codecademy

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).

```
// Example of invalid key names
const trainSchedule = {
  platform num: 10, // Invalid because of the space between words.
  40 - 10 + 2: 30, // Expressions cannot be keys.
  +compartment: 'C' // The use of a + sign is invalid unless it is enclosed in quotations.
}
```

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 apple = {
  color: 'Green',
  price: {
    bulk: '$3/kg',
    smallQty: '$4/kg'
  }
};
console.log(apple.color); // 'Green'
console.log(apple.price.bulk); // '$3/kg'
```

Objets

Un *objet* est un type de données intégré pour stocker des paires clé-valeur. Les données à l'intérieur des objets ne sont pas ordonnées et les valeurs peuvent être de n'importe quel type.

Accéder à des propriétés JavaScript inexistantes

Lorsque vous essayez d'accéder à une propriété d'objet JavaScript qui n'a pas encore été définie, la valeur de `undefined` sera renvoyée par défaut.

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

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

Les objets JavaScript sont mutables

Les objets JavaScript sont *mutable* , ce qui signifie que leur contenu peut être modifié, même lorsqu'ils sont déclarés en tant que `const` . De nouvelles propriétés peuvent être ajoutées et les valeurs de propriétés existantes peuvent être modifiées ou supprimées.
C'est la *référence* à l'objet, liée à la variable, qui ne peut pas être modifiée.

```
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.
```

`for...in`Boucle JavaScript

La boucle JavaScript `for...in` peut être utilisée pour parcourir les clés d'un objet. Dans chaque itération, l'une des propriétés de l'objet est affectée à la variable de cette boucle.

```
let mobile = {
  brand: 'Samsung',
  model: 'Galaxy Note 9'
};

for (let key in mobile) {
  console.log(`${key}: ${mobile[key]}`);
}
```

Propriétés et valeurs d'un objet JavaScript

Un littéral d'objet JavaScript est entouré d'accolades `{}` . Les valeurs sont mappées aux clés de l'objet avec deux-points (`:`) et les paires clé-valeur sont séparées par des virgules. Toutes les clés sont uniques, mais les valeurs ne le sont pas.
Les paires clé-valeur d'un objet sont également appelées *propriétés* .

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

Supprimer l'opérateur

Une fois qu'un objet est créé en JavaScript, il est possible de supprimer des propriétés de l'objet à l'aide de l' `delete` opérateur. Le `delete` mot clé supprime à la fois la valeur de la propriété et la propriété elle-même de l'objet. L' `delete` opérateur ne fonctionne que sur les propriétés, pas sur les variables ou les fonctions.

```
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 passant des objets en arguments

Lorsque des objets JavaScript sont transmis en tant qu'arguments à des fonctions ou à des méthodes, ils sont transmis par *référence* et non par valeur. Cela signifie que l'objet lui-même (pas une copie) est accessible et modifiable (peut être modifié) à l'intérieur de cette fonction.

```
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);
```

Méthodes d'objet JavaScript

Les objets JavaScript peuvent avoir des valeurs de propriété qui sont *des fonctions* . Celles-ci sont appelées *méthodes* objet .

Les méthodes peuvent être définies à l'aide d' *expressions de fonction de flèche* anonymes ou d'une *syntaxe de méthode abrégée* .

Les méthodes d'objet sont appelées avec la syntaxe : `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...
*/
```

Syntaxe abrégée d'affectation de déstructuration JavaScript

L' affectation de déstructuration JavaScript est une syntaxe abrégée qui permet d'extraire les propriétés d'un objet dans des valeurs de variables spécifiques.

Il utilise une paire d'accolades (`{}`) avec des noms de propriété sur le côté gauche d'une affectation pour extraire des valeurs d'objets. Le nombre de variables peut être inférieur au nombre total de propriétés d'un objet.

```
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'
```

syntaxe abrégée du nom de propriété pour la création d'objets

La syntaxe *abrégée du nom de propriété* en JavaScript permet de créer des objets sans spécifier explicitement les noms de propriété (c'est-à-dire en déclarant explicitement la valeur après la clé). Dans ce processus, un objet est créé où les noms de propriété de cet objet correspondent à des variables qui existent déjà dans ce contexte. Les noms de propriété abrégés remplissent un objet avec une clé correspondant à l'identifiant et une valeur correspondant à la valeur de l'identifiant.

```
const activity = 'Surfing';
const beach = { activity };
console.log(beach); // { activity:
'Surfing' }
```

`this`Mot-clé

Le mot-clé réservé `this` fait référence à l'objet appelant d'une méthode et peut être utilisé pour accéder aux propriétés appartenant à cet objet. Ici, en utilisant le mot- `this` clé à l'intérieur de la fonction object pour faire référence à l' `cat` objet et accéder à sa `name` propriété.

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

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

javascript fonctionne ceci

Chaque fonction ou méthode JavaScript a un `this` contexte. Pour une fonction définie à l'intérieur d'un objet, `this` fera référence à cet objet lui-même. Pour une fonction définie en dehors d'un objet, `this` fera référence à l'objet global (`window` dans un navigateur, `global` dans 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 Flèche Fonction cette portée

Les fonctions fléchées JavaScript n'ont pas leur propre `this` contexte, mais utilisent le `this` contexte lexical environnant. Ainsi, ils sont généralement un mauvais choix pour écrire des méthodes objet.

Considérez l'exemple de code :

`loggerA` est une propriété qui utilise la notation fléchée pour définir la fonction. Puisque `data` n'existe pas dans le contexte global, l'accès `this.data` renvoie `undefined` .

`loggerB` utilise la syntaxe de la méthode. Puisque `this` fait référence à l'objet englobant, la valeur de la `data` propriété est accessible comme prévu, en retournant `"abc"` .

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

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

les getters et les setters interceptent l'accès à la propriété

Les méthodes getter et setter JavaScript sont utiles en partie parce qu'elles offrent un moyen d'intercepter l'accès et l'affectation des propriétés, et permettent d'effectuer des actions supplémentaires avant que ces modifications n'entrent en vigueur.

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

fonctions d'usine javascript

Une fonction JavaScript qui renvoie un objet est appelée *fonction de fabrique* . Les fonctions d'usine acceptent souvent des paramètres afin de personnaliser l'objet renvoyé.

```
// 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!');
    }
  };
};
```

getters et setters javascript restreints

Les propriétés des objets JavaScript ne sont ni privées ni protégées. Étant donné que les objets JavaScript sont passés par référence, il n'existe aucun moyen d'empêcher complètement les interactions incorrectes avec les propriétés de l'objet.
Une façon d'implémenter des interactions plus restreintes avec les propriétés d'objet consiste à utiliser des *méthodes getter* et *setter* .
En règle générale, la valeur interne est stockée en tant que propriété avec un identifiant qui correspond aux noms des méthodes *getter* et *setter* , mais commence par un trait de soulignement (_).

```
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';
```