Programmation en javascript

Dr Bakary Diarra
Cité collégiale

Introduction

- Découvrir les objets
- Comprendre les tableaux
- Les fonctions
- Notion de programmation asynchrone
- Les promesses et les callbacks

Les objets en javascript

- Les objets sont un concept indispensable en javascript (en savoir plus)
- Ils contiennent des propriétés (keys-values → clef-valeur) et des méthodes

```
//Présentation d'un objet

const notreObjet = {
    propriete1:"Chaine de caractère",
    propriete2:200,
    method:function(){},
    propriete4:{
        prop1:true,
        prop2:["Un","Deux","Trois"]
    },
    propriete5:["Nous","Vous"],
}
```

```
method(){}
```

- ✓ Accéder aux propriétés de l'objet : $notreObjet.propriete$
- ✓ Accéder aux méthodes
  $notreObjet.method()$ pour la valeur
  $notreObjet.method$ pour la fonction
- ✓ Accéder aux propriétés au sein de la méthode
  $method(){this.propriete}$
- ✓ Vérifier si une propriété est présente dans l'objet
  $notreObjet.hasOwnProperty(propriete)$

Les objets en javascript

- Il est possible de faire des boucles sur les objets
- La boucle porte sur les clefs (keys) pour récupérer les values

```
//Présentation d'un objet

const notreObjet = {
    propriete1:"Chaine de caractère",
    propriete2:200,
    method:function(){},
    propriete4:{
        prop1:true,
        prop2:["Un","Deux","Trois"]
    },
    propriete5:["Nous","Vous"],
}
```

```
method(){}
```

- ✓ Tableau des valeurs des propriétés: $Object.values(notreObjet)$
- ✓ Tableau des noms des propriétés: $Object.keys(notreObjet)$

```
//Parcourir les clefs de l'objet

   for (let champ in notreObjet) {

         alert(`person[${champ}]  = ${notreObjet[champ]}`)

         console console(`Element no ${indice} = ${elem}`)
    }
```

Les tableaux en javascript

- Les tableaux (arrays) sont très importants en javascript (en savoir plus)
- Ils sont définis comme suit

```
//Présentation d'un tableau (array)

const notreArray=["habits","cahiers","livres","chaussures"]
```

Accéder au contenu d'un tableau

notreArray[indice] ⟶ valeur

Exemple: notreArray[0] ⟶ "habits"

- Un tableau peut contenir des valeurs de différents types
- les éléments des bases de données sont retournés sous forme de tableau d'objets
- La connaissance de certaines fonctions(méthodes) des tableaux nous sera très utile

Les tableaux en javascript

Quelques méthodes très utiles pour les tableaux (en savoir plus)

Méthode	Description
map()	Parcours tout le tableau et retourne un nouveau tableau.
filter()	Filtre un tableau et retourne tous les éléments remplissant la condition fixée
find()	Trouve le premier élément remplissant une condition donnée
findIndex()	Trouver l'index du premier élément remplissant une condition
join()	Transforme le contenu d'un tableau en une chaine de caractère
push()	Ajoute un nouvel élément a la fin du tableau
pop()	Supprime le dernier élément du tableau
slice()	Divise un tableau a partir des indices donnés en paramètre
includes()	Vérifie si le tableau contient le paramètre passé en argument

Déstructuration des tableaux et objets

- La déstructuration (destructuring) permet de choisir certains contenu d'un objet ou d'un tableau en javascript
- Elle est très utilisée dans les frameworks de javascript (React, Vue, Express…)

```
//Tableau de départ
const notreArray=["habits","cahiers","livres","chaussures"]

//Taper la ligne ci-dessous dans node
const [a, b, c] = notreArray

//Chaque variable prend la valeur se trouvant a sa position
dans le tableau de départ
a = "habits"
b = "cahiers"
c = "livres"
```

```
//objet de départ
const notreObjet = {
    type:"Chaine de caractère",
    prix:200,
   }
//Taper la ligne ci-dessous dans node
const {type, prix} = notreObjet

//Le résultat est similaire
type = "Chaine de caractère"
prix = 200
```

Attention: pour les objets, les noms doivent rester identiques

Les opérateurs Rest et Spread

- L'operateur spread (…) permet de recopier le contenu d'un tableau ou d'un objet
- Cela évite les problèmes de référence sur les objets
- L'operateur (…) rest contient le reste du tableau après une déstructuration

```
//Tableau de départ
const notreArray = ["habits","cahiers","livres","chaussures"]

//Nouveau tableau (même élément)
const newTableauRef = notreArray

//Garder le reste du tableau (... se trouve a gauche de l'égalité)
const [a, ...rest] = notreArray    => rest = ["cahiers","livres","chaussures"]

//Nouveau tableau (copie )
const newTableau = [...notreArray]   // spread = ... se trouve a droite de l'égalité
```

Fonction (arrow function)

- Très utilisée dans la programmation asynchrone car succincte
- Attention à la version sans argument

```
//Function Classique (asynchrone.js)

function sum(a, b) {
    return a + b
}
//Autre version

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

```
//Fonction lambda (asynchrone.js)

const sum = (a, b) => {
    return a + b
}
//Pour des fonctions simples

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

//Parenthèses obligatoires, sauf pour le cas d'un argument!!

const afficher = () =>console.log('Je suis sans argument!')
```

Concept d’asynchronisme

Considérons cet exemple

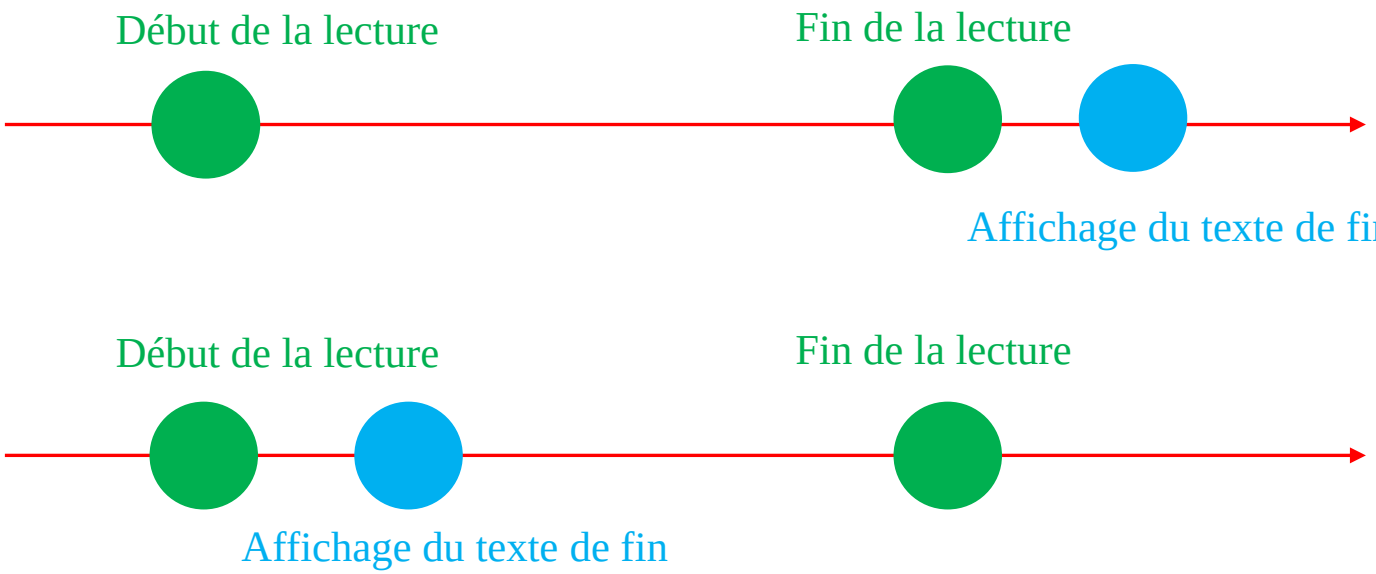
Programme bloquant (synchrone)	Programme non-bloquant (asynchrone)

`import fs from 'fs’`
 	
`let data = fs.readFileSync('./message.txt')`
 	
`console.log(data.toString())`
 	
`console.log('Fin de notre programme')`
	`import fs from 'fs’`
	
	`fs.readFile('./message.txt',function(erreur, asyncData){`
	`  if(erreur) console.log('Il y’a un souci')`
	`  console.log(asyncData.toString())`
	`})`
	
	`console.log('Fin de notre programme')`

node .\code.js Bonjour mes amis une fois Bonjour mes amis deux fois Fin de notre programme	node .\code.js Fin de notre programme Bonjour mes amis une fois Bonjour mes amis deux fois

Concept d’asyncronisme

Les étapes de l’exécution des deux programmes



1er cas

- le programme attend la fin de la lecture avant de passer aux instructions suivantes
- Il est bloquant et lent

2e cas

- le programme continue avec les instructions suivantes sans attendre
- Le résultat de la lecture est affiché des que la lecture se termine
- Non-bloquant et rapide

Les callbacks

- Fonction appelée à l'intérieur d'une autre fonction asynchrone et exécutée à la fin de cette dernière
- Très souvent sous forme de fonction lambda anonyme
- On peut définir la fonction à l'extérieur et l'appeler dans la fonction asynchrone

```
import fs from 'fs'

fs.readFile('./message.txt', (erreur, asyncData)=>{
    if(erreur) console.log('Il y'a un souci')
    console.log(asyncData.toString())
  }
)
```

Fonction lambda anonyme

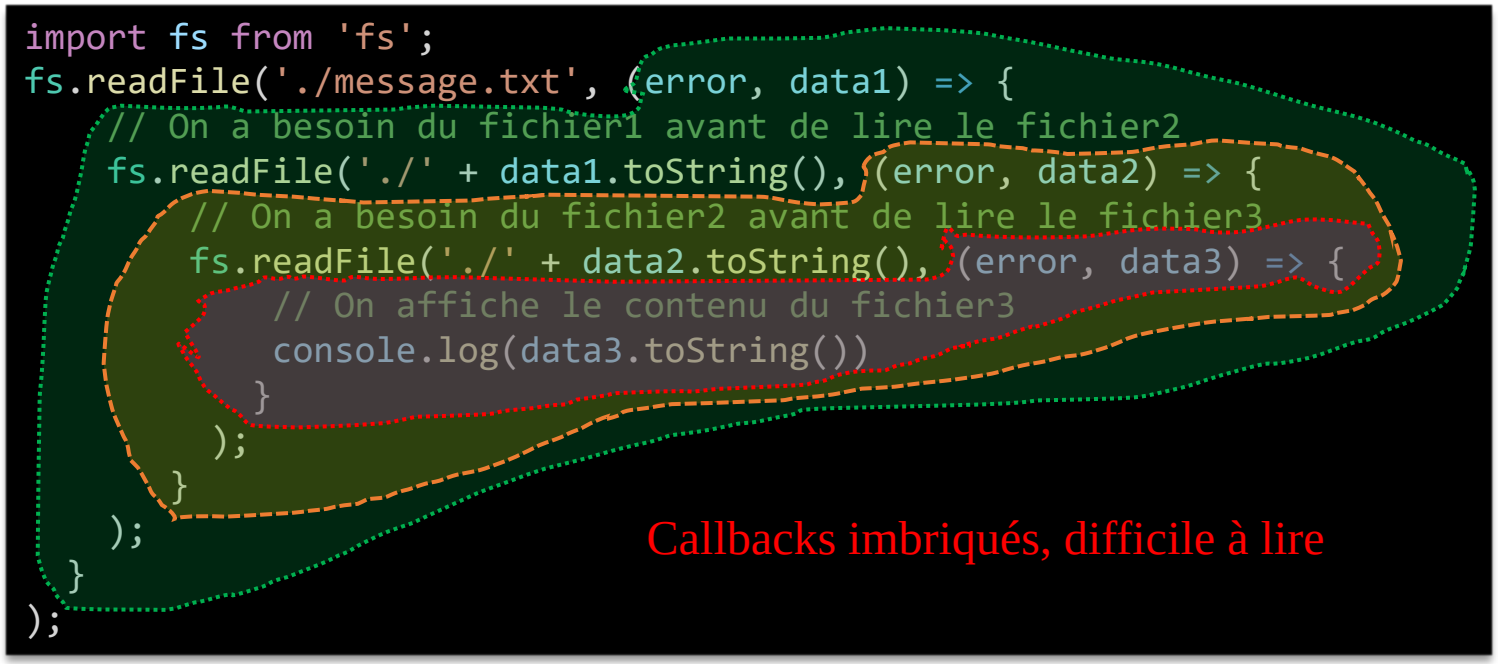
```
import fs from 'fs'

fs.readFile('./message.txt', myFunction)
```

Fonction créée à l' extérieur

Les callbacks

Enchainement des callbacks



Solutions possibles

- ✓ Modularisation
- ✓ Éviter les fonctions anonymes
- ✓ Utilisation des promesses

Inconvénient principal: le chainage de ces fonctions (voir ici)

Les promesses

Une solution pour simplifier la programmation asynchrone

- Une alternative aux callbacks
- Elles simplifient les imbrications des actions

```
import fs from 'fs'

fs.readFile('./message.txt',(err,data)=>{

  if(err) console.log('un souci')

  console.log(data.toString())
})
```

Lecture de fichier avec callbacks

```
import { readFile } from 'fs/promises';

readFile('./message.txt')

    .then(data => console.log(data.toString()))

    .catch(err => console.log('un souci', err))
```

Lecture de fichier avec les promesses

Les promesses

- Création d'une promesse se fait avec le mot clef async (en savoir plus)

```
const promesse = async (parameter) => {
    return parameter
}
```

- Utilisation/appel d'une promesse

Avec le bloc **then catch** utilisant des callbacks

```
promesse(10)
  .then(output => console.log('Sortie',output))

  .catch(err => console.log('Erreur',err))
```

Forme très pratique dans les Frameworks frontales

Avec **await** pour avoir la valeur dans une variable

```
const output = await promesse(10)

console.log('Sortie',output)
```

Forme assez simple sans callbacks
Utilisée si une promesse est appelée dans une autre

Les promesses

- La création d'une promesse se fait avec le mot clef async (en savoir plus)

```
const promesse = async (parameter) => {
    return parameter
   }
```

- Utilisation/appel d'une promesse

Avec le bloc **then catch**

```
promesse(10)
  .then(output => console.log('Sortie',output))
  .catch(err => console.log('Erreur',err))
```

Avec **await** et la gestion des erreurs

```
try {
    const output = await promesse(10)
    console.log('Sortie', output)
} catch (err) {
    console.log('Erreur', err)
}
```

Conclusion

- Comprendre les objets et les tableaux sur javascripts
- Savoir définir des fonctions
- Importance de la programmation asynchrone
- Callbacks et les promesses

Exercices

1. Soit le tableau suivant (on utilisera les fonctions des tableaux tant que possible)
   a. Ecrire une fonction qui permet de supprimer un élément du tableau et retourne un autre tableau
   ```
   const supprimerElement = (tableau, element) => {}
   ```
   b. Créer un autre tableau contenant seulement les éléments de rang impair
   c. Trouver le dernier élément du tableau
   d. Ecrire une fonction pour supprimer le nième élément du tableau
   ```
   const supprimerNthElement = (tableau, n) => {}
   ```

Exercices

2. Créer un module contenant les fonctions suivantes
   a. une fonction qui permet de trouver l'élément central (ou les éléments centraux si le nombre d'élément est pair) d'un tableau

```
const milieuTableau = (tableau) => {}
```

   b. une fonction qui permet de retourner la clef d'un objet correspondant à une valeur passée en argument

```
const clefValeur = (objet, valeur) => {}
```

   c. Ecrire une fonction qui permet de comparer
      - Deux tableaux
      - Deux objets

```
const comparerTableaux = (tableau1, tableau2) => {}
const comparerObjets = (objet1, objet2) => {}
```