(animal);
rabbit.eats; // true

Object.getPrototypeOf(rabbit) === animal; // true
Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}
```

Prototype methods

Objects without `__proto__`

- **Shallow copy** object, we can use `Object.create` to perform **an object cloning** more powerful than copying properties in `for..in`.

```
const clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

- The above call makes a truly exact copy of obj, including **all properties**: **enumerable** and **non-enumerable**, **data, accessor** properties and everything, and with the right `[[Prototype]]`.

Classes

In contrast with the old fashion `new Function`. In the modern JavaScript, there is a more advanced **class** construct, that introduces great new features which are useful for **object-oriented programming**.

- Syntax

```
class MyClass {
  prop = value; // property
  constructor(...) { // constructor
    // ...
  }
  method1(...) {} // method1
  method2(...) {} // method2
  get someProp() {} // getter method
  set someProp(value) {} // setter method
}
```

Classes

Inheritance

- Class **inheritance** is a way for one class to **extend** another class via `extends` keyword.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }
}
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}
const rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

Classes

Inheritance

- Overriding a **method**

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }
}
class Rabbit extends Animal {
  stop() {
    super.stop(); // call parent stop
    this.hide(); // and then hide
  }
}
const rabbit = new Rabbit("White Rabbit");
rabbit.stop(); // White Rabbit stands still. White Rabbit hides!
```

Classes

Inheritance

- Overriding a **constructor**

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
}
class Rabbit extends Animal {
  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }
}

const rabbit = new Rabbit("White Rabbit", 10);

alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

Classes

Inheritance

- Overriding **class fields**

```
class Animal {
  name = 'animal';
  showName() { console.log(this.name);}
}

class Rabbit extends Animal {
  name = 'rabbit';
}

new Animal().showName(); // animal
new Rabbit().showName(); // 'rabbit'
```

Classes

Static methods

We can also assign a **method** to the **class function** itself, **prepended** by `static` keyword, which is used to implement functions that belong to the **class**, but **not** to any particular **object** of it.

```
class User {
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // true
```

Classes

Static properties

They look like **regular class properties**, but prepended by `static`.

```
class Article {
  static publisher = "Ilya Kantor";
}

Article.publisher  // Ilya Kantor

//the same as
//Article.publisher = "Ilya Kantor";
```

Classes

Inheritance of static properties and methods

Static properties and methods are **inherited**.

```
class User {
  static age = 40;
  static sayHi() {
    alert('Hi')
  }
}

class Admin extends User {}

Admin.name // 40
Admin.sayHi() // 'Hi'
```

Classes

Protected properties and methods

The protected properties are usually **prefixed** with an **underscore _**.

```
class CoffeeMachine {
  _waterAmount = 0;
  set waterAmount(value) {
    this._waterAmount = value;
  }
  get waterAmount() {
    return this._waterAmount;
  }
  constructor(power) {
    this._power = power;
  }
}
```

Classes

Private properties and methods

The **privates** should start with `#`. They are only accessible from **inside** the class.

```
 class CoffeeMachine {
  #waterLimit = 200;
  #fixWaterAmount(value) {
    return this.#waterLimit;
  }
  setWaterAmount(value) {
    this.#waterLimit = this.#fixWaterAmount(value);
  }
}
const coffeeMachine = new CoffeeMachine();
coffeeMachine.#fixWaterAmount(123); // Error
coffeeMachine.#waterLimit = 1000; // Error
```

Error Handling

`try…catch` syntax

- The **try...catch** construct has two main blocks: `try`, and then `catch`

```
try {
  // code...
} catch (err) {
  // error handling
}
```

- Only works for **runtime** errors, the code must be **runnable** as valid JavaScript. It won't work if the code is **syntactically wrong**.

```
try {
  {{{{{{{{{{{{
} catch (err) {
  alert("The engine can't understand this code, it's invalid");
}
```

Error Handling

`try…catch` syntax

- It works **synchronously**,if an exception happens in **"scheduled"** code, like in `setTimeout`, then `try...catch` won't catch it:

```
try {
  setTimeout(function() {
    noSuchVariable; // script will die here
  }, 1000);
} catch (err) {
  alert( "won't work" );
}
```

Error Handling

`throw` Operator

- **throw** error in `try` block

```
const json = '{ "age": 30 }'; // incomplete data

try {
  const user = JSON.parse(json); // <-- no errors

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }

  alert( user.name );
} catch (err) {
  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}
```

Error Handling

`throw` Operator

- **rethrow** error in `catch` block

```
const json = '{ "age": 30 }'; // incomplete data
try {
  const user = JSON.parse(json);
  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }
  blabla(); // unexpected error
} catch (err) {
  if (err instanceof SyntaxError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // rethrow other non-syntax errors
  }
}
```

Error Handling

`try...catch...finally`

The `finally` clause is often used when we start doing something and want to **finalize** it in any **case of outcome**.

- With all of them

```
try {
  alert( 'try' );

  if (confirm('Make an error?')) BAD_CODE();
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

Error Handling

try...catch...finally

- `finally` and `return`

```
function func() {
  try {
    return 1;
  } catch (err) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}

alert( func() ); // first works alert from finally, and then return 1
```

Error Handling

try...catch...finally

- try...finally

```
function func() {
  // start doing something that needs completion (like measurements)
  try {
    // ...
  } finally {
    // complete that thing even if all dies
  }
}
```

Promises, async/await

Promise

- Syntax

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

- There can be **only a single result** or **an error**

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

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

Promises, async/await

Consumers: `then, catch, finally`

- `then`

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

- `catch`

```
const 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
```

Promises, async/await

Consumers: `then, catch, finally`

- `finally`

```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve/reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
.finally(() => stop loading indicator)
// so the loading indicator is always stopped before we process the result/error
.then(result => show result, err => show error)
```

Promises, async/await

Unhandled rejections

It is a **browser-specific** feature that the **unhandled rejections** can be caught **globally**.

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
  alert(event.promise); // [object Promise] - the promise that generated the error
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

Promises, async/await

Promises chaining

- We have a **sequence of asynchronous tasks** to be performed **one after another**.

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  return result * 2;//2
}).then(function(result) {
  return result * 2;//4
}).then(function(result) {
  return result * 2; //8
});
```

Promises, async/await

Returning promises

- A handler, used in `.then(handler)` may create and return a **promise**.

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  alert(result); // 4
});
```

Promises, async/await

Promise static methods

- `Promise.all(promises)`: waits for all promises to resolve and returns an array of their results. If **any of the given promises rejects**, it becomes the **error of Promise.all**, and all other results are **ignored**.
- `Promise.allSettled(promises)`: waits for all promises to **settle** and returns their **results as an array of objects** with:
  - status: `"fulfilled"` or `"rejected"`
  - **value** (if fulfilled) or **reason** (if rejected).
- `Promise.race(promises)`: waits for the **first promise to settle**, and its result/error becomes the outcome.
- `Promise.any(promises)`: waits for the **first promise to fulfill**, and its result becomes the outcome. If all of the given promises are rejected, AggregateError becomes the error of `Promise.any`.
- `Promise.resolve(value)`: makes a **resolved promise with the given value**.
- `Promise.reject(error)`: makes a **rejected promise** with the given error.

Promises, async/await

Async function

The word **async** before a function means a **function** always returns **a promise**. Other values are wrapped in a **resolved promise automatically**.

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

// the same as
async function f() {
  return Promise.resolve(1);
}
```

Promises, async/await

`await` keyword

It makes JavaScript **wait** until that promise **settles** and returns its **result**.

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

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

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

Questions?