

Les bases

JavaScript

WIK-JS-102



Wikodit - tous droits réservés 2019

La reproduction, l'utilisation ou la diffusion non autorisée de ce document est interdite sans l'autorisation de l'auteur

WIK-JS

Programme JavaScript

101 – Introduction

102 – Les bases

103 – Le prototypage et la POO

104 – Callback, asynchrones, évènements

105 – L'asynchrone avec les promesses

JavaScript

ES7, ES8, ES9, ES10

Notions

Les variables

```
const firstname = 'Jean'
let lastname = 'Jean'

hi = 'Bonjour'
lastname = 'Bon'

console.log(hi + ' ' + firstname + ' ' + lastname)
```

var Créé une variable de portée au scope


const Créé une constante de portée au block - La référence vers la valeur ne peut pas être changée

let Créé une variable de portée au block

À partir de maintenant, il est recommandé de n'utiliser que **let** et **const**

String interpolation

```
const firstname = 'Jean'  
const lastname = 'Bon'  
  
console.log(`Bonjour ${firstname} ${lastname} !`)
```



Backtick (accent grave / apostrophe à l'envers)

Les fonctions

```
function add(a, b) {  
    return a + b  
}
```

```
const result = add(2, 4)  
console.log(result)
```

```
function add(a, b) {  
    return a + b  
}
```

```
const result = add(2, 4)  
console.log(result)
```

Une fonction possède un **nom**, des **paramètres**, et un **corps**.

Une fonction est appelée avec des **arguments**, et

TP: Exercices fonctions

Exercice 1

Créer une fonction qui prend en paramètres un nom et un prénom :

- Cette fonction doit retourner la concaténation des deux.
- Assigner le retour de la fonction à une variable et affichez le.
- Modifier la valeur de cette variable par un nouveau couple “nom prénom”.
- Afficher le résultat.

Créer une fonction qui prend un nombre en paramètre et retourne son carré :

- Afficher le résultat

Les types

Le typage en JavaScript est **faible ET dynamique**.

Il n'est pas nécessaire de définir le type des variables, et une variable peut avoir plusieurs types durant son existence.

Le type est déterminé lors de l'exécution, et non pas lors de la compilation (en théorie...)

```
let a = 'chaîne'  
a = 23  
a = true
```


Les 6 types primitifs

- boolean
- number
- string
- undefined
- null
- symbol

```
const someBoolean = true
const someFalsyBoolean = false

const someString = 'Hello world!'

const somethingUndefined
const somethingDefinedToUndefined = undefined

const nul = null

const someNumber = -139.54
const somePositiveInfinity = +Infinity // 42 / 0
const someNegativeInfinity = -Infinity // 42 / -0
const someNumberThatIsNotANumber = NaN // 42 / 'toto'

const someSymbolThatWeWillNeverTalkAgain = Symbol()
```

Le 7e type : Object

OU syntaxe littérale d'objet :

```
let user = new Object()

user['name'] = 'Jean Bon'
user.age = 41
user.address = new Object()
user.address.number = 100
user.address['street'] = 'rue du Lilas'
user.address.city = 'Bordeaux'
user.address.zipcode = 33000

user.age = user.age + 1

console.log(user)
```

```
const streetType = 'street'

let user = {
  name: 'Jean Bon',
  age: 41,
  address: {
    number: 100,
    [streetType]: 'rue du Lilas',
    city: 'Bordeaux',
    zipcode: 33000,
  },
}

console.log(user)
```

Un objet est une structure complexe, qui possède des clés et des valeurs, c'est une collection de propriétés.

Le 7e type : Object

Un objet est persistant en mémoire, **seule la référence à l'objet est donné à la variable.**

Ici, on a stocké la référence de `user.address` dans `otherUser.address`, la valeur est donc non pas identique, mais c'est le même pour les deux.

Modifier `user.address`, revient à modifier `otherUser.address` et vice-versa

```
let otherUser = {  
  name: 'Emilie Bond',  
  age: 28,  
  address: user.address  
}  
  
user.address.city = 'Le Bouscat'  
  
console.log(otherUser)  
console.log(user)
```

Passage par valeur

Adresse	Valeur
0x3e	a => 4
0x3f	param => 4
0x40	
0x41	
0x42	

Mémoire Vive

En javascript, tout type **primitif** est passé par **valeur**

```
function log(param) {  
  console.log(param)  
  param = 5  
  console.log(param)  
}  
  
const a = 4  
log(a)  
console.log(a)
```

Passage par référence

Adresse	Valeur
0x3e	{ pseudo: 'toto', age: 15 }
0x3f	a => 0x3e
0x40	param => 0x3e
0x41	
0x42	

Mémoire Vive

En javascript, tout **Objet** est passé par **référence**

```
function log(param) {  
  console.log(param)  
  param.pseudo = 'tata'  
  console.log(param)  
}  
  
const a = {  
  pseudo: 'toto',  
  age: 15,  
}  
  
log(a)  
  
console.log(a)
```

Et les listes ?

Les listes/tableaux sont appelées **Array** en JavaScript, et elles sont **ordonnées**.

```
let list = new Array()
list.push('toto')

// Ou syntaxe littérale de tableau :

const a = [
  123,
  'why not a string or boolean ?',
  true,
  'or an object ?',
  { foo: 'bar' },
]

console.log(a[0])
console.log(a[4].foo)
console.log(a.length)
```

Mais alors, quel est le type d'un tableau ?

Les Arrays sont tout simplement des **Object**, avec un fonctionnement spécifique

Et les listes ?

Quelques fonctions pratiques

Il en existe plein d'autres !

```
const messages = [  
  'Hello all!'  
]  
  
messages.push('Also hello to you, you just got')  
messages.push(10)  
messages.push('years')  
messages.push('sdfadsfas')  
  
console.log(messages)  
  
const lastItem = messages.pop()  
console.log(lastItem, messages)  
  
const firstItem = messages.shift()  
console.log(firstItem, messages)  
  
messages.unshift('Hello world!')  
console.log(messages)  
  
const sentence = messages.join(' ')  
console.log(sentence)  
  
const allWords = sentence.split(' ')  
console.log(allWords)  
  
messages.reverse()  
console.log(messages)  
  
console.log(messages.indexOf('something not in the array'))  
console.log(messages.indexOf('years'))
```

Surprenant ?

Quel est le type d'une fonction ?

Une fonction, est aussi un **Object**,
à la différence qu'elle peut-être appelée

```
function hello() {}  
  
console.log(hello.name)  
console.log(hello.prototype)
```


Encore plus de surprise !

Que remarquez-vous ?

Est-ce logique ?

```
typeof 'une chaine'  
typeof true  
typeof undefined  
typeof { name: 'Jean' }  
typeof [0, 1, 2, 3]  
typeof null  
typeof function () { }
```

Les conditions

Faible égalité

```
let a = '2'  
  
if (a == 2) {  
  console.log('a est égal à 2')  
} else {  
  console.log('a n\'est pas égal à 2')  
}
```

Stricte égalité, le type est pris en compte

```
let a = '2'  
  
if (a === 2) {  
  console.log('a est le chiffre 2')  
} else {  
  console.log('a n\'est pas le chiffre 2')  
}
```

Condition ternaire

```
let a = '2'  
console.log(a == '2' ? 'a égal 2' : 'a n\'est pas égal à 2')
```

Les conditions

Opérateurs de comparaison

<	Inférieur à
>	Supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égalité faible
===	Égalité stricte
!=	Inégalité faible
!==	Inégalité stricte

Opérateurs logiques

&&	ET logique
	OU logique
!	NON logique

```
let a = '2'

if (typeof a !== undefined && a !== null) {
  if (a === 1) {
    console.log('a est 1')
  } else if (a >= 5 || a === 10) {
    console.log('a est soit supérieur ou égal à 5, ou égal à 10')
  } else {
    console.log('a est inférieur à 5 et différent de 1')
  }
} else {
  console.log('a est null ou non défini')
}
```

TP: Exercices fonctions

Exercice 2

Créer un objet user qui comporte un nom, une description et un budget.

Créer une fonction qui prend en paramètre un user.

Si le budget est undefined/null, remplir le champ description de l'objet user avec "Tu as oublié ton porte feuille".

- Si le budget est entre 0 et 5 euros, remplir le champ description de l'objet user avec "Il fallait travailler cet été".
- Si le budget est égal 5 euros, remplir le champ description de l'objet user avec "Tu as le droit à une bière".
- Sinon, remplir le champ description de l'objet user avec "Tu peux payer ta tournée".

Afficher l'objet user sous la forme "Bravo, (nom), (description). Voici le rappel de ton budget : (budget)"

Les fonctions

```
// Fonction nommée (hoisted)  
console.log(sum(7, 4)) // => 11  
function sum(x, y) {  
    return x + y  
}
```

```
// Fonction anonyme  
console.log(diff(7, 4))  
// => ERREUR  
const diff = function (x, y) {  
    return x - y  
}  
console.log(diff(7, 4)) // => 3
```

```
// Fonction anonyme avec flèche (conserve le scope)  
const times = (x, y) => { return x * y }  
// Fonction anonyme avec return implicite (conserve le scope)  
const times = (x, y) => x * y
```

Si un seul paramètre, les parenthèses des fonctions fléchées sont facultatives

Fonction tableau avec callback

```
const list = [1, 4, 9, 16]
const doubleOfEverything = list.map(x => {
  return x * 2
})

console.log(doubleOfEverything)
```

OU

```
const list = [1, 4, 9, 16]
const doubleOfEverything = list.map(x => x * 2)
console.log(doubleOfEverything)
```

Les itérations (for)

```
const fruits = ['apple', 'orange', 'strawberry', 'blueberry']
const l = fruits.length
for (let i = 0; i < l ; i++) {
  console.log(1, fruits[i])
}
```

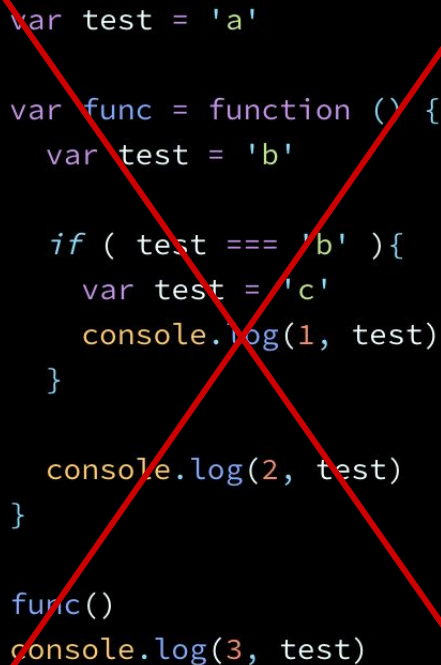
```
for (let i = 0; i < l ; i++) {
  if (fruits[i] === 'apple') { continue }
  console.log(3, fruits[i])
  if (fruits[i] === 'strawberry') { break }
}
```

```
for (let i = fruits.length; i-- ; ) {
  console.log(2, fruits[i])
}
```

Les itérations (while)

```
let found = false
let i = 0
while (!found) {
  if (fruits[i] === 'strawberry') {
    found = true
  }
  console.log(4, fruits[i])
  i++
}
```


La portée des variables : le scope



```
var test = 'a'

var func = function () {
  var test = 'b'

  if ( test === 'b' ){
    var test = 'c'
    console.log(1, test)
  }

  console.log(2, test)
}

func()
console.log(3, test)
```

```
let test2 = 'a'

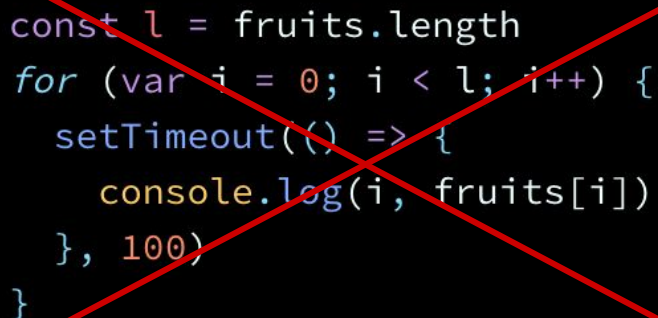
const func2 = function() {
  let test2 = 'b'

  if ( test2 === 'b' ) {
    let test2 = 'c'
    console.log(11, test2)
  }

  console.log(12, test2)
}

func2()
console.log(13, test2)
```

Attention au scope !



```
const l = fruits.length
for (var i = 0; i < l; i++) {
  setTimeout(() => {
    console.log(i, fruits[i])
  }, 100)
}
```

```
const l = fruits.length
for (let i = 0; i < l; i++) {
  setTimeout(() => {
    console.log(i, fruits[i])
  }, 100)
}
```

setTimeout : Exécute la fonction en paramètre après un certain temps (en ms)

NE JAMAIS UTILISER LE CODE DE GAUCHE !

Array destructuration

ES5

```
let tbl = [ 10, 20, 30 ]  
let [ a, b ] = tbl
```

```
console.log(1, a)  
console.log(2, b)
```

```
[ b, a ] = [ a, b ]  
console.log(3, a)  
console.log(4, b)
```

```
let tbl = [10, 20, 30]  
let a = tbl[0]  
let b = tbl[1]
```

```
console.log(1, a)  
console.log(2, b)
```

```
const tmp = b  
b = a  
a = tmp
```

```
console.log(3, a)  
console.log(4, b)
```

Object destructuration

ES5

```
const user = {  
  firstname: 'Jean',  
  lastname: 'Dubois',  
  age: 30,  
}  
  
const { firstname, age } = user  
console.log(1, firstname, age)  
  
// Mais aussi:  
const city = 'Bordeaux'  
const address = {  
  city,  
  zipcode: '33000',  
}  
console.log(2, address.city)
```

```
const user = {  
  firstname: 'Jean',  
  lastname: 'Dubois',  
  age: 30,  
}  
  
const firstname = user.firstname  
const age = user.age  
console.log(1, firstname, age)  
  
// Mais aussi:  
const city = 'Bordeaux'  
const address = {  
  city: city,  
  zipcode: '33000',  
}  
console.log(2, address.city)
```

Le paramètre rest

S'utilise en paramètre de fonction comme son nom l'indique

```
function multiply(multiplier, ...numbers) {  
  return numbers.map((number) => {  
    return multiplier * number;  
  })  
}  
  
// 4 args  
console.log(multiply(2, 1, 4, 5))  
  
// 2 args  
console.log(multiply(3, 4))  
  
// many args  
console.log(multiply(5, 3, 4, 5, 3, 1, 4, 5, 6))
```

...

L'opérateur spread

■ ■ ■ aussi

Pour les objets

Pour les arrays

```
const firstArray = ['bien', 'oui']  
const secondArray = ['et', 'en', 'non']  
  
const thirdArray = [...firstArray, ...secondArray]  
  
console.log(thirdArray)
```

```
const identity = {  
  firstname: 'Jonathan',  
  lastname: 'Hash'  
}  
  
const address = {  
  city: 'Pas loin',  
  country: 'France'  
}  
  
const user = {  
  ...identity,  
  ...address,  
}  
  
console.log(user)
```

TP: Exercices fonctions

Exercice 3

Créer un tableau “numbers” contenant 10 nombres entiers pairs et impairs.

- Parcourir le tableau et afficher si le nombre est pair ou impair.

Exercice 4

Créer une fonction retournant si une chaîne de caractère (passée en paramètre) est un palindrome.

TP: Exercices fonctions

Exercice 5

Créer une fonction qui prend deux tableaux en paramètres et qui renvoie un tableau avec les éléments qui diffèrent.

- (Cf: lodash fonction `_.difference`). **Ne pas utiliser lodash ou les fonctions existantes !**

Exercice 6

Créer une fonction qui prend en paramètres un tableau de string et un séparateur et qui renvoie la concaténation des strings du tableau séparées par le séparateur.

- (Cf: lodash fonction `_.join`). **Ne pas utiliser lodash ou les fonctions existantes !**

Félicitations !!

WIK-JS-102 burned :)