howest hogeschool

Web Technology Closures

Dieter Mourisse

Functions

Functions in JavaScript

There are a couple of ways to define a function in JavaScript

Function declaration

```
function sum_dec(a, b) {
   return a + b;
}

const result_dec = sum_dec(5, 2);
```

Function expression

```
const sum_expr = function (a, b) {
   return a + b;
}

const result_expr = sum_expr(5, 2);
```

Function expression with arrow notation

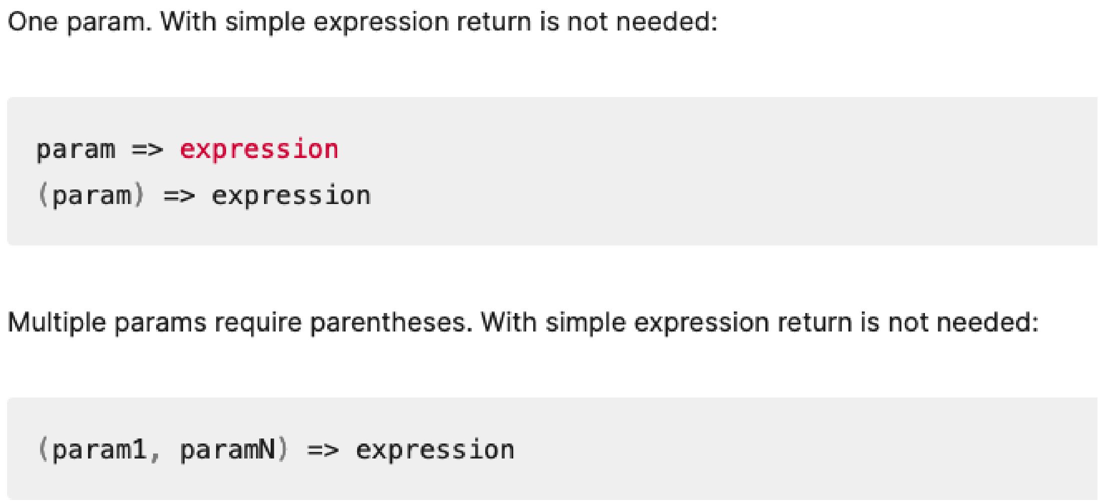
```
const sum_arrow = (a, b) => a + b;
const result_arrow = sum_arrow(5, 2);
```

not hoisted

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#defining_functions

Arrow notation

An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.



One param. With simple expression return is not needed:

```
param => expression
(param) => expression
```

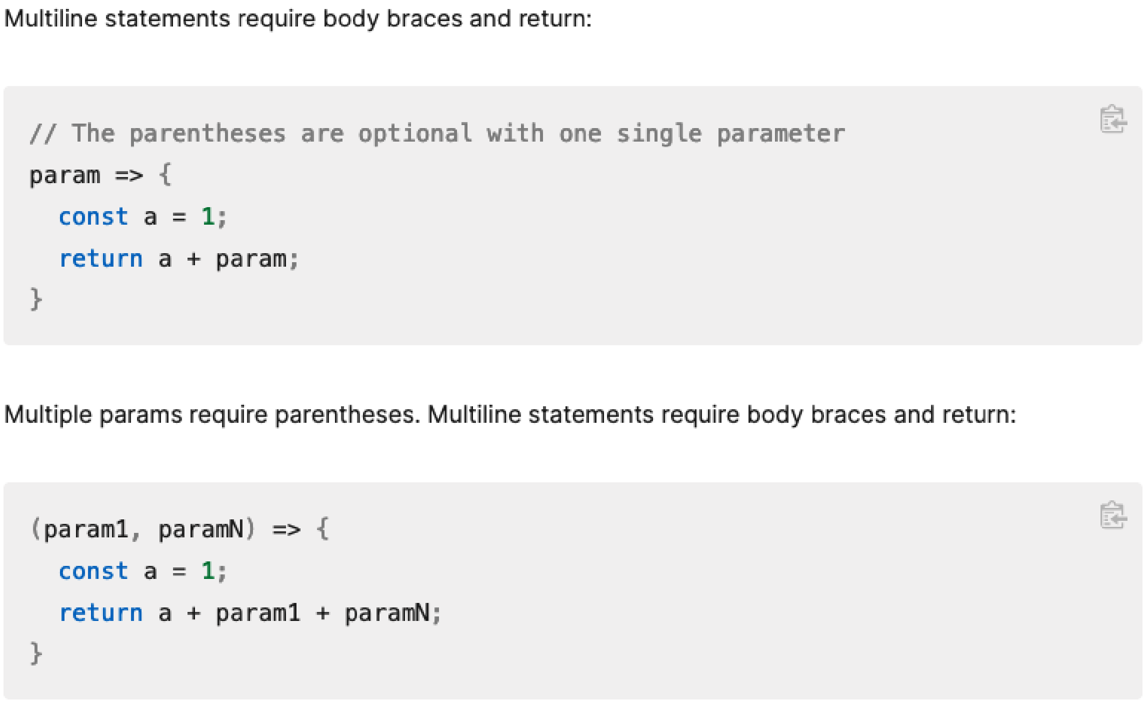
Multiple params require parentheses. With simple expression return is not needed:

```
(param1, paramN) => expression
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Arrow notation

An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.



Multiline statements require body braces and return:

```
// The parentheses are optional with one single parameter
param => {
  const a = 1;
  return a + param;
}
```

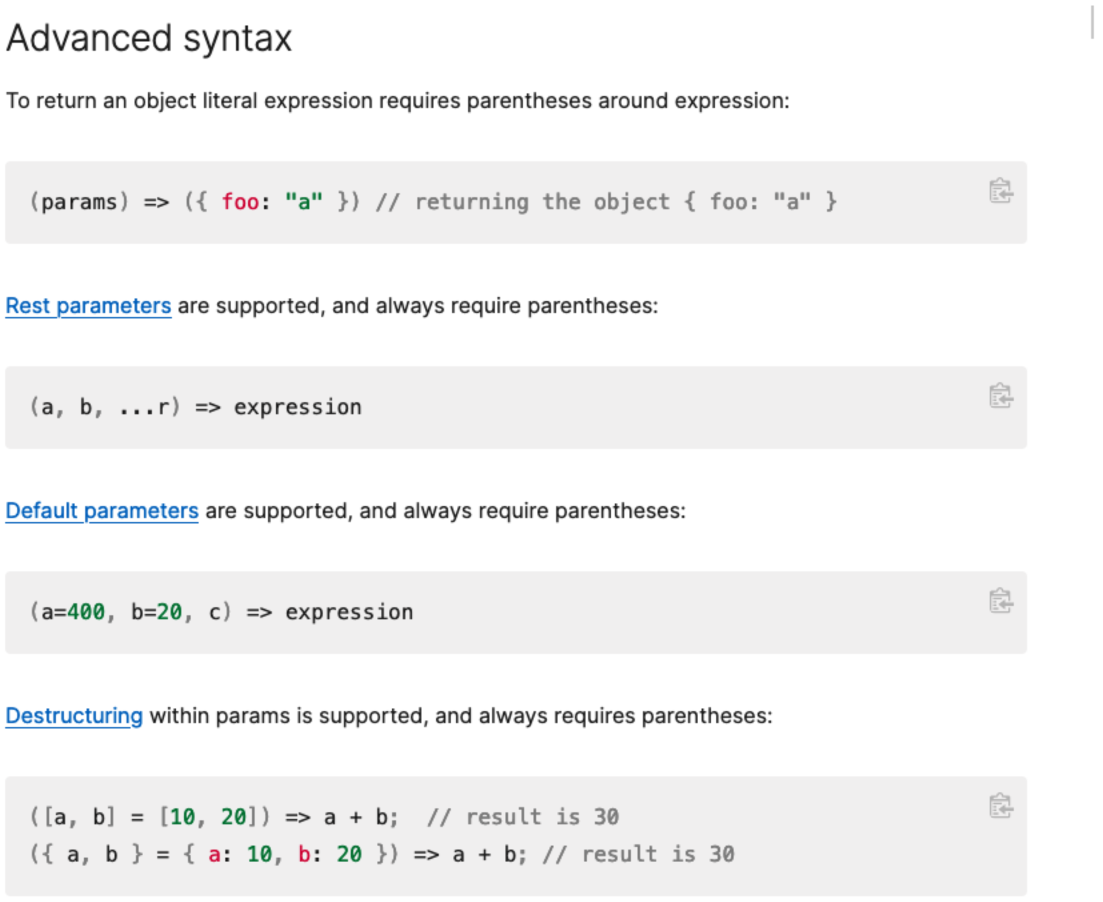
Multiple params require parentheses. Multiline statements require body braces and return:

```
(param1, paramN) => {
  const a = 1;
  return a + param1 + paramN;
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Arrow notation

An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.



Advanced syntax

To return an object literal expression requires parentheses around expression:

```
(params) => ({ foo: "a" }) // returning the object { foo: "a" }
```

Rest parameters are supported, and always require parentheses:

```
(a, b, ...r) => expression
```

Default parameters are supported, and always require parentheses:

```
(a=400, b=20, c) => expression
```

Destructuring within params is supported, and always requires parentheses:

```
([a, b] = [10, 20]) => a + b;  // result is 30
({ a, b } = { a: 10, b: 20 }) => a + b; // result is 30
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

IIFE (Immediately Invoked Function Expression)

An **IIFE** (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
const result = (function (a, b) { return a + b })(5, 3);

const result = ((a, b) => a + b)(5, 3);
```

They can be used to avoid polluting the global scope

https://developer.mozilla.org/en-US/docs/web/JavaScript/Reference/Operators/function#using_an_immediately_invoked_function_expression_iife

IIFE (Immediately Invoked Function Expression)

Because our application could include many functions and global variables from different source files, it's important to limit the number of global variables.

If we have some initiation code that we don't need to use again, we could use the IIFE pattern.

https://developer.mozilla.org/en-US/docs/web/JavaScript/Reference/Operators/function#using_an_immediately_invoked_function_expression_iife

Higher order functions

First-class Function

A programming language is said to have **First-class functions** when functions in that language are treated like any other variable.

For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.

https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function

Higher order Functions

A **higher-order function** (HOF) is a function that does at least one of the following:

- Takes one or more functions as arguments
- Returns a function as its result

Higher order function	
.forEach	The **forEach()** method executes a provided function once for each array element.
.map	The **map()** method **creates a new array** populated with the results of calling a provided function on every element in the calling array.
.filter	The **filter()** method creates a shallow copy of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function.
.reduce	The **reduce()** method executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element.
.addEventListener	The **addEventListener()** method of the EventTarget interface sets up a function that will be called whenever the specified event is delivered to the target.

Our own higher order functions (function arguments)

```
function combine(element, function1, function2) {
   return [function1(element), function2(element)];
}

const double = n => 2 * n;
const square = n => n * n;



const result_combine = combine(3, double, square);
```

Execute function1 and function2 on element and return both results in an array.

```
[ 6, 9 ]
```

```
function ourOwnMap(array, theFunction) {
    let result = []
    for (let element of array) {
        result.push(theFunction(array));
    }
    return result;
}
```

```
function ourOwnFilter(array, theFunction) {
        let result = [];
        for (let element of array) {
                if (theFunction(element)) result.push(element);
        }
}
```

```
function ourOwnForEach(array, theFunction) {
    for (let element of array) {
        theFunction(element);
    }
}
```

```
function ourOwnReduce(array, theFunction, initialValue) {
        let acc = initialValue;
        for (let element of array) {
                acc = theFunction(acc, element);
        }
        return acc;
}
```

Our own higher order functions (returning functions)

Suppose we want functions isDivisibleBy2, isDivisibleBy3, isDivisibleBy4, …

```
function isDivisibleBy2(number) {
    return number % 2 === 0;
}

function isDivisibleBy3(number) {
    return number % 3 === 0;
}

function isDivisibleBy4(number) {
    return number % 4 === 0;
}
```

```
function isDivisibleByN(number, n)
{
    return number % n === 0;
}
```

Our own higher order functions (returning functions)

```
const isDivisibleByN = function (number, n) { return number % n == 0 };
const arr = [3, 2, 10, 7, 15, 25, 9];
const result = arr.filter(isDivisibleByN);
```

← [2, 10, 25]

number	n	result
3	0	false
2	**1**	**true**
10	**2**	**true**
7	3	false
15	4	false
25	**5**	**true**
9	6	false

Our own higher order functions (returning functions)

```
const isDivisibleByN = function (number, n) { return number % n == 0 };
const arr = [3, 2, 10, 7, 15, 25, 9];
const result = arr.filter(element => isDivisibleByN(element, 2));
```

← [2, 10]

number	n	result
3	2	false
2	**2**	**true**
10	**2**	**true**
7	2	false
15	2	false
25	2	false
9	2	false

Our own higher order functions (returning functions)

```
const isDivisbleBy2 = function (element) {
   return isDivisibleByN(element, 2);
}

const result = arr.filter(isDivisibleBy2);
```

isDivisibleByTwo is a function with one parameter.
When calling it with one parater, it will call the function isDivisibleByN with the arguments element and two

Our own higher order functions (returning functions)

```
const divisibleChecker = function (divisor) {
        const divisibleByDivisor = function (number) {
                return number % divisor == 0;
        }
        return divisibleByDivisor;
}

const isDivisibleBy2 = divisibleChecker(2);
const isDivisibleBy3 = divisibleChecker(3);
const result = arr.filter(isDivisibleBy2);
```

divisibleByDivisor is a function with one parameter. It checks whether *number* is divisible by *divisor*.

divisibleChecker is a function with one parameter.

When calling it with one parameter, it will **return** the **function** divisibleByDivisor that has one argument.

Depending on the value for the *divisor* argument, for each value of *divisor* it will return a different function.

Our own higher order functions (returning functions)

```
const divisibleChecker = function (divisor) {
   return number => number % divisor == 0;
}

const result = arr.filter(divisibleChecker(2));
```

howest
hogeschool

Closures

Lexical scope

The word *lexical* refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

```
const safari = "Apple";

function init() {
    const firefox = "Mozilla";
    function displayName() {
        const chrome = "Google";
    }
    displayName();
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures#lexical_scoping

Variable shadowing

In computer programming, **variable shadowing** occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope.

```
const name = "Apple"
function init() {
        const name = "Mozilla";
        function displayName() {
                const name = "Google";
                console.log(name);            → Google
        }
        displayName();
        console.log(name);            → Mozilla
}

init();
console.log(name);            → Apple
```

Look at most inner scope first for **declaration** of variable.

Variable shadowing

In computer programming, **variable shadowing** occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope.

Look at most inner scope first for **declaration** of variable.

```
const name = "Apple"
function init() {
        let name = "Mozilla";
        function displayName() {
                name = "Google";
                console.log(name);          →  Google
        }
        displayName();
        console.log(name);          →  Google
}

init();
console.log(name);          →  Apple
```

Closure

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
const divisibleChecker = function (divisor) {
        const divisibleByDivisor = function (number) {
                return number % divisor == 0;
        }
        return divisibleByDivisor;
}

const isDivisibleBy2 = divisibleChecker(2);
```

Part of the outer scope of the function divisibleChecker

We still have access to that outer scope, where the value for *divisor* is 2

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

Example

Question

- We need a function that returns an integer
- Increments each time we call the function

```
let counter = 0;

function increment() {
   counter += 1;
   return counter;
}

console.log(increment());
```

The variable *counter* can be modified everywhere in our code, it is a global variable.

Question

- We need a function that returns an integer
- Increments each time we call the function

```
function increment() {
   let counter = 0;
   counter += 1;
   return counter;
}
```

This function will always return 1, a new variable *counter* is created every time the *increment* function is called.

Question

- We need a function that returns an integer
- Increments each time we call the function

```
function makeCounter() {
   let counter = 0;
   function increment() {
      counter += 1;
      return counter;
   }
   return increment;
}

const increment = makeCounter();
const increment2 = makeCounter();

console.log(increment());
console.log(increment());
console.log(increment2());
```

```
1
2
1
```

The function *makeCounter* will return an *increment* function, each function that is returned has their own *counter* variable that is incremented whenever the function is called.

The *makeCounter* function is however part of the global scope, even if we only need one counter.

Question

- We need a function that returns an integer
- Increments each time we call the function

```
function makeCounter() {
   let counter = 0;
   return () => counter += 1;
}
```

The function *makeCounter* will return an *increment* function, each function that is returned has their own *counter* variable that is incremented whenever the function is called.

```
const increment = makeCounter();
const increment2 = makeCounter();
```

The *makeCounter* function is however part of the global scope, even if we only need one counter.

```
console.log(increment());
console.log(increment());
console.log(increment2());
```

```
1
2
1
```

Question

- We need a function that returns an integer
- Increments each time we call the function

```
const increment = (function () {
   let counter = 0;
   return () => counter += 1;
})();

console.log(increment());
console.log(increment());
```

IIFE

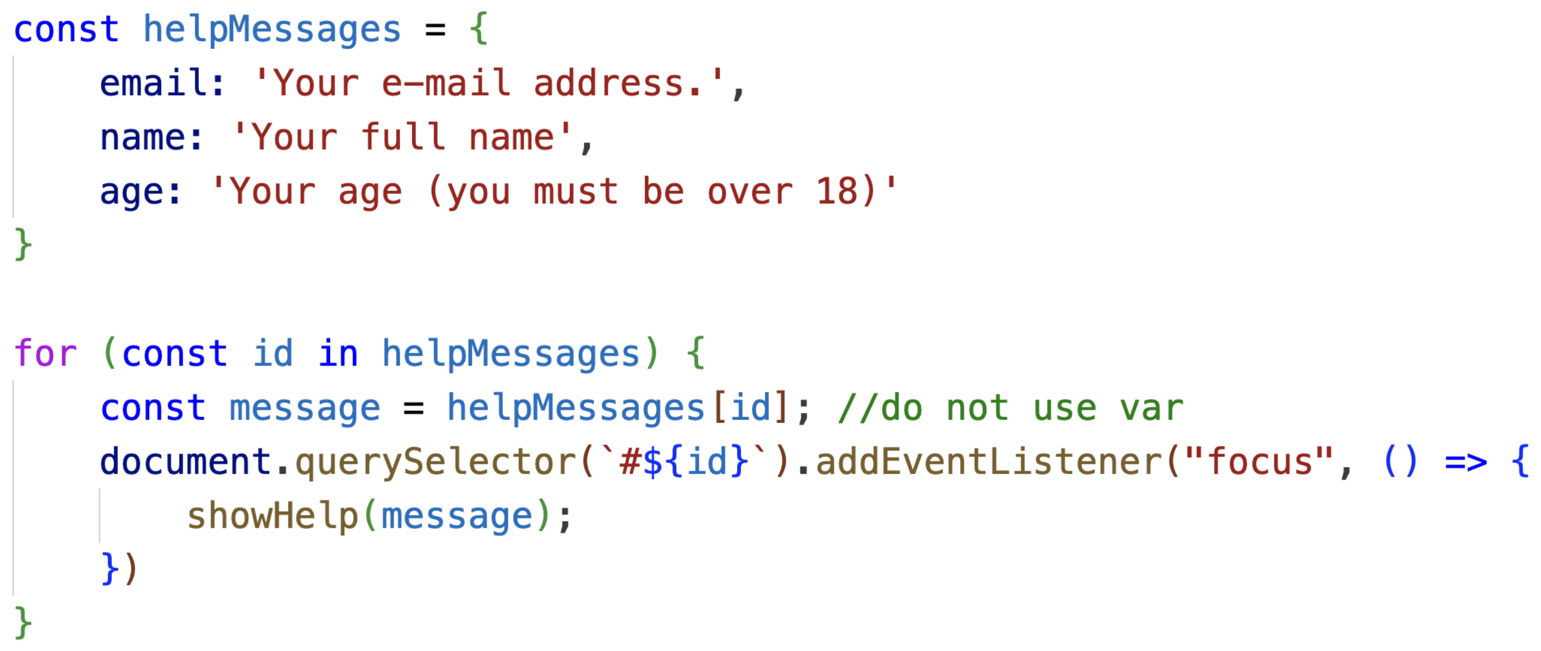
1

2

Increment and decrement

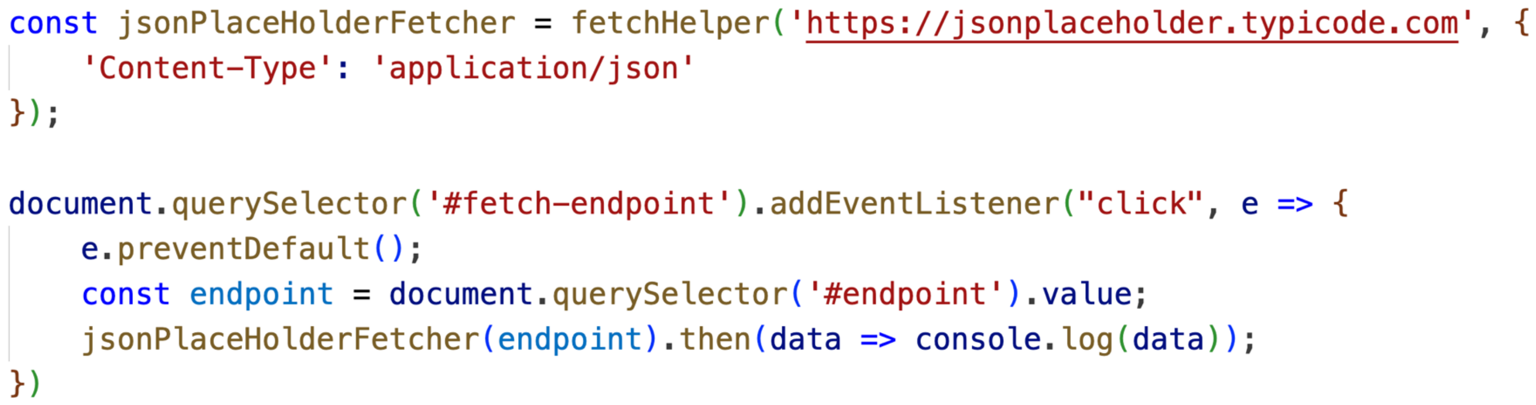
```
const { increment, decrement } = (function () {
   let counter = 0;
   const increment = () => counter += 1;
   const decrement = () => counter -= 1;

   return { increment: increment, decrement: decrement };
})();

console.log(increment());
console.log(increment());
console.log(decrement());
console.log(increment());
```

```
1
2
1
2
```



```
const helpMessages = {
    email: 'Your e-mail address.',
    name: 'Your full name',
    age: 'Your age (you must be over 18)'
}

for (const id in helpMessages) {
    const message = helpMessages[id]; //do not use var
    document.querySelector(`#${id}`).addEventListener("focus", () => {
        showHelp(message);
    })
}
```

```
let history = {};
return async function (endpoint, method = 'GET') {
    if (endpoint in history) {
        return history[endpoint];
    }

    return fetch(`${baseURL}/${endpoint}`, {
        method: method,
        headers: headers,
    })
        .then(response => response.json())
        .then(data => {
            history[endpoint] = data;
            return data;
    });
}
```



```
const jsonPlaceHolderFetcher = fetchHelper('https://jsonplaceholder.typicode.com', {
    'Content-Type': 'application/json'
});

document.querySelector('#fetch-endpoint').addEventListener("click", e => {
    e.preventDefault();
    const endpoint = document.querySelector('#endpoint').value;
    jsonPlaceHolderFetcher(endpoint).then(data => console.log(data));
})
```

Web Technology
Objected Oriented Programming

Dieter Mourisse

The this keyword

The this keyword

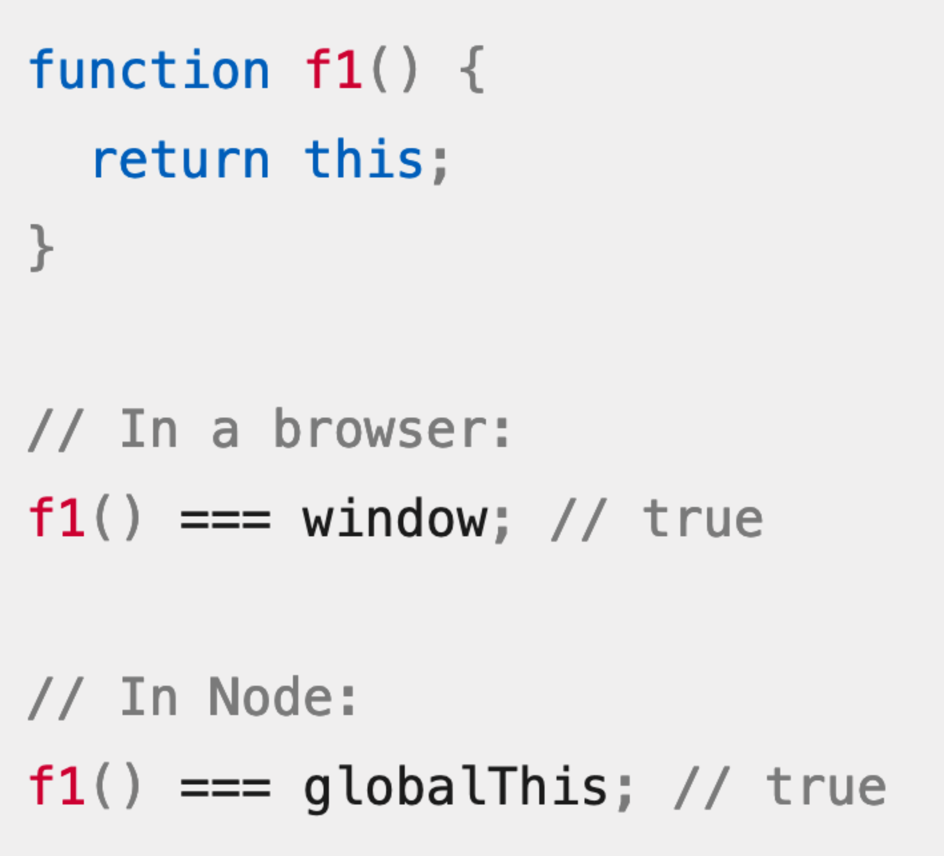
The *this* keyword references the object that is executing the current function.

When it is a method function inside an object, *this* references the object.

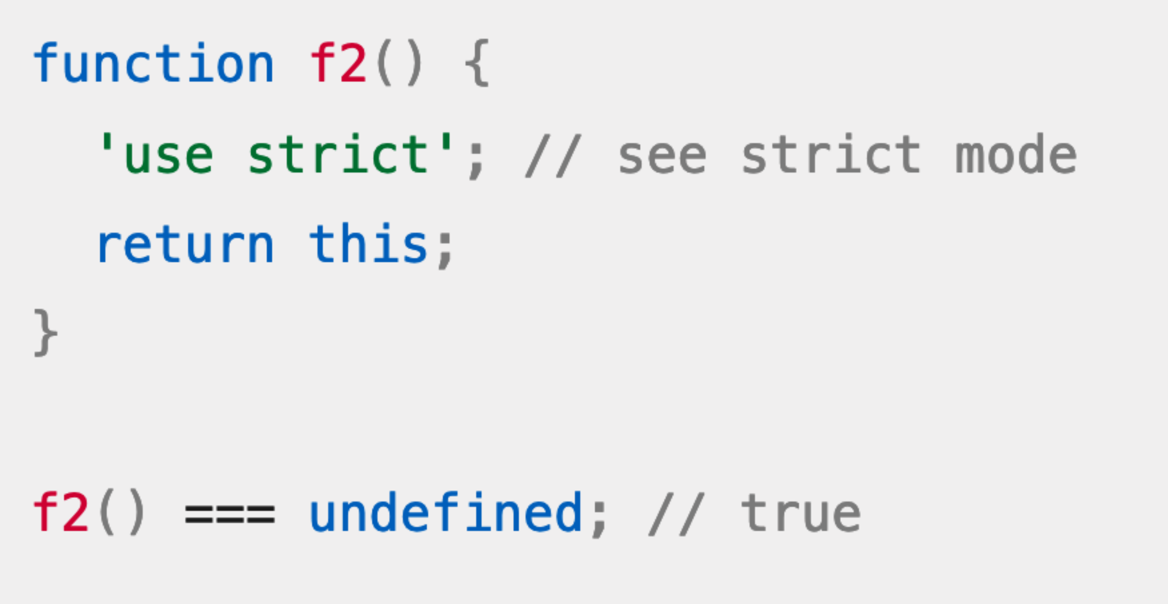
When it is a function in the global space, *this* references the window object in the browser or the global object in cjs node. When in strict mode or esm node *this* is undefined.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

Function context



```
function f1() {
  return this;
}

// In a browser:
f1() === window; // true

// In Node:
f1() === globalThis; // true
```



```
function f2() {
  'use strict'; // see strict mode
  return this;
}

f2() === undefined; // true
```

Node.js top-level

In the top-level code in a Node cjs module, *this* is equivalent to *module.exports*.

```
console.log(this === module.exports);
```

In the top-level code in a Node es6 module, *this* is undefined.

```
console.log(this === undefined);
```

Object Context

```
const alice = {
   name: ”Alice",
   objFunction: function() { return this }
}

console.log(alice.objFunction() === alice); //true
```

The function *objFunction* is called on the *alice* object, so *this* will be equal to the *alice* object.

Arrow function

In arrow functions, this retains the value of the enclosing lexical context's this. In global code, it will be set to the this of the global object:

```
const bob = {
   name: ”Bob",
   objFunction: () => this
}

console.log(bob.objFunction() === module.exports); //true (common js)
```

Arrow function

In arrow functions, this retains the value of the enclosing lexical context's this. In global code, it will be set to the global object:

```
function myFunction() {
   const bob = {
      name: "Bob",
      objFunction: () => this
   }
   return bob.objFunction();
}

console.log(myFunction() === globalThis); //true
```

Arrow function

In arrow functions, this retains the value of the enclosing lexical context's this. In global code, it will be set to the global object:

```
carol = {
   objFunction: function() {
      f = () => this;
      return f();
   }
}

console.log(carol.objFunction() === carol); // true
```

Example

```
"use strict";

const colours = ["red", "green", "blue"];

document.querySelector("#element").addEventListener("click", function(e) {

    console.log(this === e.currentTarget);
    const that = this;

    colours.forEach(function() {
        console.log(this === undefined) // true
        console.log(that === document.querySelector("#element")); // true
    })
})
```

Example

```
"use strict";

const colours = ["red", "green", "blue"];

document.querySelector("#element").addEventListener("click", function(e) {

    console.log(this === e.currentTarget); // true
    const that = this;

    colours.forEach(function() {
        console.log(this === document.querySelector("#element")); // true
    }, this)
})
```

forEach has an optional second parameter, the value to be use as *this* when executing the callback function.

Example

```
"use strict";

const colours = ["red", "green", "blue"];

document.querySelector("#element").addEventListener("click", function(e) {
    colours.forEach(() => {
        console.log(this === document.querySelector("#element")) // true
    });
})
```

When using arrow function, *this* is taken from the enclosing lexical scope

When provided, second argument of *forEach* will be ignored

Favour *e.currentTarget* over *this*

Call/apply/bind

Call

The **call()** method calls the function with a given this value and arguments provided individually.

```
const person = {
   name: "Mattias De Wael"
}

function greeting(...titles) {
   const titlestring = titles.join(" ");
   return `Hello ${titlestring} ${this.name}`;
}

let result = greeting("dr.", "ir."); // Hello dr. ir. undefined

result = greeting.call(person, "dr.", "ir."); // Hello dr. ir. Mattias De Wael
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call

Apply

The **apply()** method calls the specified function with a given this value, and arguments provided as an array (or an array-like object).

```
const person = {
   name: "Mattias De Wael"
}

function greeting(...titles) {
   const titlestring = titles.join(" ");
   return `Hello ${titlestring} ${this.name}`;
}

const result = greeting.apply(person, ["dr.", "ir."]) // Hello dr. ir. Mattias De Wael
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

Bind

The **bind()** method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

```
const person = {
    name: "Mattias De Wael"
}

function greeting(...titles) {
    const titlestring = titles.join(" ");
    return `Hello ${titlestring} ${this.name}`;
}

greetingForPerson = greeting.bind(person);

result = greetingForPerson("dr.", "ir."); // Hello dr. ir. Mattias De Wael
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

Objects (pre ES6)

Object without methods

```
const alice = {
   name: "Alice",
   surname: "Anderson",
   courses: ["Programming Fundamentals", "Databases"]
}

const bob = {
   name: "Bob",
   surname: "Birch",
   courses: ["Web Technology", "Information Modelling"]
}
```

Adding methods

```
const alice = {
    name: "Alice",
    surname: "Anderson",
    courses: ["Programming Fundamentals", "Databases"],

    fullName: function() {
        return `${this.name} ${this.surname}`
    }
}

alice.fullName();
```

Method name

Properties of the object

You can’t use arrow functions

Adding methods

```
const myCounter = {
    counter: 0,

    increase: function() {
        this.counter += 1;
        return this.counter;
    }
}

myCounter.increase();
```

Method name

Properties of the object

You can’t use arrow functions

This is tedious work when you are working with multiple objects of the same “class”, for instance multiple persons that have a firstName, lastName and a fullName method.

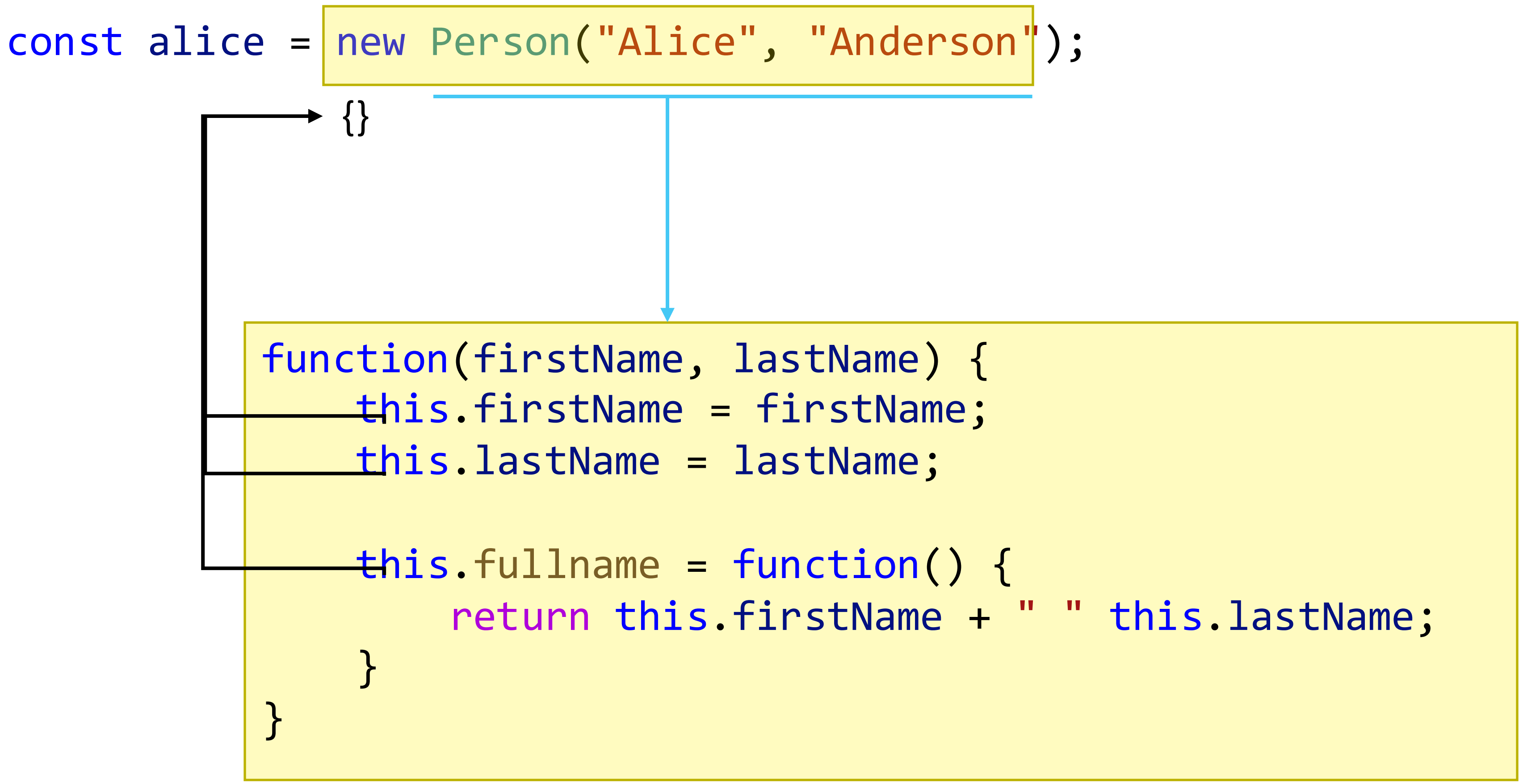
Constructor function

```
'use strict'


const Person = function(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.fullname = function() {
        return this.firstName + " " + this.lastName;
    }
}

const alice = new Person("Alice", "Anderson");
let bob = new Person("Bob", "Birch");
```

This is a constructor function. It adds fields and functions to the this object.

The new keyword will create and return a new empty object and execute the function on that object.

Constructor function

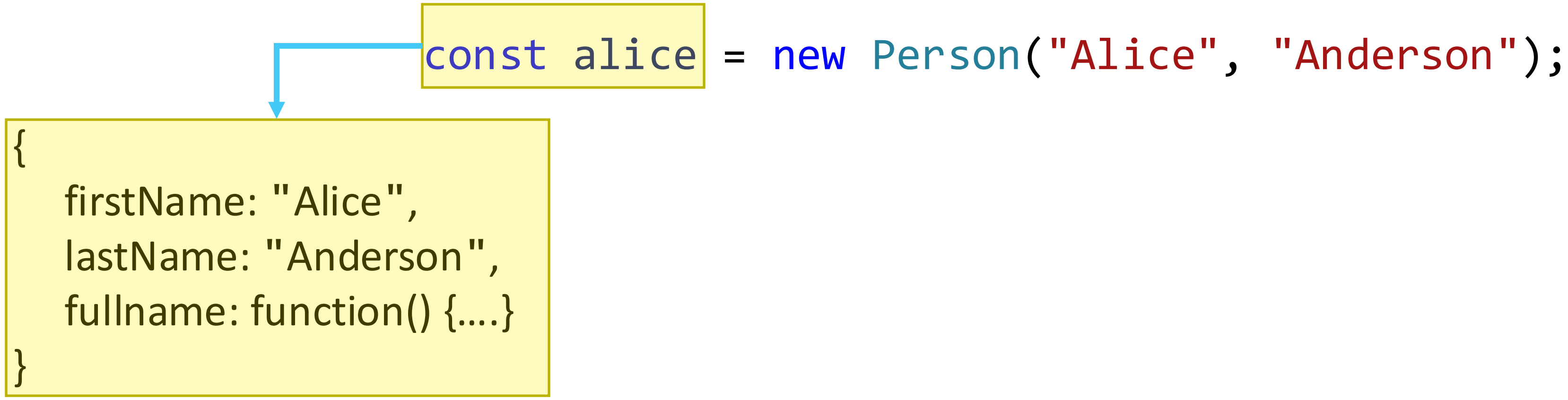


```
const alice = new Person("Alice", "Anderson");
{}
```

```
function(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.fullname = function() {
        return this.firstName + " " this.lastName;
    }
}
```

The new keyword will create and return a new empty object and execute the function in that object.

Constructor function



```
const alice = new Person("Alice", "Anderson");
```

```
{
    firstName: "Alice",
    lastName: "Anderson",
    fullname: function() {….}
}
```

The new keyword will create and return a new empty object and execute the function in that object.

```
function(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.fullname = function() {
        return this.firstName + " " + this.lastName;
    }
}
```

Constructor function

```
'use strict'

const Person = function(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.fullname = function() {
        return this.firstName + " " + this.lastName;
    }
}

let alice = new Person("Alice", "Anderson");
let bob = new Person("Bob", "Birch");
```

Problem: for every created object there is a copy of the implementation of fullname.

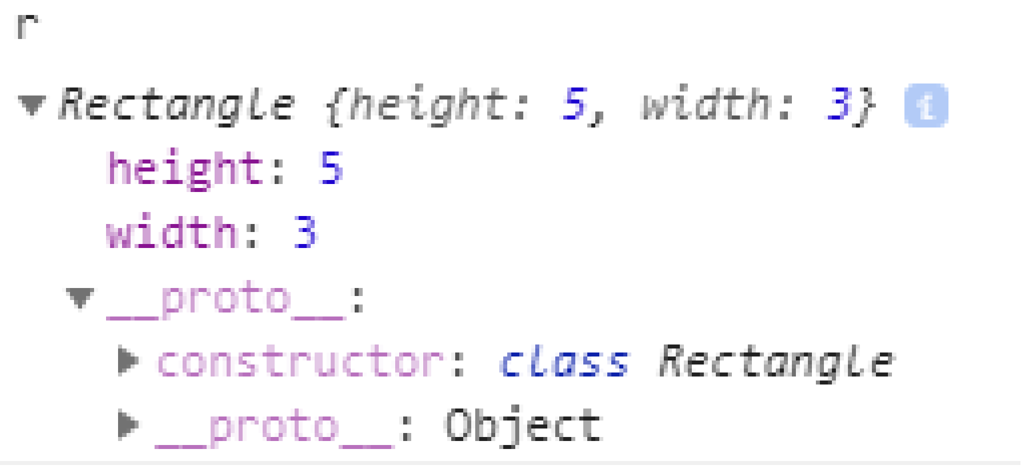
ES6 Classes

ECMAScript 6 gives us extra syntax so that we can write classes in a way similar to Java.

Defining a class

```
class Rectangle {

    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

const r = new Rectangle(5, 3);

console.log(r);
```

The method with name constructor gets called when creating a new object of that class.

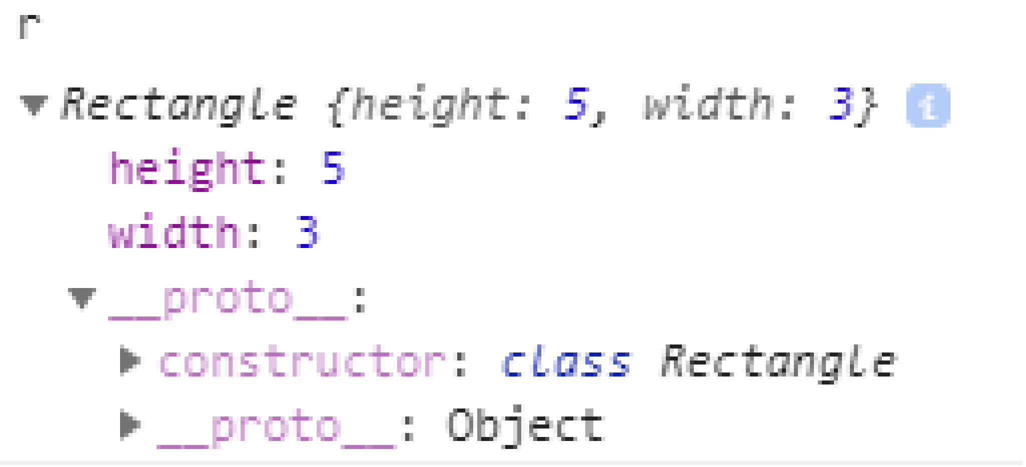


```
r
▼Rectangle {height: 5, width: 3}
    height: 5
    width: 3
  ▼__proto__:
    ▶constructor: class Rectangle
    ▶__proto__: Object
```

Defining a class

```
class Rectangle {

    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

const r = new Rectangle(5, 3);

console.log(r);
```

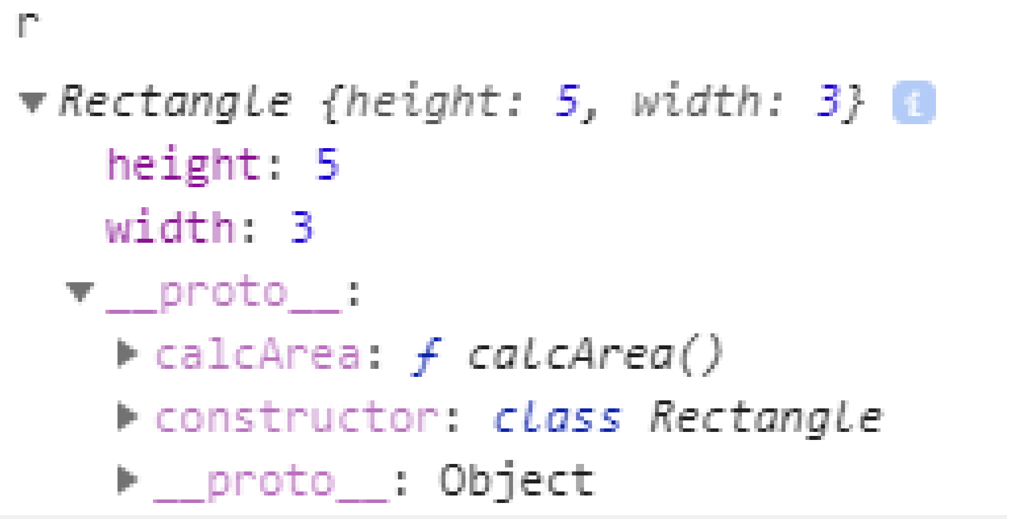
Add two (public) fields to the object



```
r

▼Rectangle {height: 5, width: 3}
    height: 5
    width: 3
  ▼__proto__:
    ▶constructor: class Rectangle
    ▶__proto__: Object
```

Adding a method to a class

```
class Rectangle {

    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    calcArea()
    {
        return this.height * this.width;
    }
}

const r = new Rectangle(5, 3);
console.log(r.calcArea());
```

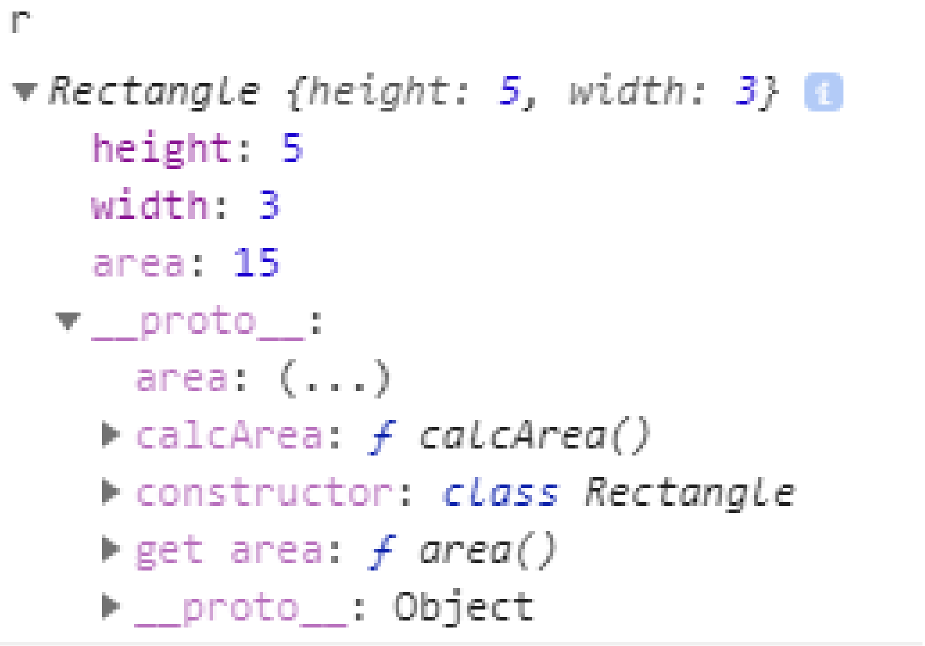


```
r
▼ Rectangle {height: 5, width: 3} ℹ
    height: 5
    width: 3
  ▼ __proto__:
    ▶ calcArea: ƒ calcArea()
    ▶ constructor: class Rectangle
    ▶ __proto__: Object
```

Writing a getter method

```
class Rectangle {

    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    get area() {
        return this.calcArea();
    }

    calcArea()
    {
        return this.height * this.width;
    }
}

const r = new Rectangle(5, 3);
console.log(r.area);
```

You can write a get method. This get method can then be called as a property on the object. Use these for *calculated* properties



```
r
▼ Rectangle {height: 5, width: 3}
    height: 5
    width: 3
    area: 15
  ▼ __proto__:
      area: (...)
    ▶ calcArea: ƒ calcArea()
    ▶ constructor: class Rectangle
    ▶ get area: ƒ area()
    ▶ __proto__: Object
```

Writing a getter method

```
class Rectangle {

    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    get area() {
        return this.calcArea();
    }

    calcArea()
    {
        return this.height * this.width;
    }
}

const r = new Rectangle(5, 3);
console.log(r.area);
```

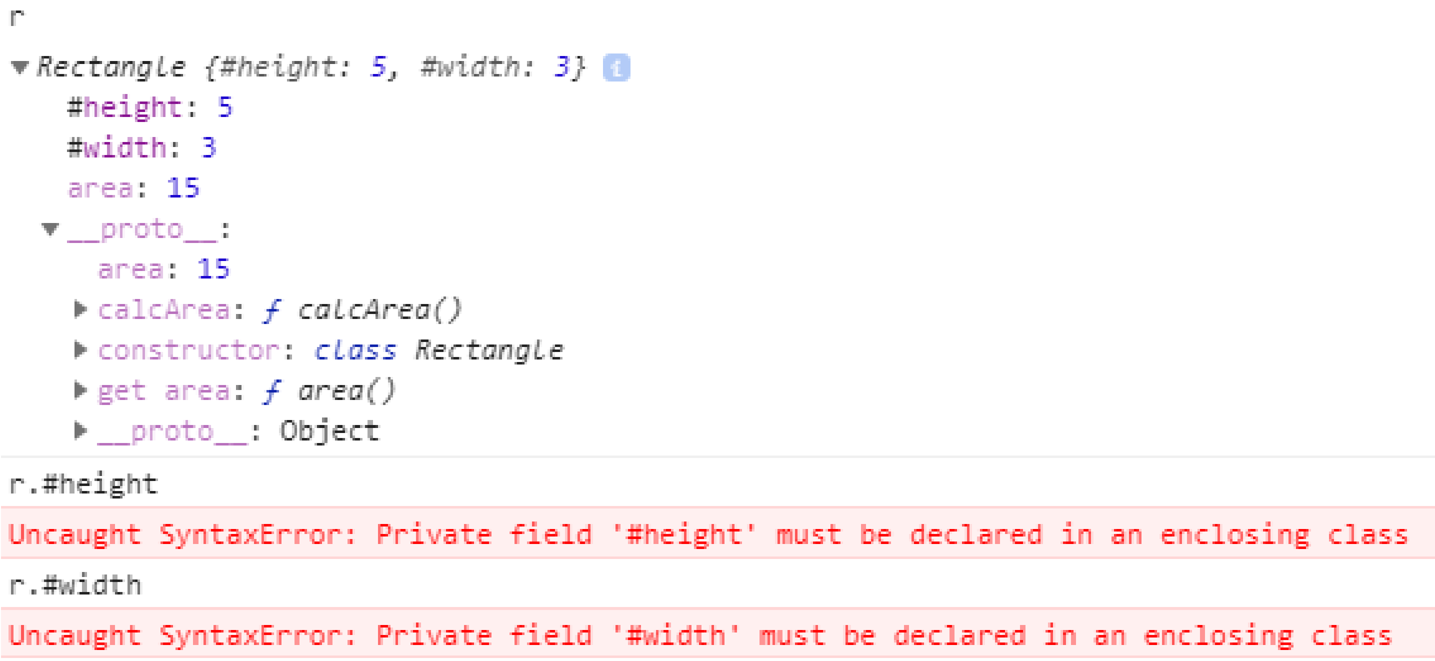
getter methods are only used for **calculated** properties. Regular fields can be accessed with the this.property notation.

regular fields height and width, these do not have a get method.

Private fields

```
class Rectangle {

    #height;
    #width;

    constructor(height, width) {
        this.#height = height;
        this.#width = width;
    }

    get area() {
        return this.calcArea();
    }

    calcArea()
    {
        return this.#height * this.#width;
    }
}
```

Private variables should be put up-front the class. They must start with a #, this is what makes them private.

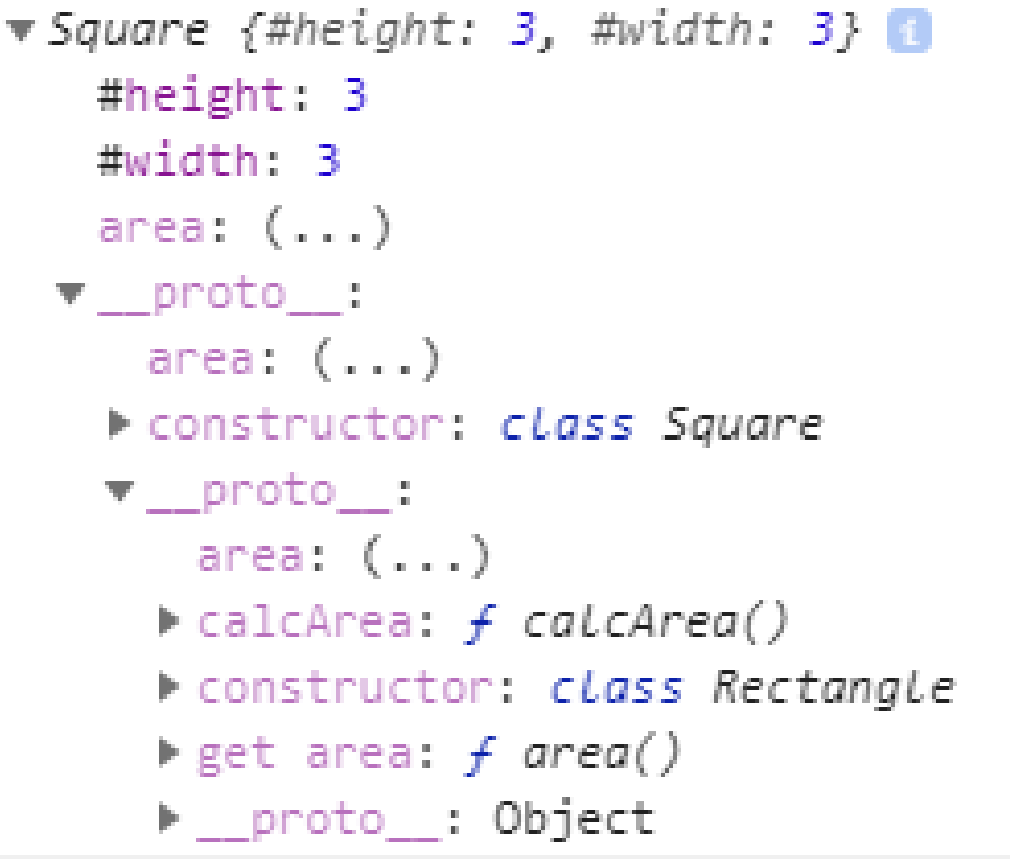


```
r
▼Rectangle {#height: 5, #width: 3}
    #height: 5
    #width: 3
    area: 15
  ▼__proto__:
      area: 15
    ▶calcArea: ƒ calcArea()
    ▶constructor: class Rectangle
    ▶get area: ƒ area()
    ▶__proto__: Object
r.#height
Uncaught SyntaxError: Private field '#height' must be declared in an enclosing class
r.#width
Uncaught SyntaxError: Private field '#width' must be declared in an enclosing class
```

Inheritance in ES6

```
class Rectangle {
    …
}

class Square extends Rectangle {

    constructor(size) {
        super(size, size);
    }

}
```

```
const s = new Square(3);
```



```
▼ Square {#height: 3, #width: 3}
    #height: 3
    #width: 3
    area: (...)
  ▼ __proto__:
      area: (...)
    ▶ constructor: class Square
    ▼ __proto__:
        area: (...)
      ▶ calcArea: ƒ calcArea()
      ▶ constructor: class Rectangle
      ▶ get area: ƒ area()
      ▶ __proto__: Object
```

Overriding methods

```
class Rectangle {
  #height;
  #width;

  constructor(height, width) {
    this.#height = height;
    this.#width = width;
  }

  toString() {
    return `Rectangle(${this.#width},
      ${this.#height})`;
  }

  get width() {
    return this.#width;
  }
}
```

```
class Square extends Rectangle {

  constructor(size) {
    super(size, size);
  }

  toString() {
    return `Square(${this.width})`
  }
}
```

Method toString exists in Object, Rectangle and Square.

We created a public getter for the private field width so that we can access it in our derived class

this vs super

The **super** keyword is used to access properties on an object literal or class's [[Prototype]], or invoke a superclass's constructor.

- To access the member in the instance or in the derived class instead of the one in the base class, use *this*.
- If a member does not exist in the instance or in the derived class but exists in the base class, use *this*.
- To bypass the member of the derived class and access the one in the base class, use *super*.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super

howest
hogeschool

Sets and Maps

Set

The **Set** object lets you store unique values of any type,
whether primitive values or object references.

```
teachers() {
    const teachers = new Set();
    for (let course of this.courses) {
        teachers.add(course.teacher);
    }
    return teachers;
}
```

```
teachersBob = bob.teachers();
for (const teacher of teachersBob) {
    console.log(teacher);
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Set

The **Set** object lets you store unique values of any type, whether primitive values or object references.

```
teachers() {
    const teachers = new Set();
    for (let course of this.courses) {
        teachers.add(course.teacher);
    }
    return teachers;
}
```

```
teachersBob.forEach((value) => {console.log(value)});
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Set

Set.prototype.add()
Inserts a new element with a specified value in to a Set object,
if there isn't an element with the same value already in the Set.

Set.prototype.forEach()
Calls callbackFn once for each value present in the Set object, in insertion order. If a thisArg parameter is provided, it will be used as the this value for each invocation of callbackFn.

Set.prototype.clear()
Removes all elements from the Set object.

Set.prototype.has()
Returns a boolean asserting whether an element is present with the given value in the Set object or not.

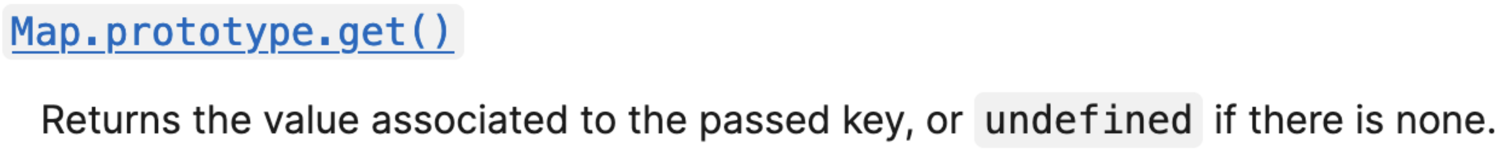
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Map

The **Map** object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values) may be used as either a key or a value.

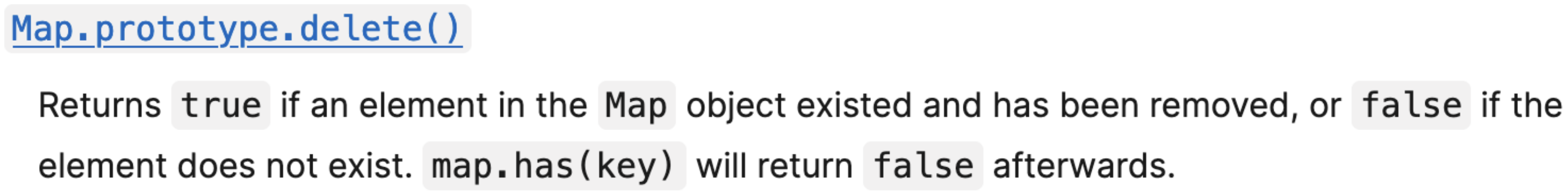
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Map



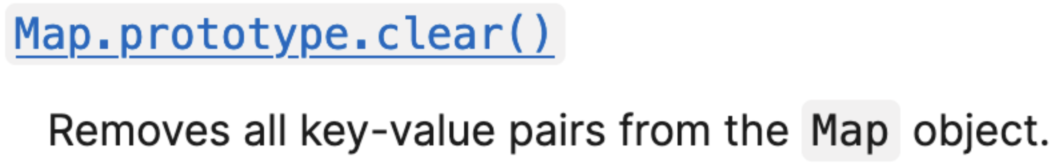
`Map.prototype.get()`

Returns the value associated to the passed key, or `undefined` if there is none.



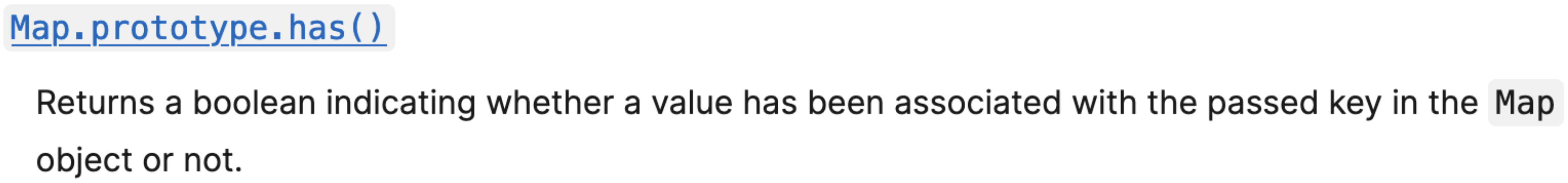
`Map.prototype.delete()`

Returns `true` if an element in the `Map` object existed and has been removed, or `false` if the element does not exist. `map.has(key)` will return `false` afterwards.



`Map.prototype.clear()`

Removes all key-value pairs from the `Map` object.

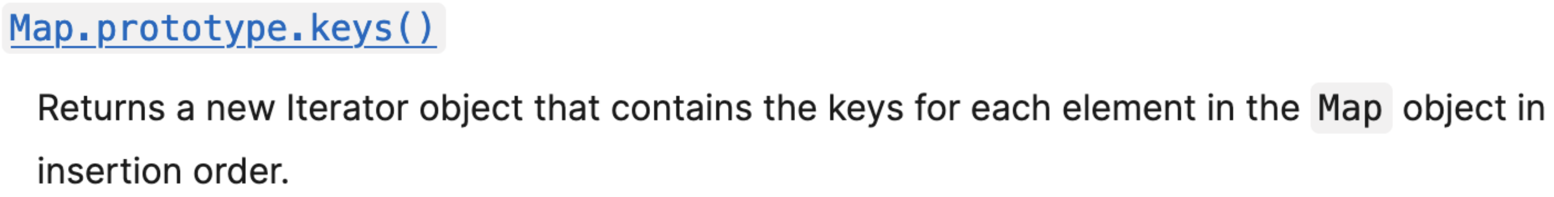


`Map.prototype.has()`

Returns a boolean indicating whether a value has been associated with the passed key in the `Map` object or not.

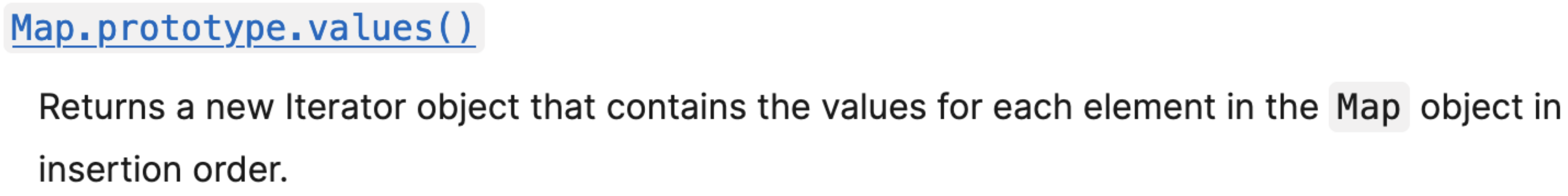
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Map



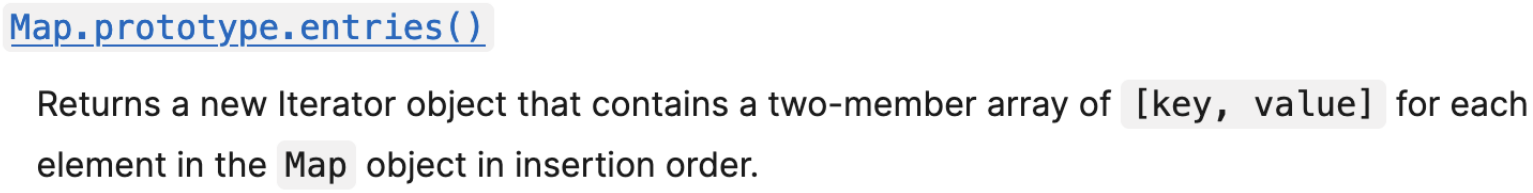
`Map.prototype.keys()`

Returns a new Iterator object that contains the keys for each element in the `Map` object in insertion order.



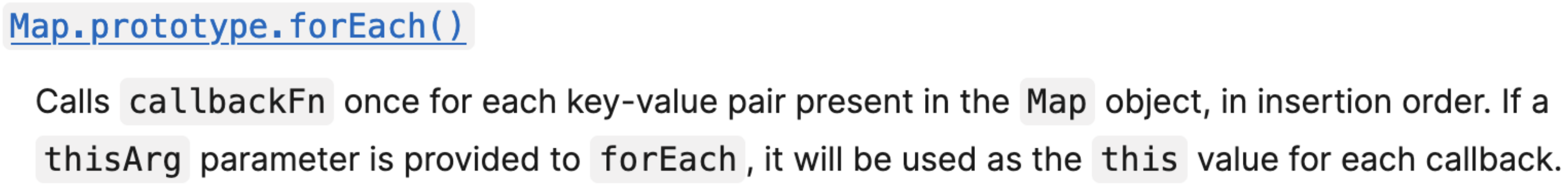
`Map.prototype.values()`

Returns a new Iterator object that contains the values for each element in the `Map` object in insertion order.



`Map.prototype.entries()`

Returns a new Iterator object that contains a two-member array of `[key, value]` for each element in the `Map` object in insertion order.



`Map.prototype.forEach()`

Calls `callbackFn` once for each key-value pair present in the `Map` object, in insertion order. If a `thisArg` parameter is provided to `forEach`, it will be used as the `this` value for each callback.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Enums

Using strings

```
const Seasons = {
  Summer: "summer",
  Autumn: "autumn",
  Winter: "winter",
  Spring: "spring"
}
```

```
class Date {
  constructor(day, month) {
    this.day = day;
    this.month = month;
    this.season = Date.dateToSeason(this.day, this.month);
  }

  static dateToSeason(day, month) {
    let season;
    if (1 <= month && month < 3 || month == 3 && day < 21) {
      season = Seasons.Winter;
    } else if (month <= 5 || month == 6 && day < 21) {
      season = Seasons.Spring;
    } else if (month <= 8 || month == 9 && day < 21) {
     season = Seasons.Summer;
    } else if (month <= 11 || month == 12 && day < 21) {
      season = Seasons.Autumn;
    } else {
      season = Seasons.Winter;
    }
    return season;
  }
}
```

Using ints

```
const Seasons = {
   Summer: 0,
   Autumn: 1,
   Winter: 2,
   Spring: 3
}
```

```
class Date {
   constructor(day, month) {
      this.day = day;
      this.month = month;
      this.season = Date.dateToSeason(this.day, this.month);
   }

   static dateToSeason(day, month) {
      let season;
      if (1 <= month && month < 3 || month == 3 && day < 21) {
         season = Seasons.Winter;
      } else if (month <= 5 || month == 6 && day < 21) {
         season = Seasons.Spring;
      } else if (month <= 8 || month == 9 && day < 21) {
        season = Seasons.Summer;
      } else if (month <= 11 || month == 12 && day < 21) {
         season = Seasons.Autumn;
      } else {
         season = Seasons.Winter;
      }
      return season;
   }
}
```

This approach has some problems, definitions from unrelated enums can for example overlap and cause problems.

It is also semantically incorrect, seasons aren’t really strings or integers.

Symbol

Symbol is a built-in object whose constructor returns a symbol primitive — also called a **Symbol value** or just a **Symbol** — that's guaranteed to be unique.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol

Symbol

Symbol is a built-in object whose constructor returns a symbol primitive — also called a **Symbol value** or just a **Symbol** — that's guaranteed to be unique.

```
console.log(Symbol("summer") === Symbol("summer")); // false
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol

Using Symbol

```
const Seasons = {
    Summer: Symbol("summer"),
    Autumn: Symbol("autumn"),
    Winter: Symbol("winter"),
    Spring: Symbol("spring")
};
```

The seasons object can still be modified.

```
class Date {
  constructor(day, month) {
    this.day = day;
    this.month = month;
    this.season = Date.dateToSeason(this.day, this.month);
  }

  static dateToSeason(day, month) {
    let season;
    if (1 <= month && month < 3 || month == 3 && day < 21) {
      season = Seasons.Winter;
    } else if (month <= 5 || month == 6 && day < 21) {
      season = Seasons.Spring;
    } else if (month <= 8 || month == 9 && day < 21) {
     season = Seasons.Summer;
    } else if (month <= 11 || month == 12 && day < 21) {
      season = Seasons.Autumn;
    } else {
      season = Seasons.Winter;
    }
    return season;
  }
}
```

Using Symbol

```
const Seasons = Object.freeze({
        Summer: Symbol("summer"),
        Autumn: Symbol("autumn"),
        Winter: Symbol("winter"),
        Spring: Symbol("spring")
});
```

Object.freeze freezes an object so that it can’t be modified anymore.

```
class Date {
  constructor(day, month) {
    this.day = day;
    this.month = month;
    this.season = Date.dateToSeason(this.day, this.month);
  }

  static dateToSeason(day, month) {
    let season;
    if (1 <= month && month < 3 || month == 3 && day < 21) {
      season = Seasons.Winter;
    } else if (month <= 5 || month == 6 && day < 21) {
      season = Seasons.Spring;
    } else if (month <= 8 || month == 9 && day < 21) {
     season = Seasons.Summer;
    } else if (month <= 11 || month == 12 && day < 21) {
      season = Seasons.Autumn;
    } else {
      season = Seasons.Winter;
    }
    return season;
  }
}
```