

Apprendre la syntaxe JavaScript : objets

Restrictions dans les propriétés de nommage

Les noms de clé d'objet JavaScript doivent respecter certaines restrictions pour être valides. Les noms de clé doivent être soit des chaînes, soit des identificateurs valides ou des noms de variable (c'est-à-dire des caractères spéciaux tels que ceux - qui ne sont pas autorisés dans les noms de clé qui ne sont pas des chaînes).

```
// Exemple de noms de clés invalides
const TrainSchedule = {
  platform num : 10 , // Invalide à cause
de l'espace entre les mots.
  40 - 10 + 2 : 30 , // Les expressions
ne peuvent pas être des clés.
  + compartiment : 'C' // L'utilisation
d'un signe + n'est valide que s'il est
entre guillemets.
}
```

Notation par points pour accéder aux propriétés de l'objet

Les propriétés d'un objet JavaScript sont accessibles en utilisant la notation par points de cette manière : `object.propertyName` . Les propriétés imbriquées d'un objet sont accessibles en enchaînant les noms de clé dans le bon ordre.

```
const pomme = {
  couleur : 'Vert' ,
  prix : {
    vrac : '$3/kg' ,
    petiteQté : '$4/kg'
  }
} ;

consoler . bûche ( pomme . couleur ) ; //
'Vert'

consoler . bûche ( pomme . prix . vrac )
; // '$3/kg'
```

Objets

An *object* is a built-in data type for storing key-value pairs. Data inside objects are unordered, and the values can be of any type.

Accessing non-existent JavaScript properties

When trying to access a JavaScript object property that has not been defined yet, the value of `undefined`

```
const classElection = {
  date: 'January 12'
```

will be returned by default.

```
};
```

```
console.log(classElection.place); //  
undefined
```

JavaScript Objects are Mutable

JavaScript objects are *mutable*, meaning their contents can be changed, even when they are declared as `const`. New properties can be added, and existing property values can be changed or deleted. It is the *reference* to the object, bound to the variable, that cannot be changed.

```
const student = {  
  name: 'Sheldon',  
  score: 100,  
  grade: 'A',  
}  
  
console.log(student)  
// { name: 'Sheldon', score: 100, grade:  
  'A' }  
  
delete student.score  
student.grade = 'F'  
console.log(student)  
// { name: 'Sheldon', grade: 'F' }  
  
student = {}  
// TypeError: Assignment to constant  
variable.
```

JavaScript for...in loop

The JavaScript `for...in` loop can be used to iterate over the keys of an object. In each iteration, one of the properties from the object is assigned to the variable of that loop.

```
let mobile = {  
  brand: 'Samsung',  
  model: 'Galaxy Note 9'  
};  
  
for (let key in mobile) {  
  console.log(`${key}: ${mobile[key]}`);  
}
```

Properties and values of a JavaScript object

A JavaScript object literal is enclosed with curly braces `{}`. Values are mapped to keys in the object with a colon `:`, and the key-value pairs are separated by commas. All the keys are unique, but values are not. Key-value pairs of an object are also referred to as *properties*.

```
const classOf2018 = {  
  students: 38,  
  year: 2018  
}
```

Delete operator

Once an object is created in JavaScript, it is possible to remove properties from the object using the `delete` operator. The `delete` keyword deletes both the value of the property and the property itself from the object. The `delete` operator only works on properties, not on variables or functions.

```
const person = {
  firstName: "Matilda",
  age: 27,
  hobby: "knitting",
  goal: "learning JavaScript"
};

delete person.hobby; // or delete
person[hobby];

console.log(person);
/*
{
  firstName: "Matilda"
  age: 27
  goal: "learning JavaScript"
}
*/
```

javascript passing objects as arguments

When JavaScript objects are passed as arguments to functions or methods, they are passed by *reference*, not by value. This means that the object itself (not a copy) is accessible and mutable (can be changed) inside that function.

```
const origNum = 8;
const origObj = {color: 'blue'};

const changeItUp = (num, obj) => {
  num = 7;
  obj.color = 'red';
};

changeItUp(origNum, origObj);

// Will output 8 since integers are
// passed by value.
console.log(origNum);

// Will output 'red' since objects are
// passed
// by reference and are therefore
// mutable.
console.log(origObj.color);
```

JavaScript Object Methods

JavaScript objects may have property values that are *functions*. These are referred to as object *methods*. Methods may be defined using anonymous *arrow function expressions*, or with *shorthand method syntax*. Object methods are invoked with the syntax: `objectName.methodName(arguments)`.

```
const engine = {
  // method shorthand, with one argument
  start(adverb) {
    console.log(`The engine starts up
    ${adverb}...`);
  },
  // anonymous arrow function expression
  with no arguments
  sputter: () => {
    console.log('The engine
    sputters...');
  },
};

engine.start('noisily');
engine.sputter();

/* Console output:
The engine starts up noisily...
The engine sputters...
*/
```

JavaScript destructuring assignment shorthand syntax

The JavaScript *destructuring assignment* is a shorthand syntax that allows object properties to be extracted into specific variable values.

It uses a pair of curly braces (`{ }`) with property names on the left-hand side of an assignment to extract values from objects. The number of variables can be less than the total properties of an object.

```
const rubiksCubeFacts = {
  possiblePermutations:
    '43,252,003,274,489,856,000',
  invented: '1974',
  largestCube: '17x17x17'
};

const {possiblePermutations, invented,
largestCube} = rubiksCubeFacts;
console.log(possiblePermutations); //
'43,252,003,274,489,856,000'
console.log(invented); // '1974'
console.log(largestCube); // '17x17x17'
```

shorthand property name syntax for object creation

The *shorthand property name* syntax in JavaScript allows creating objects without explicitly specifying the property names (ie. explicitly declaring the value after the key). In this process, an object is created where the

```
const activity = 'Surfing';
const beach = { activity };
```

property names of that object match variables which already exist in that context. Shorthand property names populate an object with a key matching the identifier and a value matching the identifier's value.

this Keyword

The reserved keyword `this` refers to a method's calling object, and it can be used to access properties belonging to that object.

Here, using the `this` keyword inside the object function to refer to the `cat` object and access its `name` property.

```
console.log(beach); // { activities:
'Surfing' }
```

```
const cat = {
  name: 'Pipey',
  age: 8,
  whatName() {
    return this.name
  }
};

console.log(cat.whatName());
// Output: Pipey
```

javascript function this

Every JavaScript function or method has a `this` context. For a function defined inside of an object, `this` will refer to that object itself. For a function defined outside of an object, `this` will refer to the global object (`window` in a browser, `global` in Node.js).

```
const restaurant = {
  numCustomers: 45,
  seatCapacity: 100,
  availableSeats() {
    // this refers to the restaurant
    object
    // and it's used to access its
    properties
    return this.seatCapacity -
    this.numCustomers;
  }
}
```

JavaScript Arrow Function this Scope

JavaScript arrow functions do not have their own `this` context, but use the `this` of the surrounding lexical context. Thus, they are generally a poor choice for writing object methods.

Consider the example code:

`loggerA` is a property that uses arrow notation to define the function. Since `data` does not exist in the global context, accessing `this.data` returns `undefined`.

`loggerB` uses method syntax. Since `this` refers to the enclosing object, the value of the `data` property is

```
const myObj = {
  data: 'abc',
  loggerA: () => {
    console.log(this.data); },
  loggerB() { console.log(this.data);
},
};
```

accessed as expected, returning "abc" .

```
myObj.loggerA();    // undefined  
myObj.loggerB();    // 'abc'
```

getters and setters intercept property access

JavaScript getter and setter methods are helpful in part because they offer a way to intercept property access and assignment, and allow for additional actions to be performed before these changes go into effect.

```
const myCat = {  
  _name: 'Snickers',  
  get name() {  
    return this._name  
  },  
  set name(newName) {  
    //Verify that newName is a non-empty  
    string before setting as name property  
    if (typeof newName === 'string' &&  
        newName.length > 0) {  
      this._name = newName;  
    } else {  
      console.log("ERROR: name must be a  
non-empty string");  
    }  
  }  
}
```

javascript factory functions

A JavaScript function that returns an object is known as a *factory function*. Factory functions often accept parameters in order to customize the returned object.

```
// A factory function that accepts  
'name',  
// 'age', and 'breed' parameters to  
return  
// a customized dog object.  
const dogFactory = (name, age, breed) =>  
{  
  return {  
    name: name,  
    age: age,  
    breed: breed,  
    bark() {  
      console.log('Woof!');  
    }  
  };  
};
```

javascript getters and setters restricted

JavaScript object properties are not private or protected. Since JavaScript objects are passed by reference, there is no way to fully prevent incorrect interactions with object properties.

One way to implement more restricted interactions with object properties is to use *getter* and *setter* methods.

Typically, the internal value is stored as a property with an identifier that matches the *getter* and *setter* method names, but begins with an underscore (`_`).

```
const myCat = {  
  _name: 'Dottie',  
  get name() {  
    return this._name;  
  },  
  set name(newName) {  
    this._name = newName;  
  }  
};  
  
// Reference invokes the getter  
console.log(myCat.name);  
  
// Assignment invokes the setter  
myCat.name = 'Yankee';
```

 Save  Print  Share ▼